Performance Prediction for Concurrent Database Workloads

Jennie Duggan
Brown University

Ugur Cetintemel
Brown University

Olga Papaemmanouil
Brandeis University

Eli Upfal
Brown University

ABSTRACT

Current trends in data management systems, such as cloud and multi-tenant databases, are leading to data processing environments that concurrently execute heterogeneous query workloads. At the same time, these systems need to satisfy diverse performance expectations. In these newly-emerging settings, avoiding potential Quality-of-Service (QoS) violations heavily relies on performance predictability, i.e., the ability to estimate the impact of concurrent query execution on the performance of individual queries in a continuously evolving workload.

This paper presents a modeling approach to estimate the impact of concurrency on query performance for analytical workloads. Our solution relies on the analysis of query behavior in isolation, pairwise query interactions and sampling techniques to predict resource contention under various query mixes and concurrency levels. We introduce a simple yet powerful metric that accurately captures the joint effects of disk and memory contention on query performance in a single value. We also discuss predicting the execution behavior of a time-varying query workload through query-interaction timelines, i.e., a fine-grained estimation of the time segments during which discrete mixes will be executed concurrently. Our experimental evaluation on top of PostgreSQL/TPC-H demonstrates that our models can provide query latency predictions within approximately 20% of the actual values in the average case.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems Modeling Techniques

General Terms

Measurement, Performance

1. INTRODUCTION

Concurrent query execution facilitates improved resource utilization and aggregate throughput, while making it a challenge to accurately predict individual query performance. Modeling the performance impact of complex interactions that arise when multiple queries share computing resources and data is difficult albeit critical for a number of tasks such as Quality of Service (QoS) management in the emerging cloud-based database platforms, effective resource allocation for time-sensitive processing tasks, and user experience management for interactive database systems.

Consider a cloud-based database-as-a-service platform for data analytics. The service provider would negotiate service level agreements (SLAs) with its users. Such SLAs are often expressed in terms of QoS (e.g., latency, throughput) requirements for various query classes, as well as penalties applied if the QoS targets are violated. The service provider has to allocate sufficient resources to user queries to avoid such violations, or else face consequences in the form of lost revenue and damaged reputation. Thus, it is important to be able to accurately predict the runtime of an incoming query on the available machines, as well as its impact on the existing queries, so that the scheduling of the query does not lead to any QoS violations. The service provider may have to scale up and allocate more cloud resources if it deems that existing resources are insufficient to accommodate the incoming query.

This paper addresses the performance prediction problem for analytical concurrent query workloads. Specifically, we study the following problem: "Given a collection of queries $q_1, q_2, q_3, ..., q_n$, concurrently executing on the same machine at arbitrary stages of their execution, predict when each query will finish its execution."

To predict QoS, we target time elapsed rather than metrics such as throughput or resource utilization. We elected execution latency because in practice it has the greatest impact on user experience. Users are generally more concerned about when a query returns its output, rather than how many are sharing a host or the number of physical I/Os required.

We propose a two-phase solution for this problem.

- **Model Building** We build a composite, multivariate regression model that captures the execution behavior of queries under concurrency. We use this model to predict the logical I/O latency for each query in a given workload mix.
- **Timeline Analysis** We analyze the execution timeline of the workload to predict the termination points for individual queries. This timeline analysis starts by predicting the first query to complete and then repeatedly performs prediction for the remaining queries.

One of our key ideas is to use a *logical* I/O-based metric, called BAL, to quantify the performance impact of concurrently executing queries. This simple yet highly-predictive metric gives us an aggregate level cost without having to model the more complex underlying physical systems individually.

Furthermore, we extend our modeling framework to support time-changing workloads by calculating incremental predictions of the execution latency for discrete mixes as they occur. When an incoming query is added to a mix, we project how it will impact the

SIGMOD'11, June 12–16, 2011, Athens, Greece.

currently running queries as well as estimate the execution latency for the new addition. We extend this system to allow for longer-term resource planning by modeling a queue of queries.

This paper makes the following novel contributions:

- We argue and experimentally demonstrate that BAL, a simple metric that captures the latency of a logical I/O, is a robust indicator for query performance even in the presence of concurrency. BAL facilitates the modeling of the joint effects of physical I/O delay and memory contention with a single value, normalized over the execution of the query.
- We develop a multivariate regression model that predicts the impact of higher-degree concurrent query interactions derived primarily from isolated (degree-1) and pairwise (degree-2) concurrent execution samples.
- We show experimental results obtained from a PostgreSQL / TPC-H study that supports our claims and verifies the effectiveness of our predictive models. Our predictions are on the average within nearly 20% of the actual values, while timeline analysis leads to additional improvements, reducing our average error to an average of 9% with periodic prediction re-evaluation.

The rest of the paper is organized as follows. Section 2 looks at model building for query interactions by examining changes in logical I/O latency. After that we consider how to predict the logical I/O latency metric in Section 3. Next we examine training techniques for these models in Section 4. Immediately following in Section 5 we leverage these models to produce end-to-end latency estimates by simulating the lifetime of the query. Section 6 provides experimental evaluation of this work. In Section 7 we examine related work, and finally we conclude in Section 8.

2. QUERY LATENCY INDICATORS

The goal of our framework is to predict the response latency of concurrently running queries. Query latency is directly affected by resource contention in the underlying hardware. As the number of queries that are presently executing goes up their query latency increases as they share access to disk, CPU time as well as memory. Therefore our first step was to identify effective *query latency indicators*, i.e., metrics that can capture the resource contention in the underlying hardware and can be used to predict the latency of concurrent queries. In this section we discuss the various query latency indicators we examined. We also explore in more detail our most effective indicator, the *Buffer Access Latency (BAL)* metric. We continue with a brief description of our workload.

2.1 Analytical Workload

In this work we target analytical workloads. We assume that all queries are derived from a set of known query classes (e.g., TPC-H templates) and that they are primarily I/O-bound. We target medium-weight queries. For the context of this work, we define medium-weight queries as the 10 classes in TPC-H with latency on and around the median latency of all of query classes.

The query classes in our workload stress different parts of our system and the impact of concurrency on their latencies fluctuates at varying magnitudes. In Table 1, we study the variance of the 10 medium-weight TPC-H query classes. The tables shows the variance of query latency (and their average latency) when queries are executed in different multiprogramming levels (i.e., number of queries executing concurrently). We note that a multiprogramming level of x means that x queries run concurrently. For all queries, their average latency increases as we increase the degree of concurrency. Moreover, the variance of latencies also increases. Some query classes, such as 7 and 18 exhibit much more uncertainty in their latency under concurrency. These queries are very I/O-bound and require complex data access patterns of cascading joins and aggregation. This causes them to be more sensitive to the queries they are running with because it causes them to speed up or slow down based on the amount of sharing that goes on among their working set. In contrast, query class 6 has a simple execution plan (consisting of a table scan and a light aggregate). In this work we formulate predictions for individual queries based on their template and the query mix(es) they are a part of.

Query Class	MPL 2	MPL 3	MPL 4	MPL 5
3	456 (141)	2663 (178)	2596 (214)	10750 (279)
4	398 (156)	1883 (199)	3564 (223)	5638 (279)
5	56 (129)	2219 (160)	3539 (198)	10424 (257)
6	345 (114)	160 (126)	1115 (160)	1666 (185)
7	343 (155)	2547 (197)	4595 (222)	9050 (307)
8	218 (136)	1231 (151)	7771 (225)	5236 (253)
10	126 (141)	2139 (203)	2699 (214)	6667 (270)
14	408 (117)	430 (126)	909 (159)	1681 (183)
18	98 (141)	2630 (186)	2044 (206)	18821 (312)
19	488 (119)	694 (147)	1637 (177)	1354 (192)

Table 1: Variance in observed latencies (in seconds) for different multiprogramming levels (MPLs). In parentheses are the average latencies under the same conditions.

