Performance and Resource Modeling in Highly-Concurrent OLTP Workloads

Barzan Mozafari* Carlo Curino† Alekh Jindal* Samuel Madden*

*CSAIL MIT
{barzan,alekh,madden}@csail.mit.edu

†Microsoft
ccurino@microsoft.com

ABSTRACT

Database administrators of Online Transaction Processing (OLTP) systems constantly face difficult questions. For example, "*What is the maximum throughput I can sustain with my current hardware?*", "*How much disk I/O will my system perform if the requests per second double?*", or "*What will happen if the ratio of transactions in my system changes?*". Resource prediction and performance analysis are both vital and difficult in this setting. Here the challenge is due to high degrees of concurrency, competition for resources, and complex interactions between transactions, all of which non-linearly impact performance.

Although difficult, such analysis is a key component in enabling database administrators to understand which queries are eating up the resources, and how their system would scale under load. In this paper, we introduce our framework, called DBSeer, that addresses this problem by employing statistical models that provide resource and performance analysis and prediction for highly concurrent OLTP workloads. Our models are built on a small amount of training data from standard log information collected during normal system operation. These models are capable of accurately measuring several performance metrics, including resource consumption on a per-transaction-type basis, resource bottlenecks, and throughput at different load levels. We have validated these models on MySQL/Linux with numerous experiments on standard benchmarks (TPC-C) and real workloads (Wikipedia), observing high accuracy (within a few percent error) when predicting all of the above metrics.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational databases

Keywords

OLTP, Performance Predictions, Multi-tenancy

1. INTRODUCTION

Operating a large database management system (DBMS) or a multi-tenant "database-as-a-service" [16] is a challenging and stressful task for database administrators (DBA), especially as the DBMS starts experiencing heavy concurrent load. Although some databases provide tools for measuring the run-time of an individual query, many performance problems are a result of *interactions* between concurrent queries, which existing systems are not capable of modeling. Many transactions that run fine in isolation become much slower when run together, as they interact in complex ways and contend for shared resources. Load that is added over time may cause resources that were previously abundant to become constrained, and query performance to plummet. Applications may generate unpredictable, time-varying load that puts strain on different resources (e.g., RAM, disk, or CPU) at different times. To handle all these scenarios, we need a way to *attribute* system load, on a per-resource basis to different queries, transactions, or applications. This attribution enables a number of useful applications, including:

- **Diagnosis / Performance Inspection:** Why is a given query running slow (in the presence of concurrency)? Which transaction groups are causing the spike of lock wait times in the DBMS?
- **Run-time Performance Isolation:** In a database supporting multiple applications, which application/transaction is using more than its allocated share of resources? At what rate should transactions of a given application be admitted/ dropped in order to avoid any SLA (service-level agreement) violations?
- **Billing:** What is the actual contribution of each workload to the overall resource consumption?

A second challenge for DBAs is to understand how database resource consumption and performance vary as load on the system changes, e.g., when partitioning the database, or when a sudden increase in popularity of a website imposes unexpected load on the back-end database To address the second challenge, we need *what-if analysis* tools to allow DBAs to answer two other classes of questions:

- **Performance Prediction:** What will the performance (e.g. latency or throughput) of a given query or application be if the rate of 'new order' transactions doubles?
- **Provisioning:** Which resource (e.g., disk, CPU, RAM) will bottleneck first if the load on the system increases? What is the hardware required to deliver a desired throughput?

To address these challenges, we introduce our approach, called DBSeer, for statistical *performance modeling and prediction*. Our techniques involve collecting a limited set of low-overhead statistics from the DBMS and the operating system, measured during normal system operation. We then use these statistics to build offline models for the given DBMS and the workload(s) running on it. These models allow us to perform attribution and what-if analysis by assessing the resource (e.g., CPU, disk, RAM, cache, DB locks) requirements of individual queries or applications and estimating how those requirements change as the database grows in size, as queries in the workloads change, or as allocated resources vary.

In DBSeer, we develop two classes of models and compare their performance. First, we develop *black-box models*, that make min-

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

imal assumptions about the nature of the underlying system, and train statistical regression models to predict future performance based on past performance statistics. Second, we develop *white-box models* that take the major components of the underlying database system into account, to enable more accurate predictions. Specifically, we develop white-box models for disk I/O, lock contention, and memory utilization of MySQL. Although these models are focused on MySQL, we believe that even solving performance prediction in the context of this one system represents an important step forward, as MySQL alone is used by millions of users. Moreover, in Section 8.5, we report preliminary (but promising) results in applying our models to another DBMS (PostgreSQL).

As we show in our experiments, the trade-off between these two classes of models is that black-box models are more general but are also less effective in making predictions outside of the range of inputs on which they were trained. Unfortunately, many interesting questions (including many what-if scenarios) require such "out of range" predictions, e.g., predicting performance when dramatic and heretofore unseen changes happen in the workload. This is why developing white-box models is also necessary: they are less general than black-box models (as they make assumptions about the nature of the database) but they provide higher extrapolation power.

Our approach in DBSeer[1] is specifically designed for highly concurrent OLTP (*transaction processing*) applications . These applications run lightweight transactions that read or write a few records at a time. We focus on this class of problems because OLTP settings are most frustrating for DBAs, due to their high levels of concurrency and the complex interactions between transactions (i.e., competition for different resources such as locks, cache, I/O). Such competitions can lead to non-linear effects, where a small change in load can trigger a large change in performance. Though some prior work has addressed performance prediction in the case of OLAP (a.k.a. *analytical*) databases [12, 3, 8], this problem has not been well studied in OLTP. Thus, a key contribution of our work is to develop non-linear models that capture highly concurrent locking and logging operations of OLTP workloads.

In summary, we make several contributions towards modeling transactional workload, including:

- **Resource Models:** we have developed white and black-box models for predicting different resources, including CPU, RAM, network, disk I/O, and lock contention. Our primary contribution here is a set of novel white-box models for predicting disk I/O and lock contention.
- **Extracting transaction types:** we have developed highly accurate clustering techniques to automatically extract and summarize "classes of similar transactions" from a query log that allow us to accurately group similar transactions. We show that this clustering is able, for example, to identify the 5 transaction classes in TPC-C, and the major query types in Wikipedia.
- **Evaluation:** we evaluate our models on a real database system, both using the well-known TPC-C benchmark and the real-life traces of Wikipedia, showing that we can predict the maximum throughput within 0-25% error. Additionally, we show that white-box models can avoid over-provisioning by at least 9× and predict disk I/O from 4× to 100× more accurately than simple black-box models when predicting resource utilization over a wide range of transaction rates.

2. SOLUTION OVERVIEW

In this paper, we focus on the problem of *resource prediction*. Given a set of transaction types (we describe our method for deriving these below) running at a certain rate (transactions per second, or TPS), with a certain mixture (fraction of each transaction type in the overall workload), the goal is to predict the CPU usage, disk I/O, minimum amount of RAM, network consumption, and time spent in lock contention. Such models are important for understanding how close a DBMS is to saturation and which resource will saturate first, as well as for diagnosis/investigation of performance problems (e.g., attributing utilization of particular resources to particular transactions).

2.1 DBSeer Overview

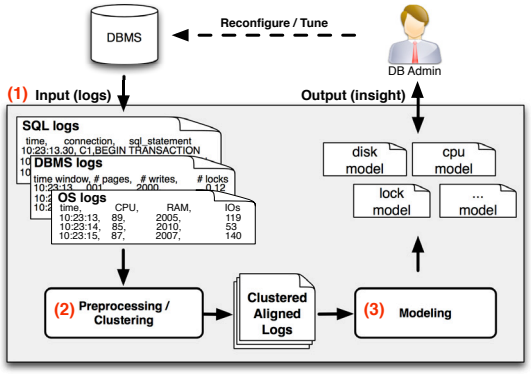

Figure 1: Workflow in DBSeer.

DBSeer consists of the following steps, shown in Figure 1:

1. **Collecting Logs.** We observe a DBMS during normal operation, i.e., running without modification in its production state. We collect standard SQL query logs, as well as various DBMS and OS statistics (over periods of hours or days).

2. **Preprocessing / Clustering.** We align (by time) and join the various logs, and extract a set of transaction types to categorize the types/classes of transactions that the system runs. We automatically cluster the transactions based on their SQL statements and the different tables that they have accessed. We also construct a concise profile of each transaction type, summarizing its access pattern to different tables in the database. These summaries are used in our lock contention model. Logging and clustering are described in more detail in Section 3. Note that the transaction types only differ based on features relevant for performance and resource consumption. This allows us to bound the number of classes we consider, while significantly improving the accuracy of our models. This also allows users to query the system with what-if scenarios that include changing the mixture of transactions (ratios of each transaction type in the workload).

3. **Modeling.** We build white- and black-box models to predict the resource utilization (CPU, RAM, disk I/O, Locks, etc.) of the system for different mixes and rates of transaction types. Our models can accurately answer attribution and what-if questions (e.g., for provisioning, diagnosis, etc) for any given mix of transactions or target transaction rate. This is shown in the figure as the Output/Insight arrow. Even for white-box models we relied on general knowledge of the functional component within the DBMS, without the need to modify nor access the source code of the system. A DB administrator (DBA) uses our system by providing as input a set of transaction logs (we use [1] to collect logs), and querying the resulting models to test a series of hypotheses. The DBA or automated tools running on his/her behalf can leverage our models to reconfigure or tune the DBMS, or perform other operational actions (migration, provisioning, etc.). Details of our models for different resources are presented in Section 4–6. Finally, we combine all these models in Section 7

[1]Source code for DBSeer is available at `http://dbseer.org`.

to accurately predict the maximum throughput that the system can deliver for a given mixture and rate.

