# A Bayesian Approach to Online Performance Modeling for Database Appliances using Gaussian Models

Muhammad Bilal Sheikh  Umar Farooq Minhas  Omar Zia Khan
Ashraf Aboulnaga  Pascal Poupart  David J Taylor
Cheriton School of Computer Science, University of Waterloo, Canada
{*mbsheikh, ufminhas, ozkhan, ashraf, ppoupart, dtaylor*} *@uwaterloo.ca*

Technical Report CS-2011-13

**Abstract**

In order to meet service level agreements (SLAs) and to maintain peak performance for database management systems (DBMS), database administrators (DBAs) need to implement policies for effective workload scheduling, admission control, and resource provisioning. Accurately predicting response times of DBMS queries is necessary for a DBA to effectively achieve these goals. This task is particularly challenging due to the fact that a database workload typically consists of many concurrently running queries and an accurate model needs to capture their interactions. Additional challenges are introduced when DBMSes are run in dynamic cloud computing environments, where workload, data, and physical resources can change frequently, on-the-fly. Building an efficient and highly accurate *online* DBMS performance model that is robust in the face of changing workloads, data evolution, and physical resource allocations is still an unsolved problem. In this work, our goal is to build such an online performance model for database appliances using an *experiment-driven* modeling approach. We use a Bayesian approach and build novel Gaussian models that take into account the interaction among concurrently executing queries and predict response times of individual DBMS queries. A key feature of our modeling approach is that the models can be updated online in response to new queries or data, or changing resource allocations. We experimentally demonstrate that our models are accurate and effective – our best models have an average prediction error of 16.3% in the worst case.

## 1 Introduction

Database appliances are becoming a popular way of deploying database management systems (DBMSes) in today's cloud computing environments. A database appliance is a virtual machine (VM) with a pre-installed and preconfigured copy of an operating system and a DBMS, ready to go out of the box. Using a database appliance reduces the total cost of ownership by saving multiple hours that are typically spent on installing, configuring, and tunning a DBMS from scratch. Users can run a database appliance in an Infrastructure as a Service (IaaS) cloud, such as Amazon's Elastic Computing Cloud (EC2) [1], renting computing power on-demand. A related paradigm for deploying and using database systems is Database as a Service (DaaS), exemplified by Amazon's Relational Database Service [2] and Microsoft's AzureSQL [3]. In addition to saving setup and deployment costs, these approaches offer low-cost alternatives to in-house infrastructure procurement and management, and enable flexible resource provisioning and adapting the available resources to the dynamically changing workload. For all these reasons, database appliances are widely deployed on the cloud and their use will continue to grow.

One of the problems with database appliances is that the highly dynamic nature of the workloads and the environment makes it difficult for a database administrator (DBA) to *predict the performance of a running query or workload*. In this work, our goal is to build an *online* performance model for database appliances that can predict response times of individual DBMS queries, given a *query type* and a set of queries already executing in the DBMS, which we call a *query mix* (details in Section 3). A key feature of our models is that they can adapt to changes in the queries, data, or DBMS configuration in an online manner.

Predicting the response time of a database query before execution is very useful for many administrative tasks. For example, such a prediction can enable a database administrator (DBA) or an automatic tool to schedule workloads effectively, perform better admission control, do capacity planning, and formulate policies for effective resource provisioning. For example, given two database appliances and their respective workloads, a DBA can use a performance model to predict the effect of giving more resources (e.g., memory or CPU) to one or the other DB appliance and then decide how to efficiently partition the available resources among these appliances so that they both meet their SLAs.

Traditionally, the performance of DBMSes has been studied by constructing elaborate analytical models. However, such models need to be carefully constructed by a domain expert and are usually specific to a particular DBMS. Moreover, these models do not capture the full complexities of query execution and the interactions among concurrently executing queries in the system. Furthermore, analytical models become obsolete as soon as there are any changes to the DBMSs implementation. *Experiment-driven* modeling techniques [5, 6, 9, 18, 19] overcome the above mentioned shortcoming of analytical models and have therefore become very popular. We use an experiment-driven modeling approach in this paper.

Experiment-driven modeling relies on: (1) sampling the space of possible query executions to collect training data, and then (2) fitting statistical or machine learning models to the collected sample data. Most existing techniques [4, 5, 6, 9, 13, 18, 19, 21, 22] for database experiment-driven performance modeling rely on static models that are trained offline for specific configurations and resource allocation levels. These models cannot be updated online due to the inherent inflexibility of the learning techniques used. Thus, any changes in the workload, the database configuration, or resource allocation to the VM containing the DBMS (i.e., the database appliance) require collecting new samples and re-training of models. Collecting new samples is very costly, taking hours or days, which severely limits the applicability of prior experiment-driven modeling techniques. This limitation is especially restrictive for database appliances, since in addition to the dynamic nature of the database and queries, resource allocations can also change in an online manner.

In this work, our goal is to address this limitation and build efficient and highly accurate *online* query response time models for database appliances that take into account the interactions among concurrently running queries and can dynamically and robustly adapt in the face of changes in the workload, database or physical resource allocation, *without the need for additional sampling experiments*. We identify the use of Gaussian Process (GP) models and show how to effectively apply them to build online response time models. Gaussian Process models have been previously applied to various problems, including database performance modeling [6, 18]. We choose them in this work because they lend themselves well to online adaptation. However, we show that a simplistic approach to adapting GP models is too costly. A major contribution of this paper is to develop a novel Bayesian approach for efficient online adaptation of GP models. Our experimental results demonstrate that GP models outperform other techniques in terms of goodness of fit, accuracy, and model training/prediction time. Furthermore, the expressiveness offered by the Bayesian framework allows us to effectively leverage prior knowledge derived from sample data to learn response time models for previously unseen queries and configurations for which there is no offline sampled response time data. The high accuracy and fast convergence of online GP models make them suitable for online performance modeling of databases appliances.

## 2 Related Work

Database systems have traditionally relied on analytical performance models, with model parameters based on simple statistics. Analytical models are most prominently used in query optimizers, and there has been some recent work to adapt optimizer models online [14]. Analytical models were also used to set the multi-programming limit (MPL) of a DBMS for improved throughput [4, 22]. There has also been work on self-predicting databases that are capable of answering "what-if" questions [14, 17]. In addition to these internal database models, queuing models for multi-tier architectures also attempt to include the performance of the DBMS [21, 23]. A significant limitation of these analytical modeling techniques is that they are notoriously hard to evolve with the system and they necessarily make simplifying assumptions, so they do not capture the complex execution of dynamically changing workloads. As a result, there is increasing focus in the research community on experiment-driven performance modeling for database and multi-tier systems.

The recent literature includes several examples of experiment-driven models for database systems. Predicting

Query Run-Time (PQR) trees [10] make use of binary classification trees to represent disjoint sets of time ranges. A new query traverses through the tree reaching a leaf node which represents the predicted time range for that query. Ganapathi et al. [9] use Kernel Canonical Correlation Analysis (KCCA) to predict multiple metrics for database queries, including response times. The KCCA technique takes two covariance matrices (query feature and performance) and projects them onto two subspaces such that the projections of the two matrices are maximally correlated. The authors report a prediction time of "under a second", while our models take a total of 81 ms on average for prediction. Tozer et al. [19] use a linear regression response time model for throttling long running queries. Linear regression models are typically less accurate than the GP models that we use (which can be seen in the $R^2$ correlation coefficient). Watson et al. [21] predict quantile ranges for multi-tier applications under virtual resource allocation. They model performance as a joint distribution over performance metric and resource allocation. All of these approaches suffer from a fundamental limitation: they require re-learning the model for any change in the workload or configuration, rendering them ineffective for online performance modeling of database appliances. Another limitation of these approaches is their inability to leverage prior knowledge in a meaningful way. Furthermore, most of these approaches provide point value predictions with no confidence intervals.

In this paper, we use Gaussian Process model that rely only on query types and no additional features, and hence are DBMS agnostic. The proposed GP models are very expressive and are not constrained by a fixed function form. They can be updated efficiently in an online manner, and new highly accurate models for queries running on different configurations can be learned by using prior knowledge. As an added advantage, these models can predict not only a point (mean) response time but the complete response time distribution, which can be used to provide confidence intervals for the prediction. Gaussian Processes have been used in [6, 18] but these works, like other prior work on experiment-driven modeling, do not consider dynamically updating these models.