2.2 I/O-Based Indicators

Our framework addresses concurrent query performance predictions for analytical queries. These workloads include mostly read-only operations and typically involve high number of I/O operations. Our first goal was to identify a metric that is strongly correlated with query latency of OLAP queries and can capture the impact of concurrency on the query performance. Since I/O operations are the dominant factor of query latency we aimed to identify query latency indicators that quantify how the cost of I/O operations changed due to concurrently running queries.

We can think of an I/O operation as a sequence of access methods. When fetching a page for a query, a request is first sent to the buffer pool which is the fastest way to access data. If the requested page is not found, the lookup is passed on to the OS-level cache, which is the second fastest option. Finally, if this fails, the request is enqueued for disk access to retrieve the block containing the requested data. This disk access may be a seek or a sequential read, causing the cost to access a data block to vary dramatically. Given this sequence of access methods, concurrency affects the number of I/O operations required by a query as multiple queries need to share the buffer pool and OS-level cache and the physical disk.

We hypothesized that *block reads* from disk, *buffer pool hits* and a *multivariate model* of the two would be well-correlated with latency. We experimented with using linear regression models for predicting query latency based on these metrics for different multiprogramming levels. Our results are shown in Table 2. Our experimental setup is detailed in Section 6.1. The table shows the prediction error for each of these metrics. In general, we found that these metrics were all disrupted from performing well by inherent variance in the latency of a block from disk. In the next paragraphs we discuss the limitations of each of these metrics.

I/O Block Reads. The first metric we considered was counting the number of physical blocks read by a query. If a query's execution speed is highly tied to how fast it can complete I/Os (since

Query	I/O Blocks	BP Hits	Both	BAL
Avg	25%	25%	24%	5%
3	29%	29%	28%	5%
4	24%	25%	24%	6%
5	30%	30%	29%	3%
6	17%	17%	15%	3%
7	26%	27%	25%	4%
8	26%	27%	24%	3%
10	28%	29%	28%	6%
14	21%	21%	17%	3%
18	27%	26%	30%	12%
19	22%	22%	20%	3%

Table 2: Mean relative error ($\frac{|observed-prediction|}{observed}$) of I/O-based latency indicators trained at MPL 1-5.

reads from memory have low impact on the latency), then the number of blocks read could give us an indicator of how much query progress has been accomplished over time and how much time it requires to complete.

Table 2 shows that the latency prediction based on blocks read has an error less than 29% in most cases. This implies that there is a correlation between number of blocks read in a query's execution time and the query's latency. However, for certain query classes, such as 3 and 5, this prediction is weaker. The prediction is less strongly correlated because these queries spent considerable time on their CPUs, and during that time they relinquish control over the I/O bandwidth. These CPU-intensive query plans cause the queries to be more sensitive to the degree of overlap between their working set and that of the queries that they are executing alongside of. This is because the queries frequently make short data accesses punctuated by longer periods of CPU usage. Thus if the disk arm remains roughly in the same region that they are executing from, the cost of an I/O will remain relatively uniform. But if the data access patterns are more random, this uncertainty will be reflected in the cost of an I/O operation.

Buffer Pool Hits. We also studied if the buffer pool hits could be used as query latency indicator. Our assumption was that the latency overhead of disk access may be relatively fixed and that we could subtract the I/O operations saved by each buffer pool hit from this base overhead to estimate the query latency. We found that this metric was correlated with latency, but the prediction errors were similar to those of the I/O block reads indicator. There were a number of factors contributed to this, including the fact that the overall system has multiple layers of in-memory storage (i.e., buffer pool, OS cache) and so it was difficult to infer the savings due to memory. Also, these savings did not adequately capture the changing behavior of I/O operations, i.e., the physical blocks read and written. The mix of seeks and sequential reads was not fixed, making it harder to obtain consistently good predictions.

I/O Block Reads & Buffer Pool Hits. Finally, we experimented with multivariate regression on both of these metrics. Our assumption in this case was that these two metrics would complement each other since buffer pool hits replace disk reads (i.e., a hit to the buffer pool allows us to skip a disk access). Table 2 show that this model did not significantly improve our prediction errors.

The results in Table 2 demonstrate that the number of physical I/O blocks read and buffer pool hits are moderately correlated with query latency. This is primarily because there is a high degree of variance with regard to how long it takes to bring an individual block in from disk. If the block is a part of a sequential read, then it may occur quickly. However seeks are highly variable in their cost. Physical I/O latency per block is proportional to the distance that

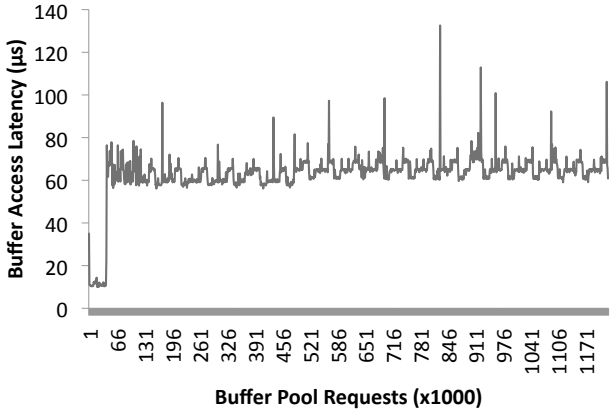

Figure 1: BAL as it changes over time for TPC-H Query 14. Averaged over 5 examples of the template in isolation.

the disk arm moves to fetch a target block. Thus in practice, the speed of reading an individual block of data can vary by up to an order of magnitude. This causes both the I/O count and buffer pool hits to have distorted latency predictions as they model the average I/O latency, when the quantity has an unknown (and routinely changing) distribution.

2.3 Buffer Access Latency (BAL) as an Indicator

We found that handling each of block access methods individually had limited utility because the interactions were too complex. To simplify the problem, we identified the initial request to the buffer pool as a gateway that all queries must go through to receive their data. When a query requests a block of data, it submits its request to the buffer pool manager. When the buffer pool receives a request it queries its levels of storage one after another until it acquires the needed disk block. The time that elapses on average between when a request is issued and when the necessary block is returned constitutes the average delay for a *logical I/O request*.

Therefore, rather than modeling each the steps of a buffer pool request independently, we use the average latency of a logical I/O operation as our query latency indicator. We call this metric *buffer access latency* or *BAL*. Averaging this metric over the duration of a query allows us to capture the interactions of disk seeks, sequential reads, OS cache hits and buffer pool hits in a normalized manner.

Each average BAL is typically an aggregate of more than a million logical I/O requests, so this allows us to infer changes in the mix of I/O types (sequential reads, seeks, etc.) for individual queries. A low average BAL when the query runs in isolation implies that we have a lightweight query that reads most of its data from memory. In contrast, high BAL readings mean that a query is requesting diverse data that is not memory-resident such as a fact table scan or an index scan, necessitating expensive seeks.

To measure the BAL metric we modified the database engine we used (PostgreSQL 8.4.3). Specifically, we intercepted the point in the buffer pool manager where the query executor issues page requests. A timer starts when the request is issued to the buffer pool. If the page is not found, then the process refers the request to the storage manager, which interfaces with the disk and operating system. The timer stops when the requisite page is returned to the query executor, regardless of its source. To generate the BAL metric, we sum up the time elapsed for each page request and divide it by the logical number of pages accessed by our query plan. In Table 2 we examine the quality of BAL in comparison to the other

I/O-based indicators we studied to predict latency. Our prediction errors drop to around 3%-6% in most cases.

One of the reasons that BAL is a good indicator for latency is because it is averaged over many samples over the lifetime of a query. Thus, even when a query goes through very complex interactions with the OS as well as other queries, it will still converge to an accurate prediction of the overall latency.

In Figure 1 we display how the typical values of BAL vary through the lifetime of a query when run in isolation. All of our BAL measurements are averages of 1000 buffer pool requests. This particular query (of TPC-H template 14) goes through a brief period of low latency in BAL while the query is warming up (i.e., it is loading all of the database binaries). It then reaches a relatively stable state where it is continuously reading blocks and joining tuples. The BAL periodically fluctuates as the query cycles through seeks and sequential reads in the course of a join. When we consider all of the queries in our workload the average standard deviation was around 33% of the query's BAL average. This variance can be attributed to the inherent noisiness of the metrics BAL is capturing (i..e, a mixture of buffer pool reads, seeks and sequential physical I/Os). BAL is a robust metric because it captures the aggregate of these complex interactions over many logical page requests, converging on a stable average behavior.