All of our models accept a mixture $(f_1, \cdots . f_J)$ and a target TPS T, where f_i represents the fraction of the total transactions run from type i and J is the total number of types. The models are built from a small selection of training data collected over a narrow range of TPS and a specific mixture (such as the conditions of a system running in production), but can predict the expected resource utilization over a wide range of TPS and mixtures. We validate this experimentally in Section 8, showing that we can observe the system at a fixed mixture, and answer what-if and attribution questions about *never-seen-before* mixtures and rates.

3. PREPROCESSING

In this section, we describe our approach to logging (Sec. 3.1), transaction clustering (Sec. 3.2), and estimating page access distribution (Sec. 3.3). These are used as input to the prediction models described in the rest of the paper.

3.1 Gathering the Log and Statistics

We collected a number of aggregate statistics about the system. These statistics are collected passively, without affecting performance of the system or varying the rate or mix of transactions that are run, using standard operating system and database logging features. Specifically, we collected (i) resource consumption statistics from the OS (Linux, in our case), (ii) load statistics (e.g., number of dirty pages, number of logical reads, etc.) from the DBMS (MySQL, in our case), and (iii) a query log, containing start-time, duration, and the SQL for each statement run by the system. Practically this consists of running dstat (`http://dag.wieers.com/home-made/dstat/`), a python-based tool to collect OS and MySQL statistics on the server, and leveraging MySQL's query logging functionalities. Also, note that most production environments routinely monitor most of these statistics for administration and auditing purposes, and refer to this as telemetry.

The result of this logging is a number of features which we use in our models. These include:

1. The SQL statements and transactions executed by the system (that will be clustered using our clustering mechanism).
2. The run-time (latency) of each transaction.
3. Aggregate OS stats, including per-core CPU usage, number of I/O reads and writes, number of outstanding asynchronous I/Os, total network packets and bytes transferred, number of page faults, number of context switches, CPU and I/O usage.
4. Global status variables from MySQL including the number of `SELECT`, `UPDATE`, `DELETE`, and `INSERT` commands executed, number of flushed and dirty pages, and the total lock wait-time.

As we are focused on non-intrusive logging, we do not collect any statistics that significantly slow down performance, such as fine-grained locking information.

3.2 Transaction Clustering

We first use the logs to group transactions into a set of types or classes. The main goal of this step is to cluster transactions into classes that exhibit a similar access pattern, i.e. they access the same tables in the same order and perform similar operations on each table. (Note that different parts of a table might be accessed with different probabilities by each class of transactions.) Extracting transaction types allows us to model different mixtures of transactions in the workload, where the workload at any point in time is represented by the total number of transactions per second as well as the fraction of each transaction type in the mixture.

Extracting Transaction Summaries: Our clustering begins by parsing the query logs and extracting a *transaction summary* from each (successfully) committed transaction, defined as:

$$[t_o(mode_1, table_1, n_1, t_1), \cdots, (mode_k, table_k, n_k, t_k)] \quad (1)$$

where k is the number of tables accessed by the transaction (e.g., if it accesses table a, then table b and then again table a, we have $k = 3$), t_o is the time lag between the `BEGIN` and the first SQL statement, and for $1 \le i \le k$, $table_i$ is the i'th table being accessed by this transaction, $mode_i$ is either w when accessing $table_i$ requires an exclusive lock (e.g. `DELETE`, `UPDATE`, `INSERT` or `SELECT . . . FOR UPDATE`), and is r if the access requires a read-only/shared lock (e.g. general `SELECT`). We define n_i as the approximate number of rows accessed from $table_i$. Finally, for $1 \le i < k$, t_i is defined as -1 when both the i'th and the $(i+1)$'th table accesses are caused by the same SQL statement. Otherwise, t_i is the time lag between the completion of the SQL statement causing the i'th access and the time that the $(i+1)$'th statement was issued. t_k is defined as the time between the last SQL statement and the final `COMMIT`. All of this information can be obtained from the SQL logs except the number of rows read or written from each table, which we estimate using the query rewriting technique described in Section 3.3.

Learning Transaction Types: Given the transaction summaries, we use the extracted features and apply the DBSCAN [9] clustering algorithm to group individual transactions based on their accesses. Note that DBSCAN is an unsupervised method (so it does not require labeled examples of each transaction type), and does not make any a priori assumptions about the number of clusters (as does, for example, k-means). As we show in Section 8.3, this clustering performs very well. In fact, it gives 0% misclassification compared to a manual clustering of the transactions based on their semantics. The input data consists of one row per transaction. Each row contains a set of transaction features. The feature set consists of 2 attributes for each table in the database, one for the number of rows read from and one for the number of rows updated/inserted in each table (many of these features will be zero as most transactions do not access all the tables). These simple features proved sufficient for clustering TPC-C and Wikipedia (as well as several other workloads [1]).

The output of this clustering is a workload summary that lists the major transaction types (along with representative queries) and provides their frequencies in the base workload. We found that the extracted classes are easy to understand and manipulate by the users, since we provide representative examples and allow the user to assign a name/label to each transaction type (e.g., the New Order transaction of TPC-C). This is a powerful abstraction that enables users to specify the rate at which *each* type of transaction will be executed (i.e., mixture and overall rate) as input to the prediction models. This allows us to explore hypothetical scenarios and understand how performance would vary for different mixtures of transactions.

3.3 Estimating Access Distributions

Our second use of the logs is to infer a rough probability distribution over all the pages in the database by access (read or write) and by transaction type—this is used in our locking and I/O prediction models to estimate conflict and update probabilities. We do this by processing the raw MySQL logs to extract individual queries/updates, and then rewriting queries to extract the primary keys accessed in the tables (using rewriting techniques in [6]). We then run our rewritten queries against the database to obtain sets of primary keys read/written by each query/update (grouped by transaction type). This analysis is done offline, imposing no overhead on the operational database. In the case of MySQL/InnoDB, primary keys are always the clustered index of a table, therefore we can assume that tuples with contiguous primary keys are stored consecutively on disk and hence, we can infer which pages are ac-

cessed. Here, we assume that all accesses to large tables are done via indexes, rather than sequential scans, which is true of well-tuned transactional applications. However, if needed, we could also use the `EXPLAIN` command to determine the exact access method in the query plan and hence, the exact set of pages that a given transaction will access. We then further aggregate these accesses into probability distributions (per table and per transaction type) necessary for our predictions.

4. MODELING DISK I/O AND RAM

In this section, we present our white-box models for disk I/O and RAM provisioning. Our models in this section are based on MySQL. While some of MySQL features are shared among other traditional DBMSs (e.g., LRU-based caching or 2-phase locking), some of the other features are specific to MySQL (e.g., the buffer pool flushing policy). Our main goal in this paper is to demonstrate that it is possible to build accurate models for a given DBMS. In particular, we focus on MySQL due to its popularity (used by millions of users). However, In Section 8.5, we also report preliminary results on applying our models to another DBMS (PostgreSQL).

Disk and memory are important aspects of performance in a database system. In fact, in a transactional database, such as MySQL, disk I/Os and RAM utilization are closely related to one another. The three main causes for disk I/Os are:

1. **Log writes**, needed to guarantee transactionality.
2. **Dirty pages write backs**, needed to bound recovery time, and allow transactionally-consistent reclamation of log files (all pages dirtied by transactions logged in the current log file need to be flushed to disk before the log file can be recycled).
3. **I/Os due to cache misses**, needed to read pages from disk that were not cached in the buffer pool, and possibly trigger eviction of dirty pages (which need to be written back to disk first).

These operations heavily depend on the size of the buffer pool: a smaller buffer pool leads to more I/Os.

In Section 4.1, we provide a brief overview of the aforementioned factors. Then, in Section 4.2, we introduce our white-box model for disk writes, which accounts for both log-related writes as well as those related to log reclamation. Finally, in Section 4.3, we present our models for disk reads and page writes caused by cache misses/evictions.

4.1 Background on Disk I/O in a DBMS

Log writes, due to their nature, can be easily modeled with linear regression, as they are proportional to the rate of each transaction type in the load (our linear models are discussed in Section 6). Therefore, in the rest of this section we focus on writes that are due to dirty page write-backs, or "flushes". Flushing happens for two main reasons: 1) *capacity misses*, when a new page is brought into the buffer pool and there are no free buffers, forcing an existing record to be flushed and 2) *log-triggered data flushes*, when the redo log file(s) is full and needs to be *rotated* or *recycled*. We handle capacity misses together with reads in Section 4.3, while in this section we focus on *log-triggered data flushes*.

Log-triggered data flushes exist because the dirty pages corresponding to the recycled log records need to be flushed to disk before recycling the log file. In a typical implementation, multiple log files are used in a circular fashion. Before recycling an old log file, the DBMS guarantees that all buffer pool pages dirtied by transactions logged in that file have been flushed back to disk. This means that the DBMS may temporarily stop serving transactions to flush dirty pages (this is needed to guarantee transactionality). Since this can cause a performance-hiccup for database users, modern DBMSs try to avoid this visible disruption of performance/service. This is achieved by means of a combination of heuristics that strike a balance between avoiding stalls due to log recycling and amortizing multiple writes to a page in the buffer pool before writing it back to disk, thus limiting the I/O pressure. In MySQL, this process is referred to as *adaptive flushing* of the buffer pool pages. Modeling the net effect of all these complex heuristics is a challenging task, and is one of our main contributions in this paper which is described next.

