Modeling and Exploiting Query Interactions in Database Systems

Mumtaz Ahmad
University of Waterloo
m4ahmad@uwaterloo.ca

Ashraf Aboulanaga
University of Waterloo
ashraf@cs.uwaterloo.ca

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Kamesh Munagala
Duke University
kamesh@cs.duke.edu

ABSTRACT

The typical workload in a database system consists of a mixture of multiple queries of different types, running concurrently and interacting with each other. Hence, optimizing performance requires reasoning about query mixes and their interactions, rather than considering individual queries or query types. In this paper, we show the significant impact that query interactions can have on workload performance. We present a new approach based on planning experiments and statistical modeling to capture the impact of query interactions. This approach requires no prior assumptions about the internal workings of the database system or the nature or cause of query interactions, making it portable across systems. As a concrete demonstration of the potential of capturing, modeling, and exploiting query interactions, we develop a novel interaction-aware query scheduler that targets report-generation workloads in Business Intelligence (BI) settings. Under certain assumptions, the schedule found by this scheduler is within a constant factor of optimal. An experimental evaluation with TPC-H queries on IBM DB2 demonstrates that our scheduler consistently outperforms (up to 4x) conventional schedulers that do not account for query interactions.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

The typical workload in a database system consists of a mixture of queries of different types, running concurrently and interacting with each other. The interaction among queries can have a significant effect on performance. Hence, optimizing performance requires reasoning about *query mixes* and their interactions, rather than considering individual queries or query types. Current trends like server consolidation and offering database applications as a service [5] are causing database systems to support more heterogeneous clients concurrently, leading to richer query mixes. For example, Salesforce.com supports many concurrent clients running customer relationship management applications on the same backend database.

A query Q_1 that runs concurrently with another query Q_2 could impact Q_2's performance in different ways, either negatively or positively. For example, the resource demands of Q_1 and Q_2 could interfere with each other, with the interference happening at one or more of different physical resources like CPU, L1 or L2 cache, memory, and I/O bandwidth. Moreover, the queries can interfere at internal resources inside the database system such as latches, locks, and buffer pools. In such cases, the presence of more concurrent instances of Q_1 will degrade Q_2's performance significantly. On the other hand, queries running concurrently in a mix may positively affect each other. For example, Q_1 may bring data into the buffer pool that is then used by Q_2.

Previous work related to query interactions – e.g., on lock or buffer-pool contention [14, 15], and on multi-query optimization [25] – is highly scattered. We are not aware of any work that considers query interactions in a general way that encompasses the various types of interactions that occur in modern database systems. This lack of attention is surprising because interactions are a major cause of performance problems in database systems, and database administrators spend many hours trying to track them down. In this paper, we will show the significant impact that query interactions can have on database performance.

A major hurdle posed by query interactions is in finding effective ways to capture and model them. As we hinted above, there is a large spectrum of possible causes for interactions that includes resource-related, data-related, and configuration-related dependencies. Sometimes, interactions are benign. However, depending on the system setting, the effect of interactions can vary all the way from severe performance degradation to huge performance gains. Furthermore, an interaction that occurs when a database system runs on one hardware configuration may not happen when the same system runs on a different hardware configuration.

The implication of these challenges is that the analytical cost models used today by database query optimizers to cost query plans will not work for modeling interactions. (Current cost models work on a per query plan basis, and are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'08, October 26–30, 2008, Napa Valley, California, USA.
Copyright 2008 ACM 978-1-59593-991-3/08/10 ...$5.00.

very inaccurate at estimating the overall behavior of multiple concurrent query plans.) To make the conventional approach work for modeling query interactions, we will need to develop models for the complex internal behavior of each distinct database system, and how this behavior depends on hardware characteristics, resource allocation, and data properties; a seemingly impossible task.

In this paper, we propose an entirely different and practical approach to capture and model interactions. First, we measure the impact of interactions in terms of how they affect the average completion time of queries. Completion time is an intuitive and universal metric that is oblivious to the actual cause of the interaction. Second, we propose a proactive *experiment-driven* approach to tease out the significant interactions that exist in a query workload. This approach is based on running a small set of carefully-chosen query mixes from the workload, and measuring how the average completion time of different queries is affected by running them in a mix instead of in isolation. We will show that most significant interactions can be captured in practice by "sampling" very few mixes.

While the experiment-driven approach has to be repeated for each new database and hardware setting, it has two important advantages: (i) it works independent of the root cause of interaction because the effect of any interaction will be captured in the monitoring data collected from the experiments; and (ii) it supports incremental update as query workloads evolve over time, as well as on-line maintenance based on the monitoring data available when query mixes run in the production setting. In this paper, we will show how the data generated from the experiments can be used to track significant interactions – e.g., which query types when present in a mix can degrade the presence of a specific query type the most? – and also for generating performance models for tasks like smart query scheduling.

Third, we provide an end-to-end use case that shows how interactions can be modeled and exploited to gain huge performance improvements in database systems. Our use case considers the problem of scheduling large batches of queries in Business Intelligence (BI) settings [21]. In particular, we have developed a query scheduler, called *QShuffler* (for *Query Shuffler*), that focuses on *throughput-oriented* workloads like those encountered in report generation systems. In such systems, there is a fixed number of report types that a user can request, but the reports requested during a certain period vary depending on user activity. Depending on user activity, multiple reports may be requested over a short period of time. The goal of the system is to minimize the *total completion time* for generating all the reports (i.e., to maximize throughput). For example, we may have a batch of 100 reports that need to be generated every night. In this case, the response time of individual queries is not important as long as all the queries are executed within the specified time window. BI systems like Cognos [9] and Business Objects [7] are examples of such systems.

Concretely, the goal of QShuffler is to schedule appropriate query mixes for a given query workload W to minimize W's total completion time. We show that schedulers used in commercial and research database systems today (e.g., first come first serve, shortest job first) rely on the characteristics of individual queries, and can produce suboptimal schedules when significant inter-query interactions exist. By taking into account the interactions among different queries in a query mix when making scheduling decisions, QShuffler can provide performance improvements up to 4x over conventional schedulers. Under heavy load, interaction-aware query scheduling can turn an otherwise unresponsive system into one that processes its workload in a timely fashion.

Apart from understanding query interactions, QShuffler's performance gains come from a novel algorithm for scheduling a large batch of queries. This algorithm uses a linear-programming-based formulation of the scheduling problem. Given accurate performance models for estimating query completion times in the presence of interactions, this algorithm is guaranteed to produce a schedule that is within a constant additive factor of the optimal schedule. However, we face a tradeoff here because increases in model accuracy come at the cost of observing the performance of more query mixes through experiments. Fortunately, QShuffler's scheduling algorithm is very robust to model inaccuracies. As long as the performance models can distinguish the good query mixes from the bad ones, the algorithm can find efficient schedules that are far better than the schedules found by conventional schedulers.