## 3 Solution Overview

Our goal is to build a query-interaction-aware response time model for database appliances that is able to adapt itself to changing workloads and resource allocations in an online fashion. We assume that each query submitted to the DBMS belongs to a specific query type $Q_i$, where $1 \leq i \leq T$ and $T$ is the total number of query types. A workload $W$ comprises zero or more instances of each query type. The mix of concurrently running queries, $m_j$, is represented as a vector $< N_{1j}, ..., N_{Tj} >$, where $N_{ij}$ represents the number of instances of query type $Q_i$ in the mix $m_j$. Query interactions have been shown to significantly affect the response time of an incoming query [5, 6, 19]. Therefore, our response time models should take into account the particular mix of queries, $m_j$, or at least the total number of queries in this mix $l$ given by $\sum_{i=1}^{T} N_{ij} = l$. It is always true that $l \leq M$, where $M$ is the *multi-programming limit (MPL)* of the DBMS and is specified by the DBA. Given the query mix currently running on the system, we want to be able to predict the response time of individual incoming queries. That is, we want to find a function $f(.)$ such that

$$\hat{r}_{ij} = f(m_j, Q_i)$$

where $\hat{r}_{ij}$ is the estimated response time of a query type $Q_i$ running in a mix $m_j$. The function $f(.)$ can be based on the distribution of queries in the mix, or just on the total number of queries in the mix ($l$). We use an *experiment-driven black box* modeling approach to collect training data consisting of samples $S_i =< m_j, r_{ij} >$, where $r_{ij}$ is the actual response time of a query type $Q_i$ running in a mix $m_j$. We then use this data to learn models for predicting query response times (i.e., to learn $f(.)$).

The response time of a particular query depends not only on the current load and the query mix but also on different tunable configuration parameters (e.g., buffer pool size, MPL, available CPU, and memory). One very important goal of our modeling approach is to adapt our models as these parameters change, without offline sample generation and model re-learning. To achieve this, we build two separate models: **1) response time model:** a model to predict query response times for each query type of interest and **2) configuration model:** a model to predict the response time model's parameters for different configurations. We present response time and configuration models in Section 5 and 7, respectively.

Figure 1 shows the overall workflow of our model learning process. As a first step, a DBA will generate different training query mixes using our workload generator module which takes $M$ (MPL) and $T$ (total query types) as input

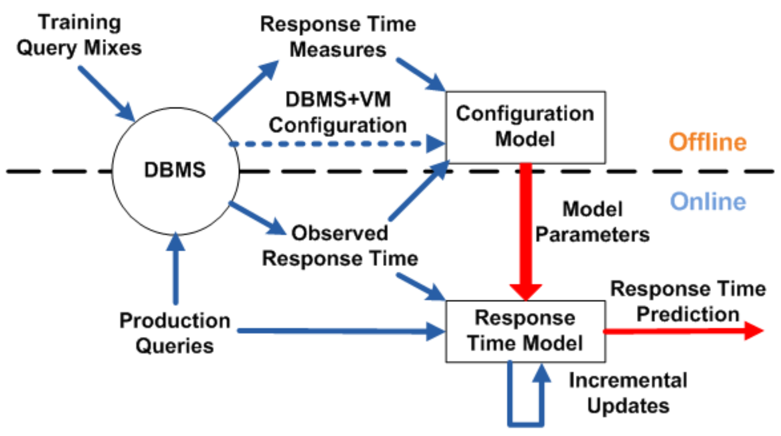

Figure 1: Model learning workflow

(Section 4). Using the generated query mix samples, the DBA will then run a series of experiments for different configurations (e.g., buffer pool size, total physical memory, CPU, etc.) using a client coordinator. Our client coordinator is a client-side program implemented in C++. The client coordinator creates client threads, each with a separate connection to the DBMS. Each thread then selects and runs a query from a given query mix. Once the desired query mix is running, a separate thread executes a target query and measures its response time. This results in training samples of the form $S_i =< m_j, r_{ij} >$. These samples are then used to learn an offline configuration model (Section 7) which accumulates all configuration parameters and the corresponding response time model parameters for the selected configurations. The configuration model is then used *online* by the client coordinator to initialize query response time models (Section 5) for new unseen configurations and query types for which there are no offline trained models. The configuration model takes the observed response time of the most recently executed instance of a query type, and the DBMS+VM configuration to periodically generate parameters for that query type's response time model. During the online phase, the configuration model can generate new model parameters for the response time model of a given query type when the query is run in a new configuration. To detect the need for generating the new model parameters, the configuration model monitors the DBMS+VM configuration and the observed execution time of queries of this type. The response time model, which is specific to each configuration and query type, updates itself by incorporating the most recent response time data (details in Section 6).

Note that in our current implementation the client coordinator is external to the DBMS, which is suitable for applications such as capacity planning, workload scheduling, or admission control, among others. Alternatively, the client coordinator can be integrated with the DBMS, to provide, for example, more accurate statistics for query optimization or dynamically controlling the *MPL* for maximum performance.

## 4 Generating Training Workloads

Like other experiment-driven approaches, we require sampling experiments to collect data for training our models. However, since our models adapt dynamically they are not as sensitive to the sampling policy used as models in prior work [6, 19]. A simple sampling approach that guarantees relatively good coverage of the space of possible query mixes will provide a good starting point for an offline trained model. The model will then be able to correct itself quickly even if the initial training samples are not the best representative of the running workload.

### 4.1 Uniform Sampling

A direct consequence of constructing the sample query mixes used for training by sampling each query type uniformly in the range $[0 - M]$ is that the distribution of load, where the load is the total number of queries in the mix, $l$, approximates a normal distribution around the mean value of $l$ (by the central limit theorem). The variance decreases

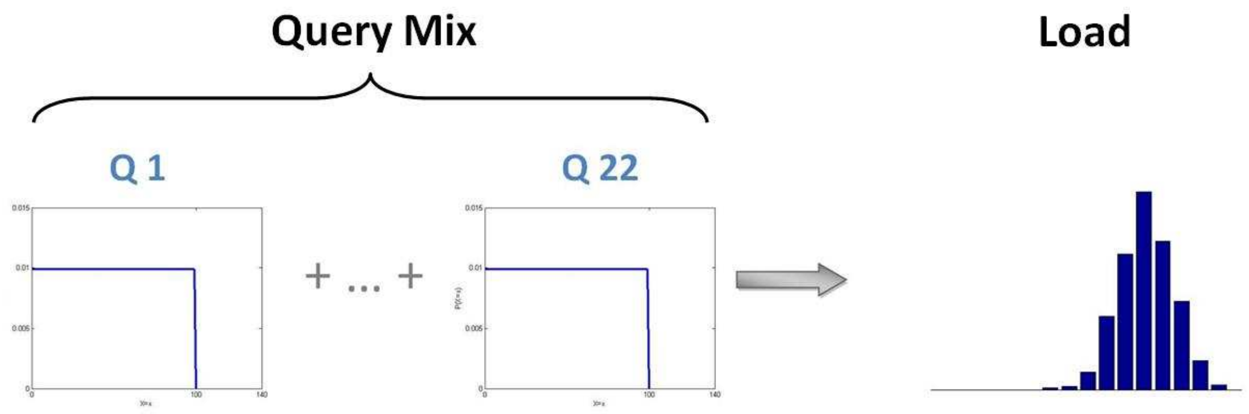

Figure 2: Consequence of Unifrom sampling.

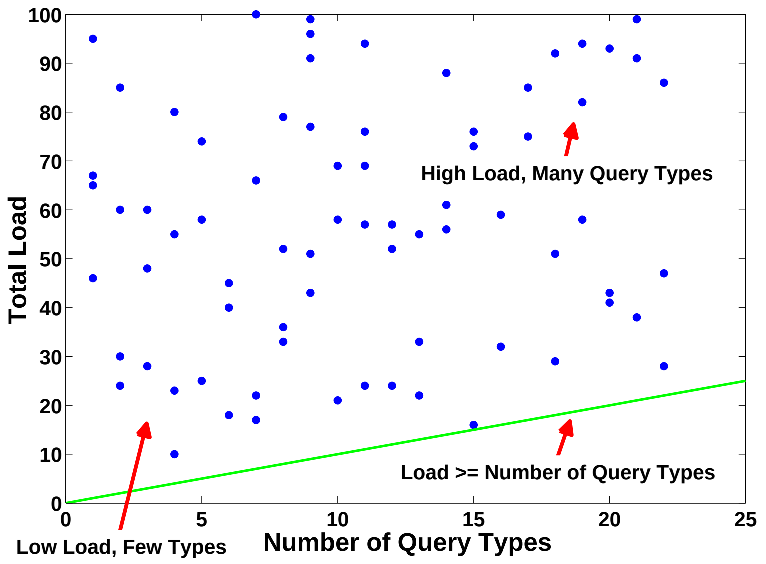

Figure 3: Workload Characterization based sampling. The number of queries (x-axis) contributing to the total load (y-axis)

as the number of identical uniform distributions, one for each query type, increases (shown in Figure 2). A detailed discussion of the exact distribution of the sum of identically and non-identically distributed uniform distributions can be found in [8]. Based on the upper limit of each query type allowed in the mix, the distribution over load is very narrow. This is not desirable for learning response time models for widely varying loads.