4.2 Disk Write Model

While different DBMSs use different heuristics for maintaining the balance between eagerly writing pages or lazily flushing them at the log rotation time, in the following we provide a simple analysis based on *conservation of flow* that abstracts the internal details and complicated heuristics of a particular DBMS (MySQL in our case) while still providing a reasonable prediction of the overall I/O behavior. To achieve this goal, we leverage the probability of pages being dirtied and evicted, as follows.

Probability of a page being dirtied. Let D be the number of pages in the database. For any given mixture of transactions, say $\tilde{f} = (f_1, \cdots, f_l)$, we build a probability distribution[2] $\tilde{p}_{write}$ over all the pages in the database, where every transaction drawn from this mixture writes to the i'th page with probability $p_{write,i}$. In other words, $\sum_{i=1}^{D} p_{write,i} = 1$. Here, for simplicity, we assume that a transaction only accesses one page. (similar analysis can be done when each transaction accesses multiple pages).

Given $\tilde{p}_{write}$, we would like to estimate the expected number of *unique* dirty pages after executing n transactions, drawn at random according to their corresponding weight in the mixture $\tilde{f}$. Here, we assume that different transactions arrive independently of each other. Let us denote this value with T_n, which can be written as

$$T_n = \sum_{i=1}^{D} T_{n,i} \tag{2}$$

where $T_{n,i}$ is the probability of the i'th page being written to, at least once (i.e. being dirtied), calculated as:

$$T_{n,i} = 1 - (1 - p_{write,i})^n \tag{3}$$

where $(1 - p_{write,i})^n$ is the probability of the i'th page staying clean (i.e., never been written to) after n transactions.

Next, we need to model the log rotation process using these probabilities. Note that, at any point in time, every page falls into exactly one of these three categories : (C1) where a page is dirty and the first transaction that made it dirty is logged in the old log, (C2) where a page is dirty and its first dirtying transaction is logged in the current (new) log, and finally (C3) where a page is still clean (i.e. is identical to its copy on the disk). Let $P_{1,i}$, $P_{2,i}$, and $P_{3,i}$ represent the probability of the i'th page belonging to each of these categories, respectively. Clearly, $P_{1,i} + P_{2,i} + P_{3,i} = 1$ for $i = 1, \cdots, D$. Let $d_{1,t}$, $d_{2,t}$ and $d_{3,t}$ represent the number of pages in categories (C1), (C2) and (C3) at time t, respectively. Also, let L be the maximum capacity of each log file, i.e., each log file can log up to L transactions on average. Clearly, the log needs to be rotated (i.e., the old log has to be deleted) at least as often as every L transactions, i.e. when the new log is full. However, the log rotation can only happen at time t if $d_{1,t} = 0$, otherwise a system crash could lead to data loss in any of the pages in category (C1), i.e., a loss of durability. Moreover, if a log rotation happens at t, we will have $d_{2,t+1} = 0$, i.e. the new log will be empty at the beginning of time $t + 1$.

In the following, we show how $P_{1,i}$, $P_{2,i}$, and $P_{3,i}$ can be expressed

[2]We use the notation $\tilde{X}$ to denote finite probability distributions, where X_i denotes the probability of the i'th outcome.

in terms of $T_{n,i}$. First, however, we provide the intuition behind our analysis.

Abstracting the main idea behind MySQL's adaptive flushing. The main idea behind MySQL's I/O heuristics, such as adaptive flushing (`http://bit.ly/bRN04A`), is that the flush rate (flow of pages out) should roughly match the rate at which pages are dirtied, such that at the time a log rotation happens, there will be no dirty pages waiting to be flushed. Specifically, if the system is running n transactions per second adaptive flushing chooses a flush rate of $F_t(n)$, where:

$$F_t(n) = \frac{d_{1,t}}{\frac{l_t}{n}} = \frac{d_{1,t} \cdot n}{l_t} \tag{4}$$

Here, l_t denotes the current capacity of the new log ($0 < l_t \leq L$) at time t, and thus, the new log is expected to get full in $\frac{l_t}{n}$ seconds given the current rate, over which $d_{1,t}$ pages need to be flushed back to disk. Thus, adaptive flushing attempts to write $\frac{d_{1,t} \cdot n}{l_t}$ dirty pages back to disk during each unit of time. Note that in reality MySQL uses several other heuristics to decide when to flush a page, but in Section 8 we show that this simplified model is a good enough predictor of the average flush rate.

Estimating flush rate (Monte-Carlo baseline). Given this basic model of flushing, the goal of the rest of this section is to predict the expected flush rate $F(n)$ for a given TPS of n, where $F(n) = E[F_t(n)]$, without directly observing the $d_{1,t}$ and l_t values. One approach to estimate $F(n)$ is to simply perform a MC (Monte-Carlo) simulation by randomly initializing l_0 and $d_{1,0}$ (to account for bias caused by different starting states), then for each t, selecting $\binom{D}{d_{t,0}}$ pages (as dirty ones), then selecting $F_t(n)$ out of these $d_{1,t}$ pages to flush, and then repeating this process for many values of l_0 and $d_{1,0}$ until we converge to a value for $F(n)$. This MC simulation, however, has several drawbacks. First, it provides little insight into the contribution of different characteristics of the workload or different tuning parameters towards the overall flush-rate (e.g., there is no way to attribute flushes to a given transaction type). Secondly, the straightforward MC simulation is quite slow—for TPC-C, we found it to be 6 to 10 orders of magnitude slower than the solution that is proposed in the rest of this section. This is due to the large number of variables and the numerous possible initial configurations. For instance, the TPC-C workload with 32 warehouses has about 750,000 different pages, and thus, in order to accurately estimate $F(n)$, we need to repeat the simulation for a reasonable portion of all the 2^{750000} possible sets of dirty pages, as starting states!

Estimating flush rate (Iterative approach). Instead, we develop a simple iterative algorithm that converges much more quickly and provides very accurate predictions as well as insight into how different transactions contribute to the overall I/O. Before presenting the details of our algorithm, we first simplify equation (4). Assuming that there is no log rotation at t and $t+1$, in expectation we have:

$$d_{1,t+1} = d_{1,t} - F_t(n)$$

Thus, the number of outstanding dirty pages can only decrease, since the old log is no longer being appended to. We also have $l_{t+1} = l_t - n$, since n log records are written to the new log at each time step. Substituting these two values into equation (4) for $F_{t+1}(n)$ we get:

$$F_{t+1}(n) = \frac{(d_{1,t} - F_t(n)) \cdot n}{l_t - n} = \frac{(d_{1,t} - \frac{d_{1,t} \cdot n}{l_t}) \cdot n}{l_t - n} = \frac{d_{1,t} \cdot n}{l_t}$$

Thus, in the absence of log rotations, $F_{t+1}(n) = F_t(n)$. By induction:

$$F_t(n) = \frac{d_{1,t_0} \cdot n}{L} \tag{5}$$

where t_0 is the time step immediately after any log rotation (i.e. log was rotated at $t_0 - 1$) and t is any time before the next log rotation. In other words, $t_0 \leq t < t_0 + \frac{L}{n}$. This holds because $l_{t_0} = L$, i.e., the new log is empty right after a log rotation.

Equation (5) is more desirable for estimating the expected flush rate, i.e. $E[F_t(n)]$, since n and L are time-independent. Thus, we only need to estimate $E[d_{1,t}]$. Due to the linearity of expectation, we have:

$$E[d_{1,t}] = \sum_{j=1}^{D} P_{1,j} \tag{6}$$

$$\text{Similarly,} \quad E[d_{2,t}] = \sum_{j=1}^{D} P_{2,j} \tag{7}$$

$$\text{and} \quad E[d_{3,t}] = \sum_{j=1}^{D} P_{3,j} \tag{8}$$

This equation, gives us a direct formula for estimating the average disk I/O (or flush rate), if we can accurately estimate $P_{1,i}$, $P_{2,i}$, and $P_{3,i}$. However, these variables are inter-dependent, since as pages are dirtied and flushed to disk, the probabilities change. To handle this, we define three time-series $\{P_{1,j,t}\}$, $\{P_{2,j,t}\}$, and $\{P_{3,j,t}\}$, where $P_{i,j,t}$ is the probability of the j'th page belonging to category Ci (where $i = 1, 2, 3$) at time t (where $t = 0, 1, 2, \cdots$). The final step of our modeling is to find the relationship between $\{P_{1,j,t}\}$, $\{P_{2,j,t}\}$, and $\{P_{3,j,t}\}$. Here, we omit several derivations (which are presented in detail in our technical report [15]), and present the final result, showing that for $m \geq 1$:

$$P_{1,i,t+m} = P_{1,i,t}\left(1 - \frac{n}{L}\right)^m$$

$$P_{2,i,t+m} = 1 - P_{1,i,t+m} - P_{3,i,t+m}$$

$$P_{3,i,t+m} = \begin{cases} P_{3,i,t} \cdot \left(1 - \frac{n}{L}\right)^m + P_{1,i,t} \cdot \frac{n}{L} \cdot m \cdot \frac{(1-\frac{n}{L})^m}{1-\frac{n}{L}} & \text{if } T_{n,i} = \frac{n}{L}, \\ P_{3,i,t} \cdot \left(1 - T_{n,i}\right)^m + P_{1,i,t} \cdot \frac{n}{L} \cdot \frac{(1-T_{n,i})^m - (1-\frac{n}{L})^m}{\frac{n}{L} - T_{n,i}} & \text{if } T_{n,i} \neq \frac{n}{L}. \end{cases}$$

Our algorithm uses these equations by iteratively incrementing t, until they converge, thus estimating the values of $P_{1,i}$, $P_{2,i}$, and $P_{3,i}$. Having these values, we can use equation (6) and eventually (5) to estimate the flush rate.