1.1 Summary of Contributions and Roadmap

- **Query interactions:** To the best of our knowledge, this work is the first to capture, model, and exploit query interactions in a general way. In Section 2 we show the significant impact that interactions can have and motivate why modeling them is nontrivial.
- **Experiment-driven modeling:** Section 3 presents our new approach based on planning experiments and statistical modeling to capture the impact of query interactions. This approach requires no prior assumptions about the internal workings of the database system or the nature or cause of query interactions; making it portable across systems.
- **Interaction-aware scheduling:** Section 4 describes our novel scheduling algorithm that is aimed at BI report generation workloads. These are an important class of workloads (see, e.g., [21]), and our work can provide significant performance improvements here.
- **Evaluation:** Section 5 presents an experimental study using TPC-H queries on DB2, showing up to 4x improvements over (interaction-unaware) scheduling algorithms used in database systems today.

2. IMPACT OF QUERY INTERACTIONS

Consider a database system whose workload W comes from applications that generate queries belonging to a fixed set of *query types*. These query types could be different SQL query templates (e.g., SQL queries with parameter markers). For example, in the report generation settings we consider, each client application is responsible for generating one or more reports, where each report contains specific SQL templates that are instantiated with parameter values at run time. Different parameter values give rise to different query results. These templates can be identified from the application source code or by monitoring application execution.

Let $Q_1, Q_2, \ldots, Q_T$ be the T query types in a database system whose multi-programming level (MPL) is M. The MPL represents the number of queries that execute concurrently in the system at any time. A set of queries that execute concurrently in the system is referred to as a *query mix*. Query mix m_i can be represented as a vector $\langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$,

Symbol	Description
M	Multiprogramming level
T	Number of query types
Q_j	Query type j
t_j	Average execution time of a Q_j query when running alone
$m_i = \langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$	A query mix, m_i, with N_{ij} instances of each query type j
A_{ij}	Average completion time of a Q_j query when running in mix m_i
W	Workload to be scheduled

Table 1: Notation used in the paper

where N_{ij} is the number of instances of query type Q_j in m_i, and $\sum_{j=1}^{T} N_{ij} = M$. Table 1 summarizes our notation.

In order to understand different queries can interact and impact each other, we take the concrete case of TPC-H, a decision-support benchmark. In TPC-H there are 22 query types. Table 2 shows the run time of the 12 longest running TPC-H queries, on a 1GB database, when they run alone in the system, which we denote by t_j.[1] Next, Table 3 shows two mixes consisting of the 6 longest running query types from Table 2 with an MPL $M = 30$. For each mix, the table shows the query frequencies, N_{ij}, and the average run time in seconds for each query type, A_{ij}. The behavior of queries changes from mix to mix depending on interaction among queries. For example, we run the same number of instances of Q_7 in both mixes, but A_{ij} for Q_7 in m_2 is almost twice the A_{ij} for Q_7 in m_1.

Further, Table 4 shows the run time of the above 6 longest running TPC-H queries on a 10GB database when they are alone in the system. Table 5 shows two query mixes for this setting with MPL $M = 10$. Mix m_1 in this table presents an interesting case of "positive" interaction for Q_7. The run time of Q_7 when it is alone in the system, t_j, is 102.06 seconds. In m_1, we observe an average run time for Q_7, A_{ij}, of 72.66 seconds per query. Thus, Q_7 *benefits* from being run in this mix. Note that what we are seeing here is not the typical benefit of concurrent execution, where the individual response times of a set of queries will increase but the total time for executing the whole set of queries is less than executing the queries one at a time. Instead, what we are seeing is that a single instance of Q_7 is taking, on average, less time to finish in a mix than if it was running alone.

Thus, interactions in mixes of concurrently running queries can be both negative (where $A_{ij} > t_j$) and positive (where $A_{ij} < t_j$). Interestingly, Mix m_1 in Table 5 shows both positive and negative interactions: while Q_7 benefits from running in the mix, the average completion times of the three other concurrent query types are degraded severely.

Modeling query interactions can be difficult, as we show in the following example. Figure 1 shows three-way interactions for mixes consisting of Q_1, Q_7, and Q_{21} on a 1GB TPC-H database, where the MPL $M = 30$. To simplify the presentation, we fix N_{ij} for Q_{21} at 6, 9, and 12, and we vary the number of instances of the other two query types and observe the effect on the average run time A_{ij} of Q_1. When $N_{ij} = 6$ for Q_{21}, A_{ij} of Q_1 first decreases and then increases. For $N_{ij} = 9$ and 12, the effect on A_{ij} of Q_1 is even more complicated. Thus, even for this apparently simple case of three-way interaction, the behavior is non-trivial to model.

[1]See Section 5 for details of our experimental setup.

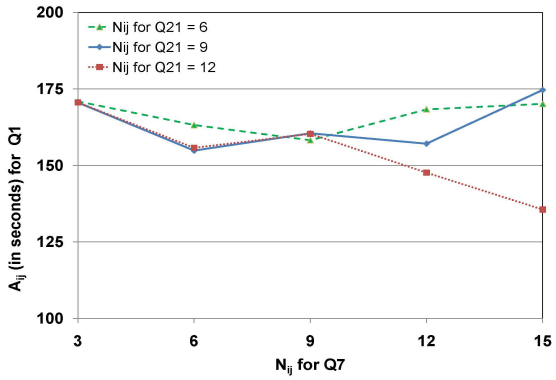

Figure 1: Effect on Q_1 of different mixes

3. EXPERIMENTAL MODELING

In the previous section, we showed the impact that interactions can have on query completion times. We now consider how this impact can be estimated so that we can answer questions like: which queries have the largest impact on query Q_j's average completion time when running concurrently with Q_j? Or, consider the mix m_1 in Table 3. Mix m_1 contains 11, 8, 3, 2, 2, and 4 instances respectively of query types Q_1, Q_7, Q_9, Q_{13}, Q_{18}, and Q_{21} from TPC-H. Given this information, can we estimate the average completion time, A_{ij}, of each query when running in this mix?

One approach that can be tried is to develop analytical formulas to estimate A_{ij} for mixes. Historically, analytical formulas have been used successfully by database query optimizers to estimate the execution cost of query plans. For example, standard database textbooks give the formula to estimate the cost of a block nested loop join based on the number of blocks in the inner and outer tables and the amount of memory given to the join.

