Tuning Database Configuration Parameters with iTuned

Songyun Duan, Vamsidhar Thummala, Shivnath Babu*
Department of Computer Science
Duke University
Durham, North Carolina, USA
{syduan,vamsi,shivnath}@cs.duke.edu

ABSTRACT

Database systems have a large number of configuration parameters that control memory distribution, I/O optimization, costing of query plans, parallelism, many aspects of logging, recovery, and other behavior. Regular users and even expert database administrators struggle to tune these parameters for good performance. The wave of research on improving database manageability has largely overlooked this problem which turns out to be hard to solve. We describe iTuned, a tool that automates the task of identifying good settings for database configuration parameters. iTuned has three novel features: (i) a technique called Adaptive Sampling that proactively brings in appropriate data through planned experiments to find high-impact parameters and high-performance parameter settings, (ii) an executor that supports online experiments in production database environments through a cycle-stealing paradigm that places near-zero overhead on the production workload; and (iii) portability across different database systems. We show the effectiveness of iTuned through an extensive evaluation based on different types of workloads, database systems, and usage scenarios.

1. INTRODUCTION

Consider the following real-life scenario from a small to medium business (SMB) enterprise. Amy, a Web-server administrator, maintains the Web-site of a ticket brokering company that employs eight people. Over the past few days, the Web-site has been sluggish. Amy collects monitoring data, and tracks the problem down to poor performance of queries issued by the Web server to a backend database. Realizing that the database needs tuning, Amy runs the database tuning advisor. (SMBs often lack the financial resources to hire full-time database administrators, or DBAs.) She uses system logs to identify the workload W of queries and updates to the database. With W as input, the advisor recommends a database design (e.g., which indexes to build, which materialized views to maintain, how to partition the data). However, this recommendation fails to solve the current problem: Amy had already designed the database this way based on a previous invocation of the advisor.

*Supported by NSF CAREER and faculty awards from IBM.

VLDB '09, August 24-28, 2009, Lyon, France

Amy recalls that the database has *configuration parameters*. For lack of better understanding, she had set them to default values during installation. The parameters may need tuning, so Amy pulls out the 1000+ page database tuning manual. She finds many dozens of configuration parameters like buffer pool sizes, number of concurrent I/O daemons, parameters to tune the query optimizer's cost model, and others. Being unfamiliar with most of these parameters, Amy has no choice but to follow the tuning guidelines given. One of the guidelines looks promising: if the I/O rate is high, then increase the database buffer pool size. However, on following this advice, the database performance drops even further. (We will show an example of such behavior shortly.) Amy is puzzled, frustrated, and undoubtedly displeased with the database vendor.

Many of us would have faced similar situations before. Tuning database configuration parameters is hard but critical: bad settings can be orders of magnitude worse in performance than good ones. Changes to some parameters cause local and incremental effects on resource usage, while others cause drastic effects like changing query plans or shifting bottlenecks from one resource to another. These effects vary depending on hardware platforms, workload, and data properties. Groups of parameters can have nonindependent effects, e.g., the performance impact of changing one parameter may vary based on different settings of another parameter.

iTuned: Our core contribution in this paper is *iTuned*, a tool that automates parameter tuning. iTuned can provide a very different experience to Amy. She starts iTuned in the background with the database workload W as input, and resumes her other work. She checks back after half an hour, but iTuned has nothing to report yet. When Amy checks back an hour later, iTuned shows her an intuitive visualization of the performance impact each database configuration parameter has on W. iTuned also reports a setting of parameters that is 18% better in performance than the current one. Another hour later, iTuned has a 35% better configuration, but Amy wants more improvement. Three hours into its invocation, iTuned reports a 52% better configuration. Now, Amy asks for the configuration to be applied to the database. Within minutes, the actual database performance improves by 52%.

We now present a real example to motivate the technical innovations in iTuned. Figure 1 is a *response surface* that shows how the performance of a complex TPC-H query [19] in a PostgreSQL database depends on the *shared_buffers* and *effective_cache_size* parameters. *shared_buffers* is the size of PostgreSQL's main buffer pool for caching disk blocks. The value of *effective_cache_size* is used to determine the chances of an I/O hitting in the OS file cache, so its recommended setting is the size of the OS file cache. The following observations can be made from Figure 1 (detailed explanations are given later in Section 7):

- The surface is complex and nonmonotonic.

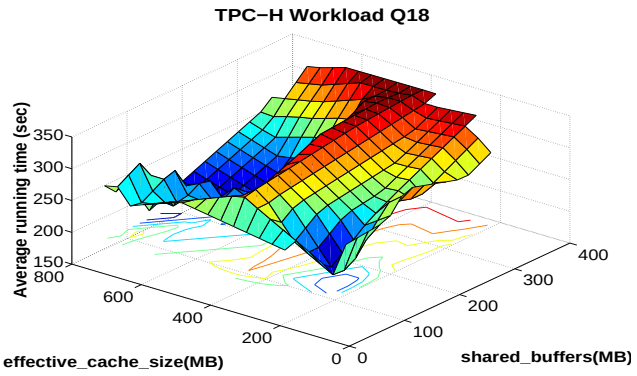

Figure 1: 2D projection of a response surface for TPC-H Query 18; Total database size = 4GB, Physical memory = 1GB

- Performance drops sharply as *shared_buffers* is increased beyond 20% (200MB) of available memory. This effect will cause an "increase the buffer pool size" rule of thumb to degrade performance for configuration settings in this region.
- The effect of changing *effective_cache_size* is different for different settings of *shared_buffers*. Surprisingly, the best performance comes when both parameters are set low.

Typical database systems contain few tens of parameters whose settings can impact workload performance significantly [13].[1] There are few automated tools for holistic tuning of these parameters. The majority of tuning tools focus on the logical or physical design of the database. For example, index tuning tools are relatively mature (e.g., [4]). These tools use the query optimizer's cost model to answer *what-if questions* of the form: how will performance change if index I were to be created? Unfortunately, such tools do not apply to parameter tuning because the settings of many high-impact parameters are not accounted for by these models.

Many tools (e.g., [17, 20]) are limited to specific classes of parameters like buffer pool sizes. IBM DB2's Configuration Advisor recommends default parameter settings based on answers provided by users to some high-level questions (e.g., is the environment OLTP or OLAP?) [12]. All these tools are based on predefined models of how parameter settings affect performance. Developing such models is nontrivial [21] or downright impossible because response surfaces can differ markedly across database systems (e.g., DB2 Vs. PostgreSQL), platforms (e.g., Linux Vs. Solaris, databases run in virtual machines [16]), workloads, and data properties.[2] Furthermore, DB2's Configuration Advisor offers no further help if the recommended defaults are still unsatisfactory.

In the absence of holistic parameter-tuning tools, users are forced to rely on trial-and-error or rules-of-thumb from manuals and experts. How do expert DBAs overcome this hurdle? They usually run *experiments* to perform what-if analysis during parameter tuning. In a typical experiment, the DBA would:

- Create a replica of the production database on a test system.
- Initialize database parameters on the test system to a chosen setting. Run the workload that needs tuning, and observe the resulting performance.

Taking a leaf from the book of expert DBAs, iTuned implements an experiment-driven approach to parameter tuning. Each experiment gives a point on a response surface like Figure 1. Reliable techniques for parameter tuning have to be aware of the underlying response surface. Therefore, a series of carefully-planned experiments is a natural approach to parameter tuning. However, running experiments can be a time-consuming process.

Users don't always expect instantaneous results from parameter tuning. They would rather get recommendations that work as described. (Configuring large database systems typically takes on the order of 1-2 weeks [12].) Nevertheless, to be practical, an automated parameter tuning tool has to produce good results within a few hours. In addition, several questions need to be answered like: (i) which experiments to run? (ii) where to run experiments? and (iii) what if the SMB does not have a test database platform?

1.1 Our Contributions

To our knowledge, iTuned is the first practical tool that uses planned experiments to tune database configuration parameters. We make the following contributions.

Planner: iTuned's experiment planner uses a novel technique, called *Adaptive Sampling*, to select which experiments to conduct. Adaptive Sampling uses the information from experiments done so far to estimate the utility of new candidate experiments. No assumptions are made about the shape of the underlying response surface, so iTuned can deal with simple to complex surfaces.

Executor: iTuned's experiment executor can conduct online experiments in a production environment while ensuring near-zero overhead on the production workload. The executor is controlled through high-level policies. It hunts proactively for idle capacity on the production database, hot-standby databases, as well as databases for testing and staging of software updates. The executor's design is particularly attractive for databases that run in cloud computing environments providing pay-as-you-go resources.

Representation of uncertain response surfaces: iTuned introduces *GRS*, for *Gaussian process Representation of a response Surface*, to represent an approximate response surface derived from a set of experiments. GRS enables: (i) visualization of response surfaces with confidence intervals on estimated performance; (ii) visualization and ranking of parameter effects and inter-parameter interactions; and (iii) recommendation of good parameter settings.

Scalability: iTuned incorporates a number of features to reduce tuning time and to scale to many parameters: (i) a sensitivity-analysis algorithm that quickly eliminates parameters with insignificant effect; (ii) planning and conducting parallel experiments; (iii) aborting low-utility experiments early, and (iv) workload compression.

Evaluation: We demonstrate the advantages of iTuned through an empirical evaluation along a number of dimensions: multiple workload types, data sizes, database systems (PostgreSQL and MySQL), and number of parameters. We compare iTuned with recent techniques proposed for parameter tuning both in the database [5] as well as other literature [18, 23]. We consider how good the results are and the time taken to produce them.