4.2.1 Algorithm

Figure 2 shows the pseudo code for estimating the flush rate. The main idea behind this algorithm is to use equations above (line 2.4 to 2.6), to estimate $F(n)$ (line 2.3), and repeat this process until our estimator for $F(n)$ (i.e., $avgF$ in Figure 2) converges, i.e. it does not change more than a small value ϵ. The main difference between the algorithm and the aforementioned equations is that here, for efficiency, we pre-calculate the common expressions that remain constant throughout the iterations (e.g., T, Tp, $cachedCoef$ and sum). In each iteration, the algorithm predicts the flush rate in two stages. First, the algorithm chooses $m = \lfloor L/n \rfloor - 1$ which is the longest time interval without a log rotation, in order to estimate all the variables right before the next log rotation using the values of those variables from the last log rotation (lines 2.4–2.6). The second step is to apply the log rotation (lines 2.7–2.9) step, which simply involves swapping the $P(1, i)$ pages dirtied in the previous iteration into the old log, and discarding the previous $P(2, i)$ old pages. To compute the actual pages flushed during each iteration (F), we compute $d1$ using $P1$ (line 2.2) and then compute F from $d1$ using adaptive flushing (line 2.3), equation (4), ensuring that this value never exceeds

Algorithm I/O-predictor($\tilde{f}, n$)

1: $avgF \leftarrow \inf, iter \leftarrow 0, F \leftarrow 0,$
$P_{1,i} \leftarrow 0, P_{2,i} \leftarrow 0, P_{3,i} \leftarrow 1.$
$period = \lfloor \frac{L}{n} \rfloor$ %time period between subsequent log rotations
$m = period - 1$ %longest interval without a log rotation
For $i = 1, \cdots, D$:
$T(i) = 1 - (1 - p_{write,i})^n$
$Tp(i) = (1 - T(i))^m,$
$cachedCoef(i) = (1 - n/L)^m,$
If $T(i) = \frac{n}{L}$,
$sum(i) = m * Tp(i)/(1 - T(i)),$
Else,
$sum(i) = \frac{(1-T(i))^m - (1-\frac{n}{L})^m}{\frac{n}{L} - T(i)};$
2: While $|avgF - F| > \epsilon$,
2.1: $avgF = (iter * avgF + F)/(iter + 1),$
2.2: $d1 = \sum_{i=1}^{D} P_{1,i},$
2.3: $F = min(\frac{d1}{period}, MaxFlushRate),$
%For the time step right before the log rotation, we have:
2.4: $P(3,i) = P(3,i) * Tp(i) + (n/L) * P(1,i) * sum,$
2.5: $P(1,i) = P(1,i) * cachedCoef,$
2.6: $P(2,i) = 1 - P(1,i) - P(3,i),$
%Now flush the rest of the old log and rotate the logs
2.7: $P(3,i) = P(3,i) + P(1,i),$
2.8: $P(1,i) = P(2,i),$
2.9: $P(2,i) = 0$
Return $avgF$

Figure 2: Expected I/O (flush rate) prediction algorithm (without page clustering).

$MaxFlushRate$, the maximum number of physical pages that a particular machine can perform.

Optimization, clustering similar pages: One of the major sources of time and space complexity in the pseudo code of Figure 2 is the total number of pages, i.e. D. However, after careful examination, it is apparent that one can cluster the pages based on the $p_{write,i}$ values. Thus, in the optimized version of this algorithm, we first cluster the D into $K(D, \epsilon)$ partitions (non-overlapping clusters) such that the i'th and j'th pages fall within the same partition if and only if $|p_{write,i} - p_{write,j}| \leq \epsilon$, where ϵ is a small enough constant. This is equivalent to running a K-means clustering over the $p_{write,i}$ values for the smallest K whose inter-cluster distance is no larger than ϵ. Choosing a proper value for ϵ is fairly straightforward, as one can use the algorithm's equations to find the largest value of ϵ for which the final error of the algorithm is less than user's tolerance threshold (We omit the math due to lack of space. However, note that our algorithm still works even without using this optimization).

This simple optimization further reduces the space and time complexity. For instance in TPC-C with 32 warehouses, we have $D \approx 750,000$ but there are only 8 unique values in the $p_{write,i}$'s, i.e. $K_{TPC-C}(D, 0) = 8$.

Once the pages are clustered based on their $p_{write,i}$ values, the algorithm in Figure 2 remains the same except:

1. The algorithm takes an extra input parameter, i.e. ϵ.

2. Before the first line, run the clustering algorithm with ϵ, find $K(D, \epsilon)$. Then, replace D with $K(D, \epsilon)$, and for $i = 1, \cdots, K(D, \epsilon)$ use the mean of the i'th cluster for $p_{write,i}$.

3. Replace line 2.2 in Figure 2 with a weighted sum, namely $d1 = \sum_{i=1}^{D} P_{1,i} \cdot K_i$ where K_i is the number of pages in the i'th cluster.

4.3 Disk Reads and RAM Provisioning

As noted above, RAM and disk I/O are tightly coupled, as more RAM can reduce the number of capacity misses in the buffer pool, reducing the number of page reads and dirty page writes. Simply creating a buffer pool that is as large as possible, however, doesn't guarantee any particular level of capacity-related misses, so some model is needed to predict the rate at which capacity misses will occur for a given buffer pool size, transaction mixture, and per-transaction-type TPS target. This will then allow us to predict the maximum TPS that can be sustained given the available RAM and disk bandwidth, and also the measure the contribution of any one transaction to the overall load of the system.

To do this we built a Monte-Carlo simulation of the buffer pool. To estimate the miss rate for a database with N pages of RAM, we allocate an N element list bp. Using the page access distributions for each transaction type (derived as described in Sec. 3.3), we derive a *combined access distribution* that represents the probability of each of the D pages in the database being touched by the input mixture. We then simulate accesses to these D pages of the database by randomly selecting pages according to this combined distribution. When a page is accessed, it is added to the head of bp if it is not already present, other wise it is moved to the head of the list. When a page is added a counter C_{read} of the number of misses is incremented. If the access is a write, a bit on the page is set to mark it as dirty. If bp already contains N elements, the last element (tail) of bp is removed, and if the dirty bit is set, a counter C_{write} of the number of flushes is incremented. This simulates the behavior of an LRU cache eviction policy. We can then compute the number of page reads and flushes per second by dividing C_{read} and C_{write} by the TPS.

We further refine our simulation model to use LRU2 (which is used by MySQL), where when a page is first accessed it is added to somewhere in the middle of the list, and then moved to the head on a subsequent access. This prevents sequential scan operations of large tables from evicting all of the pages in the buffer pool. In Section 8.4 we show that this simple approach is able to estimate the number of buffer pool misses for a given mixture and RAM size.

In summary, combining our cache and log rotation models, our I/O model predicts that if we are running n TPS for a time period t, we will read $C_{read} \cdot t \cdot n$ data pages and write back $C_{write} \cdot t \cdot n + F_t(n)$ data pages, in addition to any sequential log I/O.

5. LOCK CONTENTION MODEL

In this section we develop a model of two-phase locking (2PL) that, given a mixture of concurrent transaction, allows us to predict the expected delay that a transaction will incur. The non-linearity of lock-contention makes this a challenging problem.

Our model for lock waits is based on an adaptation of Thomasian's model of two-phase locking (2PL) [21] (there have been a number of other locking models, as described in Sec. 9, but Thomasian's appears to be the most realistic).

In order to make Thomasian's model work for real workloads, we made several extensions and modifications to it. We first provide a high-level description of Thomasian's approach and then briefly describe our own modifications to the original model.

5.1 A Summary of Thomasian's 2PL Analysis

Input parameters. Thomasian's model assumes that the incoming transactions belong to a fixed number of *transaction classes*, $C_1, \cdots, C_J$, and the database consists of a fixed number of non-overlapping *regions* (e.g., rows, tables, etc), $D_1, \cdots, D_I$. All transactions in a class have the same access pattern to the database, consisting of a probability of accessing different regions. Thomasian's model also assumes that all transactions of class C_j always access the same (fixed) number of locks, say K_j, and therefore consist of $K_j + 1$ steps: an initialization step followed by K_j steps each preceded by a lock request. The processing times of transaction steps are assumed to be exponentially distributed and the mean processing times for the n'th step of a transaction of C_j is denoted by $S_{j,n}$. Transactions in

C_j access objects in the i'th region in their n'th step with probability $g_{j,n,i}$, and hence $\sum_{i=1}^{I} g_{j,n,i} = 1$.