### 4.2 Workload Characterization Based Sampling

To solve the problem of uniform sampling, we look at the query mix from two perspectives: the overall load and the query types contributing to that load as shown in Figure 3. To cover the spectrum we generate uniformly distributed samples across two dimensions: 1) total number of queries, and 2) the number of different types of queries running (termed the *interaction level* in [6]). Sampling is done by first randomly selecting the level of load from 0 to $M$, and then randomly selecting the number of queries that contribute to this level of load. The value of this variable ranges from *1* to *min* $\{$*total query types, random load level*$\}$. These two values are used to control the random generation of the number of queries of each type. This sampling process generates a distribution over loads that covers the workload spectrum from low load to high load.

Our workload generator uses characterization based sampling to generate query mixes. The client coordinator runs the generated workload and collects the training samples. In our experiments, it takes under 20 hours to run these experiments for a workload with about 600 different query mixes. Next, we present details of the Gaussian models for predicting query response times and their online adaptation. Section 7 describes the configuration model and how it is used.

## 5 Gaussian Response Time Models

*Bayesian Networks (BN)* fall under the broader class of *probabilistic graphical models* where dependencies between variables are encoded using conditional probability distributions. More specifically, a BN is a directed acyclical graph (DAG) in which each node $X_i$ has a conditional probabilistic distribution $P(X_i|Parents(X_i))$ quantifying the effect of parents on the node. The Bayesian framework offers a number of advantages over alternative modeling approaches including flexibility and consistent semantic interpretation. We choose to develop BN based models for two reasons: 1) The Bayesian framework provides a theoretical basis for modeling uncertainty using probabilities. This allows us to predict not only the mean response time but also the complete distribution for the response time of a particular query. This uncertainty in the response time model can be directly propagated to a decision making system (e.g., autonomic provisioning tool, throttling system, etc.) 2) The Bayesian framework offers the ability to encode expert or prior knowledge using prior probability distributions. This inherent flexibility allows us to specify effective priors for dynamic models using offline learned models.

Our proposed modeling approach uses non-linear Gaussian Process models. However, we start by presenting simpler, linear Gaussian models that are easier to learn and adapt to online changes. The BNs for linear Gaussian models are shown in Figure 4.

### 5.1 Linear Gaussian Models

Our proposed linear models encode each conditional probability distribution using a *linear Gaussian distribution* as $P(Y|x) = N(\beta_0 + \sum_i \beta_i x_i, \sigma^2)$. Such networks are referred to as *Gaussian Bayesian networks* [12]. Linear models are particularly attractive because they are easier to build and given their low model generation and update cost, they can be easily used in an online setting.

For a Bayesian network model, the likelihood of model parameters $\theta$ given data $D$ is the probability of observing data $D$ given parameters $\theta$ and can be written as $LK(\theta : D)$. We use Maximum Likelihood Estimation (MLE) [7] for finding the model parameters most likely to have produced the observations in the training data. We adopt the common practice of taking the log of the likelihood to turn products into sums for easy derivation of gradients. Using this approach we define two linear Gaussian models, which we describe next.

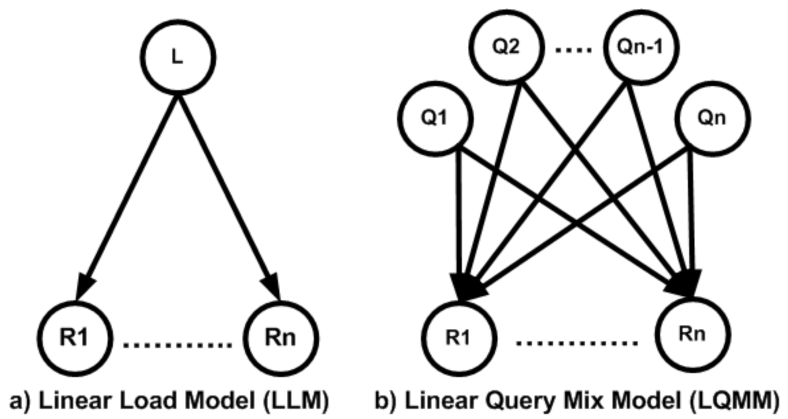

Figure 4: Bayesian Networks for linear Gaussian models

### 5.1.1 Linear Load Model

The first linear model that we develop is what we call the *linear load model (LLM)*. As shown in Figure 4(a), in this model the response time of a query is modeled only as a function of load ($l$) i.e., total number of concurrently executing queries. We model the probability of the random variable $R_i$ representing the response time of query type $Q_i$, conditional upon the random variable $L$ representing load, as a Gaussian distribution given by: $P(R_i|L) = N(\beta_0 + \beta_1 l; \sigma^2)$. The parameters $\theta_{R_i|L}$ $(\beta_0, \beta_1, \sigma^2)$ can be estimated by maximizing the log-likelihood $lk_{R_i}$, as shown in Equation 1, where $l_k$ and $r_{il_k}$ are the load and response time for the *kth* sample in the training data $D$ *($S_k =< l_k, r_{il_k} >$)*

$$lk_{R_i}\left(\theta_{R_i|L} : D\right) \quad = \quad -\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2}\sum_k \left[\frac{1}{\sigma^2}\left(\beta_0 + \beta_1 l_k - r_{il_k}\right)^2\right] \tag{1}$$

By taking the derivatives of $lk_{R_i}(\theta_{R_i|L} : D)$ with respect to $\beta_0$ and $\beta_1$ and setting to 0 we get Equations 2 and 3.

$$E_D[R_i] = \beta_0 + \beta_1 E_D[L] \tag{2}$$

$$E_D[R_i L] = \beta_0[L] + \beta_1 E_D[L^2] \tag{3}$$

where $E_D$'s are the respective means and moments $(R_i, L, R_i L, L^2)$ observed in the sample data $D$ for query type $Q_i$. Solving these linear equations gives us $\beta_0$ and $\beta_1$. $\sigma^2$ is given by Equation 4.

$$\sigma^2 = Cov_D[R_i; R_i] - \beta_1 Cov_D[L; L] \tag{4}$$

Once these parameters have been learned, we have a Gaussian distribution $N(\beta_0 + \beta_1 l; \sigma^2)$ over the predicted response times for each value of load $l$ on the database for a query type $Q_i$. The mean of the Gaussian distribution $(\beta_0 + \beta_1 l)$ is treated as the model's response time prediction $\hat{r}_{il}$ for a particular load $l$.

### 5.1.2 Linear Query Mix Model

*Linear query mix model (LQMM)* is similar to LLM but considers the query mix instead of the aggregated load. A query mix network is depicted in Figure 4(b) where each query of type $Q_i$ directly impacts the response time of all queries. Similar to the load model, response time is a Gaussian distribution centered on the linear weighted sum of the number of queries of each type running on the database. The probability distribution and the likelihood function are shown in Equation 5 and 6.

$$P(R_i|M_j) = N(\beta_0 + \beta_1 N_{1j} + \beta_2 N_{2j} ... + \beta_T N_{Tj}, \sigma^2) \tag{5}$$

$$lk_{R_i}(\theta_{R_i|M} : D) = -\frac{1}{2}log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_k(\beta_0 + \beta_1 N_{1k} + ... + \beta_T N_{Tk} - r_{ik})^2 \quad (6)$$

The likelihood function can be differentiated with respect to $\beta_0$ to $\beta_T$ to yield $T+1$ linear equations which can be solved for the values of $\beta_0$ to $\beta_T$. The variance $\sigma^2$ can be computed as shown in Equation 7.

$$\sigma^2 = Cov_D[R_i; R_i] - \sum_j^T \sum_k^T \beta_j \beta_k Cov_D[N_j; N_k] \quad (7)$$

## 5.2 Non-Linear Gaussian Process Models

Query performance varies in a complex non-linear way with varying the query mix, the hardware, and the DBMS configuration. For example, if a query involves a join, the behavior of this join varies significantly and in a non-linear way depending on whether the data fits in memory or needs to be read from disk. Therefore, the linear models developed in the previous section may not be sufficiently accurate and we need non-linear models. It is difficult to use any non-linear parametric model (e.g., a cubic model) since response times may not follow a parametric distribution. An alternative is to use non-parametric models that can learn arbitrary functions over load and query mix for response times. In this paper we use such a model, and in particular we use Gaussian Processes (GP) [15].

A Gaussian Process is a collection of infinite random variables, any finite number of which have a joint Gaussian distribution. GP based models are flexible, probabilistic and operate in a Bayesian framework which makes them suitable for modeling uncertainty and exploiting prior knowledge. In this section we develop GP based response time models that are trained offline. We describe in Section 6 how we adapt these models in an online setting. We train three different models and their variants:

1. ***Gaussian Process Load Model (GPLM):*** We train a GP load model where the response time ($r_{il}$) of a query type $Q_i$ is a function of load ($l$) similar to LLM.
2. ***Gaussian Process Mix Model (GPMM):*** GPMM is a non-parametric variant of LQMM and models the response time $r_{ij}$ of a query of type $Q_i$ as a function of query mix $m_j$.
3. ***Gaussian Process Mix + Load Model (GPMLM):*** GPMLM models the response time $r_{ijl}$ as a function of both query mix ($m_j$) and load ($l$). We show in Section 8 that GPMLM is not only more accurate then other GP based models but also more robust.