2. ABSTRACTION OF THE PROBLEM

Response Surfaces: Consider a database system with workload W and d parameters $x_1, \ldots, x_d$ that a user wants to tune. The values of parameter x_i, $1 \leq i \leq d$, come from a known domain $dom(x_i)$. Let *DOM*, where $DOM \subseteq \Pi_{i=1}^{d} dom(x_i)$, represent the space of possible settings of $x_1, \ldots, x_d$ that the database can have. Let y denote the performance metric of interest. Then, there exists a response surface, denoted S_W, that determines the value of y for workload W for each setting of $x_1, \ldots, x_d$ in *DOM*. That is, $y = S_W(x_1, \ldots, x_d)$. S_W is unknown to iTuned to begin with. The core task of iTuned is to find settings of $x_1, \ldots, x_d$ in *DOM* that give close-to-optimal values of y. In iTuned:

- Parameter x_i can be one of three types: (i) database or system

[1] The total number of database configuration parameters may be more than a hundred, but most have reasonable defaults.

[2] Section 7 provides empirical evidence.

configuration parameters (e.g., buffer pool size); (ii) knobs for physical resource allocation (e.g., % of CPU); or (iii) knobs for workload admission control (e.g., multi-programming level).
- y is any performance metric of interest, e.g., y in Figure 1 is the time to completion of the workload. In OLTP settings, y could be, e.g., average transaction response time or throughput.
- Because iTuned runs experiments, it is very flexible in how the database workload W is specified. iTuned supports the whole spectrum from the conventional format where W is a set of queries with individual query frequencies [4], to mixes of concurrent queries at some multi-programming level, as well as real-time workload generation by an application.

Experiments and Samples: Parameter tuning is performed through experiments planned by iTuned's planner, which are conducted by iTuned's executor. An experiment involves the following actions that leverage mechanisms provided by the executor (Section 5):

1. Setting each x_i in the database to a chosen setting $v_i \in dom(x_i)$.
2. Running the database workload W.
3. Measuring the performance metric $y = p$ for the run.

The above experiment is represented by the setting $X = \langle x_1 = v_1,\ldots,x_d = v_d\rangle$. The outcome of this experiment is a *sample* from the response surface $y = S_W(x_1,\ldots,x_d)$. The sample in the above experiment is $\langle X, y\rangle = \langle x_1 = v_1,\ldots,x_d = v_d, y = p\rangle$.

As iTuned collects such samples through experiments, it learns more about the underlying response surface. However, experiments cost time and resources. Thus, iTuned aims to minimize the number of experiments required to find good parameter settings.

3. OVERVIEW OF ITUNED

Gridding: *Gridding* is a straightforward technique to decide which experiments to conduct. Gridding works as follows. The domain $dom(x_i)$ of each parameter x_i is discretized into k values $l_{i1},\ldots,l_{ik}$. (A different value of k could be used per x_i.) Thus, the space of possible experiments, $DOM \subseteq \Pi_{i=1}^{d} dom(x_i)$, is discretized into a grid of size k^d. Gridding conducts experiments at each of these k^d settings. Gridding is reasonable for a small number of parameters. This technique was used in [18] while tuning four parameters in the Berkeley DB database. However, the exponential complexity makes gridding infeasible (curse of dimensionality) as the number of parameters increase. For example, it takes 22 days to run experiments via gridding for $d = 5$ parameters, $k = 5$ distinct settings per parameter, and average run-time of 10 minutes per experiment.

SARD: The authors of [5] proposed *SARD* (Statistical Approach for Ranking Database Parameters) to address a subset of the parameter tuning problem, namely, ranking $x_1,\ldots,x_d$ in order of their effect on y. SARD decides which experiments to conduct using a technique known as the *Plackett Burman (PB) Design* [11]. This technique considers only two settings per parameter—giving a 2^d grid of possible experiments—and picks a predefined $2d$ number of experiments from this grid. Typically, the two settings considered for x_i are the lowest and highest values in $dom(x_i)$. Since SARD only considers a linear number of corner points of the response surface, it can be inaccurate for surfaces where parameters have nonmonotonic effects (Figure 1). The corner points alone can paint a misleading picture of the shape of the full surface.[3]

Adaptive Sampling: The problem of choosing which experiments to conduct is related to the sampling problem in databases. We can consider the information about the full response surface S_W to be stored as records in a (large) table T_W with attributes $x_1,\ldots,x_d,y$. An example record $\langle x_1 = v_1,\ldots,x_d = v_d, y = p\rangle$ in T_W says that the performance at the setting $\langle x_1 = v_1,\ldots,x_d = v_d\rangle$ is p for the workload W under consideration. Experiment selection is the problem of sampling from this table. However, the difference with respect to conventional sampling is that the table T_W is never fully available. Instead, we have to pay a cost—namely, the cost of running an experiment—in order to sample a record from T_W.

The gridding and SARD approaches collect a predetermined set of samples from T_W. A major deficiency of these techniques is that they are not *feedback-driven*. That is, these techniques do not use the information in the samples collected so far in order to determine which samples to collect next. (Note that conventional random sampling in databases is also not feedback-driven.) Consequently, these techniques either bring in too many samples or too few samples to address the parameter tuning problem.

iTuned uses a novel feedback-driven algorithm, called *Adaptive Sampling*, for experiment selection. Adaptive Sampling analyzes the samples collected so far to understand how the surface looks like (approximately), and where the good settings are likely to be. Based on this analysis, more experiments are done to collect new samples that add maximum utility to the current samples.

Suppose n experiments have been run at settings $X^{(i)}$, $1 \le i \le n$, so far. Let the corresponding performance values observed be $y^{(i)} = y(X^{(i)})$. Thus, the samples collected so far are $\langle X^{(i)}, y^{(i)}\rangle$. Let $X^\star$ denote the best-performing setting found so far. Without loss of generality, we assume that the tuning goal is to minimize y.

$$X^\star = \arg\min_{1\le i\le n} y(X^{(i)})$$

Which sample should Adaptive Sampling collect next? Suppose the next experiment is done at setting X, and the performance observed is $y(X)$. Then, the *improvement IP(X)* achieved by the new experiment X over the current best-performing setting $X^\star$ is:

$$IP(X) = \begin{cases} y(X^\star) - y(X) & \text{if } y(X) < y(X^\star) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Ideally, we want to pick the next experiment X so that the improvement *IP(X)* is maximized. However, a proverbial chicken-and-egg problem arises here: the improvement depends on the unknown value of $y(X)$ which will be known only after the experiment is done at X. We instead compute *EIP(X)*, the *expected improvement* when the next experiment is done at X. Then, the experiment that gives the maximum expected improvement is selected.

The n samples from experiments done so far can be utilized to compute *EIP(X)*. We can estimate $y(X)$ based on these samples, but our estimate will be uncertain. Let $\hat{y}(X)$ be a random variable representing our estimate of $y(X)$ based on the collected samples. The distribution of $\hat{y}(X)$ captures our current uncertainty in the actual value of performance at setting X. Let $\texttt{pdf}_{\hat{y}(X)}(p)$ denote the *probability density function* of $\hat{y}(X)$. Then:

$$X_{next} = \arg\max_{X\in DOM} EIP(X) \quad (2)$$

$$EIP(X) = \int_{p=-\infty}^{p=+\infty} IP(X)\texttt{pdf}_{\hat{y}(X)}(p)dp \quad (3)$$

$$EIP(X) = \int_{p=-\infty}^{p=y(X^\star)} (y(X^\star) - p)\texttt{pdf}_{\hat{y}(X)}(p)dp \quad (4)$$

The challenge in Adaptive Sampling is to compute *EIP(X)* based on the $\langle X^{(i)}, y^{(i)}\rangle$ samples collected so far. The crux of this challenge is the generation of the probability density function of the estimated value of performance $y(X)$ at any setting X.

[3] The authors of SARD mentioned this problem [5]. They recommended that, before invoking SARD, the DBA should split each parameter x_i with nonmonotonic effect into distinct artificial parameters corresponding to each monotonic range of x_i. This task is nontrivial since the true surface is unknown to begin with. Ideally, the DBA, who may be a naive user, should not face this burden.

Adaptive Sampling: Algorithm run by iTuned's Planner

1. Initialization: Conduct experiments based on Latin Hypercube Sampling, and initialize GRS and $X^{\star}$=arg $\min_i y(X^{(i)})$ with collected samples;
2. Until the stopping condition is reached, do
3. Find $X_{next} = \arg \max_{X \in DOM} EIP(X)$;
4. Executor conducts the next experiment at X_{next} to get a new sample;
5. Update the GRS and $X^{\star}$ with the new sample; Go to Line 2;

Figure 2: Steps in iTuned's Adaptive Sampling algorithm

iTuned's Workflow: Figure 2 shows iTuned's workflow for parameter tuning. Once invoked, iTuned starts with an initialization phase where some experiments are conducted for bootstrapping. Adaptive Sampling starts with the initial set of samples, and continues to bring in new samples through experiments selected based on *EIP(X)*. Experiments are conducted seamlessly in the production environment using mechanisms provided by the executor.

Roadmap: Section 4 describes Adaptive Sampling in more detail. Details of the executor are presented in Section 5. iTuned's scalability-oriented features are described in Section 6.