General Idea. Here we provide a high level overview of Thomasian's work; see [21] for details. The approach assumes a non-homogeneous model where different transaction classes have different access patterns to the database regions. The approximate analysis is based on mean values of parameters to derive expressions for the probability of lock conflict (usually leading to transaction blocking) and the mean blocking time. The latter requires estimating the distribution of the effective wait-trees encountered by blocked transactions and the mean waiting time associated with different blocking levels. This is done is an iterative manner, as follows.

Let $U_{j,n}$ be the mean delay incurred by transaction from C_j due to encountering a lock conflict at its n'th step. At each step, $U_{j,n}$ is estimated based on (i) the probability of a conflict per transaction type and per each database region, (ii) the mean number of in-flight transactions that access the same region, and (iii) their current mean latencies (i.e., sum of their $S_{j,n}$ and $U_{j,n}$ values). This is done by calculating the mean depth of the wait-tree for each conflict. Once $U_{j,n}$ values are updated, new estimates for (i), (ii) and (iii) are calculated, because an increase in $U_{j,n}$ values will increase the aforementioned parameters as well. This iterative process continues until the estimates converge. The final lock delay for transactions of class j is $\sum_{i \in 1 \ldots n} U_{j,i}$.

5.2 Our Extensions to the Original Model

Despite several advantages over other theoretical work on 2PL modeling, some of the assumptions in Thomasian's original proposal [21] limit its applicability to real-life workloads and database systems, which we have addressed. The main difference between his model and our approach is that rather than assuming a particular set of transaction classes and database regions and distribution of page accesses, we learn these values from our test data. Moreover, instead of assuming a fixed processing time, we update transaction times during the analysis.

1. To decide on the database regions, we use the access distribution derived from the log (see Sec 3.2) and the same ϵ-page-clustering technique as in Section 4.2.1 to define each region as a cluster of pages in the database where each page is equally likely to be accessed.

2. To estimate $g_{j,n,i}$ values, we average the numbers extracted in the summaries of transactions of the same type (See Sec 3.2).

3. Measuring $S_{j,n}$ values is not as easy as computing regions and g values. The $S_{j,n}$ mean values are not constant and change depending on the number of active in-flight transactions. We address this problem by performing a linear regression (LR) to estimate $S_{j,n,m}$ which is the value of $S_{j,n}$ with m outstanding transactions. Our regression model is built using the transaction latencies of different transaction classes recorded in our log files.

4. The model also assumes "infinite resources", such that once the $S_{j,n}$ values are provided in the input, they do not change based on the number of active transactions in the system, as if each transaction is executing on its virtual processor with a processing rate independent of the number of active transactions in the system. We dynamically re-adjust these values at each iteration of the algorithm as follows. Based on Little's law from queuing theory we know that the average number of in-flight transactions at any point in time is $M = R * T$ where R is the average latency of a transaction and T is the system's throughput. Since our goal is to estimate the expected value of the lock waits in a steady state, without loss of generality, we can assume that T remains constant. At each iteration of the algorithm our estimation of R and M are refined. Therefore, at each iteration, instead of the original $S_{j,n}$ values, we use the adjusted values $S_{j,n} * c/M$ where c is the number of physical cores on the target machine. Although this linear approximation is not entirely accurate, it reasonably compensates for the decoupling assumption of the original model, as validated in our experiments.

5. The original model assumes that all the locks are exclusive. This is not true as in most workloads a considerable portion of the locks are read-only (i.e., shared). We solve this problem by using a Theorem from [20] stating that a database region with cardinality D_i in which locks are requested uniformly, in exclusive-mode with probability b and inclusive-mode with probability $1 - b$, is approximately equivalent to a region with cardinality D'_i in which all the locks are uniformly requested in exclusive-mode, where

$$D'_i = \frac{D_i}{1 - (1 - b)^2}$$

Thus, we adjust the original region cardinalities to compensate for the lack of inclusive locks in Thomasian's analysis.

6. The original model assumes that the transactions in each class always request the same number of locks, i.e. K_j is fixed for C_j. This assumption is again un-realistic. For instance, the 'delivery' transaction in TPC-C can request anywhere from 10 to 140 locks. Therefore, in our implementation we first cluster the transactions into different types, and then further re-partition the transactions of the same type into several transaction classes based on the number of locks that they acquire. This requires that we re-adjust the original mixture, i.e., substitute the f_j's with new probabilities for each class. We obtain these probabilities using the normalized frequencies in the training data per each transaction type. We also ignore very rare transaction classes (i.e., when $f_j < \epsilon$).

The accuracy of this improved model is evaluated in Section 8.

6. BLACK-BOX MODELS

In Sections 4 and 5, we introduced our white-box models (for RAM, disk I/O and lock contention) which are somewhat specific to MySQL, but as we will show in our experiments, enable us to make accurate predictions, well outside of the range of our training data. In this section, we present several choices of black-box models using off-the-shelf machine learning techniques for regression, which make minimal assumptions about the underlying system, and hence, are less specific to MySQL. In Section 8, we study the accuracy trade-off of these two types of models under different scenarios. Below, we divide the resources into linear and non-linear and discuss our models separately.

Linear Resources: Out of the resources that play a major role in the performance of a database, the following ones tend to grow generally linearly with the counts of each transaction type: CPU, network utilization and the number of log writes.

For example, for CPU, if a given transaction requires a certain number of CPU cycles to finish, it is likely that its execution involves almost the same instructions, regardless of others transactions that are running concurrently. Thus, for a given mixture $(f_1, \cdots, f_J)$, and a given TPS T, the CPU usage can be approximated as:

$$CPU = a_0 + a_1 \cdot f_1 \cdot T + \cdots + a_J \cdot f_J \cdot T$$

where a_i coefficients can be learned from training data (e.g. by minimizing squared error). Network I/O can be modeled similarly, as the number of messages sent for a transaction does not depend significantly on the concurrent transactions with which it is executing. Finally, the number of log writes and the number of logical records updated per second are also linearly related to the number of transactions of each type that are executed per second. Hence, these metrics can be predicted accurately across a wide range of mixtures and TPSs given a modest amount of training data.

Non-linear Resources: We have experimented with many statistical

regression models to predict the lock contention and disk I/O, including (but not limited to) all regression algorithms that come with Weka (`www.cs.waikato.ac.nz/ml/weka/`) library. However, in this paper, we only mention a few models that either yield better accuracy or represent major approaches to regression.

Polynomial fitting. By specifying the degree of a polynomial, one can use the training data to learn the coefficients of the polynomial that minimize the squared error of the fit. In particular, theoretically it has been shown that lock wait times, in a perfect 2PL scheme, asymptotically converge to a *quadratic* function of the multi-programming level [11]. Thus, a reasonable block-box model of lock wait times can be:

$$LockWaitTime = a_0 + \sum_i a_i \cdot f_i \cdot T + \sum_{i,j} a_{i,j} \cdot f_i \cdot f_j$$

where a_i and $a_{i,j}$ coefficients need to be learned.

Kernel Canonical Correlation Analysis (KCCA). KCCA [4] is a powerful kernel-based technique that considers a pair of multivariate datasets and finds dimensions along which the two datasets are maximally correlated, whereby kernel functions are used as metrics of similarity. In our context, transaction counts are the first dataset, while performance metrics (lock, or I/O) constitute the second. Since KCCA has been successfully used in performance prediction for analytical workloads [10], we wanted to see if it could also be applied to transactional workloads.

Decision Trees. Decision trees are a well-known technique for both clustering and regression. For regression, the target value of a given test data is predicted as the average target value of the corresponding leaf node in the tree. We used Matlab's implementation of decision tree regression with default parameters, specifically with leaf-merging, no pruning and minimum of 1 item for forming a leaf node (all for maximizing the extrapolation power).

Neural Networks. Feed forward neural networks are another important class of regression techniques, used to fit an input-output relationship. Again, we used Matlab's implementation (called *fitnet*) with 10 hidden layers and Levenberg-Marquardt back-propagation algorithm as the training procedure.

7. THROUGHPUT PREDICTION

Now that we have described our white- and black-box models for individual resources, we briefly describe how we predict the overall throughput of the system and identify the bottleneck resource. Each model produces an estimate of resource utilization at a given TPS rate. To determine maximum system throughput, we need to identify the TPS at which each model predicts the resource will be saturated (e.g., the point where the disk I/O model predicts the disk will be saturated). For disk and CPU, we determined per-resource saturation points using the methodology for measuring the maximum resource capacity of a machine described in Curino et al. [5]. For disk, the maximum number of page flushes we could generate on our test machine was about 600 pages per second; for CPU, the maximum CPU load we could generate was about 90%. We use these to predict TPS maximums T_{disk} and T_{cpu}, based on the maximum TPS at which our disk model predicts an I/O rate of 600 page flushes per second, or our CPU model predicts a 90% usage.

For lock contention, our model produces an estimate of the latency for each transaction class, at a specified TPS. Using Little's Law (which says that the number of in-flight transactions is equal to the arrival rate, or TPS, times the mean transaction latency), we can compute the number of in-flight transactions. Since almost all production databases or websites use admission control to limit the number of outstanding clients, the maximum TPS T_{lock} we can achieve (based on our lock contention model) is the one at which this client limit is reached. Alternatively, users can specify a maximum per-transaction latency L, which we can use to select a TPS that keeps the latency below L. To estimate the overall maximum throughput of the system, we compute $T = \min(T_{disk}, T_{cpu}, T_{lock})$. The bottleneck is the resource that dictates this minimum throughput.