However, developing accurate analytical formulas to estimate the properties of query mixes will require a detailed understanding of all possible causes of inter-query interactions. Interactions can arise from a variety of causes: resource limitations, locking, configuration parameter settings (including misconfigurations), properties of the hardware or the software implementation, correlation or skew in the data, and others. This space of potential causes is large, not fully known ahead of time, and can vary from one database system to another. Thus, general-purpose analytical formulas are hard to develop. We propose a different approach.

Our approach is based on running a small set of carefully-chosen query mixes from the possible input workloads; to collect *samples* of the form shown in Table 3. Each sample gives a measure of how the average completion time of different queries is affected by running them in a specific mix. The full set of collected samples can be analyzed to identify various interactions (and non-interactions). In addition, statistical models can be trained from the collected samples, and then used to estimate the average completion of a query when it runs in any given mix.

This approach does not depend on what the root causes of interaction are because the effect of any significant interaction will show up in the samples. The rest of this section gives the full details of our approach. The effectiveness of this approach will be shown empirically in Section 5.

Sampling: The more critical aspect of our approach is identifying which samples to collect. Each sample is collected by

Query Type	$Q1$	$Q9$	$Q21$	$Q18$	$Q13$	$Q7$	$Q6$	$Q20$	$Q8$	$Q3$	$Q10$	$Q5$
Run Time t_j (sec)	10.07	9.66	7.3	7.12	6.12	5.76	4.77	4.48	4.15	3.41	2.65	2.60

Table 2: Average run time of different TPC-H query types on a 1GB database

	$Q1$		$Q7$		$Q9$		$Q13$		$Q18$		$Q21$	
Mix	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}
m_1	11	143.9	8	144.6	3	211.2	2	97.8	2	149.8	4	127.5
m_2	2	361.7	8	298.6	1	476.0	18	121.2	0	0.0	1	231.2

Table 3: A_{ij} for different query types in query mixes on a 1GB database

scheduling an *experiment* where a selected query mix m_i is run purely to observe the average query completion times (A_{ij} values) of the queries contained in m_i. In our current implementation, when the database system is running a query mix as part of an experiment, it does not process the production workload. Thus, experiments expose a tradeoff:

- On one hand, more experiments bring in more samples, which will lead to a better understanding of query interactions as well as more accurate models to estimate A_{ij} values for given mixes.
- On the other hand, experiments add to the system overhead, so we want to minimize the number of experiments needed.

In our empirical evaluation, we consider how many samples (experiments) are needed to produce fairly-accurate models for estimating A_{ij} values. (Our sampling techniques will be described momentarily.) We will show that these values can be estimated with reasonable accuracy from few samples (50-60). In particular, the accuracy obtained from these samples is good enough for our query scheduler to produce efficient schedules that outperform the schedules produced by conventional schedulers.

While these results may seem surprising at first, it should be understood that our scheduling algorithm performs well as long as it can distinguish the bad mixes (where the performance of one or more queries is degraded severely) from the good ones. Statistical models need far fewer samples to separate the bad mixes from the good ones than what they need to predict all A_{ij} values with high accuracy. An analogy from query optimization is relevant here. Cost models used by query optimizers can be notoriously bad at estimating absolute plan completion times, but they have been successful because of their ability to distinguish the bad plans from the good ones.

A straightforward approach to pick experiments is to sample randomly from the space of possible mixes. As an attempt to selectively cover different parts of the space of mixes with few experiments, we developed a new sampling approach called *corner, diagonal, and random (CDR) sampling*. CDR sampling works as follows.

- For MPL M, we start by running T experiments where we sample the "corner" points of the space, i.e., the mixes $\langle M, 0, \ldots, 0\rangle$, $\langle 0, M, \ldots, 0\rangle$, $\ldots$, $\langle 0, 0, \ldots, M\rangle$.
- Next, we sample "diagonally". We first run the mix with equal number of occurrences of each query type, i.e., $\langle \frac{M}{T}, \frac{M}{T}, \ldots, \frac{M}{T}\rangle$. Then, we take a fixed number of random samples from the space of possible mixes, with a constraint that there has to be at least k instances of each query type. k is varied across a small range of values in $1, \ldots, \frac{M}{T} - 1$.
- Finally, we take some samples completely at random (like random sampling) from the full space of mixes.

After collecting the samples, we can analyze this data to identify important interactions (like we did in Section 2) or to train statistical models.

Statistical Models: In this paper, we consider three types of statistical models: *linear models*, *quadratic models*, and *regression trees*. The statistical model is a *pluggable component* of our framework, so any other appropriate statistical model can be used. However, in our empirical evaluation (Section 5) we have found that these simple models consistently give accurate and robust estimates.

A linear model uses the following structure to compute $\widehat{A_{ij}}$, the estimate of A_{ij} for mix i and query type j:

$$\widehat{A_{ij}} = \beta_0 + \sum_{k=1}^{T} \beta_k N_{ik}$$

A quadratic model uses a second-degree polynomial in N_{ij} to compute $\widehat{A_{ij}}$ as:

$$\widehat{A_{ij}} = \beta_0 + \sum_{k=1}^{T} \beta_k N_{ik} + \sum_{k=1}^{T}\sum_{l=1}^{T} \beta_{kl} N_{ik} N_{il}$$

The β parameters in both models are regression coefficients that will be estimated while learning the model from data, e.g., using the popular method of least squares estimation.

Regression trees are piecewise regression models [18, 34]. Each piece in such a model corresponds to a partition of the space of mixes of the form $N_{ij} \leq const$. Partitioning is carried out recursively, beginning with the full set of samples, and the set of partitions is presented as a binary decision tree. The nonleaf nodes in the tree define the partitioning conditions. Each leaf node L is associated with a constant or a function which is used to predict $\widehat{A_{ij}}$ for all mixes that match the criteria along the path from the root node to L. Efficient software packages are available to learn piecewise constant, piecewise linear, and other types of regression trees from given samples (we use [34]).

Incremental Model Maintenance: One important question is whether the sample collection and model learning has to be done from scratch each time a query type is added or deleted. The answer is no. For example, when a new query type Q is added, all we need are a few new samples with nonzero number of instances of Q. These samples can be used to update the linear regression, quadratic regression, or regression tree models incrementally.

4. QSHUFFLER

In this section, we provide an end-to-end use case that shows how interactions can be modeled and exploited to gain huge performance improvements in database systems. We develop a query scheduler, called QShuffler, that addresses the problem of scheduling large batches of queries in Business Intelligence (BI) settings [21].