4. ADAPTIVE SAMPLING

4.1 Initialization

As the name suggests, this phase bootstraps Adaptive Sampling by bringing in samples from an initial set of experiments. A straightforward technique is random sampling which will pick the initial experiments randomly from the space of possible experiments. However, random sampling is often ineffective when only a few samples are collected from a fairly high-dimensional space. More effective sampling techniques come from the family of *space-filling designs* [14]. iTuned uses one such sampling technique, called *Latin Hypercube Sampling (LHS)* [11], for initialization.

LHS collects m samples from a space of dimension d (i.e., parameters $x_1, \ldots, x_d$) as follows: (1) the domain $dom(x_i)$ of each parameter is partitioned into m equal subdomains; and (2) m samples are chosen from the space such that each subdomain of any parameter has one and only one sample in it. The set of "*" symbols in Figure 3 is an example of m=5 samples selected from a d=2 dimensional space by LHS. Notice that no two samples hit the same subdomain in any dimension.

LHS samples are very efficient to generate because of their similarity to *permutation matrices* from matrix theory. Generating m LHS samples involves generating d independent permutations of 1,. . .,m, and joining the permutations on a position-by-position basis. For example, the d=2 permutations {1,2,3,4,5} and {4,5,2,1,3} were combined to generate the m=5 LHS samples in Figure 3, namely, (1,4), (2,5), (3,2), (4,1), and (5,3).

However, LHS by itself does not rule out bad spreads (e.g., all samples spread along the diagonal). iTuned addresses this problem by generating multiple sets of LHS samples, and finally choosing the one that maximizes the minimum distance between any pair of samples. That is, suppose l different sets of LHS samples $L_1, \ldots, L_l$ were generated. iTuned will select the set $L^{\star}$ such that:

$$L^{\star} = \arg \max_{1 \le i \le l} \min_{X^{(j)}, X^{(k)} \in L_i, j \neq k} \texttt{dist}(X^{(j)}, X^{(k)})$$

Here, `dist` is a common distance metric like Euclidean distance. This technique avoids bad spreads.

4.2 Picking the Next Experiment

As discussed in Section 3 and Equation 4, iTuned has to compute the expected improvement *EIP(X)* that will come from doing the next experiment at any setting X. In turn, *EIP(X)* needs the probability density function $\texttt{pdf}_{\hat{y}(X)}(p)$ of the current estimate of performance $\hat{y}(X)$ at X. We use a *model-driven* approach—similar in spirit to [6]—to obtain the probability density function.

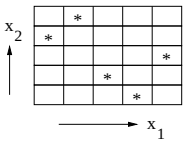

Figure 3: Example set of five LHS samples

The model used in iTuned is called the *Gaussian process Representation of a response Surface (GRS)*. GRS models $\hat{y}(X)$ as a Gaussian random variable whose mean $u(X)$ and variance $v^2(X)$ are determined based on the samples available so far. Starting from a conservative estimate based on the bootstrap samples, GRS improves the precision in estimating $y(X)$ as more experiments are done. In this paper, we will show the following attractive features of GRS:

- GRS is powerful enough to capture the response surfaces that arise in parameter tuning.
- GRS enables us to derive a closed form for *EIP(X)*.
- GRS enables iTuned to balance the conflicting tasks of *exploration* (understanding the surface) and *exploitation* (going after known high-performance regions) that arise in parameter tuning. It is nontrivial to achieve this balance, and many previous techniques [5, 18] lack it.

Definition 1. Gaussian process Representation of a response Surface (GRS): GRS models the estimated performance $\hat{y}(X)$, $X \in DOM$, as: $\hat{y}(X) = \vec{f}^{t}(X)\vec{\beta} + Z(X)$. Here, $\vec{f}^{t}(X)\vec{\beta}$ is a *regression model*. $Z(X)$ is a *Gaussian process* that captures the residual of the regression model. We describe each of these in turn. □

$\vec{f}(X) = [f_1(X), f_2(X), \ldots, f_h(X)]^t$ in the regression model $\vec{f}^{t}(X)\vec{\beta}$ is a vector of basis functions for regression [22]. $\vec{\beta}$ is the corresponding $h \times 1$ vector of regression coefficients. The t notation is used to represent the matrix transpose operation. For example, some response surface may be represented well by the regression model: $\hat{y} = 0.1 + 3x_1 - 2x_1x_2 + x_2^2$. In this case, $\vec{f}(X) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]^t$, and $\vec{\beta} = [0.1, 3, 0, -2, 0, 1]^t$. iTuned currently uses linear basis functions.

Definition 2. Gaussian process: $Z(X)$ is a Gaussian process if for any $l \geq 1$ and any choice of settings $X^{(1)}$,. . .,$X^{(l)}$, where each $X^{(i)} \in DOM$, the joint distribution of the l random variables $Z(X^{(1)}), \ldots, Z(X^{(l)})$ is a multivariate Gaussian. □

A multivariate Gaussian is a natural extension of the familiar unidimensional normal probability density function (the "bell curve") to a fixed number of random variables [6]. A Gaussian process is a generalization of the multivariate Gaussian to any arbitrary number $l \geq 1$ of random variables [14]. A Gaussian process is appropriate for iTuned since experiments are conducted at more and more settings over time.

A multivariate Gaussian of l variables is fully specified by a vector of l means and an $l \times l$ matrix of pairwise covariances [6]. As a natural extension, a Gaussian process $Z(X)$ is fully specified by a *mean function* and a pairwise *covariance function*. GRS uses a zero-mean Gaussian process, i.e., the mean value of any $Z(X^{(i)})$ is zero. The covariance function used is $\texttt{Cov}(Z(X^{(i)}), Z(X^{(j)})) = \alpha^2 corr(X^{(i)}, X^{(j)})$. Here, $corr$ is a pairwise correlation function defined as $corr(X^{(i)}, X^{(j)}) = \Pi_{k=1}^{d} \texttt{exp}(-\theta_k |x_k^{(i)} - x_k^{(j)}|^{\gamma_k})$. $\alpha, \theta_k \geq 0, \gamma_k > 0$, for $1 \leq k \leq d$, are constants.

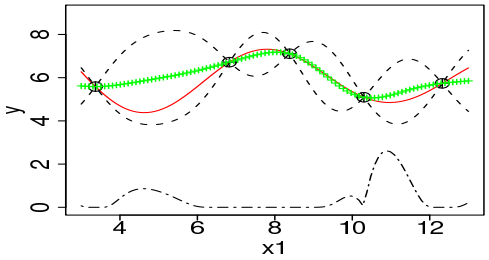

Figure 4: GRS from five samples (from Example 1)

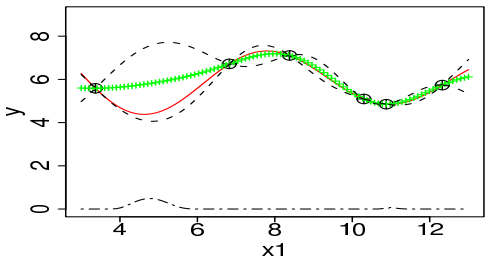

Figure 5: Example of EIP computation (from Example 2)

GRS's covariance function $\texttt{Cov}(Z(X^{(i)}), Z(X^{(j)}))$ represents the predominant phenomenon in response surfaces that if settings $X^{(i)}$ and $X^{(j)}$ are close to each other, then their respective residuals are correlated. As the distance between $X^{(i)}$ and $X^{(j)}$ increases, the correlation decreases. The parameter-specific constants θ_k and γ_k capture the fact that each parameter may have its own rate at which the residuals become uncorrelated. Section 4.3 describes how these constants are set.

Lemma 1. Probability density functions generated by GRS: Suppose the n samples $\langle X^{(i)}, y^{(i)}\rangle$, $1 \leq i \leq n$, have been collected through experiments so far. Given these n samples and a setting X, GRS models $\hat{y}(X)$ as a univariate Gaussian with mean $u(X)$ and variance $v^2(X)$ given by:

$$u(X) = \vec{f}^t(X)\vec{\beta} + \vec{c}^t(X)\mathbf{C}^{-1}(\vec{y} - \mathbf{F}\vec{\beta}) \quad (5)$$

$$v^2(X) = \alpha^2(1 - \vec{c}^t(X)\mathbf{C}^{-1}\vec{c}(X)) \quad (6)$$

$\vec{c}(X) = [corr(X, X^{(1)}), \ldots, corr(X, X^{(n)})]^t$, $\mathbf{C}$ is an $n \times n$ matrix with element i, j equal to $corr(X^{(i)}, X^{(j)})$, $1 \leq i, j \leq n$, $\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^t$, and $\mathbf{F}$ is an $n \times h$ matrix with the ith row composed of $\vec{f}^t(X^{(i)})$.

Proof: Given in the technical report [7]. □

The intuition behind Lemma 1 is that the joint distribution of the $n+1$ variables $\hat{y}(X^{(1)}), \ldots, \hat{y}(X^{(n)}), \hat{y}(X)$ is a multivariate Gaussian (follows from Definitions 1 and 2). Conditional distributions of a multivariate Gaussian are also Gaussian. Thus, the conditional distribution of $\hat{y}(X)$ given $\hat{y}(X^{(1)}), \ldots, \hat{y}(X^{(n)})$ is a univariate Gaussian with mean and variance as per Equations 5 and 6.