8. EXPERIMENTS & EVALUATION

The goal of our experiments is to understand the accuracy of our previously presented white- and black-box models at predicting resource utilization across a range of TPS and mixtures on real workloads, and to look at the end-to-end performance of our models on several "what if" and "attribution" scenarios, including:

- Finding bottlenecks: we show that we can identify the maximum throughput and the bottleneck resource for a given workload, TPS, and mixture, and that our bottleneck estimates are accurate enough to allow low-overhead provisioning of a system.
- Attribution: we show that we can predict the contribution of a given transaction type to the overall resource consumption of a system, given a workload, TPS, and mixture.

Overall, our results are promising, showing that we can often achieve low errors on individual models – e.g., within 20% relative error for CPU, 20% error for Disk writes and 0.5GB for RAM – and that we can identify bottleneck resources accurately in almost all cases and predict maximum throughput with a mean error of under 25% across a range of benchmarks.

Experimental Setup. Our experimental testbed is composed of two identical Dell PowerEdge R710 servers (one to generate load and the other one dedicated to MySQL) interconnected via a single gigabit Ethernet switch. Each server has two quad core Intel Xeon E5530 2.4 GHz processors, 24GB of RAM, and 6 disks (2TB 7200 RPM SAS) configured as a single RAID 5 volume. We used MySQL 5.5.7 running on Ubuntu 10.10 with kernel 2.6.35.

Datasets. We produced several datasets with different mixtures and different TPS, using both TPC-C benchmark and real-life Wikipedia transactions, all replayed using [1]. For each of these data sets we divided the workload into transaction types using our transaction clustering technique (we measure the accuracy of this clustering versus manual labeling in Section 8.3) and collected transaction summaries and access distributions as described in Section 3. Our main dataset consists of runs of TPC-C and Wikipedia at the following TPSs: 100, 200, 300, 400, 500, 600, 700, 800, 900. To further challenge our system we generated workloads sweeping very different mixtures of transaction types. For instance, in TPC-C, we gradually increased the fraction of the 'New Order' transactions from 0% to 88%, while decreasing the fraction of 'Payment' transactions from 88% to 0%, while keeping 'Order Status', 'Delivery', and 'Stock Level' at their original rate, namely 4% each. This is because most of the updates come from the first two types, allowing for better evaluation of our performance models. For Wikipedia, as with TPC-C, we varied the mixture of the two most frequent transactions, while keeping the other three at a fixed frequency.

Training Time. Across all of our experiments (some containing 30,000 training samples), training time of our white-box models for RAM, disk I/O and lock contention were all below 15 minutes. The black-box models exhibited a wider range of training times (from seconds for linear regression and curve fitting) to a few minutes for others. KCCA did not easily scale to large datasets (over 100 samples) and hence, we first clustered the training samples into at most 100 clusters before invoking KCCA. Since all our models are meant to be trained offline, we believe that these times are reasonable.

8.1 Verifying Throughput Prediction

In this section, we show that by using our resource models we

can accurately predict the saturation point of each resource, which in turn can be used to predict the bottleneck resource and the overall maximum throughput, as described in Sec 7.

We compared our combined models ("Our combined WB model"), which consist of our white-box models for lock and disk I/O plus linear regression (LR) for CPU to simple linear regression on the CPU vs transaction counts ("LR for CPU"), and a simple linear regression on the number of page flushes vs transaction counts ("LR for #PF"). We chose linear regression since it seems the most natural choice for DBA, e.g. when the load is twice, expect twice the resources. Also, linear regression has been proposed by the previous work [2] as a more effective model for predicting the disk I/O compared to other types of regression such as Gaussian processes. We also present results of decision tree regression for predicting the maximum throughput via projecting the disk's flush rate ("Dec. Tree for #PF"). We omit other black-box models, as they all performed very poorly at predicting maximum throughput.

To compute the actual maximum throughput (i.e., "ground truth"), we warmed the buffer pool and then gradually increased the offered TPS until the performance leveled off at some maximum rate. We randomly generated 20 mixtures of TPC-C with different ratios of transaction types. We ran these mixtures on MySQL at a low to moderate throughput and collected test data which we used to train both our models as well the two LR models and the decision tree regression model. We used each model to predict the maximum expected throughput, and compared to ground truth.

The results of our experiments are shown in Figure 3, showing the average relative errors of each model at estimating the maximum throughput on different subsets of the mixtures. Specifically, we grouped the 20 mixtures into three sets, one for I/O bound mixtures, one for lock-bound mixtures, and one for CPU-bound mixtures, and computed the average performance of each model on each subset (only a few of our mixtures were I/O bound, and approximately equal numbers were CPU and lock bound). Our model's average error ranges between 0-25%, with its worst error on lock-bound mixtures. Our I/O model produces error that is less than 1% on average (mostly because our models for sequential logging and log rotation are quite accurate, as we show later in this section). Note that the LR-based CPU model performs much better on CPU bound workloads, and that the LR-based page flush model does better on the I/O bound workloads, but in all cases our model does better. Decision tree regression performed poorly across different workloads. This is expected, as decision trees are not capable of much extrapolation beyond the range of their training data, which is a requirement for accurate estimation of maximum throughput.

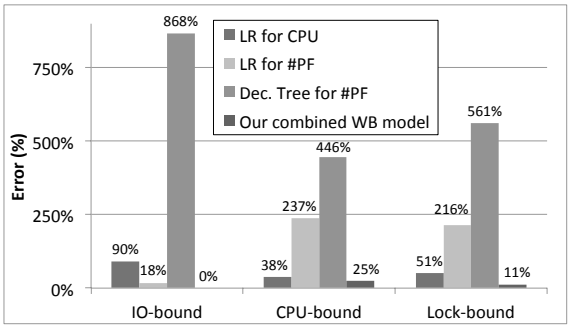

Figure 3: Max throughput prediction.

We also looked at the ability of our system to predict the bottleneck resource. On 19 out of 20 test data sets, we correctly predicted the true bottleneck. In the one case where we mispredicted, our model said the workload would be CPU bound, with a maximum throughput of 1578 TPS, while in reality the system was lock-bound and was able to run 1740 TPS (our locking model estimated the system would be able to run 1759 TPS). Hence, we underestimated the actual maximum TPS of the system by about 10%.

Although this is not a comprehensive study of system provisioning, the results of these experiments can also be used to look at our effectiveness at estimating whether a system is provisioned to handle a certain level of peak load. On these 20 data sets, our worst max throughput error is an overestimate of 44%, with mean error of 12%. In comparison, the worst case performance of the regression based models is 180% (49% mean) for the CPU model and 762% (200% mean) for the page flush model. Analyzing the detailed under and over-estimation of the maximum throughout by the other models, this results suggests that an administrator trying to estimate the resource requirements of a database system by linearly extrapolating CPU or I/O will over-provision the true resource needs of the system anywhere between 9× to two orders of magnitude more than if he used our models for provisioning. Similarly, he might under-provision the system between 8× to 10× using these models than if he used our combined models.

8.2 Attribution and Multi-tenancy Scenarios

In addition to estimating maximum throughput, our models allow us to answer questions about how much a given transaction type or tenant (e.g. in a database-as-a-service [16]) contributes to the overall load of a system. This is important for an administrator, e.g. to understand the impact on performance of migrating a database to a server. In this section, we present a few preliminary experiments showing the effectiveness of our models in two such scenarios.

In the first scenario, we consider two TPC-C workloads, $W1$ and $W2$, each running a single transaction together on a single machine. $W1$ runs the "new order" transaction at 275 TPS and $W2$ runs the "stock level" transaction at 225 TPS. Together, they result in 2.3 MB/sec of writes. Suppose the administrator wants to estimate how much the disk usage will be reduced if $W1$ is moved to another machine. We predict $W1$ alone will write 1.7 MB/sec, and in reality it writes 1.5 MB/sec, representing a 13.3% overestimation on our part. Note that a simple linear estimate would be unlikely to accurately answer such types of question, because as shown in Section 8.4 disk I/Os are highly nonlinear.

In the second scenario, we consider a full TPC-C workload running concurrently with a full Wikipedia workload, together on a single machine. This is meant to be representative of a multi-tenant database-as-a-service. We run Wikipedia at 200 TPS, and TPC-C at 100 TPS, each in isolation, and then predict the combined load when they are run together on the same machine. In this case, we are able to predict the CPU utilization of the combined workload to within 13% relative error (our pred = 4.6% CPU, actual = 4.0%), the disk write throughput to within 3% (our pred = 2.97 MB/s, actual = 3.01 MB/s), and the combined read throughput to within 97% (our pred = 330 KB/s, actual = 160 KB/sec). Although the relative error of our cache model is high (for reasons explained in Section 8.4 below), the absolute error is quite small; our later experiments show that our read model does quite well when the buffer pool is more heavily utilized.

These results show that our models are useful in several practical scenarios of great importance to database administrators. We now turn to detailed evaluations of our individual models.

8.3 Verifying Transaction Type Clustering

To test our transaction clustering, we ran our clustering algorithm described in Section 3.2 on a small log of SQL queries from Wikipedia and TPC-C. For TPC-C, we used the log of 7,361 transactions and for each we estimated the number of rows

read and written (i.e., sum of rows updated, inserted or deleted) from each of the tables. Since the TPC-C benchmark has 9 tables, our dataset consisted of 18 numerical features. Applying Weka's (`www.cs.waikato.ac.nz/ml/weka/`) implementation of the DBSCAN clustering algorithm (with density parameters of $\epsilon = 0.9$ and minimum number of points 6) on this dataset, we obtained 5 clusters, precisely matching TPC-C transaction types.