Type	$Q1$	$Q9$	$Q21$	$Q18$	$Q13$	$Q7$
t_j (sec)	294.61	578.61	570.37	554.56	101.27	102.06

Table 4: Average run time of different TPC-H query types on a 10GB database

	$Q1$		$Q7$		$Q9$		$Q13$		$Q18$		$Q21$	
Mix	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}
m_1	1	1897.42	2	72.66	5	2919.29	0	0.0	2	1904.12	0	0.0
m_2	0	0.0	0	0.0	0	0.0	1	284.7	9	1053.31	0	0.0

Table 5: A_{ij} for different query types in query mixes on a 10GB database

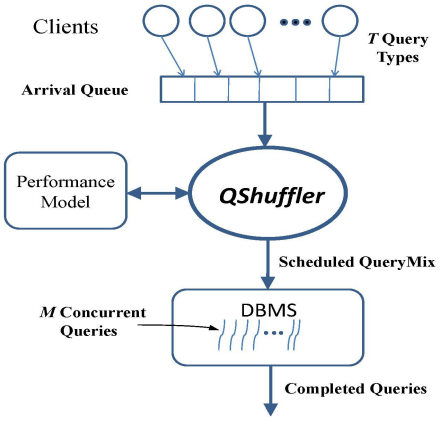

Figure 2: Problem setting

The workload to be scheduled, W, comes from a set of clients (e.g., report-generation applications) that each issues a bounded set of queries belonging to one of the T possible types $Q_1, Q_2, \ldots, Q_T$. The clients place their queries in an arrival queue, and QShuffler schedules queries from this queue. Figure 2 illustrates our problem setting. Let I_j denote the total number of queries of type Q_j in W, thus $|W| = \sum_{j=1}^{T} I_j$. QShuffler has to choose an entire sequence of query mixes to schedule in order to complete W.

The objective of QShuffler is to choose the sequence of query mixes to schedule so that the total completion time of W is minimized (which is equivalent to maximizing throughput). In our report generation scenario, this objective corresponds to producing all the reports requested in a certain period as fast as possible to stay within the available time budget. Since most database systems do not preempt queries once they start, we focus on non-preemptive scheduling. There is work on preemptive scheduling of plan operators (e.g., [6]), but our focus is on scheduling entire queries.

The rest of this section presents our scheduling algorithm. Under certain assumptions, this algorithm generates a schedule whose total completion time is within an additive constant factor of the total completion time of the optimal schedule. The algorithm considers a large set of mixes $X = \{m_1, m_2, \ldots, m_{|X|}\}$ such that the schedule chosen for W will consist of: (i) a subset of mixes selected from X, and (ii) a specification of how the I_j instances of each query type should be executed using the selected mixes. Next, we describe how X is picked and how the schedule is chosen.

The space X of mixes considered: X is a systematic enumeration of a very large subset of the full space of query mixes. For an MPL M and number of query types T, the space of possible mixes is a bounded T-dimensional space. The total size of this space is the number of ways we can select M objects from T object types, unordered and with repetition. This is an *M-selection* from a set of size T, and the number of possible selections is given by $S(T, M) = \binom{M+T-1}{M}$ [26].

If we restrict the space of mixes by assuming that queries of the same type can be scheduled only in batches of size b, then we get a subspace of size $S(T, \frac{M}{b}) = \binom{\frac{M}{b}+T-1}{\frac{M}{b}}$. We use values of $b \in$ [1-10], and set X to be the corresponding subspace of the full space. This strategy has consistently given us very good results.

Linear program to pick a subset of X: QShuffler uses a *linear program (LP)* [28] to pick the subset of X used in the chosen schedule. Intuitively, an LP optimizes an objective function over a set of variables subject to some constraints. The inputs to the LP used by the scheduler consist of the set of query mixes $m_i \in X$ and I_j, $1 \le j \le T$, the total number of instances of each query type Q_j to be scheduled. The LP contains an unknown variable n_i ($n_i \ge 0$) corresponding to each mix $m_i \in X$. n_i is the total time for which queries will be scheduled with mix m_i in the chosen schedule.

The chosen schedule should perform the work required to complete all I_j input instances of each query type Q_j. This requirement can be written in the form of T constraints:

$$\sum_{i=1}^{|X|} n_i \frac{N_{ij}}{A_{ij}} \ge I_j, \quad \forall j \in \{1, \ldots, T\} \tag{1}$$

Recall that N_{ij} denotes the number of instances of query type Q_j in the mix m_i, and A_{ij} denotes the average completion time of a query of type Q_j in m_i. (N_{ij} and A_{ij} are constants that depend only on m_i and Q_j. Section 3 shows how A_{ij} values can be estimated for all the mixes in X.) Suppose the work needed to complete the execution of one instance of Q_j is 1. Then, the total work required to complete the execution of I_j instances of Q_j in the input workload is I_j. $\frac{N_{ij}}{A_{ij}}$ denotes the fraction of this work that gets completed per unit time when mix m_i is scheduled. Thus, $\sum_{i=1}^{|X|} n_i \frac{N_{ij}}{A_{ij}}$ denotes the total work done for Q_j in the chosen schedule, which must not be less than I_j. This reasoning explains the T constraints presented in Equation 1 that the LP should work with.

The objective of the LP is to find the schedule with the minimum total time to completion. Since only one mix will be scheduled at any point in time, the LP's optimization objective can be written naturally as Minimize $\sum_{i=1}^{|X|} n_i$. We can solve the LP using any LP solver; we use the highly-efficient *CPLEX* tool [11]. In the LP solution, some n_i variables will be set to nonzero values and the rest will be zero. Next, we show how the mixes with nonzero n_i are used in the chosen schedule. We have the following lemma.

LEMMA 4.1. *The number of nonzero n_i in the LP solution is at most T, assuming $T \leq |X|$.*

The above lemma follows from linear-programming theory where it is the case that the number of variables set to nonzero values in the LP solution will not be greater that the number of constraints in the LP [28]. Recall from Equation 1 that our LP has T constraints, one per query type.

4.1 Bound on Degradation from Optimal

It follows from Lemma 4.1 that the LP will choose at most T mixes out of the $|X|$ mixes given as input. We can pick any order in which to schedule the chosen mixes. For each chosen mix, the respective n_i value found by the LP gives the total time for which query instances should be executed with that mix. In this way, we can generate a complete schedule from the LP solution.

However, this schedule assumes that we can preempt queries that are running when the time (n_i) assigned to a mix expires; the LP may have chosen to finish running these queries using one or more other mixes. Since instantaneous query preemption is not easy in most database systems (recall Section 2), we need to transform the possibly preemptive schedule generated by the LP to an efficient preemption-free schedule. We give Theorem 4.2 which uses the following notation: (i) Among all mixes with nonzero n_i in the LP solution, let a_j be the maximum average completion time for a query of type Q_j. Let a_{max} be the maximum among all a_j; (ii) Let $OPT = \sum_{i=1}^{|X|} n_i$ be the time to completion computed by the LP for the input query workload.