GRS will return $u(X)$ from Equation 5 if a single predicted value is asked for $\hat{y}(X)$ based on the n samples collected. Note that $\vec{f}^t(X)\vec{\beta}$ in Equation 5 is a plug in of X into the regression model from Definition 1. The second term in Equation 5 is an adjustment of the prediction based on the errors (residuals) seen at the sampled settings, i.e., $y^{(i)} - \vec{f}^t(X^{(i)})\vec{\beta}$, $1 \leq i \leq n$. Intuitively, the predicted value at setting X can be seen as the prediction from the regression model combined with a *correction term* computed as a weighted sum of the residuals at the sampled settings; where the weights are determined by the correlation function. Since the correlation function weighs nearby settings more than distant ones, the prediction at X is affected more by actual performance values observed at nearby settings.

Also note that the variance $v^2(X)$ at setting X—which is the *uncertainty* in GRS's predicted value at X—depends on the distance between X and the settings $X^{(i)}$ where experiments were done to collect samples. Intuitively, if X is close to one or more settings $X^{(i)}$ where we have collected samples, then we will have more confidence in the prediction than the case when X is far away from all settings where experiments were done. Thus, GRS captures the uncertainty in predicted values in an intuitive fashion.

Example 1. *The solid (red) line near the top of Figure 4 is a true one-dimensional response surface. Suppose five experiments are done, and the collected samples are shown as circles in Figure 4. iTuned generates a GRS from these samples. The (green) line marked with "+" symbols represents the predicted values $u(X)$ generated by the GRS as per Lemma 1. The two (black) dotted lines around this line denote the 95% confidence interval, namely, $(u(X) - 2v(X), u(X) + 2v(X))$. For example, at $x_1 = 8$, the predicted value is 7.2 with confidence interval (6.4, 7.9). Note that, at all points, the true value (solid line) is within the confidence interval; meaning that the GRS generated from the five samples is a good approximation of the true response surface. Also, note that at points close to the collected samples, the uncertainty in prediction is low. The uncertainty increases as we move further away from the collected samples.* □

Lemma 1 gives the necessary building block to compute expected improvements from experiments that have not been done yet. Recall from Lemma 1 that, based on the collected samples $\langle X^{(i)}, y^{(i)}\rangle$, $1 \leq i \leq n$, $\hat{y}(X)$ is a Gaussian with mean $u(X)$ and variance $v^2(X)$. Hence the probability density function of $\hat{y}(X)$ is:

$$\texttt{pdf}_{\hat{y}(X)}(p) = \frac{1}{\sqrt{2\pi}v(X)} \exp\left(\frac{-(p - u(X))^2}{2v^2(X)}\right) \quad (7)$$

Theorem 1. Closed form for *EIP(X)*: The expected improvement from conducting an experiment at setting X has the following closed form:

$$EIP(X) = v(X)(\mu(X)\Phi(\mu(X)) + \phi(\mu(X))) \quad (8)$$

Here, $\mu(X) = \frac{y(X^\star) - u(X)}{v(X)}$. Φ and ϕ are $N(0, 1)$ Gaussian cumulative distribution and density functions respectively.

Proof: Given in the technical report [7]. □

Intuitively, the next experiment to run should be picked either from regions where uncertainty is large, which is captured by $v(X)$ in Equation 8, or where the predicted performance values can improve over the current best setting, which is captured by $\mu(X)$ in Equation 8. In regions where the current GRS from the observed samples is uncertain about its prediction, i.e., where $v(X)$ is high, exploration is preferred to reduce the model's uncertainty. At the same time, in regions where the current GRS predicts that performance is good, i.e., $\mu(X)\Phi(\mu(X)) + \phi(\mu(X))$ is high, exploitation is preferred to potentially improve the current best setting $X^\star$. Thus, Equation 8 captures the tradeoff between exploration (global search) and exploitation (local search).

Example 2. *The dotted line at the bottom of Figure 4 shows* EIP*(X) along the x_1 dimension. (All* EIP *values have been scaled uniformly to make the plot fit in this figure.) There are two peaks in the* EIP *plot. (I)* EIP *values are high around the current best sample (X^* with x_1=10.3), encouraging local search (exploitation) in this region. (II)* EIP *values are also high in the region between x_1=4 and x_1=6 because no samples have been collected near this region; the higher uncertainty motivates exploring this region. Adaptive Sampling with conduct the next experiment at the highest* EIP *point, namely, x_1=10.9. Figure 5 shows the new set of samples as well as*

the new EIP*(X) after the GRS is updated with the new sample. As expected, the* EIP *around* x_1*=10.9 has reduced.* EIP*(X) now has a maximum value at* x_1*=4.7 because the uncertainty in this region is still high. Adaptive Sampling will experiment here next, bringing in a sample close to the global optimum at* x_1*=4.4.*

4.3 Overall Algorithm and Implementation

Figure 2 shows the overall structure of iTuned's Adaptive Sampling algorithm. So far we described how the initialization is done and how *EIP(X)* is derived. We now discuss how iTuned implements the other steps in Figure 2.

Finding the setting that maximizes EIP: Line 3 in Figure 2 requires us to find the setting $X \in DOM$ that has the maximum *EIP*. Since we have a closed form for *EIP*, it is efficient to evaluate *EIP* at a given setting X. In our implementation, we pick $k = 1000$ settings (using LHS sampling) from the space of feasible settings, compute their *EIP* values, and pick the one that has the maximum value to run the next experiment.

Initializing the GRS and updating it with new samples: Initializing the GRS with a set of $\langle X^{(i)}, y^{(i)} \rangle$ samples, or updating the GRS with a newly collected sample, involves deriving the best values of the constants α, θ_k, and γ_k, for $1 \leq k \leq d$, based on the current samples. This step can be implemented in different ways. Our current implementation uses the well-known and efficient statistical technique of *maximum likelihood estimation* [9, 22].

When to stop: Adaptive Sampling can stop (Line 2 in Figure 2) under one of two conditions: (i) when the user issues an explicit stop command once she is satisfied with the performance improvement achieved so far; and (ii) when the maximum expected improvement over all settings $X \in DOM$ falls below a threshold.

5. ITUNED'S EXECUTOR: A PLATFORM FOR RUNNING ONLINE EXPERIMENTS

We now consider where and when iTuned will run experiments. There are some simple answers. If parameter tuning is done before the database goes into production use, then the experiments can be done on the production platform itself. If the database is already in production use and serving real users and applications, then experiments could be done on an offline test platform. Previous work on parameter tuning (e.g., [5, 18]) assumes that experiments are conducted in one of these settings.

While the two settings above—preproduction database and test database—are practical solutions, they are not sufficient because:

- The workload may change while the database is in production use, necessitating retuning.
- A test database platform may not exist (e.g., in an SMB).
- It can be nontrivial or downright infeasible to replicate the production resources, data, and workload on the test platform.

iTuned's executor provides a comprehensive solution that addresses concerns like these. The guiding principle behind the solution is: exploit underutilized resources in the production environment for experiments, but never harm the production workload. The two salient features of the solution are:

- **Designated resources:** iTuned provides an interface for users to designate *which* resources can be used for running experiments. Candidate resources include (i) the production database (the default for running experiments), (ii) standby databases backing up the production database, (iii) test database(s) used by DBAs and developers, and (iv) staging database(s) used for end-to-end testing of changes (e.g., bug fixes) before they are applied to the production database. Resources designated for experiments are collectively called the *workbench*.

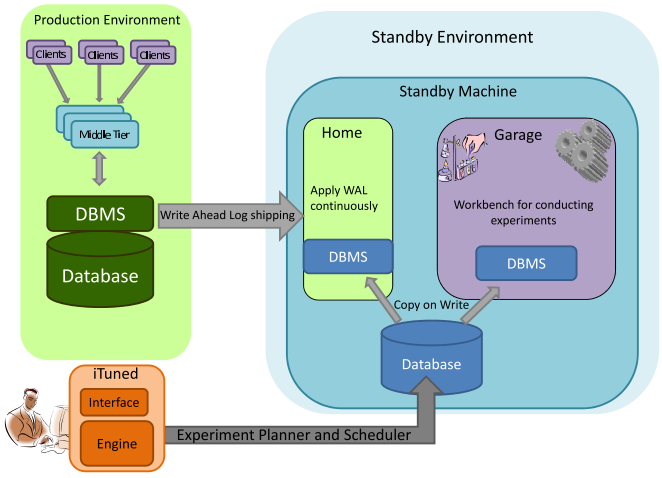

Figure 6: The executor in action for standby databases

- **Policies:** A policy is specified with each resource that dictates *when* the resource can be used for experiments. The default policy associated with each of the above resources is: "if the CPU, memory, and disk utilization of the resource for its *home use* is below 10% (threshold t_1) for the past 10 minutes (threshold t_2), then the resource can be used for experiments." Home use denotes the regular (i.e., nonexperimental) use of the resource. The two thresholds are customizable. Only the default policy is implemented currently, but we are exploring other policies.

iTuned's implementation consists of: (i) a front-end that interacts with users, and (ii) a back-end comprising the planner, which plans experiments using Adaptive Sampling, and the executor, which runs planned experiments on the workbench as per user-specified (or default) policies. Monitoring data needed to enforce policies is obtained through system monitoring tools.

The design of the workbench is based on splitting the functionality of each resource into two: (i) *home use*, where the resource is used directly or indirectly to support the production workload, and (ii) *garage use*, where the resource is used to run experiments. We will describe the home/garage design using the standby database as an example, and then generalize to other resources.