2.4 Predicting Query Latency using BAL (B2L)

We used a linear regression model to predict the end-to-end query latency using the BAL metric. The intuition is that an I/O-bound query will have latency directly proportional to how long it waits for a logical I/O on average plus some relatively fixed overhead for CPU time and fixed database support. So if the average BAL for a query α is B_α and O_α is its fixed CPU overhead then the latency of the query α L is given by the equation:

$$L = O_\alpha + P_\alpha \times B_\alpha \quad (1)$$

Our system finds the fixed overhead of each query class and this comprises the y-intercept (O_α) of the model. The P_α coefficient allows us to model the number of logical page requests required by the query plan. Therefore, if the average latency of a logical I/O (i.e., the BAL metric) of a query running in a query mix is known, we can predict its latency using the above model. We call this end-to-end latency prediction model *BAL to Latency* or *B2L*.

We use the coefficient of determination (R^2) to assess the goodness of fit of the model to the data. 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 to 1, with higher values denoting a better fit. It captures the error of the model, SS_{err}, by calculating the sum-squared errors of each data point in the training set, normalized by the total variability of the data, SS_{tot}, calculated by taking the sum of squared deviations of the individual data points from the mean. R^2 is calculated as $1 - SS_{err}/SS_{tot}$.

The strength of this above model is demonstrated in its R^2 values for each query class in our training set. We train on an average 259 examples of each template. The R^2 coefficients varied from 0.83-0.99 for the 10 queries in our TPC-H workload; all of our queries were very well-fitted by this model.

We found that TPC-H templates conform well to this model. Queries within the same class rarely change their query execution plan. We capture this consistency with the slope of our linear model. In practice, we found that our query templates did not have their I/O request count vary by greater than 5%, despite operating with range predicates and indices. In some cases, such as a skewed data distribution or a highly varying range predicate, we would need to create multiple models per query template. In that case, we would subdivide our templates into predicate ranges as described in [1].

More details of our experiments are shown in Section 6. In summary, BAL was our strongest query performance indicator. In the next section, we discuss how this metric can be predicted for concurrent workloads.

3. BUFFER ACCESS LATENCY PREDICTION

As we explored in the previous section, the average buffer access latency of a query is a consistent indicator of its latency throughout changing query mixes and concurrency levels. In this section, we propose a technique for predicting this metric by modeling the interactions of query templates based on observations among pairs. At a high level, we aim to quantify how a query's performance changes as it shares resources with competing processes. To capture this effect, we first characterize how a query class runs in isolation. We then extrapolate how interacting with other query classes impacts its performance.

3.1 Modeling Query Interactions

To predict the concurrent performance for a query class, we need to quantify its average BAL metric under contention. In this section we create a prediction model for BAL for light-to-moderate levels of query contention. We start by examining (a) the resource utilization of a query class under optimal conditions (i.e., queries running in isolation), and (b) the resources that concurrent queries will expect to consume. For our purposes, resource needs refer primarily to the I/O bandwidth and memory used, whose joint effects are captured by the BAL of each query. We estimate how much these queries will contend with each other on a template-by-template basis, by examining how their BAL is affected when they are interacting in pairs.

For example, when TPC-H Query 6 is run with Query 14, both enjoy a speedup of about 10 seconds on average. This is because both of their query execution plans are dominated by sequential scans on the fact table. In contrast, when Q19 is run with Q7, they slow down by approximately 50% and 30% respectively. In this case, the two queries share less overlap in their working sets and both require significant memory to do expensive joins.

To better characterize how queries interact, we first looked at all pairwise interactions. For a workload of n queries, we executed all unique combinations (a total of 55 pairwise combinations in our case of 10 query classes). These pairings allowed us to characterize the magnitude and sign (positive or negative to denote slow down or speed up respectively) of how each query affects others in the workload. We also studied the incremental utility of building our model on multiple, higher-degree interactions, which did not yield any consistent or notable improvements while being more expensive. Hence, we concluded that models based on pairwise interactions, enhanced with additional statistics, strike a good balance between practicality and accuracy.

Specifically, by analyzing how pairs affect each other, we model the BAL at higher degrees of concurrency. We do this by building a composite of the BAL based on a base cost of the individual BALs of the queries in isolation. We then incorporate the cost of interactions indicated by the change in BAL due to direct and indirect interactions among the queries. We use this pairwise interaction rate and isolated runs to build our multivariate regression model. We call this BAL estimation model *BALs to concurrent BAL* or *B2cB*, which we describe in detail next.

Variables	R^2
I	0.167
I & C	0.169
I & C & D	0.219
I & C & D & G	0.358

Table 3: R^2 values for different BAL prediction models (variables: Isolated (I), Complement Sum (C), Direct (D), Indirect (G)) Training on multiprogramming level 3.

3.1.1 Predicting BAL Under Concurrency

In our model for predicting the concurrent BAL we distinguish between *primary* and *complement* queries. When we predict the concurrent BAL of a query, we call it the *primary*. Queries that are running concurrently with the primary are its *complements*. The intuition behind our model is that when we are dealing with a non-overloaded state, the cost of contention increases linearly and is a function of fixed costs for each query in the mix as well as variable costs for each pairwise interaction among the queries (divided into direct and indirect interaction with the primary query). For example, if we assume round-robin scheduling, then the cost (i.e., delay) to obtain a data block with concurrency (i.e., its concurrent BAL) equals to the cost to obtain a data block when no concurrent queries exist, plus the cost of one average page read of each of the complement concurrent queries, plus any observed interaction cost among all of the queries in the workload.

Our BAL prediction model has the following four independent variables provided from training data:

- *Isolated*: This variable represents the BAL of the primary query when it is run in isolation. Examining isolated BAL creates a baseline to assess the concurrent BAL. It provides us the default page access time for the primary query under optimal circumstances. We denote the isolated BAL of a query i as T_i.

- *Complement*: Similarly, we need to consider the resource requirements of the complement queries in our mix. The complement variable is the sum of complement queries' isolated BALs. Summing the BAL of the complements allows us to build an estimate of the rate at which they consume I/O bandwidth given fair scheduling. This is a rough estimate of the "cost" of complement queries.

- *Direct*: This variable is the sum of the change in BAL for the primary when paired with each of its complements. The direct variable captures the change in the primary's BAL from direct query interactions. We quantify a direct query interaction $T_{i/j}$ as the average BAL of query i measured while running concurrently with query j. The change in this measurement is $\Delta T_{i/j} = T_{i/j} - T_i$. If this quantity is positive, we are experiencing slowdown. If it is negative, then we have a beneficial interaction between queries.

- *Indirect*: Finally we examine the change in BAL between the complement queries. This gives us a glimpse into how much contention they are creating and how much their interaction will affect the primary query. Note that their interactions may not be symmetrical ($\Delta T_{i/j} \neq \Delta T_{j/i}$). For example, an equi-join requires dedicated memory to complete its comparisons efficiently. If it is run concurrently with a table scan, the table scan will not be as affected by the join because it uses each tuple only once. The scanning query disproportionately affects the latency of the joining query due to it having access to less memory. We define the indirect variable as the sum of the change in BAL for each complement when run with each other complement.