THEOREM 4.2. *We can produce a preemption-free schedule S whose time to completion for the input workload is at most $OPT + a_{max}T$ if the following assumption holds: reducing the number of query instances in a mix will not increase the average completion time of any query type in that mix.*

Proof Sketch: The assumption in Theorem 4.2 is that interactions among queries should not affect queries positively; which holds if the interactions arise due to software or hardware resource limitations (e.g., locking, I/O bottlenecks). Such interactions are perhaps the most common ones in practice. We can construct a preemption-free schedule S with the property stated in Theorem 4.2 as follows:

1. Pick one among the remaining mixes with a nonzero n_i. Suppose we picked mix m_i.
2. Schedule input query instances with m_i for time n_i.
3. Now wait until all scheduled queries finish. Do not schedule any more queries with m_i. Set $n_i = 0$ for m_i.
4. If there are more mixes with nonzero n_i, go to Step 1.

Note that S does not preempt running queries. For each of the query mixes with nonzero n_i in the LP solution, S takes at most $n_i + a_{max}$ time. (a_{max} more time than n_i will be required if S just scheduled the "longest running query" when time n_i expires for the current mix; S will have to wait until this query finishes before starting the next mix.) Since there are at most T mixes with nonzero n_i (Lemma 4.1), S will finish in time $OPT + a_{max}T$. □

4.2 Robustness of the Chosen Schedule

The LP requires estimates of the A_{ij} values for the mixes in X. Section 3 shows how QShuffler estimates these values through statistical modeling. Even if these models are not very accurate, we have observed that the LP chooses a good subset of mixes. However, the n_i values output by the LP as the time to run each mix become less reliable. In this case, we can use a technique that is slightly different from the one in the proof sketch to generate a preemption-free schedule from the LP solution. While this technique is more robust to modeling errors, the generated schedule does not have a provable bound on total completion time.

Without loss of generality, let the mixes with nonzero n_i in the LP solution be $m_1, m_2, \ldots, m_T$, with respective n_i values $n_1, n_2, \ldots, n_T$. (It does not matter if less than T mixes have nonzero n_i.) We partition the total number of instances I_j of query type Q_j among $m_1, m_2, \ldots, m_T$ in proportion to the fraction of work related to Q_j that the LP solution assigned to each mix, namely:

$$n_1 \frac{N_{1j}}{A_{1j}} \ : \ n_2 \frac{N_{2j}}{A_{2j}} \ : \ \cdots \ : \ n_T \frac{N_{Tj}}{A_{Tj}}$$

Once the entire input workload $I_1, I_2, \ldots, I_T$ has been partitioned among the mixes $m_1, m_2, \ldots, m_T$, these mixes are scheduled in decreasing order of n_i values. For each mix m_i, we schedule queries from the set of instances assigned to m_i until they all complete, then we move to the next mix.

We also considered the possibility of starting queries from the next mix as soon as individual queries from the current mix complete. However, when we start new queries in this manner, we are effectively running new mixes that may not be part of the solution to the LP. These mixes have unpredictable performance and may render our schedule highly sub-optimal; hence our decision to allow scheduled mixes to run to completion.

4.3 Scalability of Linear Programming

Our LP solver can handle a very large number of mixes in X (variables in the LP) in real-time, e.g., an LP with close to 0.34 million variables was processed within 8 seconds. Thus, with an LP, X can be a very large subset of the full space of possible query mixes; increasing the chances of finding the best subset of mixes for the chosen schedule. This benefit would be lost if we use an approach based on *integer programming (IP)* since IP is computationally intractable.

5. EXPERIMENTS

5.1 Experimental Setup

Machine and database: Our experiment were run on machines with dual 3.4GHz Intel Xeon CPUs and 4.0GB of RAM running Windows Server 2003. The database server we use is DB2 version 8.1. We use the TPC-H database with scale factors of 1GB and 10GB. Unless we note otherwise, we always use the 1GB database with the standard TPC-H data generator that generates uniform data. The exceptions to this are Section 5.3, in which we use a 10GB database and Section 5.4 in which we use a data set with a skewed data distribution. The buffer pool size of the database was set to 400MB for the 1GB database, and 2.4GB for the 10GB database. We leave all other tuning parameters of DB2 at their default value. We used the DB2 Design Advisor to recommend indexes for the TPC-H workload. In our experiments, we vary MPL M from 20 to 60. The default MPL for DB2 (or the *number of agents* in DB2 terms) is 200.

Query workload: Our workloads use the 12 longest running TPC-H query types shown in Table 2, with different parameter values for each instantiation chosen according to TPC-H rules. These queries are also identified as long running in the disclosure reports of commercial benchmark runs.

Scheduling algorithms: We experimented with 3 different scheduling algorithms: QShuffler, SJF, and FCFS. Since FCFS scheduling is sensitive to the arrival order of queries, we define different arrival orders for FCFS as follows: We arrange the query types in our workload in the order in which they are specified in the TPC-H benchmark (i.e., $Q1$ first, then $Q3$, ..., $Q21$). We then go through the list of query types in a round robin manner, placing p randomly generated instances of each query type in the arrival queue until all queries are in the queue. The parameter p specifies the degree of skew in the workload arrival order. As p increases, more queries of the same type arrive together.

Performance metric: Our performance metric is *total completion time* for the workload. Since the workload queries are fixed in each experiment, minimizing total completion time is equivalent to maximizing throughput.

5.2 Scheduler Effectiveness

Figure 3 shows the total completion time of the schedule chosen by QShuffler for a workload consisting of 60 instances of each of the 12 longest running TPC-H query types, for a total of 720 queries. The figure shows the completion times on a 1GB database for different MPLs. The completion times increase slightly with MPL, which is expected. However, the slope of the increase is very low, which means that the system is not thrashing and we are within the load capacity of the system for this workload. We measure the performance of other algorithms in terms of *slowdown compared to QShuffler schedule*, defined as: $\frac{\text{completion time of alternate schedule}}{\text{completion time of QShuffler schedule}} \times 100\%$.

Figures 4, 5, and 6 show the performance for different MPLs of QShuffler, FCFS, and SJF for $p = 5$, 25, and 50, respectively. The figures show that QShuffler is significantly better than FCFS and SJF. The approximation made by the algorithm to get a non-preemptive schedule and its reliance on model accuracy do not reduce its effectiveness.

The figures show that as MPL increases, FCFS and SJF are not able to keep up with the increased load on the system and their performance degrades compared to QShuffler. As MPL increases, there are more interactions that come into play, and QShuffler is able to take these interactions into account when choosing the schedule.