All database systems support one or more hot standby databases whose home use is to keep up to date with the (primary) production database by applying redo logs shipped from the primary. If the primary fails, a standby will quickly take over as the new primary. Hence, the standby databases run the same hardware and software as the production database. It has been observed that standby databases usually have very low utilization since they only have to apply redo log records. In fact, [8] mentions that enterprises that have 99.999% (five nines) availability typically have standby databases that are idle 99.999% of the time.

Thus, the standby databases are a valuable and underutilized asset that can be used for online experiments without impacting user-facing queries. However, their home use should not be affected, i.e., the recovery time on failure should not have any noticeable increase. iTuned achieves this property using two *resource containers*: the home container for home use, and the garage container for running experiments. iTuned's current implementation of resource containers uses the *zones* feature in the Solaris OS [15]. CPU, memory, and disk resources can be allocated dynamically to a zone, and the OS provides isolation between resources allocated to different zones. Resource containers can also be implemented using virtual machine technology which is becoming popular [16].

The home container on the standby machine is responsible for

Feature	Description and Use
Sensitivity analysis	Identify and eliminate low-effect parameters
Parallel experiments	Use multiple resources to run expts in parallel
Early abort	Identify and stop low-utility expts quickly
Workload compression	Reduce per-experiment running time without reducing overall tuning quality

Table 1: Features that improve iTuned's efficiency

applying the redo log records. When the standby machine is not running experiments, the home container runs on it using all available resources; the garage lies idle. The garage container is *booted*—similar to a machine booting, but much faster—only when a policy fires and allows experiments to be scheduled on the standby machine. During an experiment, both the home and the garage containers will be active, with a partitioning of resources as determined by the executor. Figure 6 provides an illustration. For example, as per the default policy stated earlier, home and garage will get 10% and 90%, respectively, of the resources on the machine.

Both the home and the garage containers run a full and exactly the same copy of the database software. However, on booting, the garage is given a *snapshot* of the current data (including physical design) in the database. The garage's snapshot is logically separate from the snapshot used by the home container, but it is physically the same except for *copy-on-write* semantics. Thus, both home and garage have logically-separate copies of the data, but only a single physical copy of the data exists on the standby system when the garage boots. When either container makes an update to the data, a separate copy of the changed part is made that is visible to the updating container only (hence the term copy-on-write). The redos applied by the home container do not affect the garage's snapshot. iTuned's implementation of snapshots and copy-on-write semantics leverages the Zettabyte File System [15], and is extremely efficient (as we will show in the empirical evaluation).

The garage is *halted* immediately under three conditions: when experiments are completed or the primary fails or there is a policy violation. All resources are then released to the home container which will continue functioning as a pure standby or take over as the primary as needed. Setting up the garage (including snapshots and resource allocation) takes less than a minute, and tear-down takes even less time. The whole process is so efficient that recovery time is not increased by more than a few seconds.

While the above description focused on the standby resource, iTuned applies the same home/garage design to all other resources in the workbench (including the production database). The only difference is that each resource has its own distinct type of home use which is encapsulated cleanly into the corresponding home container. *Thus, iTuned works even in settings where there are no standby or test databases.*

6. IMPROVING ITUNED'S EFFICIENCY

Experiments take time to run. This section describes features that can reduce the time iTuned takes to return good results as well as make iTuned scale to large numbers of parameters. Table 1 gives a short summary. The first three features in Table 1 are fully integrated into iTuned, while workload compression is currently a simple standalone tool.

6.1 Eliminating Unimportant Parameters Using Sensitivity Analysis

Suppose we have generated a GRS using n samples $\langle X^{(i)}, y^{(i)} \rangle$. Using the GRS, we can compute $E(y|x_1{=}v)$, the expected value of y when $x_1{=}v$ as:

$$E(y|x_1{=}v) = \frac{\int_{dom(x_2)} \ldots \int_{dom(x_d)} \hat{y}(v, x_2, \ldots, x_d) dx_2 \cdots dx_d}{\int_{dom(x_2)} \ldots \int_{dom(x_d)} dx_2 \cdots dx_d} \quad (9)$$

Equation 9 averages out the effects of all parameters other than x_1. If we consider l equally-spaced values $v_i \in dom(x_1)$, $1 \leq i \leq l$, then we can use Equation 9 to compute the expected value of y at each of these l points. A plot of these values, e.g., as shown in Figure 4, gives a visual feel of the overall effect of parameter x_1 on y. We term such plots *effect plots*. In addition, we can consider the variance of these values, denoted $V_1 = \texttt{Var}(E(y|x_1))$. If V_1 is low, then y does not vary much as x_1 is changed; hence, the effect of x_1 on y is low. On the other hand, a large value of V_1 means that y is sensitive to x_1's setting.

Therefore, we define the *main effect* of x_1 as $\frac{V_1}{\texttt{Var}(y)}$ which represents the fraction of the overall variance in y that is explained by the variance seen in $E(y|x_1)$. The main effects of the other parameters $x_2, \ldots, x_d$ are defined in a similar fashion. Any parameter with low main effect can be set to its default value with little negative impact on performance, and need not be considered for tuning.

6.2 Running Multiple Experiments in Parallel

If the executor can find enough resources on the workbench, then iTuned can run $k > 1$ experiments in parallel. The batch of experiments from LHS during initialization can be run in parallel. Running k experiments from Adaptive Sampling in parallel is non-trivial because of its sequential nature. A naive approach is to pick the top k settings that maximize EIP. However, the pitfall is that these k settings may be from the same region (around the current minimum or with high uncertainty), and hence redundant.

We set two criteria for selecting k parallel experiments: (I) Each experiment should improve the current best value (in expectation); (II) The selected experiments should complement each other in improving the GRS's quality (i.e., in reducing uncertainty). iTuned determines the next k experiments to run in parallel as follows:

1. Select the experiment $X^{(i)}$ that maximizes the current EIP.
2. An important feature of GRS is that the uncertainty in prediction (Equation 6) depends only on the X values of collected samples. Thus, after $X^{(i)}$ is selected, we update the uncertainty estimate at each remaining candidate setting. (The predicted value, from Equation 5, at each candidate remains unchanged.)
3. We compute the new EIP values with the updated uncertainty term $v(X)$, and pick the next sample $X^{(i+1)}$ that maximizes EIP. The nice property is that $X^{(i+1)}$ will not be clustered with $X^{(i)}$: after $X^{(i)}$ is picked, the uncertainty in the region around $X^{(i)}$ will reduce, therefore EIP will decrease in that region.
4. The above steps are repeated until k experiments are selected.

6.3 Early Abort of Low-Utility Experiments

While the exploration aspect of Adaptive Sampling has its advantages, it can cause experiments to be run at poorly-performing settings. Such experiments take a long time to run, and contribute little towards finding good parameter settings. To address this problem, we added a feature to iTuned where an experiment at $X^{(i)}$ is aborted after $\Delta \times t_{min}$ time if the workload running time at $X^{(i)}$ is greater than $\Delta \times t_{min}$. Here, t_{min} is the workload running time at the best setting found so far. By default, $\Delta = 2$.

6.4 Workload Compression

Work on physical design tuning has shown that there is a lot of redundancy in real workloads which can be exploited through workload compression to give 1-2 orders of magnitude reduction in

tuning time [3]. The workload compression technique from [3] first partitions the given workload based on distinct query templates, and then picks a representative subset per partition via clustering. To demonstrate the utility of workload compression in iTuned, we came up with a modified approach. We treat a workload as a series of executions of query mixes, where a query mix is a set of queries that run concurrently. An example could be $\langle 3Q_1, 6Q_{18} \rangle$ which denotes three instances of TPC-H query Q_1 running concurrently with six instances of Q_{18}. We partition the given workload into distinct query mixes, and pick the top-k mixes based on the overall time for which each mix ran in the workload.

7. EMPIRICAL EVALUATION

Our evaluation setup involves a local cluster of machines, each with four 2GHz processors and 3GB memory, running PostgreSQL 8.2 on Solaris 10. One machine runs the production database while the other machines are used as hot standbys, test platforms, or workload generators as needed. Recall from Section 5 that iTuned's policy-based executor can conduct experiments on the production database, standbys, and test platforms. By default, we use one standby database for experiments. Our implementation of GPR uses the *tgp* package [9].

7.1 Methodology and Result Summary

We first summarize the different types of empirical evaluation conducted and the results obtained.

- Section 7.2 breaks down the overhead of various operations in the API provided by iTuned's executor, and shows that the executor is noninvasive and efficient.
- Section 7.3 shows real response surfaces that highlight the issues motivating our work, e.g., (i) why database parameter tuning is not easy for the average user; (ii) how parameter effects are highly sensitive to workloads, data properties, and resource allocations; and (iii) why optimizer cost models are insufficient for effective parameter tuning, but it is important to keep the optimizer in the tuning loop.
- Section 7.4 presents tuning results for OLAP and OLTP workloads of increasing complexity that show iTuned's ease of use and up to 10x improvements in performance compared to default parameter settings, rule-based tuning based on popular heuristics, and a state-of-the-art automated parameter tuning technique. We show how iTuned can leverage parallelism, early aborts, and workload compression to cut down tuning times drastically with negligible degradation in tuning quality.
- iTuned's performance is consistently good with both PostgreSQL and MySQL databases, demonstrating iTuned's portability.
- Section 7.5 shows how iTuned can be useful in other ways apart from recommending good parameter settings, namely, visualizing parameter impact as well as approximate response surfaces. This information can guide further manual tuning.