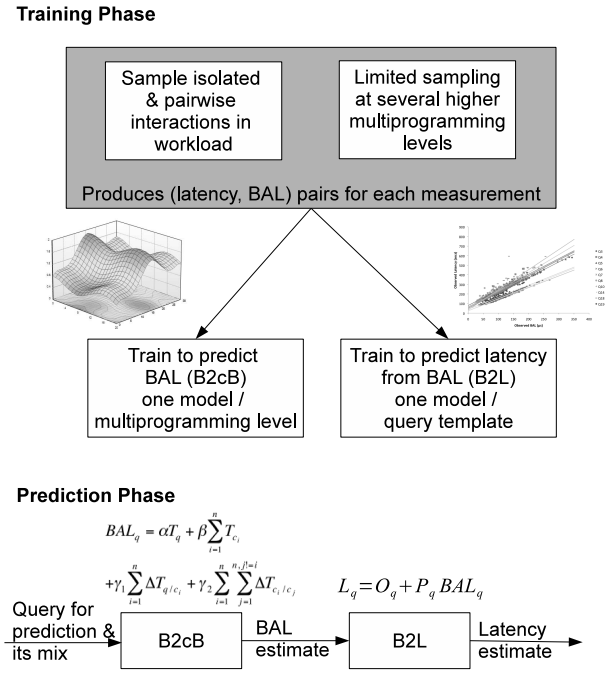

Figure 2: System model for query latency predictions of arbitrary mixes.

We found that all of these independent variables were necessary to build a complete model. As Table 3 demonstrates, the quality of our prediction model, as measured by the R^2 coefficient, improves significantly once we start considering isolated costs with interactions. Moreover, the sum-squared error is improved by greater than 50% when solving for four variables rather than relying solely on the isolated BAL.

We predict average concurrent BAL for query q while running concurrently with its complement $c_1..c_n$ as:

$$B = \alpha T_q + \beta \sum_{i=1}^{n} T_{c_i} + \gamma_1 \sum_{i=1}^{n} \Delta T_{q/c_i} + \gamma_2 \sum_{i=1}^{n} \sum_{j=1}^{n, j!=i} \Delta T_{c_i/c_j} \quad (2)$$

We solve for the coefficients α, β, γ_1 and γ_2 using linear multivariate regression for all queries in our training set, once per each multiprogramming level. We elect to do this regression at this granularity to model several degrees of contention. The regression derives our coefficients using the least sum-squared error method.

Predicting end-to-end latency. The lower portion of Figure 2 demonstrates the steps of our predicting end-to-end query latency for varying concurrency levels. Given a query and its mix of complements, we can rapidly create an average concurrent BAL estimate using the B2cB model. The concurrent B2cB is the input to our B2L model which provides the final latency estimate.

To provide a more concrete example, if we are predicting the BAL of query a and it is being run with queries b and c, we start out with the following inputs:

- T_a, T_b, T_c - the average isolated BAL of a, b and c.

- $\Delta T_{a/b}, \Delta T_{a/c}$, etc. - change in BAL of a when run with b or c ($\Delta T_{a/b} = T_{a/b} - T_a$)

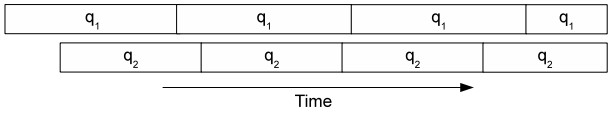

Figure 3: An example of steady state sampling for queries q_1 and q_2.

We predict the average BAL B_a of query a using Equation 2:

$$B_a = \alpha T_a + \beta(T_b + T_c) + \gamma_1(\Delta T_{a/b} + \Delta T_{a/c}) + \gamma_2(\Delta T_{b/c} + \Delta T_{c/b})$$

Then, we can predict the end-to-end query latency using Equation 1. We would apply the coefficient P_a, or the logical page coefficient for this query class and O_a, the fixed overhead for a. We would then predict our end-to-end latency by calculating:

$$L = O_a + P_a \times B_a$$

4. TRAINING THE PREDICTION MODELS

To obtain performance estimates, we need to train our prediction models. The training phase consists of running our workload in isolation, pairs as well as at several higher multiprogramming levels. This provides the coefficients for evaluating our B2L and B2cB prediction models. All of our samples are used to train both prediction systems.

To model query interactions flexibly, we must summarize their effects in a way that is agnostic to the offsets of individual query start times and representative of the variations within a query template. We model this by temporarily fixing the mix in which a query is participating and randomizing the offset at which each query starts. We call this measurement *steady state*. Steady state sampling allows us to reason about how a query will react to a continuous level of contention and a fixed set of complements. An example of steady state is shown in Figure 3

Furthermore, in order to obtain measurements that capture only interactions, our experiments are based on *warm cache*. i.e., we omit the first few samples. This allows us ignore to the overhead of the initial query set up and caching of small supporting structures This approach also allows us to sample the changing interactions from queries overlapping at different phases in their execution.

In the next paragraphs, we describe how we obtain our training set for the two prediction models (B2L and B2cB) which we described in the previous sections.

4.1 Sampling-Based Training Sets

In our framework we build our prediction model by sampling a subset of the configurations for which we are creating predictions. Experiment-driven modeling allows us to approximate the landscape of how queries in a workload interact. Our model samples queries in isolation, pairwise and higher degrees of concurrency for training. The use of our training runs are displayed in Figure 2.

First we characterize the workload that we have in terms of how each query behaves in isolation. This baseline allows us to get an estimate of what constitutes normal, unimpeded progress for a query class and how much we are speeding up or slowing down as new queries are added to the mix. In our experiments we average over many examples of each query class running in isolation with a warm cache and we record both the latency and BAL metric for each query class. The BAL-latency pair is used in the training of the B2L latency prediction model as well as as input for the first two terms of the B2cB model.

Next, we build a matrix of interactions by running all unique pairwise combinations, 55 in our case. This allows us to succinctly

Query	1	2	3	4	5
1		X			
2					X
3	X				
4			X		
5				X	

Figure 4: An example of 2-D Latin hypercube sampling.

estimate the degree of contention that each query class in a workload puts on every potential complement. As with our isolated measurements, we get both end-to-end latency as well as average BAL measurements for all of these combinations. These BAL-latency pairs are used for training the B2L model. In addition, they are used as inputs by the B2cB model to estimate BAL.

This moderate number of samples is necessary for both our BAL and latency predictions. The pairwise interactions provide us with inputs for the independent variables of B2cB. They also give us our input for our B2L model. B2L builds upon many concurrency levels in order to plot how latency grows as contention increases. Each multiprogramming level helps us complete the model.

Next, we build our model coefficients for interactions of degree greater than two. We sample at this level using Latin hypercube sampling (LHS). LHS uniformly distributes our sample selection throughout our prediction space. This is done by creating a hypercube with the same dimensionality as our multiprogramming level. We then select our samples such that every value on every plane gets intersected exactly once. Each sampling run of this technique produces at most n combinations, where n is the number of unique queries in our workload. A simple example of Latin hypercube sampling is shown in Figure 4. These sample runs are used to create the coefficients for B2cB at each multiprogramming level. We do multivariate B2cB linear regression on a set of Latin hypercube sampled data points.

LHS is a general sampling technique and is not a perfect fit for exploring this space of all possible query combinations. LHS does not take into account the difference between combinations and permutations when exploring our sampling space. For example, to the sampler, the combination of (3, 4) and (4, 3) would both be considered distinct samples. From the database's point of view they are both simply one instance of Q3 and one instance of Q4 running concurrently. We eliminated LHS in which the same combination appears more than once from our training set because each run produces data for all queries involved (i.e., when we execute mix (3,4), we record data for both Q3 and Q4).

For our training phase we used this sampling technique three times for each concurrency level. Experimentally we found that as more samples were taken we naturally get a more comprehensive picture of the cost of contention in our system. On the other hand, more sampling takes more time. We found that three LHS runs for each multiprogramming level was a good trade off between these competing objectives. Acquiring more samples did not improve our prediction accuracy.

Each LHS run consists of ten steady state combinations, resulting in 30 training combinations sampled for each multiprogramming level. Initially this may seem like many samples, but it is not that many in comparison to all possible combinations. This is especially true for higher multiprogramming levels where our set of combinations grows exponentially every time we add a query. The complexity of our training space does grow significantly as our workload becomes more complex. If we are training on a set of n queries, we must first execute it in isolation, incorporate it into all pairs and higher level sampling. For pairs, we sample all combinations with replacement, producing $(n^2 + n)/2$ steady state runs.