### 5.2.1 Bayesian Inference with Gaussian Processes

In Bayesian inference the probability of a hypothesis (*posterior probability*) depends on the likelihood of the hypothesis (based on observed data) and the prior belief (*prior probability*). For Gaussian Process based models we specify a GP prior as follows:

$$f(x) \sim GP(m(x), k(x, x')) \quad (8)$$

where $m(x)$ is a mean function and $k(x, x')$ is a covariance function. This function is also known as a *kernel function*. For a Gaussian process the joint distribution of the observed values $y$ and the predicted value $f(x_*)$ at $x_*$ is shown in Equation 9.

$$\begin{bmatrix} y \\ f(x_*) \end{bmatrix} \sim N\left( \begin{bmatrix} \mu \\ x_* \end{bmatrix}, \begin{bmatrix} K + \sigma^2 I & {k_*}^T \\ K_* & k(x_*, x_*) \end{bmatrix} \right) \quad (9)$$

This results in a posterior Gaussian Process that is used for prediction as shown in Equations 10, 11 and 12.

$$f|D \sim GP(m_D, k_D) \quad (10)$$

$$m_D(x) = m(x) + k^T(K + \sigma^2 I)^{-1}(y - m) \quad (11)$$

$$k_D(x, x') = k(x, x') - k^T(K + \sigma^2 I)^{-1}k' \quad (12)$$

where $\mu = m(x_i)$ for all samples in the training data. $k^T$ is the transpose of the vector of co-variances of $x$ with each training sample. $K$ is the covariance matrix for training data, $\sigma^2$ is the noise in samples, and $y$ is a vector of response values in the training data.

As shown in Equations 11 and 12, $m_D(x)$ is the mean predicted value, which is the sum of *prior mean m(x)* and a smooth function. Similarly, the variance for a point prediction $k_D(x, x')$ is the difference between the variance of the data and how well GP is able to explain the data at the target value. Therefore, the Gaussian Process not only captures the variance in the data but also how confident the model is when predicting at a particular point. The variance tends to increase as we move further away from data in the input space.

### 5.2.2 Specifying Prior Distributions

Bayesian inference requires specifying a prior, $m(x)$ and $k(x, x')$ in our Gaussian Process models (Equation 8). We create variants of GP models by using the following mean and kernel functions, and we evaluate their performance in Section 8.

**Mean Functions**

The approximate global shape of the function (i.e. linear, polynomial ..) is specified using a mean function $m(x)$. We use the following mean functions:

1. **0 Mean Function:** The mean and variance of the predictive distribution with **0 mean** for $f_*$ at $x_*$ is given in Equations 13 and 14.

$$E(f_*) = {k_*}^T {(K + \sigma^2 I)}^{-1} y \quad (13)$$

$$Var(f_*) = k(x_*, x_*) - {k_*}^T {(K + \sigma^2 I)}^{-1} k_* \quad (14)$$

2. **Linear Mean Function:** Based on LQMM we also experimented with the following mean function:

$$m(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_T x_T \quad (15)$$

Following the discussion on incorporating explicit basis functions in [16] we get the predictive distribution represented in Equations 16 and 17.

$$E(g_*) = E(f_*) + R^T \beta' \quad (16)$$

$$E(g_*) = E(f_*) + R^T {(H {K_y}^{-1} H^T)}^{-1} R \quad (17)$$

where $H$ is the matrix collection of all basis functions $h(x)$ for all training and test cases, $R$ is $H_* - H{K_y}^{-1}K_*$ and $\beta^*$ is given by ${(H{K_y}^{-1}H^T)}^{-1} H {K_y}^{-1} y$.

**Kernel Functions**

Kernel functions (covariance functions in Equation 8) are a measure of proximity of two input samples (load or query mix). We experimented with two kernel functions (Equations 18 and 19): 1) *Squared exponential function (SE)* with parameters $\theta = \{\eta, \sigma_f^2\}$ and 2) *Rational quadratic function (RQ)* with parameters $\theta = \{\eta, \sigma_f^2, \alpha\}$.

$$k(x_p, x_q) = \sigma_f^2 exp\left(-(x_p - x_q)^T \frac{1}{2\eta^2 I}(x_p - x_q)\right) \quad (18)$$

$$k(x_p, x_q) = \sigma_f^2 \left[ 1 + (x_p - x_q)^T \frac{1}{2\alpha\eta^2 I} (x_p - x_q) \right]^{-\alpha} \quad (19)$$

where $\eta$ is the characteristic length scale, a parameter indicating how much each dimension of the $x$ vector must change before the covariance function significantly changes. $\sigma_f^2$ is the signal strength and $\alpha$ controls the shape of the kernel function.

**Notation:** A variant of Gaussian Process Mix Model with linear mean and RQ covariance function is referred to as GPMM (LIN, RQ). The same conventions are used for referring to other variants of GP based models in the subsequent sections.

### Hyper-Parameter Learning

After selecting the mean and kernel functions, the next step is to specify the parameters associated with the mean ($< \beta_0, \beta_1, ., \beta_N >$) and kernel ($\eta, \sigma_f^2, \alpha$) functions. These parameters are referred to as the *hyper-parameters* of the model. The parameters are learned in light of training data by optimizing the log marginal likelihood given in Equation 20 using a conjugate gradients based technique.

$$\log P(y|x, \theta) = -\frac{1}{2} log|K| - \frac{1}{2} y^T K^{-1} y - \frac{n}{2} log 2\pi \quad (20)$$

A numerical approximation technique such as one based on conjugate gradients generally works well. In the optimization of the log marginal likelihood the term $-\frac{1}{2}log|K|$ is the complexity penalty which penalizes complex functions and $-\frac{1}{2}y^T K^{-1} y$ is the data fit measure [16]. The model therefore automatically selects a simpler model (Occam's razor) [16] which explains the data well without over-fitting, allowing the model to generalize well in practice as opposed to regression based models which over-fit to data.

## 6 Online Model Adaptation

In this section, we describe how Gaussian models can be refined using actual observations of query response times and how they can adapt to changes in configuration and workloads. Section 7 shows how prior knowledge can be leveraged effectively by using the configuration model to initialize the hyper-parameters of the response time models for unseen query types and resource allocations.

For the linear Gaussian models where parameters are learned using MLE, there are no hyper-parameters that need to be learned. Also, these models can be easily updated online by maintaining running or moving averages for computing the $E_D$ values, which can be updated with new data. However, as we show in Section 8, linear models have unsatisfactory prediction accuracy on some configurations. Therefore, in the following sections we focus on the non-trivial case of adapting the more accurate non-linear GP based models to an online setting.

### 6.1 Adding/Removing A Sample (Rank-1 Updates)

If we have an offline trained model for $n$ samples and we want to incorporate $x_{n+1}$ into the model we need to recompute the inverse of the new $n+1$ by $n+1$ covariance matrix ${K_{n+1}}^{-1}$. However, the time complexity of the inverse computation is $O(n^3)$ which makes the operation prohibitive in an online setting. We make use of the partitioned inverse equations presented in [11] for a positive definite matrix where we have the inverse ${K_n}^{-1}$, ${K_{n+1}}^{-1}$ can then be computed as in Equation 21.

$${K_{n+1}}^{-1} = \begin{bmatrix} F & b \\ b & D \end{bmatrix} \quad (21)$$

where $D$, $b$ and $F$ are given by Equations 22, 23 and 24

$$D = (k - {k_{n+1}}^T {K_n}^{-1} k_{n+1})^{-1} \quad (22)$$

$$b = -DK_n{}^{-1}k_{n+1} \tag{23}$$

$$F = K_n{}^{-1} - Dk_{n+1}k_{n+1}{}^T \tag{24}$$

and $k = K(x_{n+1}, x_{n+1})$ where $k_{n+1}$ is the covariance vector of $x_{n+1}$ with all $n$ samples.

Similar to adding a new sample, removing a sample is equivalent to removing the corresponding column from $K_{n+1}{}^{-1}$. If we want to remove the $j$th column, we will first need to permute the $j$th column and $j$th row to the end of the matrix. Using the above definitions we get $K_n{}^{-1} = F + Dk_{n+1}k_{n+1}{}^T$. These updates where we only add or remove a single column to the inverse covariance matrix are referred to as *rank-1 updates*.