Tuning tasks in our evaluation consider up to 30 configuration parameters. By default, we consider the following 11 PostgreSQL parameters for OLAP workloads: (P1) shared_buffers, (P2) effective_cache_size, (P3) work_mem, (P4) maintenance_work_mem, (P5) default_statistics_target, (P6) random_page_cost, (P7) cpu_tuple_cost, (P8) cpu_index_tuple_cost, (P9) cpu_operator_cost, (P10) memory allocation, and (P11) CPU allocation. Descriptions of parameters P1-P9 can be found in [13]. Parameters P10 and P11 respectively control the memory and CPU allocation to the database.

7.2 Performance of iTuned's Executor

We first analyze the overhead of the executor for running experiments. Recall its implementation from Section 5. Table 2 shows the various operations in the interface provided by the executor, and the overhead of each operation. The Create Container operation is done once to set up the OS environment for a particular tuning task; so its 10-minute cost is amortized over an entire tuning session. This overhead can be cut down to 17 seconds if the required type of container has already been created for some previous tuning task. Note that all the other operations take on the order of a few seconds. For starting a new experiment, the cost is at most 48 seconds to boot the container and to create a read-write snapshot of the database (for workloads with updates). A container can be halted within 2 seconds, which adds no noticeable overhead if, say, the standby has to take over on a failure of the primary database.

Operation by Executor	Time (sec)	Description
Create Container	610	Create a new garage (one time process)
Clone Container	17	Clone a garage from already existing one
Boot Container	19	Boot garage from halt state
Halt Container	2	Stop garage and release resources
Reboot Container	2	Reboot the garage (required for adding additional resources to a container)
Snapshot-R DB	7	Create read-only snapshot of database
Snapshot-RW DB	29	Create read-write snapshot of database

Table 2: Overheads of operations in iTuned's executor

7.3 Why Parameter Tuning is Nontrivial

The OLAP (Business Intelligence) workloads used in our evaluation were derived from TPC-H running at scale factors (*SF*) of 1 and 10 on PostgreSQL [19]. The physical design of the databases are well tuned, with indexes approximately tripling and doubling the database sizes for SF=1 and SF=10 respectively. Statistics are always up to date. The heavyweight TPC-H queries in our setting include Q1, Q7, Q9, Q13, and Q18.

Figure 1 shows a 2D projection of a response surface that we generated by running Q18 on a TPC-H SF=1 database for a number of different settings of the eleven parameters from Section 7.1. The database size with indexes is around 4GB. The physical memory (RAM) given to the database is 1GB to create a realistic scenario where the database is 4x the amount of RAM. This complex response surface is the net effect of a number of individual effects:

- Q18 (Large Volume Customer Query) is a complex query that joins the Lineitem, Customer, and Order tables. It also has a subquery over Lineitem (which gets rewritten as a join), so Q18 accesses Lineitem—the biggest table in TPC-H—twice.
- Different execution plans get picked for Q18 in different regions of the response surface because changes in parameter settings lead to changes in estimated plan costs. These plans differ in operators used, join order, and whether the same or different access paths are used for the two accesses to the Lineitem table.
- Operator behavior can change as we move through the surface. For example, hash joins in PostgreSQL change from one pass to two passes if the *work_mem* parameter is lower than the memory required for the hash join's build phase.
- The most significant effect comes from hash joins present in some of the plans. Hash partitions that spill to disk are written directly to temporary disk files in PostgreSQL; not to temporary buffers in the database or to *shared_buffers*. As *shared_buffers* is increased, memory for the OS file cache (which buffers reads and writes to disk files) decreases. Thus, disk I/O to the spilled partitions increases, causing performance degradation.

Surfaces like Figure 1 show how critical experiments are to understand which of many different effects dominate in a particular setting. It took us several days of effort, more than a hundred experiments, as well as fine-grained monitoring using *DTrace* [15] to understand the unexpected nature of Figure 1. It is unlikely that a

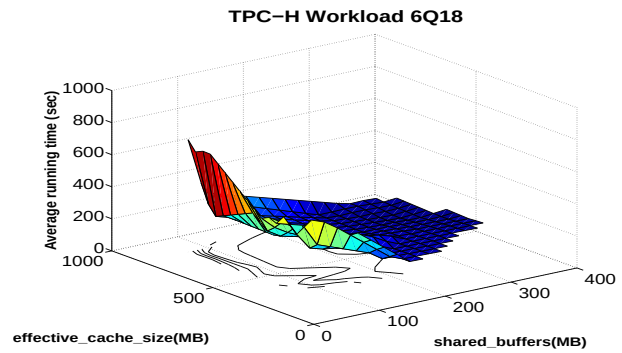

Figure 7: Impact of shared_buffers Vs. effective_cache_size for workload W4 (TPC-H SF=10)

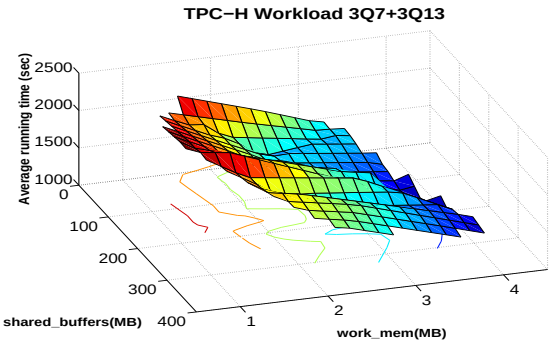

Figure 8: Impact of shared_buffers Vs. work_mem for workload W5 (TPC-H SF=10)

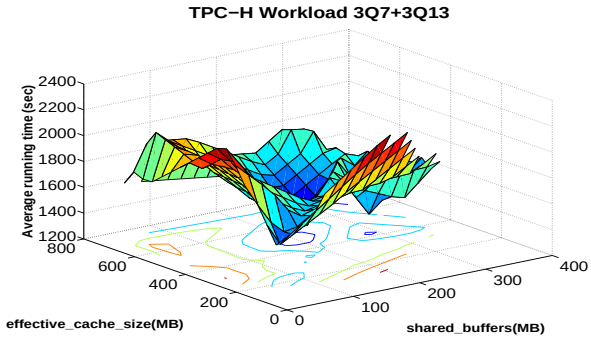

Figure 9: Impact of shared_buffers Vs. effective_cache_size for workload W5 (TPC-H SF=10)

non-expert who wants to use a database for some application will have the knowledge (or patience) to tune the database like we did.

The average running time of a query can change drastically depending on whether it is running alone in the database or it is running in a concurrent mix of queries of the same or different types. For example, consider Q18 running alone or in a mix of six concurrent instances of Q18 (each instance has distinct parameter values). At the default parameter setting of PostgreSQL for TPC-H SF=1, we have observed the average running time of Q18 to change from 46 seconds (when running alone) to 1443 seconds (when running in the mix). For TPC-H SF=10, there was a change from 158 seconds (when running alone) to 578 seconds (when running in the mix).

Two insights come out from the results presented so far. First, query optimizers compute the cost of a plan independent of other plans running concurrently. Thus, optimizer cost models cannot capture the true performance of real workloads which consist of query mixes. Second, it is important to keep the optimizer in the loop while tuning parameter settings because the optimizer can change the plan for a query when we change parameter settings. While keeping the optimizer in the loop is accepted practice for physical design tuning (e.g., [4]), to our knowledge, we are the first to bring out its importance and enable its use in configuration parameter tuning.

Figure 7 shows a 2D projection of the response surface for Q18 when run in the 6-way mix in PostgreSQL for TPC-H SF=10. The key difference between Figures 1 (Q18 alone, TPC-H SF=1) and 7 (Q18 in 6-way mix, TPC-H SF=10) is that increasing *shared_buffers* has an overall negative effect in the former case, while the overall effect is positive in the latter. We attribute the marked effect of *shared_buffers* in Figure 7 to the increased cache hits across concurrent Q18 instances. Figures 8 and 9 show the response surface for a workload where *shared_buffers* has limited impact. The highest impact parameter is *work_mem*. This workload has three instances of Q7 and 3 instances of Q13 running in a 6-way mix in PostgreSQL for TPC-H SF=10. All these results show why users can have a hard time setting database parameters, and why experiments that can bring out the underlying response surfaces are inevitable.

7.4 Tuning Results

We now present an evaluation of iTuned's effectiveness on different workloads and environments. iTuned should be judged both on its *quality*—how good are the recommended parameter settings?—and *efficiency*—how soon can iTuned generate good recommendations? Our evaluation compares iTuned against:

- **Default** parameter settings that come with the database.
- **Manual rule-based** tuning based on heuristics from database administrators and performance tuning experts. We use an authoritative source for PostgreSQL tuning [13].
- **Smart Hill Climbing (SHC)** is a state-of-art automated parameter tuning technique [23]. It belongs to the hill-climbing family of optimization techniques for complex response surfaces. Like iTuned, SHC plans experiments while balancing exploration and exploitation (Section 4.2). But, SHC lacks key features of iTuned like GRS representation of response surfaces, executor, and efficiency-oriented features like parallelism, early aborts, sensitivity analysis, and workload compression.
- **Approximation to the optimal setting:** Since we do not know the optimal performance in any tuning scenario, we run a large number of experiments offline for each tuning task. We have done at least 100 (often 1000+) experiments per tuning task over the course of six months. The best performance found is used as an approximation of the optimal. This technique is labeled *Brute Force*.

iTuned and SHC do 20 experiments each by default. iTuned uses the first 10 experiments for initialization. Strictly for the purposes of evaluation, by default iTuned uses only early abort among the efficiency-oriented techniques from Section 6.