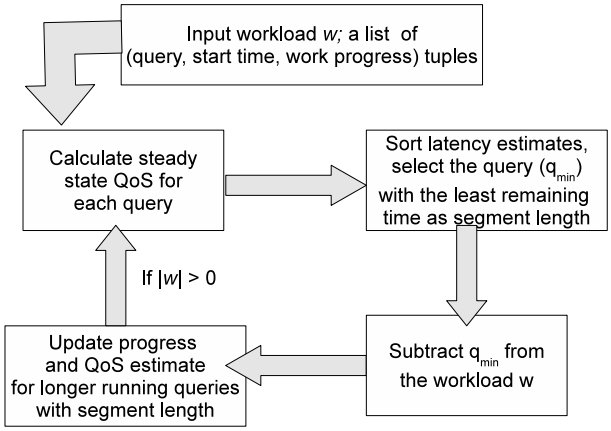

Figure 5: Just-in-Time evaluation flowchart

If we sample m multiprogramming levels greater than two and we run l LHS samples, our number of required training runs can be calculated as $n + (n^2 + n)/2 + lmn$. In our case, this comes to 155 sampled combinations.

In practice, the total training period took approximately two days in our modest setup. This may seem like considerable time, but we are only required to train once and can then use the results for any arbitrary mixes indefinitely. It is also worth noting that this up-front training period is what allows our model to be extremely lightweight once it reaches the evaluation phase (i.e., when it is producing query latency estimates). The cost of creating an estimate is negligible once the model is trained. It is only the cost of applying the B2cB model (summing the primary, complement, direct and indirect I/O contributions) and providing the output to a B2L model ($L = O_\alpha + P_\alpha * B_\alpha$).

5. TIMELINE ANALYSIS: PERFORMANCE PREDICTION FOR CHANGING MIXES

Using the B2cB and B2L methods we can predict the latency of individual queries being executed if the mix does not change during a query's execution. This system is useful for simple cases, where we only want an estimate for how long a query will run in a very homogeneous mix. However, in most circumstances the query mix over which we are evaluating is constantly changing as new queries are submitted by users and pre-existing ones terminate at varying intervals. For example, in a production system, managers and other decision-makers submit queries when they are at work and would benefit from an estimated time of arrival for the results. With modeling we can give them real time feedback of how long a new query will run and how much it will affect their currently executing workload using "what-if"-style analysis.

This type of system necessitates an incremental evaluation of a larger set of circumstances. We need to consider all of the mixes that will happen during a query's execution as the number and/or type of complement queries varies. This system must quantify the slowdown (or speedup) caused by these mixes, and estimate what percentage of the query's work will be accomplished in each mix.

We propose two formulations for evaluating our predictions. In the first scenario new queries are being submitted for immediate execution. We refer to this as *just-in-time (JIT) modeling (JIT)*. In JIT the number of queries monotonically decreases as members of the currently executing batch completes. In the second scenario we consider a queue-based approach, where our system is given a fixed multiprogramming level and an ordered list of queries to run. In this case we also model the mixes that will occur, by projecting when queries from the queue will start during the query's execution. Next, we describe these modeling scenarios.

5.1 Just-in-Time Modeling

Just-in-Time modeling allows us to ask the questions: "If I schedule a query now, how long will it take? How will it affect the presently running queries' completion times?" JIT estimates are more flexible in that they support more than one multiprogramming level in our estimate and model real-time changes in the workload.

Also, this more incremental approach allows us to refine our estimates as time progresses. As we predict latency every time a query is added we can correct past estimates by examining how the query has progressed since we last forecasted its end-to-end latency. In the context of an SLA, this may allow us to prevent QoS violations from happening by giving us the ability to intervene and load balance as time passes. Experimentally we saw an average approximately 10% error in our QoS estimates with this approach.

This JIT modeling requires estimates of the latency for each query in each mix and the percentage of its execution time each mix will occupy. The JIT algorithm is charted in Figure 5. In our timeline-based QoS estimator, first we look at the progress of all the n queries that are presently executing in a mix. We create a list of the time that has elapsed since each presently executing query began and initialize our performance estimate with this quantity. We also record the number of logical I/O requests that have been serviced for each query. This second metric gives us an estimate of what percentage of the query's work has been completed.

Next we must look at the estimated QoS for each query in the proposed mix, operating under the temporary assumption that the mix does not change. We can use the techniques in the previous section to create end-to-end latency estimates for each query in the workload under these steady state conditions. This first estimates BAL using the B2cB prediction model, which we then translate into latency using B2L model. We multiply the steady state estimate by the percentage of the query that we project remains based on the number of logical I/Os serviced.

After this we sort the steady state estimates and pick the query with the minimum remaining time as our first *segment* of evaluation. This defines the period over which we evaluate our first discrete mix. We select this query q_{min} and its estimated latency l_{min} as the time when our mix state will change next. We subtract q_{min} from the mix and we update the progress of each query q that is not equal to q_{min} by taking the ratio of l_{min}/l_q and multiplying it by the logical I/O requests remaining. We also add l_{min} to our estimate for each query in the workload that is not terminating. Finally, we subtract q_{min} from our workload because we project that it will have ended at this prediction point. We keep iteratively predicting and eliminating the query with the least time remaining until we have completed our estimates for all queries in the workload.

To summarize, we start with n queries and project a completion time for each in n segments. Each segment contains one less query than the previous segment as we subtract the query that we forecast will terminate next. At each concurrency level greater than two, we use our B2cB and B2L models to create QoS estimates. For isolated and pairwise cases we use the latencies recorded during the training phase.

5.2 Queue Modeler

Another scenario under which this system is be useful is for estimating how long a query will take if we have a fixed multiprogramming level. In [14] the authors discussed how multiprogramming level is a common "knob" for optimizing DBMS performance while scheduling concurrent resources. In this formulation,

Algorithm 1 Queue modeling algorithm where q_p is the primary query, p_i is the progress of query i, R_i is the number of logical page requests for i. Iterating over all queries in workload w

$q_{min} = \emptyset, t = 0$
while $q_{min} \neq q_p$ **do**
  for each q_i in w **do**
    $l_i =$ EstimateTimeRemaining(q_i, p_r, w)
  end for
  w = sort(l, q) // sort to find the query with the lowest remaining time estimate
  $t+ = l_0$ // add minimum time to primary's estimate
  for $i = 1; i \leq |w|;$ ++i **do**
    $p_i+ = (l_0/l_i) * (R_i - p_i)$
  end for
  $w_0 = get\_queue\_next()$ // replace the query with the least remaining time with the next one in the queue
end while
return t;

we model our changing mix by looking ahead in the queue when we project that our current mix will end. We refer to this scenario as a *Queue Modeler (QM)*.

The Queue Modeler requires access to a queue of queries submitted. QM works very similarly to JIT predictions, except it examines the currently executing workload and models the addition of the next query in the queue as a replacement when a current query will terminate. This system allows us to give an end-to-end estimate of progress without starting the execution of some of the queries that are included in our prediction.

The Queue Modeler is shown in Algorithm 1. Queue modeling starts with a list of status information for each presently executing query, much like JIT. This too is a pair of the query execution time and the progress the query has made in terms of logical I/O requests. We add the new query to the list (q_p) with its progress and current latency at zero. Next we model the steady state latency of each query in the mix and sort the resulting estimates of latency remaining. The query with the least projected time remaining, q_{min}, is estimated to continue running for l_{min} time. Next we update the progress of the remaining queries by taking the ratio of l_{min} to their projected time remaining. We update their latency estimate by adding q_{min}'s projected completion time to model the amount of time that the query will remain in this mix. Finally we replace q_{min} with the next one on the queue. We continue this cycle until the query for which where are creating a prediction is q_{min}.

6. EXPERIMENTAL EVALUATION

In this section we set out to verify the accuracy and utility of the four modules in our framework. First we look at the accuracy of B2L, or how well our experiment-driven model can use buffer access latency to predict end-to-end query latency under varying concurrency levels and query mixes. Next, we examine how well our B2cB model can predict BAL for use with B2L. After that we explore the accuracy of combining these two techniques to produce end-to-end latency estimates. Finally, we experiment with creating prediction on random, dynamically generated workloads at a fixed multiprogramming level using our JIT and queue modeler timeline analysis techniques.