The figures also show that as p increases, the performance gap between QShuffler and FCFS also increases. FCFS is the scheduling algorithm used by all database systems that we are aware of, and this experiment shows that its sensitivity to arrival order can significantly degrade its performance. However, for this experiment, SJF consistently turns out to be the worst policy overall. Interestingly, SJF is the optimal scheduling policy if query interactions are ignored, and the fact that it is the worst policy in this experiment demonstrates the importance of modeling query interactions when scheduling. To illustrate the potential benefit of QShuffler (or, conversely, the opportunity lost by using FCFS), we note that for $p = 50$ the performance gain of QShuffler over FCFS is up to 60%. This gain comes "for free" simply by scheduling the queries in the correct way.

5.3 Scalability and Robustness of QShuffler

We study the scalability of QShuffler in two dimensions: query types T and size of the data set.

As T increases, the space of possible query mixes increases, which affects both model building and scheduling. To test

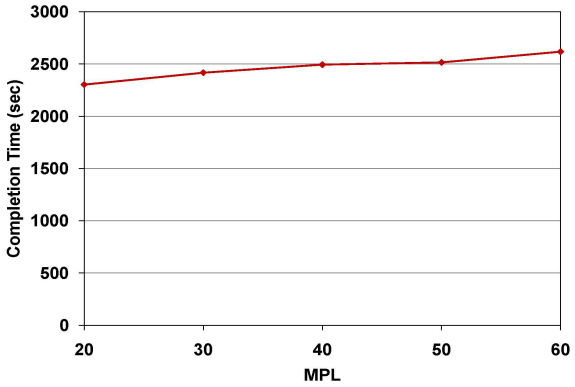

Figure 3: QShuffler Execution Time

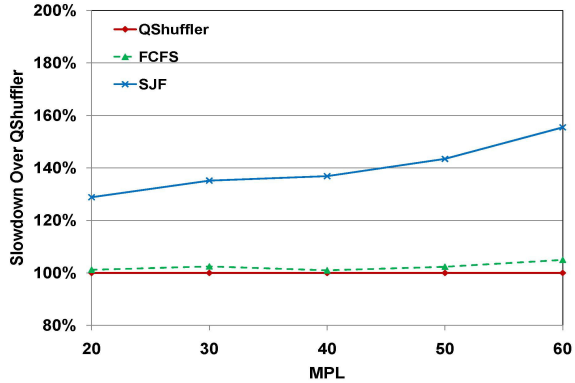

Figure 4: Scheduling for $p = 5$

QShuffler for higher T, we use a workload comprised of 21 of the 22 queries in the TPC-H benchmark. We do not use $Q15$ since it creates and drops a view, which is not supported in our current implementation. The workload consists of 60 queries of each type, for a total of 1220 queries. Figure 7 shows the performance of the different scheduling algorithms for this workload for MPL 30 and varying p. The completion time for QShuffler in this case is 2554 seconds. The figure shows that, as in previous experiments, QShuffler performs better than both FCFS and SJF.

To test QShuffler for larger database sizes, we use the 10GB TPC-H database. Since the hardware is unchanged from the 1GB case, the queries place a much higher load on the system and have much higher run times in the 10GB case. Therefore, we experiment with only the 6 longest running query types from Table 2. The run times of these 6 queries in the 10GB setting are given in Table 4. The workload consists of 10 queries of each type for a total of 60 queries, and MPL is set to 10. The arrival order of the queries is determined based on the parameter p, and we use $p = 2, 5$, and 10 since we have a lower MPL than the 1GB case. Figure 8 shows the performance of the different scheduling algorithms for this workload. The completion time for QShuffler in this case is 1.78 hours and it is significantly better than the other algorithms, e.g., beating FCFS (7.43 hours) by a factor of 4.2. Once again, this shows the potential of interaction-aware scheduling. On examining the different mixes we found that, unlike the 1GB case, the homogeneous mixes (i.e., containing one query type only) are among the best mixes. As the arrival patterns become more skewed, FCFS is able to hit these good mixes through its default behavior.