## 6.2 Data Replacement Policy

To update the models of different query types online we maintain a set of recently observed response times for each query type. We replace old samples in this set with newly observed samples and cap the number of samples maintained for each query type to 400, which we refer to as the *sample limit* $C$. There are two motivations for setting a cap on the number of samples. First, for all the experiments that we conducted we saw the greatest gains in accuracy for the first 100 samples and diminishing returns subsequently. Second, and more importantly, old response time data for particular query mixes becomes irrelevant given new response time samples for a close enough query mix. In fact, stale response time data, if kept, results in inaccurate predictions over time similar to an offline trained model which is not updated online. This happens because of data evolution and changes in database access patterns over time. We add new samples using rank-1 updates (Section 6.1) unconditionally as long as we have less than $C$ samples. This is the case when we need to learn a model for a query online and is discussed in Section 7.1. Once we have $C$ samples, we replace the sample that is closest to a new sample in the input space. A very suitable measure of proximity in the input space is the kernel function (the covariance function).

## 6.3 Data Reuse Policy

To better adjust to dynamic changes in the system, like addition of queries and changes in resource allocations we employ two policies for reuse of data accumulated by the model of each query type:

**1. Keeping prior data with new kernel function:** Whenever resource allocation is changed, we keep the existing data in models and recompute the $K^{-1}$ matrix with the new kernel function.

**2. Extending prior data:** To model the impact of new query types on the existing query models (by adding samples containing this new query type), we extend the dimension of the query mix from $T$ to $T + 1$ for all existing samples. This extension does not require re-computation of the covariances between the samples.

# 7 Configuration Model

Even when real-time data is incorporated into the model and the data reuse policies are adopted as described in the previous section, an online response time model takes too long to converge below reasonable error rates if uninformative hyper-parameters are used for the model. This is demonstrated experimentally in Section 8.4.3. For the online response time models to converge quickly we need good hyper-parameters. In our approach, we use the configuration model (described next) to predict these hyper-parameters for response time models.

During the offline training phase, we learn the response time models for different configurations. We explored how the kernel hyper-parameters $\theta$ are affected by factors such as the average response time of a query, the variance of the query response time, and the configuration parameters including buffer pool, memory, and CPU allocation. We found that the query response time, buffer pool, and CPU allocation (in that order) accounted for a significant amount of variation in the values of the hyper-parameters for response time models. Therefore, we build a *Gaussian Process Configuration Model (GPCM)* to leverage prior knowledge from the offline trained models to predict hyper-parameters for new queries on seen and unseen configurations. GPCM is maintained offline and is updated with the

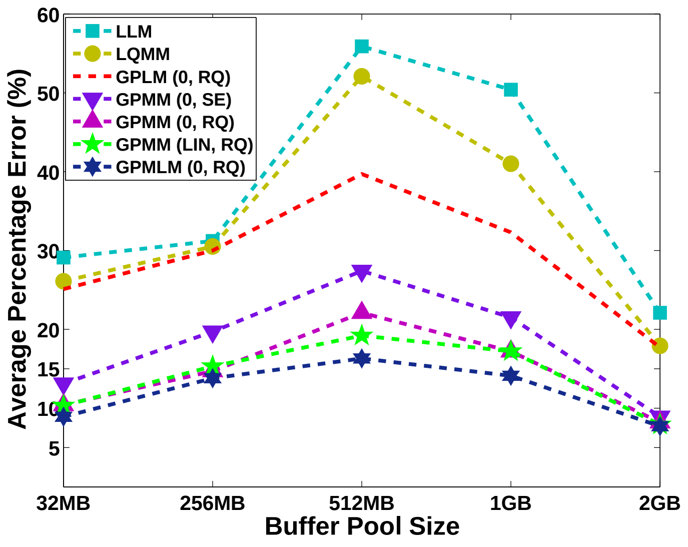

Figure 5: Average percentage error of different models

hyper-parameters of a newly trained response time model for a particular query on a configuration. GPCM is query type agnostic and only encodes the mean and variance in response time and the configuration parameters as follows:

$$[R_{MEAN}, R_{SD}, BP, CPU_{NUM}, CPU, MEM] \Rightarrow [\eta, \sigma_f^2, \alpha]$$

Note that there are two GPCMs for the SE based GP models for predicting $\eta, \sigma_f^2$, and three GPCMs for RQ based GP models for predicting each of $\eta, \sigma_f^2, \alpha$. $BP$ and $MEM$ are encoded in MBs (e.g. 2GB as 2048) and CPU in MHz (e.g. 3.0GHz as 3000) where as $R_{MEAN}$ and $R_{SD}$ are encoded in seconds.

## 8 Experimental Evaluation

### 8.1 Experimental Setup

For our experiments we use an IBM blade server with 2 dual core AMD Opteron processors 2216 HE at 2.2 GHz, 8 GB physical memory, and a 45 GB SCSI hard drive. As our software configuration, we use 64-bit Ubuntu Server 10.04 running Linux Kernel 2.6.32-21 with the PostgreSQL 8.4.4 DBMS (referred to as Postgres) and Xen 3.2.3. In this work, we build and train response time models for TPC-H benchmark queries [20]. TPC-H is a decision support (DSS) benchmark that consists of 22 query types. These query types model a real world data warehousing environment where complex ad-hoc queries are expected to be run against the data warehouse. We use a scale factor 1 database with a total size of 2.3 GB on disk. Note that our techniques are generally applicable to other DSS and online transaction processing (OLTP) style workloads. Training models for these other workloads is a part of our future work.

### 8.2 Model Accuracy

We start by comparing the accuracy of the proposed GP models and the linear Gaussian models. We measure the accuracy of predictions using the average percentage error (APE) which is calculated by taking the average of the percentage error $\frac{|y-f(x)|}{y}$ for each prediction.

### 8.2.1 Effect of Buffer Pool Size

Figure 5 shows the relative error for all 22 TPC-H query types at different buffer pool sizes, using GP and linear Gaussian models. All models are learned offline using the response time data collected for query mix samples generated using our workload generator. The figure shows that linear models are not sufficiently accurate. They perform well for very small and very large buffer pool sizes where almost nothing fits in the buffer pool (32MB) or where almost the complete database fits in the buffer pool (2GB). However, the prediction error can be close to 50% in some cases, which renders these models ineffective for most practical purposes. Thus, the simplicity of linear Gaussian models comes at the expense of insufficient accuracy, and we do not consider them further in this paper, focusing instead on GP models. As expected, GP models perform significantly better than linear models. Even the simplest GPLM outperforms LQMM on all buffer pool sizes, demonstrating the inherent non-linearity of query response time distributions.

### 8.2.2 Effectiveness under Overload

Our results on various configurations show that the variance in response times increases significantly when the database is overloaded, i.e., $l \geq 70$. However, all GP based models were able to capture this variance in the predicted confidence intervals. An example of this is Figure 6, where the response times of TPC-H Q12 are modeled using GPMM (0, RQ) on a 2GB buffer pool. The shaded area in Figure 6 represents the 95% confidence interval for the mean response time prediction. Note that the prediction curve is smooth as a function of the 22-dimensional query mix, however it is shown only as a function of 1-dimensional load. Our results consistently showed that the predicted variance by GP models increases for overloaded databases where there is considerable variation in response times and for test cases which were very far from the training data. This ability to capture the uncertainty associated with prediction is yet another advantage of our approach and can be directly propagated to a decision making system.

### 8.2.3 Overall Accuracy

Figure 7 shows the predicted values and the actual response time values for GPMLM (0, RQ) model for around 4700 test cases for all query types on all five buffer pool sizes. Overall, GPMLM (0, RQ) is the best performing model with an average error ranging from 7.7% for the 2 GB buffer pool to 16.3% for 512 MB buffer pool, as shown in Figure 5. Note that we do not propose GPMLM with linear mean because of the significant training cost associated with linear mean based GP models as discussed in 8.3.1. For GPMLM (0, RQ), the overall correlation coefficient for all configurations and all query types is 0.94.

## 8.3 Online Adaptability

### 8.3.1 Online Costs

**Model Adaptability:** Despite the simplicity of linear models, due to the high error rates on some configurations (shown in Section 8.2) we do not adapt these models online. On the other hand, GP models with linear mean, which are highly accurate as shown in Section 8.2, pose serious scalability problems. This is because the number of hyper-parameters for the linear mean function is $T + 1$, where $T$ is the number of different query types. With around 500 samples for a query type it takes approximately one hour to learn the hyper-parameters for 22 models, one for each query type. Furthermore, the hyper-parameters of GPMM(LIN,*) are highly sensitive to the actual response times and therefore cannot be predicted using GPCM. Therefore, as with linear models, we do not adapt GP based models with prior linear mean functions to an online setting.

GP*(0,SE) are the most efficient GP based models, since there are only two hyper-parameters ($\{\eta, \sigma_f^2\}$) to learn offline or predict in an online setting. However, the extra flexibility of the RQ kernel function with three hyper-parameters ($\{\eta, \sigma_f^2\, \alpha\}$) pays off with respect to model accuracy. Additionally, since the number of hyper-parameters for GP*(0,RQ) is independent of $T$, we adapt GP*(0,RQ) based models online and evaluate their performance in the following sections. The time to learn hyper-parameters for SE and RQ based models with 0 mean, 22 query types, and 500 samples per query type, is approximately 4 and 7 minutes, respectively.