We gathered data in two phases for our steady state experiments. Initially, we create our training data consisting of several steady state mixes. Our training set draws from samples at multiprogramming levels 1-5. Using the training data set we build our prediction models for the concurrent buffer access latency (B2cB) and the end-to-end latency (B2L). The details of this training process are in Section 4. B2cB produces coefficients for the multivariate prediction model for each multiprogramming level. B2L produces its simple linear model for each query class.

Next we looked at a test data set, consisting of randomly selected mixes that are disjoint from the training data. Our test data set is drawn from two Latin hypercube sample sets per multiprogramming level, producing 20 steady state measurements. Each steady state measurement is comprised of at least 5 queries per template in the mix.

Finally, we experimented with completely arbitrary mixes, relaxing our steady state assumption. This allowed us to evaluate our timeline analysis. Here we looked at both incremental re-evaluation of our predictions (JIT) as well as creating only one prediction when each query started (queue modeler).

6.1 Setup

We experimented with a TPC-H query workload consisting of moderate weight queries that exert strain on different parts of the database system. We targeted OLAP because the queries are longer running, giving us a more time to characterize their interactions. Analytical queries tend to be more highly I/O-bound, allowing us to focus on this element for our interaction modeling rather than a more complex multidimensional model that would have to take into account CPU slicing and other resource sharing.

We worked with a mix of ten moderate weight queries. Specifically, our workload is comprised of TPC-H queries 3, 4, 5, 6, 7, 8, 10, 14, 18 and 19 on an instance of TPC-H, scale factor 10. We selected the ten queries on and around the median execution time for all of the TPC-H templates. We focus on medium-weight queries because they are long running enough such that we can make useful inferences about their behavior. They are also light enough to allow for us to experiment with meaningful multiprogramming levels.

Furthermore, we wanted to predict realistic workloads. We study concurrent query mixes of different multiprogramming levels, however we do not address predictions for cases of very high contention. Therefore, we do not consider cases where it is likely that a mix of queries could complete all of their executions in isolation faster than running its queries concurrently.

We also elected to focus on moderate-weight queries because modeling across multiple query weights adds another layer to the problem as demonstrated in [8]. In this work they classified their queries according to a relative weight and built models for each group. In future work, we may extend our framework with a similar labeling strategy.

In our experimental configuration we added the primary key indexes provided by the standard TPC-H implementation. The benchmark was deployed on a modified version of PostgreSQL 8.4.3. The source code was instrumented to measure BAL. All of our trials were run on a machine with an Intel i7 2.8 GHz processors and 8 GB of RAM. The machine ran on Ubuntu 9.04 on Linux 2.6.28-11.

6.2 Predicting Query Contention

In this section we will look at how well our B2cB and B2L system can predict BAL and end-to-end latency. All of the results below are evaluated on our test data, disjoint from the training samples.

6.2.1 B2L

First we examine the effectiveness of B2L for estimating latency based on a provided BAL. We built our linear models for each query class on our training set at multiprogramming levels 1-5. Our train-

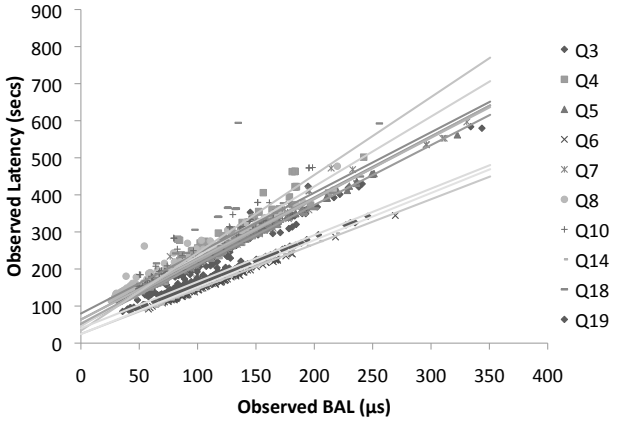

Figure 6: Fit of B2L to steady state data at multiprogramming levels 3-5. Error rates for this data are in Table 2

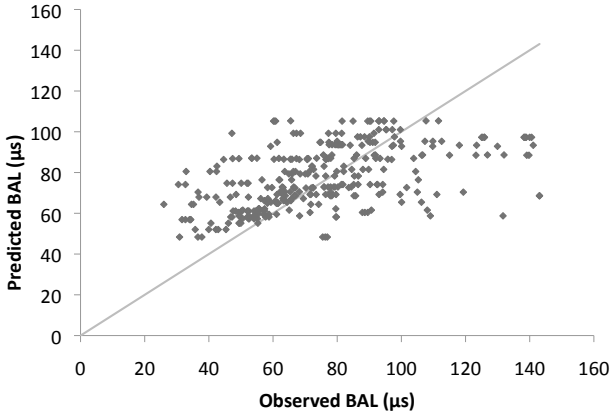

Figure 7: B2cB predictions on steady state data, multiprogramming level 3

ing phase produces simple equations of the form $L = O_\alpha + P_\alpha * B_\alpha$ for each query class in our workload.

We demonstrate the fit of our model with Figure 6. The points on this graph are all from individual queries of our test data set with concurrency levels 3-5. The lines denote the slope of an individual B2L model. Each query class fits very well with its regression model. Our evaluation data produced an average error of just 5%. This tight fit is because queries generally will have latency proportional to the rate at which they can access physical pages. This shows that despite relatively few mixes sampled and varying concurrency levels, buffer access latency is an efficient and robust estimator of query latency. To reiterate, none of the points evaluated here were a part of the training phase.

It is clear that our B2L models fall into two main trajectories. These different slopes are a consequence of our query templates having different query execution plan layouts. Query plans can be divided into processing pipelines. These pipelines delineate points in the query execution where results from the previous pipeline must be completed before the next phase of a query may start. For example, sorts require access to all of the tuples in their operation and thus require that executor start a new pipeline for them (separate from the operators that provided tuples to the sort). In future work, we may consider breaking our predictions down to the granularity of pipelines as was done for progress estimators in [7].

The queries with smaller latency growth (queries 6, 14 and 19) had a single pipeline (i.e., no materialization points such as sorts or group-by aggregates). The remaining query classes which demonstrated a faster latency increase in proportion to their BAL generally had greater than one pipeline in their query plans. This means that the overall growth rate for each BAL is higher, due to the overhead of the tuples having to traverse multiple pipelines and typically be evaluated by more operators in the query plan.

6.2.2 B2cB

Now that we have established that average BAL is a good predictor of latency in the case of varying resource contention, we turn to evaluating how well we can predict this metric. We examine the accuracy of the B2cB approach for predicting average BAL on the test data set.

The fit of B2cB predictions to measured BAL at multiprogramming level 3 in is displayed in Figure 7. This charts all of the queries in the test mixes in steady state. There were a total of 20 steady state mixes sampled for this experiment.

Our samples create a uniform fit around the trend line, denoting that the model can make reasonable "ballpark estimates" of BAL. There is some variance within queries of the same class and mix. Much of this can attributed to buffer pool contents and changing predicates for each example of a query template.

Another cause of variance is temporal shifts (i.e., queries in the same mix start at different times in relation to their complements). These shifts are displays as horizontal lines of points on this graph. We have experimented with breaking our queries into sections and evaluating our predictions on this finer-grained basis. Experimentally we did not find significant improvements by evaluating these queries on a section-by-section basis. We also found that sampling to quantify all pairs at this granularity caused our training time to grow significantly.

Also, our model works best at low–to moderate–levels of contention. As contention increases (denoted by a higher observed BAL), our estimates have reduced accuracy. This is because as we get closer to a state of overload, our growth pattern gradually shifts from linear to exponential. Modeling overload and the shift in our distribution of BAL is left to future work.

In summary, our B2cB estimates had an average error of 24-35%. As we increased our multiprogramming level, our estimates became slightly less accurate as we neared a state of overload.