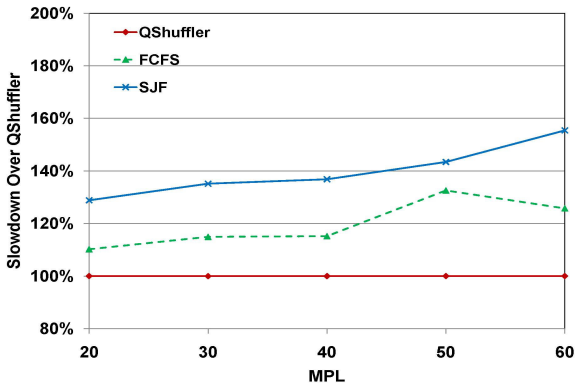

Figure 5: Scheduling for $p = 25$

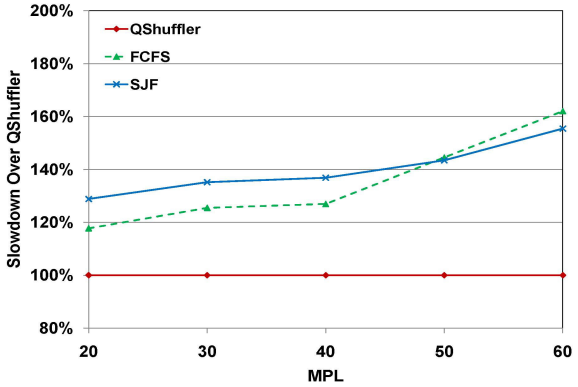

Figure 6: Scheduling for $p = 50$

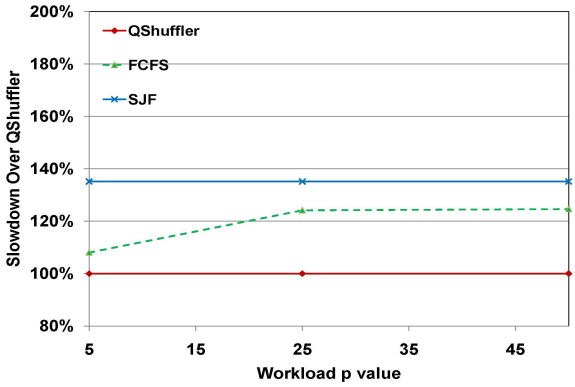

Figure 7: Scheduling for $T = 21$

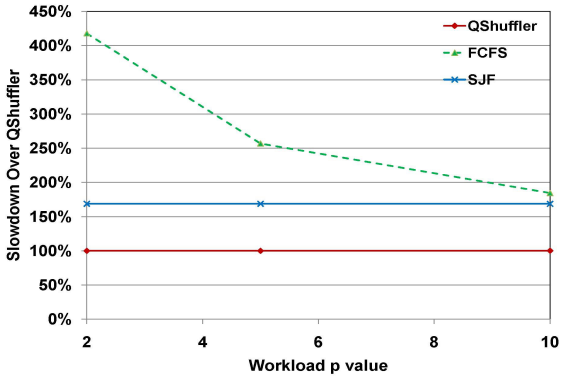

Figure 8: Scheduling for 10GB database

5.4 Scheduling for Skewed Databases

The scheduling algorithm of QShuffler makes use of average completion times at the level of query types, where a type in our current implementation is a distinct query template like in TPC-H (Section 2). If the data distribution in the database is highly skewed, then it is possible that two instances of the same query type that have different parameter values could have very different effects in the same mix. This can be dealt with by dividing each query template into several query types based on parameter ranges. Thus we would be dealing with more query types as demonstrated above in Section 5.3 . Automatically determining the best set of query types in the presence of skew is a subject of future work. However to test the robustness of our current implementation that does not have this ability, we use the skewed TPC-D/H database generator available at [31]. This database generator populates a TPC-D/H database using skewed random values that follow a Zipf distribution. This distribution has a parameter z that controls the degree of skew, where $z = 0$ generates a uniform distribution and as z increases, the data becomes more and more skewed.

We test our scheduling algorithms on a 1GB database that was generated using $z = 1$. The skew in the data exhibits itself in the varying run times for different instances of the same query type. However, we do not change the way that we define the query types for our scheduling algorithms. We continue to group all instances of a TPC-H query together into one query type, and we use the average execution time of these instances to build performance models.

Figure 9 shows our results for workloads consisting of 60 instances (with different parameter values) of the 6 longest running query types in Table 2, for a total of 360 queries. The figure shows three workloads with $p = 5$, 25, and 50, and MPL 30. The figure shows that QShuffler is consistently better than FCFS and SJF for the different workloads for all values of p. Thus, we can see that our technique, while primarily developed assuming uniform query run times for the different types, can still perform well even for skewed data distributions. Our scheduling algorithm performs well as long as it can distinguish the bad mixes (where the performance of one or more queries is degraded severely because of interactions) from the good ones.

5.5 Cost and Accuracy of Modeling

Since performance modeling is an essential part of our technique, we consider in our final experiment: (1) How accurate are our performance models? and (2) How expensive is it to build these models? To answer these questions, we sample the space of possible query mixes using CDR sampling as described in Section 3, and we use our samples to build performance models for query completion times, A_{ij}.

Mean Relative Error (MRE): We compute the accuracy of a performance model as follows. We pick $S = 100$ test samples at random from the full space of samples (different from the samples used to build the model), and we compute the model-predicted value of performance p_{est} for each test sample (i.e., the estimated A_{ij}). MRE is defined as $\frac{1}{S}\sum_{i=1}^{S}\frac{|p_{est}-p_{obs}|}{p_{obs}}$, where p_{obs} is the actual performance observed for the sample (i.e., the actual A_{ij}). MRE is a common metric for quantifying model accuracy.

Figures 10 and 11 show the MRE on the test samples vs. the number of samples used for model building for Q_{13} and Q_{18}, respectively. The figures show MRE for the three types of models presented in Section 3: linear regression models, regression trees (CART [34]), and quadratic regression mod-

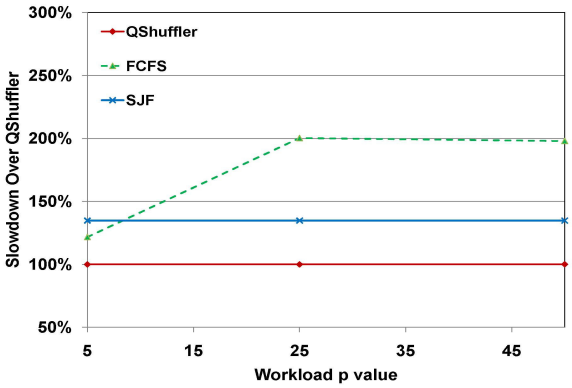

Figure 9: Scheduling for skewed data

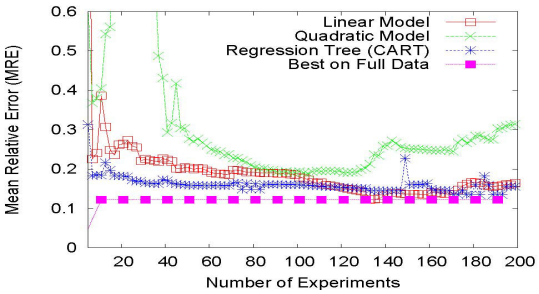

Figure 10: Model accuracy for Q_{13}

els. The "Best on Full Data" plot shows the best modeling accuracy achieved using all the samples we collected (more than 400 for our default setting). The graphs for other query types are similar that we omit them due to lack of space.

From the figures, we can see that: (1) MRE quickly converges to a value of around 25-30% with a small number of training samples (50-60), (2) simple linear models, which we use in QShuffler, are not drastically off the accuracy of the more complex regression tree models, and (3) quadratic models are the least accurate, which shows that a more complex model structure does not necessarily lead to more accuracy. Thus, we see that modeling A_{ij} can be done quite effectively: we can get good accuracy by using simple linear models and training these models with a small number of query mixes sampled from the space of possible mixes. In particular, the accuracy obtained from these samples is good enough for our query scheduler to produce efficient schedules that outperform the schedules produced by conventional schedulers. The time needed for modeling, which includes both sample collection time and model building time, is as follows for our experimental settings: (i) 3 hours for $T = 12$, 1 GB, 60 samples; and (ii) around 24 hours for $T = 6$, 10 GB, and 30 samples. We saw that just on one run in the 10 GB case we saved more than 5 hours. With repeated runs in a report generation setting, the cost of modeling is well justified by the savings in query completion time. Thus, we see that the modeling requirements of QShuffler can be effectively satisfied through simple statistical modeling.