Figure 10 compares the tuning quality of iTuned (I) with Default (D), manual rule-based (M), SHC (S), and Brute Force (B) on a range of TPC-H workloads at SF=1 and SF=10. The performance metric of interest is workload running time; lower is better. The workload running time for D is always shown as 100%, and the times for others are relative. To further judge tuning quality, these figures show the rank of the performance value that each technique finds. Ranks are reported with the prefix R, and are based on a best-to-worst ordering of the performance values observed by Brute Force; lower rank is always better. Figure 10 also shows (above I's bar) the total time that iTuned took since invocation to give the recommended setting. Detailed analysis of tuning times is done later in this section.

11 distinct workloads are used in Figure 10, all of which are nontrivial to tune. Workloads W1, W2, and W3 consist of indi-

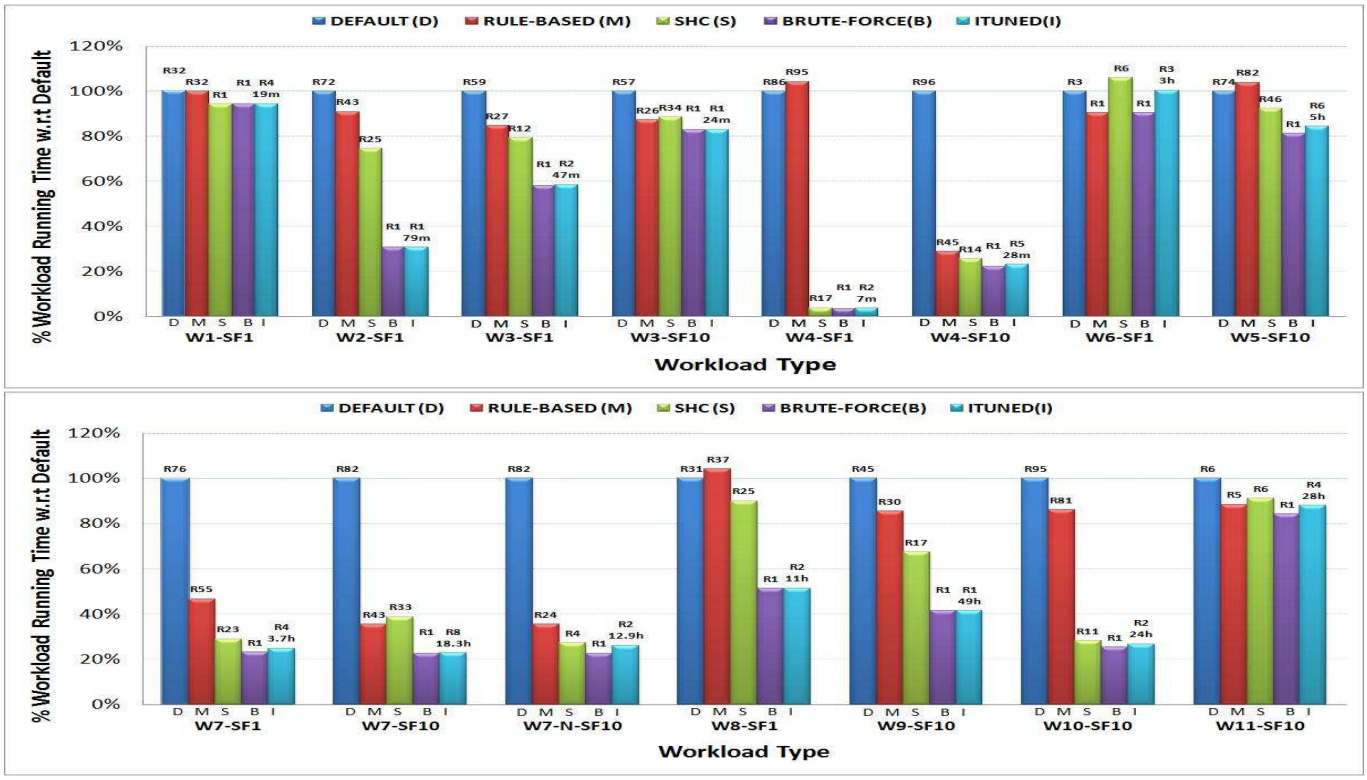

Figure 10: Comparison of tuning quality. iTuned's tuning times are shown in minutes (m) or hours (h). Ri denotes Rank i

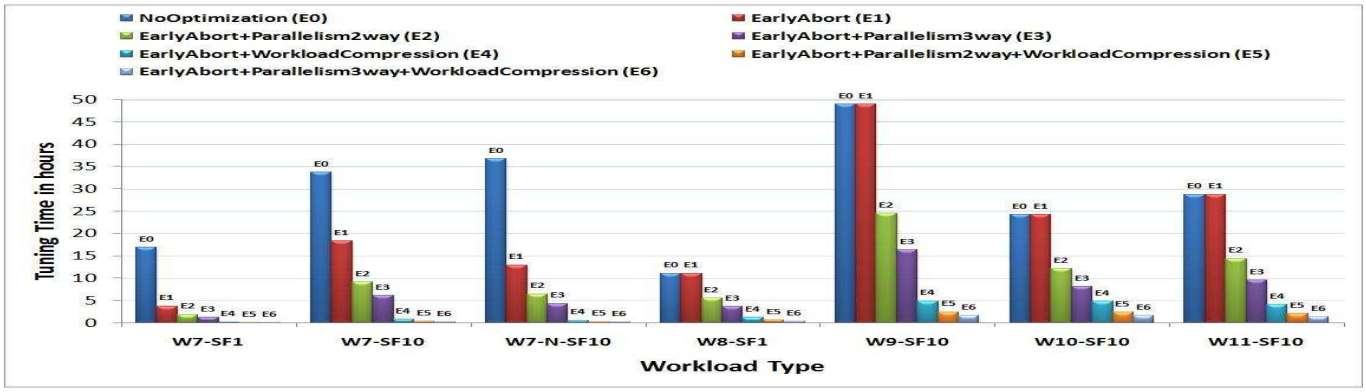

Figure 11: Comparison of iTuned's tuning times in the presence of various efficiency-oriented features

vidual TPC-H queries Q1, Q9, and Q18 respectively running at a *Multi-Programming Level (MPL)* of 1. MPL is the maximum number of concurrent queries. TPC-H queries have input parameters. Throughout our evaluation, we generate each query instance randomly using the TPC-H query generator *qgen*. Different instances of the same query are distinct with high probability.

Workloads W4, W5, and W6 go one step higher in tuning complexity because they consist of mixes of concurrent queries. W4 (MPL=6) consists of six concurrent (and distinct) instances of Q18. W5 (MPL=6) consists of three concurrent instances of Q7 and three concurrent instances of Q13. W6 (MPL=10) consists of five concurrent instances of Q5 and five concurrent instances of Q9.

Workloads W7 and higher in Figure 10 go the final step in tuning complexity by bringing in many more complex query types, much larger numbers of query instances, and different MPLs. W7 (MPL=9) contains 200 query instances comprising queries Q1 and Q18, in the ratio 1:2. W8 (MPL=24) contains 200 query instances comprising TPC-H queries Q2, Q3, Q4, and Q5, in the ratio 3:1:1:1. W9 (MPL=10), W10 (MPL=20), and W11 (MPL=5) contain 100 query instances each with 10, 10, and 15 distinct TPC-H query types respectively in equal ratios. The results for W7-N shown in Figure 10 are from tuning 30 parameters.

Figure 10 shows that the parameter settings recommended by iTuned consistently outperform the default settings, and is usually significantly better than the settings found by SHC and common tuning rules. iTuned gives 2x-5x improvement in performance in many cases. iTuned's recommendation is usually close in performance to the approximate optimal setting found (exhaustively) by Brute Force. It is interesting to note that expert tuning rules are more geared towards complex workloads (compare the M bars between the top and bottom halves of Figure 10).

As an example, consider the workload W7-SF10 in Figure 10. The default settings give a workload running time of 1085 seconds. Settings based on tuning rules and SHC give running times of 386 and 421 seconds respectively. In comparison, iTuned's best setting after initialization gave a performance of 318 seconds, which

Workload	Perf. Metric	#Params	Quality (Rank)	Tuning time (Hours)
TPC-W (MySQL)	Response time	7	R1	3.2
TPC-W (MySQL)	Throughput	7	R4	7.6
TPC-W (PostgreSQL)	Response time	20	R23	2.5
TPC-W (PostgreSQL)	Throughput	20	R8	2.5
RUBiS (MySQL)	Response time	6	R1	6.1
RUBiS (MySQL)	Throughput	6	R2	6.6

Table 3: Sample of iTuned's results for OLTP workloads

was improved to 246 seconds by Adaptive Sampling (77% improvement over default). iTuned's sensitivity analysis found the *shared_buffers* parameter to have the most impact on performance. The default setting of 32 MB for *shared_buffers* is poor. The rule-based setting of 200 MB is better, but iTuned found a setting close to 400 MB where the performance is far better.

Figure 10 shows that iTuned takes on the order of tens of hours to find good settings for complex workloads. Configuring large database management systems takes on the order of one to two weeks [12]. Thus, one to two days of time spent parameter tuning is acceptable, especially considering that iTuned gives 2x-5x improvement in performance in many cases. More importantly, Figure 11 shows that iTuned's tuning time can be reduced by orders of magnitude using the techniques we proposed in Section 6. Early Abort uses $\Delta = 2$ and workload compression picks the top mix in the workload.