6.3 Steady State Predictions

Next we examine our ability to predict end-to-end latency under steady state conditions by composing our BAL estimates with B2L. For each query in a steady state run we first predict the BAL using B2cB. Then we translate this predicted BAL to latency using the B2L models depicted in Figure 6. We hold the mix constant to allow us to model discrete mixes despite the underlying queries having different execution latencies. We will later quantify the additional challenges posed by continuously changing query mixes.

In Figure 8 we assess our model's accuracy on the test data. We divide our findings into query classes and also averaged overall. We tested this system for multiprogramming levels 3-5. The figure shows our relative error, calculated as: $\frac{|recordedQoS - prediction|}{recordedQoS}$.

For reference, we include the standard deviation for each query in each mix. This is normalized to the average latency of the sample that it came from. This is to quantify the variance-inducing conditions caused by temporal offsets and changes in our template predicates. It is averaged over multiprogramming levels 3-5.

At multiprogramming level 3, our error rate roughly equals to the average standard deviation. Our error averages to 15%, which is well-fitted considering the variety of cases handled (20 discrete

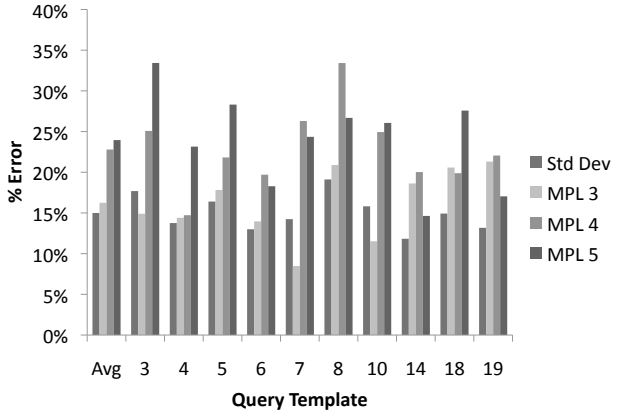

Figure 8: Steady state relative prediction errors for query latency at multiprogramming levels 3-5

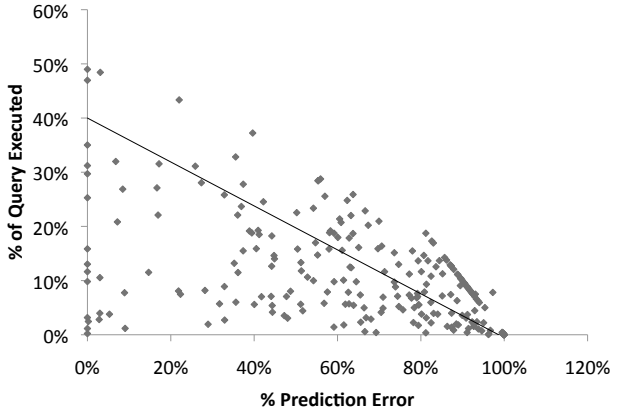

Figure 9: JIT prediction errors as we re-evaluate queries periodically at multiprogramming level 3.

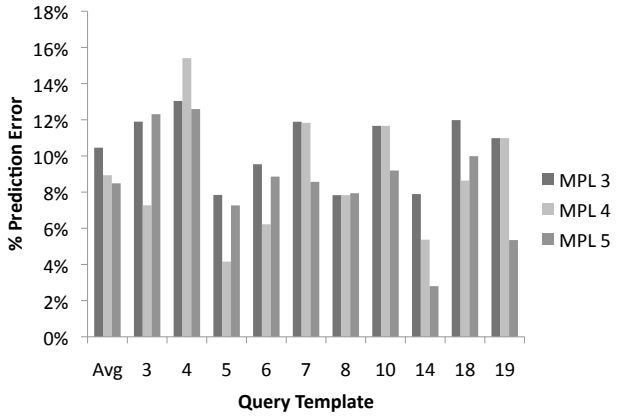

Figure 10: JIT relative latency estimation errors at various multiprogramming levels.

mixes times 3 queries per mix). Queries that have relatively simple, highly I/O dominated plans such as Q6 are modeled well. Queries that have more complex aggregation are more reliant on prefetching and consequently are sensitive to the temporal offsets of their complements. Examples of this phenomenon include Q3 and Q5. Later we will discuss the effects of relaxing this steady state assumption.

6.4 Timeline Evaluation

Next we evaluate how well our timeline approaches estimates individual query latency. Once again, we experimented with three multiprogramming levels. We ran 250 randomly generated queries (both the template and predicates were selected in this manner). These measurements excluded a warmup and cool down period for each run to allow our multiprogramming level to remain constant. All of these results are from the same set of query executions, processed for both JIT and queue modeling timeline analysis.

JIT predictions produce a latency estimate for all queries currently executing when a new query is started. We use this technique to simulate the scenario where users are submitting queries that are run immediately. This system will enable the user to estimate how long a new query will take and its projected impact on currently running queries' latencies. In this context, we work from the scenario where the user has no way of predicting the incoming workload. Thus we re-evaluate our latency predictions every time a new query is introduced into the mix.

This technique also allows us to do real-time "course corrections" for our estimates. We can see how our prediction errors converge as time passes for the JIT technique in Figure 9. Our predictions naturally follow a half-cone shaped distribution. The closer a query is to completion, the better our predictions are. We could improve our readings further by evaluating more frequently, perhaps at regular intervals rather than when we are scheduling a new query. This would be useful if we are (for example) updating a real-time progress indicator for users who are awaiting query results or running real-time load balancing.

With the JIT estimator our average error is around 10%. This is a combination of having good latency estimates paired with corrections from the re-evaluations. This is the reason that the higher multiprogramming levels generally perform better; they re-evaluate more frequently as they schedule queries at a faster rate.

This outcome demonstrates that we can (a) give users real-time guidance about how scheduling decisions will affect currently running queries as well as ones that they are considering scheduling; and (b) we can make predictions with arbitrary mixes. Thus we are not limited by our initial steady state configuration. The ability to depart from steady state estimates and still produce similar error rates is caused by the database being so large that most of it cannot fit in memory at any given time. Thus, our estimates adapt to the RAM changing its contents regularly, negating the necessity of discretely modeling buffer pool state.

Finally in Figure 11, we examine how well our predictor can handle arbitrary mixes paired with more uncertainty about the duration and content of the individual mixes. Using the queue modeler technique, we build a latency estimate from predicting both when the primary query and its complements will terminate. We model for a continuous multiprogramming level by adding new queries from a provided queue when complement queries are projected to terminate. The simulation ends when our primary query is projected to be the next query replaced by an element of the queue.

Overall, this brings our accuracy levels to an average of 21%. This is close to our estimation accuracy in steady state. This accuracy demonstrates that our framework can make predictions under realistic scenarios of queries starting and stopping at intervals offset from each other. This allows a user to make straightforward "what-if" analysis when they are considering scheduling a batch of queries. This framework could potentially enable a user to formulate different orderings of their queries and submit them for evaluation to our system. This would search for time-saving orderings of submitted queries.

These results also demonstrate that our system can tolerate some

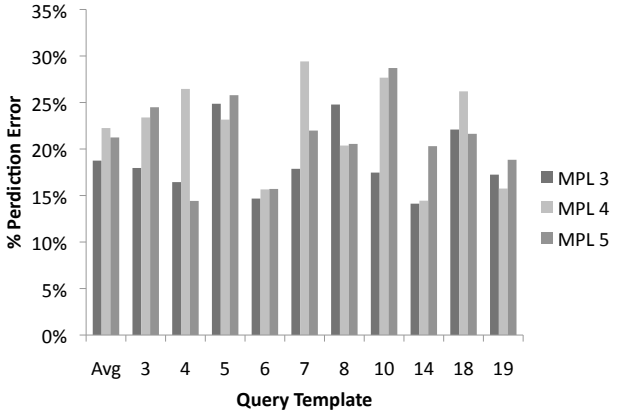

Figure 11: Queue modeling relative latency estimation errors at various multiprogramming levels.

uncertainty about the mixes being executed. Despite our steady state estimates having limited inherent errors, this does not unduly affect our mixed workload case. In many cases the timeline errors are not significant because as we are transitioning between mixes, we do this one query at a time. Thus the subsequent mixes are only slightly different from each other in terms of how they affect the primary query.