For Wikipedia we used a log of 1,000 transactions and generated features corresponding to Wikipedia's 12 tables. We achieved the same accuracy using the same parameters to DBSCAN. The extracted clusters again matched the semantic notion of different transactions in Wikipedia workload, namely 'Add to Watch List' (type 1), 'Remove from Watch List' (type 2), 'Update Page' (type 3), 'Get Page Anonymous' (type 4), and finally 'Get Page Authenticated' (type 5). In both cases, we tried a range of parameters to DBSCAN and found that it was relatively insensitive to their values.

8.4 Verifying Resource Models

We now look at the performance of our individual resource models and their ability to predict resource utilization over a range of inputs. As with maximum throughput, we compared to two baselines: 1) linear regression and 2) linear regression with clustering.

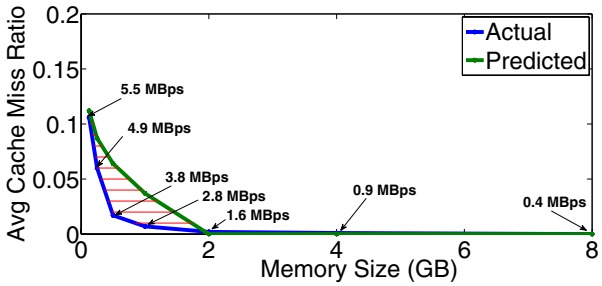

Figure 5: Cache-miss rate for memory provisioning and physical read prediction using the original TPC-C benchmark.

Disk Write Prediction. We used the datasets described at the beginning of Sec 8 to compare the accuracy of our model for predicting the data flush rate (described in Section 4.1). Figure 6 shows the results. Here, we used a limited range of training data from TPC-C where the average TPS was 900 and the fraction of 'New Order' transactions was less than 0.2 and predicted the number of page flushes on 9 different mixtures. These mixtures had an average TPS of 100 and varied the fraction of 'New Order' transactions roughly from 0% to 90% ('Payment' transactions were added to compensate, falling from roughly 90% to 0% of the load). In this experiment, the mean relative error of our white-box page flush model over the entire range was only 16%, while the errors for other black-box models ranged from 180% for LR without our transaction clustering to 3346% for decision tree regression. This dramatic difference confirms our theory that the mixture and TPS of training and test data are different (e.g. in what-if scenarios), our white-box models are significantly superior. LR-based methods are more accurate compared to others, due to their extrapolation capabilities, which methods like decision trees do not have.

We repeated similar experiments over many other ranges of data, both for TPC-C and Wikipedia. Due to lack of space, we have only summarized a few of them in Figure 4. In Figure 4, we use the notation '$D\ n_1 - n_2$' to denote the workload D (either TPC-C or Wikipedia), with a narrow range of different mixtures in the training set at the average TPS of n_1, and a complete range of different mixtures in the test data with an average TPS of n_2. In this figure, for each workload, we show different scenarios. In the left subfigure, the training TPS is close to the testing TPS (e.g., *wiki* 460 – 450 and *tpcc* 500 – 400): here the gap between the accuracy of our white-box model and that of the black-box models is considerably smaller (especially for LR with classification and decision trees), confirming our theory that we can rely on these black-model models when we have observed similar training data to the test range that we want to predict. In the right subfigure, the training TPS is either much lower than the test TPS or vise versa (e.g., *wiki* 100 – 900 and *wiki* 900–100). In these cases, the gap between our white and black box models widens dramatically: DBSeer out performs other methods by at least a factor of 4 in all cases, and up to 100× or more versus Neural Nets on Wikipedia. This gap is larger for Wikipedia mainly due to its higher degree of non-linearity in load (e.g. articles accessed/ edited almost follow a power law) compared to TPC-C. This experiment confirms our theory that white-box models can much more accurately model disk I/O when test data is far from training data.

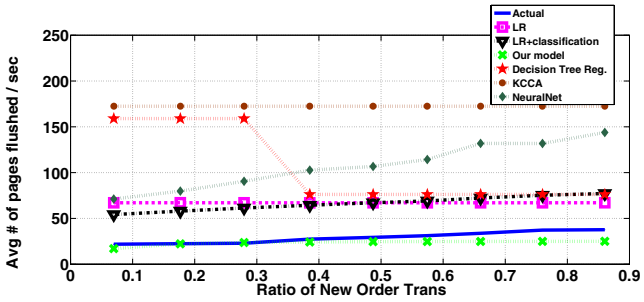

Figure 6: Predicting average page flush rate.

Disk Read/Memory Requirement Prediction. To validate our model of cache misses (which is used to relate buffer pool size to I/O), we used the original TPC-C mixture running at random TPS rates, and ran experiments for a wide range of buffer pool sizes. During each experiment, after the warm-up period of the database, we recorded the average cache-miss rate as well as the average number of physical reads per second and compared these actual numbers to the numbers predicted by our model. Figure 5 summarizes the result of this experiment, where we report the actual cache miss rate as well as that predicted by our model. As shown in this picture, our model slightly overestimates the cache miss rate. The error in memory provisioning for any cache miss rate (or equivalently, any number of physical reads) is the horizontal distance between the actual curve and our prediction curve, as illustrated with red horizontal lines in Figure 5. For instance, in order to ensure a cache miss rate of less or equal to 0.07, we need 224MB of buffer pool while our model's suggestion for the buffer pool size is 442MB, i.e. an over-provisioning. Our model follows the actual curve more closely at higher rates (i.e., above 0.06) and also at lower cache-miss rates (i.e., closer to zero) which is the ideal case where enough RAM could be allocated to dramatically reduce the amount of physical reads (and is where one would expect most OLTP databases to operate). Finally, the numbers with arrows show average disk reads in MB per second for each buffer pool size, showing a prediction of physical reads with less than 4.6MB average error.

Predicting CPU and other Linear Metrics. As mentioned in Section 6, CPU can be easily predicted using linear regression over the number of transactions from each type. We have validated the linearity of CPU across many different mixtures and TPS rates, both for TPC-C and Wikipedia. Here, due to space constraints, we only summarize a few experiments which are representative of the accuracy of linear regression for CPU prediction.

Fixed Mixtures. In Figure 7, we show the results of an experiments where we train on a Wikipedia workload of 1K articles, with a TPS

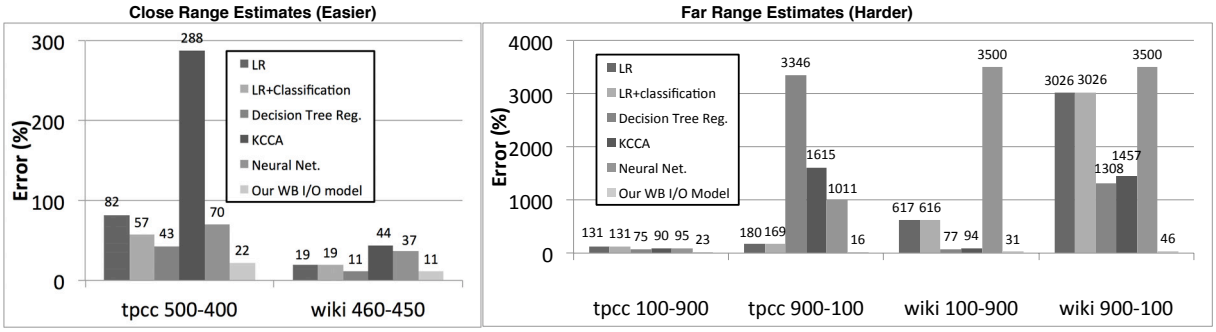

Figure 4: Disk write flush rate prediction for Wikipedia and TPC-C (percentage error) .

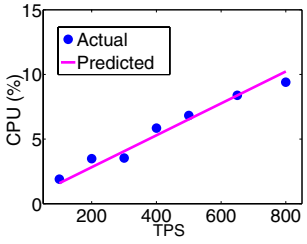

Figure 7: Predicting CPU, when varying the TPS for the Wikipedia database.

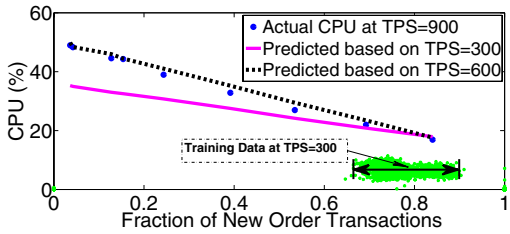

Figure 8: Predicting CPU, when varying both mixture and TPS.

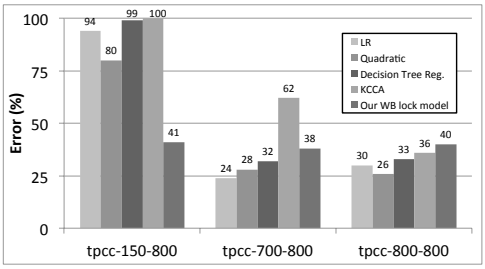

Figure 9: Predicting total lock wait times.

of 100 to 300, and then test it by predicting the CPU when increasing the TPS from 100 to 800. Our mean absolute error was 0.5 (in absolute CPU percentage) and mean relative error was 10.5%. A similar experiment for Wikipedia with 100K articles (training on a TPS between 100 to 300 and testing for CPU ranging between 400 to 800) leads to a mean absolute error of 1.5 and a mean relative error of 14.5%. For each TPS, we have shown the average CPU usage for more readability.