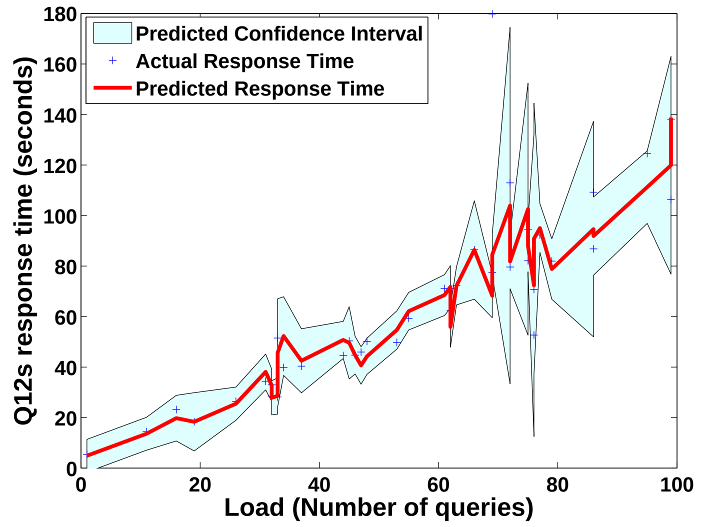

Figure 6: Response time predictions for TPC-H Q12 w/ 2 GB buffer pool (GPMM (0, RQ)).

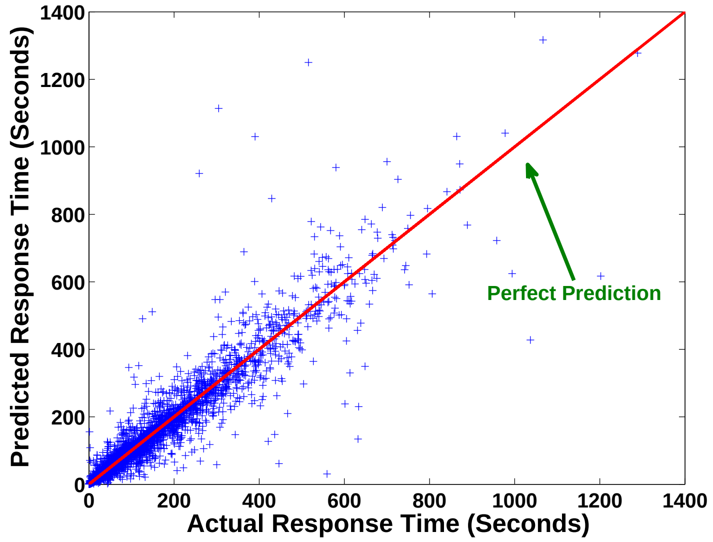

Figure 7: Overall correlation of predicted and actual response times for all 5 configurations GPMLM(0, RQ)

| | Complexity | Wall clock times ($n$ data samples) | | |
|---|---|---|---|---|
| | | **400** | **1000** | **3000** |
| GP - Point Prediction | $O(n)$ | 4ms | 17ms | 82ms |
| $K^{-1}$ Computation | $O(n^3)$ | 40ms | 413ms | 11s |
| Rank-1 $K^{-1}$ Updates | $O(n^2)$ | 12ms | 64ms | 623ms |
| Learning Hyper-Parameters | - | 13s | 19s | 1100s |

Table 1: Wall clock times for various operations

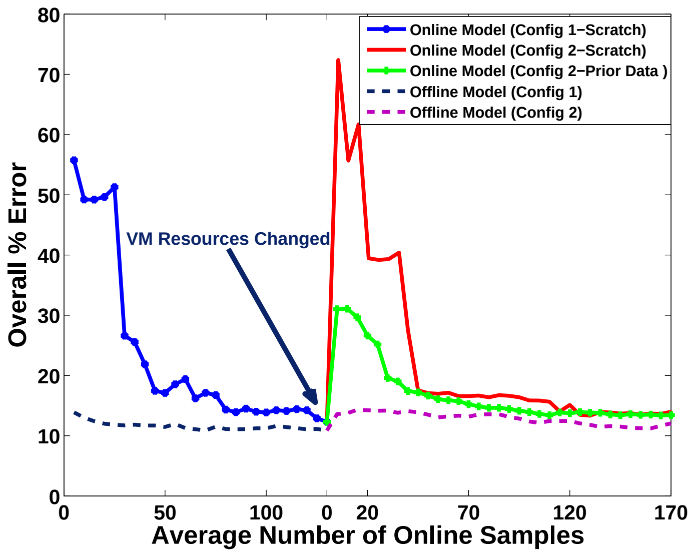

Figure 8: Online models for changing configurations

**Incremental Data Addition:** The complexity of adding or removing a new data sample using rank-1 updates is $O(n^2)$ as opposed to $O(n^3)$ for recomputing the inverse of the covariance matrix. As shown in Table 1 each new query (assuming 1000 data samples for that query's model) takes around 17ms for prediction and another 64ms for updating the model with the actual response time of the query. The prediction includes the time to calculate both the mean and variance of the predictive Gaussian distribution. The calculation of variance takes almost two thirds of the total prediction time. For an application of the response time model which does not require the predictive variance, only the mean response time can be computed to further reduce the prediction cost. Similarly, 64ms for updates include two calls, one for removing a sample and another for adding a sample. There are other opportunities that can be exploited for example, reusing the covariances computed for response time prediction. However, we have not implemented these features, leaving them for future work.

**Data Reuse Costs:** We adopt the policy of keeping prior data when new hyper-parameters are selected as in the case where resource allocations for a DBMS are changed. This requires re-computation of the covariance matrix with the new kernel function and then the inverse of the matrix. The total cost of this operation is approximately 7 seconds for 22 models with 1000 data samples each. The time taken scales linearly with the number of query types given a fixed number of samples for each model. Similarly, the cost of extending all existing data samples ($22 \times 1000$) by an additional '0' when a new query is added is reasonable (¡ 2 seconds) considering the frequency of such additions.

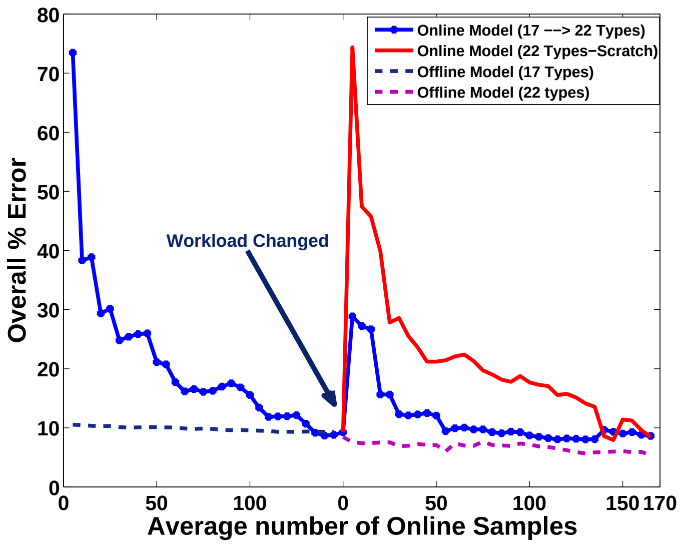

Figure 9: Online performance models for changing workload

### 8.3.2 Adapting to Dynamic Configurations

In order to model new (unseen) configurations, we learn an offline configuration model from the data collected at five buffer pool configurations on a physical machine with 2 cores at 2.2GHz with 8GB memory. We encode the CPU, memory, and buffer pool size as discussed in Section 7.

**Online Response Time Models:** Queries are generated and run online on a VM with 4GB memory and 3 cores at 2.2GHz (config 1 in Figure 8) and models are created for each query type dynamically by getting the kernel function hyper-parameters from GPCM. Note that in this case we have no prior response time data and the models are learned from scratch for this configuration. The blue line in Figure 8 shows that the average percentage error falls below 20% when the model has seen only under 40 samples on average. After the model's error rate falls below 12%, we reduce the number of VM CPUs from 3 to 2 and its memory from 4 GB to 2 GB (config 2 in Figure 8).

A simple policy is to get new hyper-parameters from the configuration model, discard the previously accumulated data by the online response time model, and learn the model for the new configuration from scratch as done for the previous configuration. As show in Figure 8 (red), a drawback of this approach is that initially before the model converges, the error rate can be very high. To prevent this from happening, the online model keeps all the existing data accumulated from previous configurations and evolves by incorporating actual observed response time data. Keeping response time data collected from the previous configuration until new data replaces it, significantly reduces the maximum error. This is shown in Figure 8 (green), where the model converges more quickly as compared to the case of starting from scratch (red). However, to reuse the previous response time data, we have to recompute the $K^{-1}$ matrix for each query's model. The cost associated with this computation is discussed in Section 8.3.1. This computation is done by the online model, whenever the resource allocation is changed.