7. RELATED WORK

The topic of performance prediction for database workloads has recently gained significant interest in the research community. There have been many studies regarding how to characterize workloads, plot query progress and predict query performance. In this section we explore each of these topics in turn.

Workload Characterization Prior work characterizing database performance [11, 17] study utilization of individual resources (such as CPU time) for a database workload and how to profile queries. In our research we target a higher level metric: query latency.

In [5] the authors analyzed how to classify workloads. Work has been done to characterize the usage of specific resources in databases including [4] and [10]. Here the authors examined memory and CPU usage under various workloads respectively.

Query Progress Indicators There has also been significant work on query progress indicators [7, 13, 12, 6]. In [7], the authors reason about the percent of the query completed, however their solution does not address direct latency predictions for database queries and does not apply on concurrent workloads. In [12], the authors estimate the remaining execution time for multiple, concurrent running queries and their approach relies on previous work on single-query progress indicators. Therefore, they rely on semantic information of the SQL queries. Our solution is simpler as it uses statistics collected through experimental sampling and we predict the total response time of the queries with similar accuracy. [6] provides a survey of existing progress indicators and recommends the best ones for use in practice.

In [13] the authors propose a single query progress indicator that can be applied on a large subset of database query types. Their solution uses semantic segmentation of the query plans and strongly relies on the optimizer's (often inaccurate) estimates for cardinality and result sizes. Although their progress estimator takes into account the system load, their work does not specifically estimate progress of concurrent queries and includes limited experiments on resource contention. In the contrary, our work primary focuses on concurrent workloads and does not rely on the optimizer's estimates. Moreover, our timeline analysis identifies the overlap between concurrent queries to continuously improve the prediction error and thus can effectively address resource contention for concurrent queries. Finally, progress estimators have been researched for MapReduce queries [15]. This work assumes a different workload model from the one we have developed for analytical queries.

Query Performance Prediction In [8] the authors use machine learning techniques to predict multiple performance metrics of analytical queries. Although their predictions include our QoS metric (query latency), their system does not address concurrent workloads. Moreover, their solution relies on statistics based on the SQL text as well as ones obtained from the query execution plan.

In [9] the authors also use machine learning to predict query performance. They predict a range for their query execution latency. Like us, their work considers concurrency, but only models based on the multiprogramming level rather than individual query mixes. In contrast, we provide a direct latency estimate and consider system strain at a higher granularity.

Furthermore, [12] examines predicting query latency with concurrency. The authors make predictions including remaining latency on queries that are executing concurrently. Like this work, the authors consider the addition and removal of queries during the time for which they are creating a prediction. In contrast to this work, it does not consider multiple, diverse query classes.

Query interactions have also been addressed and greatly advanced in [1] and [2]. In this work the authors create these concurrency-aware models to build schedules for batches of OLAP queries. Their solutions create regression models based on sampling techniques similar to the ones that we use. These systems created schedules for a list of OLAP queries to minimize end-to-end latency for a large set of queries. Our experiment-driven approach predicts the response time of *individual* queries, and presumes that the order of execution of the queries is fixed for a real-time environment.

[3] models the mixes of concurrent queries over time from a provided workload, but does not target individual query latencies, instead optimizing for end-to-end workload latencies. This approach uses an incremental evaluation approach, which our timeline analysis is inspired by. Our timeline modeling uses a very different framework to project latency.

Finally, [14] explores how to schedule OLAP mixes by varying the multiprogramming level and priority assigned to individual queries. They use this priority-based control mechanism to prevent overload and under-load, allowing the database to maintain a good throughput rate. Like this work, our model experiments for many multiprogramming levels, but our focus in this work is in predicting execution latency rather than prioritizing queries effectively.

8. CONCLUSIONS AND FUTURE WORK

This work proposes a lightweight estimator for concurrent query execution performance for analytical workloads. To the best of our knowledge it is the first to predict execution latency for individual queries for real-time reporting without using semantic information.

Our system starts with studying the relationship between BAL and quality of service as measured by query execution latency. We have demonstrated that there is a strong linear relationship between these two metrics, which we model with our system B2L. This relationship is based on the observation that as long as there is contention for a resource and we can instrument its bottleneck, then we can accurately predict latency. We produce very accurate estimates of latency given the average BAL despite this metric exhibiting moderate variance. We accomplish this because our queries are sufficiently long that we collect enough samples to produce a rep-

resentative average. This naturally is proportional to the latency because the queries are primarily I/O-bound. We predict the BAL by extrapolating higher degree interactions using pairwise BALs in a system we call B2cB.

We then adapt this baseline system to a dynamically changing workload using timeline analysis. We predict the steady state latency of each query in a workload and determine which will terminate first. We then estimate the progress of each query after the first terminates and conjecture about which will end next. We continue this cycle in two formulations: just-in-time and queue-modeler. In the former, we build our prediction based on the currently executing batch. The latter fixes our multiprogramming level and predicts when new queries will be started as old ones end.

There are several open directions for future work in the area of concurrent query performance prediction. In the short term, we plan to experiment with more advance machine learning techniques for predicting the query latency. One approach we are considering is extending the Kernel Canonical Correlation Analysis that was used in [8] to predicting single query latency in the context of concurrent execution. We can apply this technique to mapping mixes to latencies by mapping features to performance. We could also benefit from using a richer feature set including query execution plan data to refine our predictions.

We also plan to capture our BAL distribution under varying conditions using a Gaussian process model [16]. This would enable us to smooth the surface of our predictions by developing a piecewise model for predicting BAL. It may enable us to serve a wider workload range.

Another direction we plan to research in the future is incrementally tuning our estimates to workloads as they change over time. We would have to incrementally update our B2L model to handle changing table cardinalities from inserts and deletes. This would also require us to identify when the query execution plan changes in response to growing or shrinking data and how this impacts our indicator metrics.

Finally, this paper performs modeling at the granularity of query templates. We would like to generalize our solution for cases in which such templates are not available. To this end, we are interested in automatically identifying clusters of queries that exhibit similar concurrent interaction behavior and use these as the basis for predictive modeling.

9. ACKNOWLEDGEMENTS

This work was funded by in part by NSF under Grants IIS-0905553 and IIS-0916691.

10. REFERENCES

[1] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Modeling and exploiting query interactions in database systems. In *Proceeding of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 183–192, New York, NY, USA, 2008. ACM.

[2] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Qshuffler: Getting the query mix right. *Data Engineering, International Conference on*, 0:1415–1417, 2008.

[3] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Interaction-aware prediction of business intelligence workload completion times. *Data Engineering, International Conference on*, 0:413–416, 2010.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 3–14, Washington, DC, USA, 1998. IEEE Computer Society.

[5] M. Calzarossa and G. Serazzi. Workload characterization: A survey. In *Proceedings of the IEEE*, pages 1136–1150, 1993.

[6] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for SQL queries? In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 575–586, New York, NY, USA, 2005. ACM.

[7] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

[8] 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: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 592–603, Washington, DC, USA, 2009. IEEE Computer Society.

[9] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. *Autonomic Computing, International Conference on*, 0:13–22, 2008.

[10] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. volume 26, pages 15–26, New York, NY, USA, April 1998. ACM.

[11] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. *SIGARCH. News*, 26(3):39–50, 1998.

[12] G. Luo, J. Naughton, and P. Yu. Multi-query sql progress indicators. In Y. Ioannidis, M. Scholl, J. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, and C. Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 921–941. Springer Berlin / Heidelberg, 2006.

[13] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 791–802, New York, NY, USA, 2004. ACM.

[14] A. Mehta, C. Gupta, and U. Dayal. BI batch manager: a system for managing batch workloads on enterprise data-warehouses. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, pages 640–651, New York, NY, USA, 2008. ACM.

[15] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[16] C. Rasmussen. Gaussian processes in machine learning. In *Advanced Lectures on Machine Learning*, volume 3176, pages 63–71. Springer Berlin / Heidelberg, 2004.

[17] P. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Transactions on Software Engineering*, 18:347–355, 1992.