For each of the complex workloads from Figure 10, we show iTuned's tuning time with and without different techniques. It is clear that these techniques can reduce iTuned's tuning time to at most a few hours. The drop in tuning quality across all these scenarios was never more than 1%. In general, we have found workload compression to be even more effective in parameter tuning than in physical design tuning. We speculate that parameter settings are less sensitive to which queries get picked in the compressed workload compared to, say, index selection.

Because of space constraints, we have only given representative results in this paper. Table 3 gives a brief summary of other empirical results—including results for OLTP workloads, MySQL, and different performance metrics—that show iTuned's consistent good performance. TPC-W is an e-Commerce benchmark that simulates the activities of a retail website. Our experiments with TPC-W are based on a 48000-transaction workload on a 6GB database. RUBiS [1] is Web service benchmark that implements the core functionality of an auction site like eBay.

7.5 Sensitivity Analysis

This section evaluates two important features of iTuned: sensitivity analysis of database parameters and effect plots for visualization (recall Section 6.1). We use both real workloads and complex synthetic response surfaces in our evaluation. We compare iTuned's performance against SARD [5] which is described in Section 3. Recall that, unlike iTuned, SARD is not an end-to-end tuning tool, and can be misled by nonmonotonic effects of parameters.

Our concerns about SARD were validated by a simple evaluation. We chose three popular and hard benchmark functions from the optimization literature: Griewank, Rastrigin, and Rosenbrock [23]. All three functions have a global optimum of 0. We used the functions to generate response surfaces with 20 parameters each. Of these 20 parameters, 5 are important—i.e., they impact the shape of the surface significantly—while the remaining 15 are unimportant. On the Griewank and Rastrigin surfaces—which have significant nonmonotonic behavior—SARD completely failed to identify the unimportant parameters. As iTuned did experiments progressively, it never classified any important parameter as unimportant. By the time fifty experiments were done, iTuned was able to clearly separate the five important parameters from the unimportant ones.

Workload	Optimal	SARD+AS	SHC	iTuned
Griewank	0	28.6	28.7	2.0
Rastrigin	0	200.8	209.1	26.1
Rosenbrock	0	40.2	160.5	7.9
W2-SF1	95	240 (R29)	231 (R24)	95 (R1)
W3-SF1	11	43 (R20)	67 (R24)	12 (R4)
W6-SF1	390	450 (R63)	417 (R20)	403 (R5)
W8-SF1	208	208 (R1)	289 (R4)	208 (R1)

Table 4: Sensitivity analysis. For W2, W3, W6, W8, rank and performance of best setting (secs) are shown. Lower is better

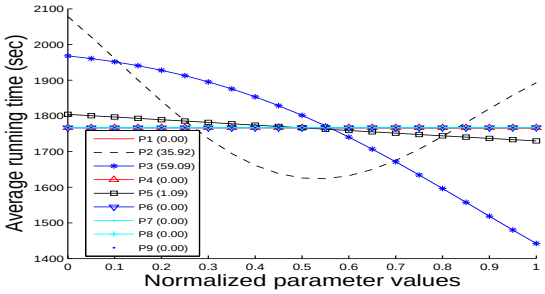

Figure 12: Effect plot for workload W5 (TPC-H SF=10)

Tables 4 gives end-to-end tuning results for three techniques: (i) SARD+AS, where SARD is used to identify the important parameters, and then Adaptive Sampling is started with the samples collected by SARD used for initialization; (ii) SHC (does not do sensitivity analysis), and (iii) iTuned. Note that lower numbers are better in all cases. iTuned clearly outperforms the alternatives.

A very useful feature of iTuned is that it can provide intuitive visualizations of its current results. Figure 12 shows an effect plot (recall Section 6.1) generated by iTuned based on 10 experiments for the workload whose surface is shown in Figures 8 and 9. The parameters P1-P9 correspond to the first nine PostgreSQL parameters listed in Section 7.1. Without knowing the actual response surface, a user can quickly grasp the main trends in parameter impact based on the effect plot. Note how the plot mirrors the trends in Figures 8 and 9.

In summary, as few as twenty experiments chosen smartly by iTuned can produce a wealth of information in a reasonable amount of time to aid both naive users and expert DBAs in tuning database configuration parameters.

8. RELATED WORK

Databases have fairly mature tools for physical design tuning (e.g., index selection [4]). However, these tools do not address configuration parameter tuning. Furthermore, these tools depend on the query optimizer's cost models which do not capture the effects of many important parameters.

Surprisingly, very little work has been done on tools for holistic tuning of the many configuration parameters in modern database systems. Most work in this area has either focused on specific classes of parameters (e.g., [17]) or on restricted subproblems of the overall parameter tuning problem (e.g., [5]). IBM DB2 provides an advisor for setting default values for a large number of parameters [12]. DB2's advisor does not generate response surfaces, instead it relies on built-in models of how various parameters affect performance [5]. As we show in this paper, predetermined mod-

els may not be accurate in a given setting. SARD (discussed in Section 3) and [18] are also related to iTuned. SARD focuses on ranking parameters in order of impact, and is not an end-to-end tuning tool. Techniques to learn a probabilistic model using samples generated from gridding were proposed in [18]. The model was then applied to tune four parameters in Berkeley DB. Gridding becomes very inefficient as the number of parameters increases. Section 7 also compared iTuned with a technique based on hill climbing (e.g., [23]) that has been used for parameter tuning. None of the above techniques have an equivalent of iTuned's executor or the efficiency-oriented features from Section 6.

Techniques for tuning specific classes of parameters include solving analytical models [20], using simulations of database performance (e.g., in Oracle database), and control-theoretic approaches for online tuning [17]. These techniques are all based on predefined models of how changes in parameter settings affect performance. Techniques to tune the CPU and memory allocations to databases running inside virtual machines were proposed in [16]. However, the focus was not on planning experiments to learn the underlying response surfaces. All the above techniques can benefit from Adaptive Sampling and iTuned's executor.

Traditional database sampling deals with the problem of sampling from a large dataset, while our approach of Adaptive Sampling is about drawing samples from a response surface that is never materialized fully. Adaptive Sampling shares goals, but not techniques, with conventional database problems like online aggregation [10], model-driven acquisitional query processing [6], and sampling for statistics estimation [2]. A two-phase adaptive method is given in [2] where the sample size required to reach a desired accuracy is decided based on a first phase of sampling. In contrast, Adaptive Sampling can adapt after each sample is collected.

Oracle 11g introduced the SQL Performance Analyzer (SPA) to help DBAs measure the impact of database changes like upgrades, parameter changes, schema changes, and gathering optimizer statistics [24]. (Quoting from [24], "it is almost impossible to predict the impact of such changes on SQL performance before actually trying them.") SPA conducts experiments where SQL statements in the workload are executed with and without applying a change. However, Oracle 11g does not provide an experiment planner that can automatically handle complex tuning tasks like parameter tuning. Finally, experiments are used to collect data in many domains like chemical and mechanical engineering, social science, and computer simulation. While iTuned shares overall guiding principles with experiment planning in these domains, the requirements and algorithms differ.

9. CONCLUSION

We described iTuned, a tool to automate the task of recommending good settings for database configuration parameters. iTuned has three novel features: (i) Adaptive Sampling to bring in appropriate data proactively through planned experiments to find high-impact parameters and high-performance parameter settings, (ii) an executor to support online experiments in production database environments through a cycle-stealing paradigm that places near-zero overhead on the production workload; and (iii) portability across different database systems. We showed the effectiveness of iTuned through an extensive evaluation based on different types of workloads, database systems, and usage scenarios.

10. REFERENCES

[1] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *OOPSLA*, 2002.

[2] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD*, 2004.

[3] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing SQL workloads. In *SIGMOD*, 2002.

[4] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD Conference*, 1998.

[5] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *SMDB*, 2008.

[6] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[7] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. Technical report, Mar. 2009. Available at www.cs.duke.edu/~shivnath/ituned.

[8] R. G. Freeman and A. Nanda. *Oracle Database 11g New Features*. McGraw-Hill Osborne Media, 2007.

[9] R. B. Gramacy. tgp: An R Package for Bayesian Nonstationary, Semiparametric Nonlinear Regression and Design by Treed Gaussian Process Models. *Journal of Statistical Software*, 19(9), June 2007.

[10] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.

[11] C. R. Hicks and K. V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 1999.

[12] S. Kwan, S. Lightstone, et al. Automatic Configuration of IBM DB2 Universal Database. In *IBM Perf. Technical Report*, Jan. 2002.

[13] Postgresql performance optimization. wiki.postgresql.org/wiki/Performance_Optimization.

[14] T. J. Santner, B. J. Williams, and W. Notz. *The Design and Analysis of Computer Experiments*. Springer, first edition, July 2003.

[15] *Dtrace, ZFS, and Zones in Solaris*. www.sun.com/software/solaris.

[16] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, 2008.

[17] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In *VLDB*, 2006.

[18] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. In *SIGMETRICS*, 2004.

[19] TPC-H and TPC-W Benchmarks from the Transaction Processing Council. `http://www.tpc.org`.

[20] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage*, 4(1), 2008.

[21] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, 2002.

[22] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.

[23] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, pages 287–296, 2004.

[24] K. Yagoub, P. Belknap, et al. Oracle's sql performance analyzer. *IEEE Data Engineering Bulletin*, 31(1), 2008.