**Comparison with Offline Trained Models:** Dashed lines in Figure 8 show the performance of offline trained models for the two unseen configurations for the queries seen online. These offline models are trained retrospectively for comparison purposes, i.e., the offline models are trained for the response time data seen during online operation. As shown in Figure 8, the online models converge quickly below reasonable error rates ( ¡ 20% ) for configurations for which we have no previous training data and ultimately approach the accuracy of models that are trained offline specifically for these configurations.

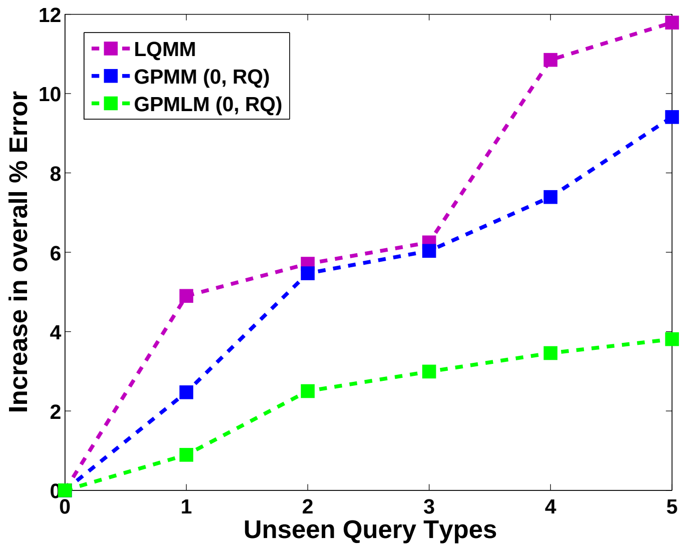

Figure 10: Impact of new queries on trained query models

### 8.3.3 Adapting to Dynamic Workloads

In addition to being used for unseen configurations the offline trained configuration model is used to learn response time models for unseen workloads. In this experiment, query mixes are generated and run against a DBMS with 2GB buffer pool size. The online response time models are learned from scratch for 17 randomly selected TPC-H query types and shown in Figure 9 (blue line). When the average accuracy of the models falls below 10%, the workload is changed by introducing the remaining 5 TPC-H query types in the system.

A simple policy for dealing with changing workloads is to discard all prior data accumulated by the online models and start learning online models from scratch (red line in Figure 9). The main drawback of this approach is that the initial error is too high and the models take longer to converge below acceptable error rates. The policy adopted for changing workloads is to extend the existing query mix vectors (Section 6). The models then evolve by incorporating new real-time response time data. As shown in Figure 9, there is a slight increase in error with the addition of 5 new query types and the models then converge quickly below an average error of less than 20%. Similar to Figure 8, the dashed lines in Figure 9 represent the error rates for the offline models trained retrospectively for the two workloads with 17 and 22 query types.

## 8.4 Model Robustness

### 8.4.1 Impact of new queries

In this experiment, we show that GP models generalize very well and are capable of modeling the addition of new query types online. We train our models offline using query mixes with 17 TPC-H query types. While the system is running, we introduce the remaining 5 query types which our trained model has never seen before. Figure 10 shows the impact of new queries on the model accuracy of existing queries for various GP models. The increase in percentage error for the model is relative to the percentage error when the model is running online and there are no new queries. Figure 10 shows that the accuracy of all the models decreases since each model is trained using query mixes with fewer query types (17) than the actual types (22) running in the system. LQMM and GPMM both suffer from addition of new queries with the overall percentage error increasing by almost 12% and 10%, respectively. However, GPMLM fares very well with an overall increase of less than 4% in percentage error. The reason for this resilience is that although the number of queries of each new query type is not captured by the query mix in GPMLM, but the contribution by

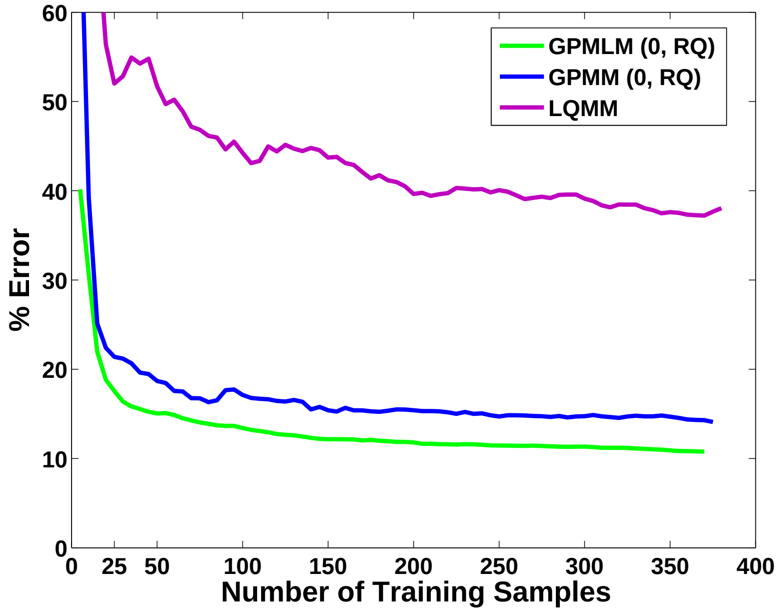

Figure 11: Average percentage error for all configurations

new queries to the overall load on the system is effectively captured by the model. These results clearly demonstrate that GPMLM is better suited to handle the addition of new query types as compared to mix only models. Note that we adopt the policy of extending the query mix samples whenever a new query type is added, resulting in a similar convergence of all query models as described in Section 8.3.3 after the initial rise in error shown in Figure 10.

### 8.4.2 Offline Model Convergence

Figure 11 shows the average percentage error for three models trained offline for 8 configurations and all query types with the number of samples. The hyper-parameters are re-learned after every 5 samples in an offline manner. The experiment shows the best models that can be learned when the parameters are re-learned after every 5 samples and the learning process taking almost 4 hours for query models for each configuration. The results show that the most gains are made for the first 50 samples, motivating us to set a cap on the number of data samples.

LQMM not only has the highest initial error rate but also has a more uneven behavior when new data samples are added and model retrained. Both GP based models perform significantly better and the initial quick drop in percentage error which we saw for all GP based models make the models suitable for online adaptation. As shown in Figure 11 GPMLM is the best performing model with an average of less than 25 samples for the model to fall below 20% error. GPMLM performed worst for VM with 1 CPU at 2.2GHz and 1GB memory and 512MB buffer pool, where the APE dropped below 20% after an average of 46 data samples. The next section evaluates the convergence of different models when they are trained online.

### 8.4.3 Online Model Convergence

In this section, we describe how models can be learned efficiently for new query types on-the-fly, as opposed to requiring offline re-training. We train an offline model for 21 query types and learn the model online for a new query type using GPMM and GPMLM models with RQ kernel and 0 mean. The experiment is repeated for all 22 TPC-H query types where the offline model is trained for the remaining 21 queries on a 512MB buffer pool allocation. For the new query type, we have no training data so we generate random query mixes online, creating and evolving a new model dynamically for the query. As described before, new response times can be incorporated into the model

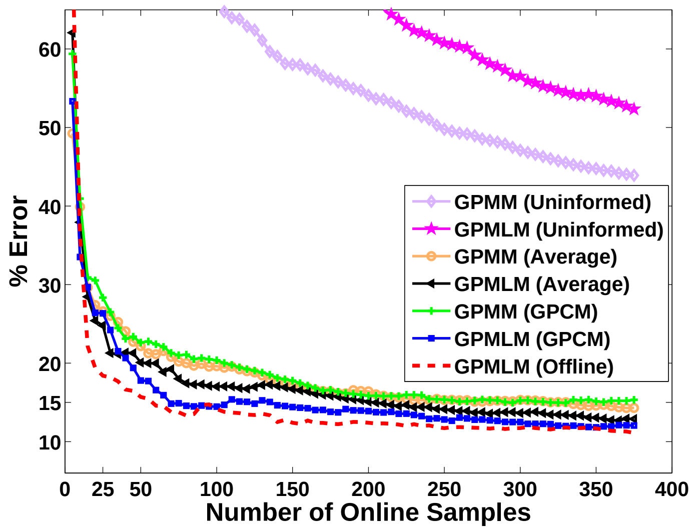

Figure 12: Comparison of hyper-parameter selection schemes

efficiently. However, hyper-parameter optimization cannot be performed online. For RQ kernel and 0 mean, our goal is to estimate values of $\eta, s_f^2, \alpha$ for the response time model of a new query type. We compare three different schemes for selecting hyper-parameters to seed the model for new query type.