Varying both Mixture and TPS. In Figure 8, we report a set of experiments where we train on a very limited range of different mixtures at around TPS=300 (shown in green) and predict the average CPU usage through a much wider range of different mixtures and at a different TPS, namely at TPS=900. The mean absolute error (MAE) across these different mixtures is 6.3 and mean relative error (MRE) is 16.7%. In the same figure, we have also reported a similar set of experiments but using a training data at TPS=600. In this case, the error is lower (MAE=1.5 and MRE=5.8%). This is because the training data at 600 TPS is more similar to the test data at 900 TPS, compared to a training data of 300 TPS.

We have run similar experiments validating the linearity of network I/O for OLTP workloads, the number of logical reads and the amount of log I/O. Due to space limitations, we omit these results.

Verifying the Contention Model. The effectiveness of our lock contention model was demonstrated in Section 8.1 where we showed that it could predict the maximum throughput of lock-bound workloads, where the CPU and I/O were far from saturation However, we have also measured the accuracy of our lock model's prediction in isolation. We ran several experiments with TPC-C in which we increased the TPS by increasing the number of 'New Order' transactions (which typically cause a lot of contention due to their write-intensive nature). Figure 9 reports the error of our white-box model as well as a few other black-box models, where we use the same '$D\ n_1 - n_2$' notation as above. The same trend can be seen here where the accuracy of the white-box dominates black-box models as the ranges of training and testing data widen. Interestingly, when the training data is similar enough to the testing data, the LR and quadratic curves are more accurate than our white-box model. This is expected, as the white-box model makes a number of simplifying assumptions, and therefore only learns a few parameters from the data. This is why the accuracy of the white-box model does not improve, even when presented with more similar training data. The overall conclusion here is the same as before, however: when answering what-if questions or predicting performance regimes far from previously seen settings, use the white-box model. In other cases, quadratic fitting or LR seem are viable choices.

8.5 Generality to Other DBMSs

The focus of our paper is to show that, for a chosen DBMS, we can build very accurate models—hence our focus on MySQL. However, an interesting question is if and to what degree our MySQL-specific models apply to other DBMSs. In this section, we present some preliminary results on evaluating our models for predicting disk writes on another popular open-source DBMS: PostgreSQL. We apply the same models that we introduced in this paper (which target MySQL) to training and testing data collected from a PostgreSQL installation (on similar hardware). The white-box model of disk writes that we introduced in Section 4.2 was based on some general characteristics of traditional DBMSs as well as the specific flushing policy of MySQL. Thus, by using the same model for PostgreSQL we study the effect of different flushing policies in these two DBMSs on the accuracy of our predictions for disk writes in PostgreSQL.

We ran several experiments with the TPC-C workload, where we varied both the TPS range and the transaction mixtures. We collected similar combinations of training and testing data for PostgreSQL to those used for the MySQL experiment of Figure 6. In this experiment, our predictions for disk flush rate in Postures had an average relative error of 19.6%. Recall, from Section 8.4, that our error in predicting the same metric for MySQL in a similar setting (i.e., Figure 6) was 16%. Thus, the error of our flush rate predictions for Postures are comparable to that of our predictions for MySQL.

Although our experiments for PostgreSQL are not as comprehensive as those for MySQL, given that our models were originally designed for MySQL, even these preliminary results are quite promising, suggesting that with modest amount of additional modeling, accurate results might be achievable for other DBMSs as well.

9. RELATED WORK

Lock Contention. There have been numerous theoretical attempts at analyzing the performance of different variations of 2-phase locking. The pioneering paper by Thomasian [21] is the theoretical basis of our lock contention model which we have modified in a number of ways as described in Section 5.

Most of this literature uses either analytical modeling or random number-driven simulations [17] (see [22] for a survey). Because of the difficulty of collecting suitable measurements, there have been only a few studies which use trace-driven simulations. Also, to our knowledge no other work besides ours has implemented or evaluated an analytical model for predicting a real-world, deployed database system. Perhaps the closest to ours is an early work [19] where they have examined the degree to which three real workloads in IBM DB2 conformed to the assumptions commonly made in the literature, i.e. without predicting the contention degree of those workloads. In [7] the authors validated their model using an *abstraction* of TPC-C but on their own database *simulator*.

Performance and IO Prediction. There has been prior research on performance modeling for a database system, but mostly for OLAP workloads [12, 3, 8] and their optimal scheduling [14]. OLAP workloads are fundamentally different than OLTP. In OLAP, queries are mostly long-running and read-only with significantly lower degrees of concurrency. Therefore, the lock contention is typically not a problem in OLAP, e.g., the maximum concurrency level in [8] is 5, compared to thousands in an OLTP workload. Also, performance requirements in OLTP tend to be more stringent (e.g., a few milliseconds). Similarly, the authors in [10] only address scenarios with no concurrency, i.e. queries running one at a time. To the best of our knowledge, no prior work has tackled this problem for highly concurrent OLTP databases. The only exception is [2] where some preliminary results on CPU and IO prediction have been reported, where linear regression (LR) is shown to outperform Gaussian regression. The IO models presented in this paper improve on LR predictions by up to 71× in terms of accuracy. Moreover, lock contention, RAM, or other resources have not been addressed in [2].

Progress Indicators [13] also consider performance prediction but they need to continuously monitor the system to refine their previous estimates and thus, are more suitable to long-running queries. Finally, an early work has proposed models for buffer pool and bottleneck analysis [18], but they depend on many detailed specifications of the hardware and have been validated only in simulation.

In [16], we discussed our overall *vision* in the DBSeer project and enumerated important problems that arise in the context of a database-as-a-service and that require performance prediction, whereas in this paper we present our solution for performance prediction in a transactional database. In [5] we studied OLTP models for CPU, I/O, and RAM but in a very different setting, where the focus was consolidating existing workloads (without changing their rates or mixtures), rather than predicting the performance of each workload across variations in transaction rate and mixture. In the current paper, we do not consider consolidation but focus on more sophisticated resource models that relax several assumptions of our previous work: 1) we do not need to observe each workload in isolation, 2) our models in this paper are designed to make predictions even when the mixtures and rates that we predict for were never observed during the training/data collection phase.

10. CONCLUSION

In this paper, we presented a series of predictive models for resource utilization in OLTP databases as well as an experimental evaluation of those models on MySQL. For some resources, like CPU, Network, and log writes, we found that black-box models based on regression work quite well, yielding relative errors of just a few percent even when predicting resource utilization at very different rates or on different transaction mixtures than where they were trained. For other resources, e.g., RAM utilization, page flushes due to log recycling and buffer pool evictions, and database locks, white-box models that model the database are needed when making predictions about system performance over a wide range of transaction rates different than those observed at training. Our white-box models consist of a iterative model for log recycling, as well as a number of optimizations to an existing lock-contention model. Overall, our results evaluating these models are encouraging, yielding relative errors at predicting maximum throughput of a system ranging from 0–25% on a TPC-C like workload, with improvements of up to two orders of magnitude versus black box models in some cases.

Acknowledgments. This work was supported by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, as well NSF grant IIS-1065219. We are grateful to Evan Jones, Y. C. Tay, Djellel E. Difallah, Purnamrita Sarkar and the anonymous reviewers for their comments.

11. REFERENCES

[1] OLTP Benchmark. http://oltpbenchmark.com.
[2] M. Ahmad and I. T. Bowman. Predicting system performance for multi-tenant database workloads. In *DBTest*. 2011.
[3] M. Ahmad, et al. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT*. 2011.
[4] F. R. Bach and M. I. Jordan. Kernel Independent Component Analysis. *Journal of Machine Learning Research*, 3:1–48, 2002.
[5] C. Curino, et al. Workload-aware database monitoring and consolidation. In *SIGMOD*. 2011.
[6] C. Curino, et al. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 2010.
[7] P. Di Sanzo, et al. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *ICPE*. 2010.
[8] J. Duggan, et al. Performance prediction for concurrent database workloads. In *SIGMOD*. 2011.
[9] M. Ester, et al. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*. 1996.
[10] A. Ganapathi, et al. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*. 2009.
[11] J. Gray, et al. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2), June 1996.
[12] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*. 2008.
[13] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL progress indicators. In *EDBT*. 2006.
[14] A. Mehta et al. BI batch manager: a system for managing batch workloads on enterprise data-warehouses. In *EDBT*. 2008.
[15] B. Mozafari, et al. DBSeer: Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. Technical report, MIT, 2013.
[16] B. Mozafari, C. Curino, and S. Madden. Resource and performance prediction for building a next generation database cloud. *CIDR*, 2013.
[17] I. K. Ryu and A. Thomasian. Analysis of database performance with dynamic locking. *J. ACM*, 37, 1990.
[18] S. Salza and M. Renzetti. Performance Modeling of Paralled Database System. *Informatica (Slovenia)*, 22(2), 1998.
[19] V. Singhal and A. J. Smith. Analysis of locking behavior in three real database systems. *VLDBJ*, 6, 1997.
[20] Y. C. Tay, R. Suri, and N. Goodman. A Mean Value Performance Model for Locking in Databases: The Waiting Case. In *SIGMOD*. 1984.
[21] A. Thomasian. On a More Realistic Lock Contention Model and Its Analysis. In *ICDE*. 1994.
[22] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *J. ACM*, 40, 1993.