**1. Uninformative Hyper-Parameters:** We set the values of all three kernel hyper-parameters to 0. Figure 12 shows that without a proper kernel function the online models converge at a very slow rate, even with the addition of new response time data. This counters any advantage gained by online modeling as the prediction error will be too high for too long. In fact, GPMLM model performs worse than the GPMM model and the APE is more than 50% even after the addition of 370 response times.

**2. Average Hyper-Parameters:** In this scheme, we simply take the average of the hyper-parameters of the 21 query models that were trained offline and use these values as hyper-parameters for the response time model of a new query type. As shown in Figure 12, the average hyper-parameter selection yields an online model that converges quickly. This also demonstrates the dependence of hyper-parameters on the configuration. However, a limitation of this approach is that when we have no offline trained model for any query, as in the case of unseen configurations, we will not have any hyper-parameters to work with.

**3. Predicting Hyper-Parameters with GPCM:** We next leverage the configuration model to predict the hyper-parameters for the response time model of a new query type. The three hyper-parameters are a function of mean response time, variance of response time, and the buffer pool. In this case the GPCM is learned for 3 buffer pool configurations (32MB, 512MB and 2GB). The configuration model waits for five response time samples and then generates hyper-parameters for initializing response time models. The GPCM generates new hyper-parameters after every *k=50* samples to ensure that we have "good" parameters based on the current average response time of a new query type. Note that when new hyper-parameters are generated, the response time model recomputes the inverse covariance matrix ($K^{-1}$). This is the same as when new hyper-parameters are generated for unseen configurations. Both GPMM and GPML models do better with hyper-parameters predicted using GPCM, requiring fewer than half the samples compared to the average hyper-parameters case to converge to an error below 20%. The dashed line in Figure 12 represents the offline model trained retrospectively for the query mixes seen online by the DBMS with optimized hyper-parameters. The results show that most gains are made for the first hundred samples, motivating us to set a cap on the number of data samples ($C$).

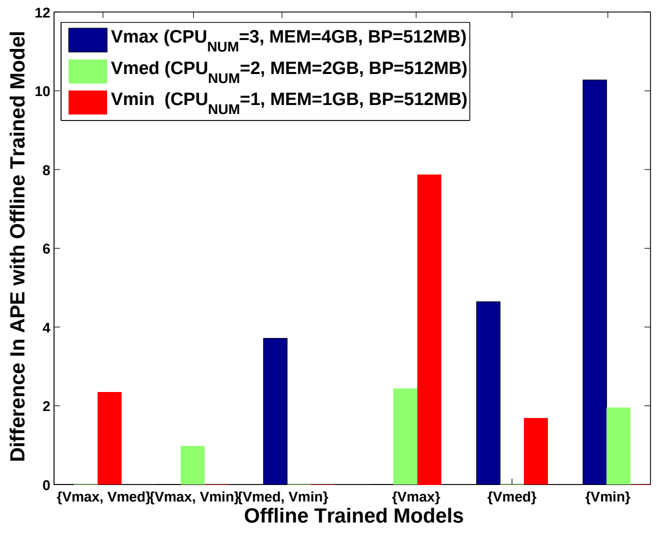

Figure 13: GPLML (0,RQ) % error for unseen configurations.

### 8.4.4 Configuration Model Accuracy

In this experiment we compare the response time model's error rate for the three different VM configurations when the hyper-parameters are predicted using GPCM to the error rate for an offline trained response time model for each configuration. The three configurations are termed $V_{min}$, $V_{med}$, and $V_{max}$ and are described in Figure 13. The x-axis represents the offline trained response time models for different configurations that are used to learn GPCM. This learned GPCM is then used to predict the hyper-parameters for configurations for which there is no offline trained model. The hyper-parameters are used to train models for all 22 TPC-H query types for the previously unseen configuration. The y-axis in the figure represents the difference in APE between the online trained model and an offline trained model. The offline model is trained by waiting till 100 samples have been observed by the online model then learning the offline model from these samples. The offline model in this case represents an ideal case where the training data is exactly the same as the test data. $V_{med}$ generally has the smallest difference in error when hyper-parameters are predicted from GPCM trained using one or two other VM configurations. On the other hand the two cases where we see the greatest increase in error for the online trained models are when GPCM is trained for $V_{max}$ and hyper-parameters are predicted for $V_{min}$ (7.8%) and similarly when GPCM is trained for $V_{min}$ and the hyper-parameters are predicted for $V_{max}$ (10.3%). The reason for this is that the configuration for which an offline model is trained is far from the configuration for which hyper-parameters are predicted. Note that the number of samples for each configuration are $T = 22$, one for each query type.

## 9 Discussion and Future Work

Since the online response time models rely on the configuration model for getting good hyper-parameters, in cases when the configuration model is learned for configurations that are very different from the configuration for which an online model is learned, the prediction error can be high. However, since the space of valid configurations is small compared to the space of possible workloads, a DBA can easily train the configuration model for a few representative configurations and still get high convergence rates for online models for various unseen configurations and workloads. A limitation of the current approach is that when a new configuration parameter is added (e.g., I/O), the offline configuration model needs to be retrained. However, we expect these additions to be relatively infrequent.

It would be interesting to explore hierarchical Bayesian networks and sampling based techniques to get a number of hyper-parameters for constructing online response time models and selecting parameters dynamically based on error rates. This would have the added advantage of relieving the DBA of training the offline configuration model even for a few configurations and the whole process could become completely online. Evaluating the trade-offs in accuracy and learning costs of such models are part of our future work.

Although we target database appliances in this work, our techniques are not specific to database workloads and can be applied to other workloads that have well defined request types where requests of a particular type are expected to behave similarly. Examples include HTTP requests in a web-server, or different servlet types in an application server. Applying our techniques to these systems is part of our future work.

## 10 Conclusion

We presented a novel experiment-driven performance model for predicting DBMS query response times using Gaussian Process models based on Bayesian learning techniques. One of the main strengths of using a Bayesian approach is that it allows the model to effectively adapt to changes in the query workload, database characteristics, or machine configuration in an online manner, without the need for re-training. This is in stark contrast to all prior work in this area, which required re-training the model for any of these changes. The ability of a response time model to adapt online to changes in configurations is highly desirable in today's cloud computing environments where changes are frequent. For example resources allocated to a database appliance can change online via the virtualization layer. Through extensive experimentation using the TPC-H benchmark, we show that our best model (GPMLM) performs very well in terms of goodness of fit, accuracy, and model training/prediction time. We also show that our models are able to quickly adapt to new unseen configurations, demonstrating their applicability to the dynamic environment of the cloud.

## References

[1] Amazon Elastic Compute Cloud (Amazon EC2). [online] http://aws.amazon.com/ec2/.

[2] Amazon Relational Database Service (Amazon RDS). [online] http://aws.amazon.com/rds/.

[3] Microsoft SQLAzure. [online] http://www.microsoft.com/en-us/sqlazure/default.aspx.

[4] M. Abouzour, K. Salem, and P. Bumbulis. Automatic tuning of the multiprogramming level in Sybase SQL Anywhere. In *Workshop on Self-managing Database Systems (SMDB)*, 2010.

[5] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Interaction-aware scheduling of report generation workloads. *VLDB Journal*, 2011.

[6] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Extending Database Technology (EDBT)*, 2011.

[7] E. B. Andersen. Asymptotic properties of conditional maximum-likelihood estimators. *Journal of the Royal Statistical Society. Series B (Methodological)*, 32(2), 1970.

[8] D. Bradley and R. Gupta. On the distribution of the sum of n non-identically distributed uniform random variables. *Annals of the Institute of Statistical Mathematics*, 54(3), 2002.

[9] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *International Conference on Data Engineering (ICDE)*, 2009.

[10] C. Gupta, A. Mehta, and U. Dayal. PQR : Predicting query execution times for autonomous workload management. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[11] W. W. Hager. Updating the inverse of a matrix. *SIAM Review*, 31, 1989.

[12] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[13] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1), 2003.

[14] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *Symposium on Modeling,Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2005.

[15] A. O'Hagan. Some bayesian numerical analysis. *Bayesian Statistics 4*, 1992.

[16] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. Cambridge: MIT Press, 2006.

[17] E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask 'what-if'? *Knowledge Eng. Review*, 21(3), 2006.

[18] V. Thummala and S. Babu. iTuned: a tool for configuring and visualizing database parameters. In *International Conference on Management of Data (SIGMOD)*, 2010.

[19] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *International Conference on Data Engineering (ICDE)*, 2010.

[20] The TPC-H Benchmark. [online] http://www.tpc.org/tpch/.

[21] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. In *International Conference on Autonomic Computing (ICAC)*, 2010.

[22] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *International Conference on Very Large Data Bases (VLDB)*, 2002.

[23] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *International Conference on Autonomic Computing (ICAC)*, 2007.