6. RELATED WORK

This paper builds on an earlier version that appeared as a poster [4]. Some recent papers have employed the concept of modeling the performance of *transaction mixes* in different application areas. These papers define a transaction mix as transactions of different types running during a

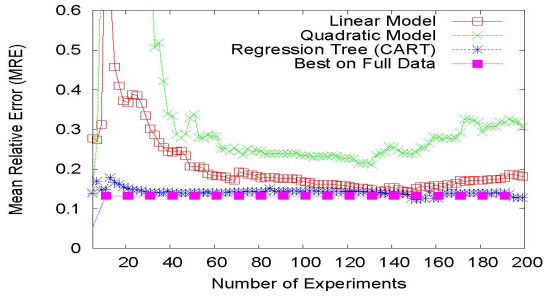

Figure 11: Modeling accuracy for Q_{18}

time interval or monitoring window, which is fundamentally different from our notion of a *concurrent* query mix. Like our work, these papers use statistical modeling. In [16], the authors propose the use of transaction-mix models for detecting performance anomalies in multi-tier enterprise applications. This work is extended in [32], where the models are also used to predict performance. In [35] and [36], the authors use transaction-mix models for resource provisioning and capacity planning in multi-tier applications. However, unlike our work, none of these papers consider the concurrent execution of transactions and the interaction among these transactions. Furthermore, we use our models to solve the problem of scheduling query mixes, while these papers focus on performance prediction.

Prior work on concurrently running query mixes generally falls into two categories: work on multi-query optimization (e.g., [25]), and work on sharing scans in the buffer pool (e.g., [23]). Both of these categories try to induce positive interactions between queries, but they are fairly restricted in the types of interactions that they consider. In our work, we can capture different kinds of both positive and negative interactions, and our scheduler can be implemented without requiring modifications to the internals of the database system, as would be required by multi-query optimization.

There is a wealth of literature on scheduling (e.g., [10]). Scheduling in database systems has been studied in the context of concurrency control, where the focus is on minimizing lock contention [14, 15]; and in real-time database systems (RTDBMS) [1, 2, 3] with the goal of minimizing missed deadlines. These works study how scheduling around critical resources can help meet this goal. In [17] and [24], the focus is on differentiating classes of requests in RTDBMS. Prioritization for resources has been studied in the context of general purpose database systems in [8, 19, 20]. Admission control and external scheduling have been studied for multi-tier applications in [12]. In [27], the optimal buffer space for a query is estimated and used to check that the memory consumption of scheduled queries does not exceed available memory. This addresses only one resource (buffer space) and does not consider interactions in the buffer pool, which can be significant. Our work is different from all these works in that reasoning about query mixes is central to our approach. Ignoring query interactions in a mix may result in suboptimal decisions. For example, [12] proposes using shortest job first as a scheduling policy, which may be the worst policy if there is query interaction (Section 5).

Another area of related work is admission control and setting the multi-programming level (or MPL) of the database system [12, 13, 22, 30]. The MPL is the number of requests served concurrently by the system. These works also do

load control, reacting in different ways to deviations in the load from optimal. However, the works on admission control and workload management in database systems generally focus on transactional workloads. BI workloads, on the other hand are very different from transactional workloads and the common approach used by commercial BI systems is to set the MPL statically [21]. In [21], the authors propose a batch BI workload manager that does admission control by admitting batches of queries such that their memory requirement equals the available memory on the system. In our work we show that query interaction can render such simple scheduling policies highly sub-optimal.

Scheduling becomes more important for systems under heavy load, since the alternative to scheduling is load shedding, for which there are several techniques (e.g., [33]). Recently, [29] proposed using shortest remaining time first (SRTF) scheduling to avoid dropping requests when the system is under load. QShuffler also avoids overload, but by taking query interaction into account we are able to make better scheduling decisions. For example, we show that SJF, the non-preemptive version of SRTF, is often a bad policy.

7. CONCLUSIONS

In this paper, we demonstrate that interactions among concurrently running queries in a *query mix* can have a significant effect on performance. Hence, we argue that it is important to take these interactions into account when making performance related decisions. We propose an experimental modeling approach for capturing interactions in query mixes, since analytical modeling of these interactions is too complex. We present QShuffler, a throughput oriented scheduler for BI report generation workloads. QShuffler determines the best schedule for the entire workload, and it is based on a linear programming formulation of the problem that results in a solution that is guaranteed to be within an additive factor of the true optimal solution. We experimentally validate the effectiveness of our modeling approach and of QShuffler using a BI benchmark on a real database system. We show that modeling is accurate enough and converges quickly, and we show that QShuffler can give us up to a four-fold improvement in performance over the default FCFS scheduler used by database systems.

8. REFERENCES

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *SIGMOD Rec.*, 17(1):71–81, 1988.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *VLDB*, 1989.

[3] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *TODS*, 17(3):513–560, 1992.

[4] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Qshuffler: Getting the query mix right. In *ICDE*, 2008.

[5] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service. In *SIGMOD*, 2008.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4), 2004.

[7] Business objects. http://www.businessobjects.com/.

[8] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *VLDB*, 1989.

[9] Cognos. http://www.cognos.com/.

[10] R. H. Conway, W. L. Maxwell, and M. L. W. *Theory of scheduling*. Addison-Wesley, 1967.

[11] CPLEX. http://www.ilog.com/products/cplex/.

[12] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW*, 2004.

[13] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *VLDB*, 1991.

[14] T. Ibaraki, T. Kameda, and N. Katoh. Cautious transaction schedulers for database concurrency control. *TSE*, 14(7), 1988.

[15] N. Katoh, T. Ibaraki, and T. Kameda. Cautious transaction schedulers with admission control. *TODS*, 10(2):205–229, 1985.

[16] T. Kelly. Detecting performance anomalies in global applications. In *Proc. Workshop on Real, Large Distributed Systems*, 2005.

[17] J. Kyoung-Don Kang Son, S.H. Stankovic. Service differentiation in real-time main memory databases. In *ISORC*, 2002.

[18] W.-Y. Loh. Regression trees with unbiased variable selection and interaction detection. *Statistica Sinica*, 12:361–386, 2002.

[19] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *ICDE*, 2004.

[20] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Improving preemptive prioritization via statistical characterization of oltp locking. In *ICDE*, 2005.

[21] A. Mehta, C. Gupta, and U. Dayal. BI Batch Manager: A system for managing batch workloads on enterprise data warehouses. In *EDBT*, 2008.

[22] A. Mönkeberg and G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *VLDB*, 1992.

[23] K. O'Gorman, A. E. Abbadi, and D. Agrawal. Multiple query optimization in middleware using query teamwork. *Software - Practice and Experience*, 35(4), 2005.

[24] H. Pang, M. J. Carey, and M. Livny. Multiclass query scheduling in real-time database systems. *TKDE*, 7(4):533–551, 1995.

[25] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2008.

[26] H. J. Ryser. *Combinatorial Mathematics*. The Mathematical Association of America, 1963.

[27] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *TODS*, 11(4):473–498, 1986.

[28] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[29] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *TOIT*, 6(1), 2006.

[30] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good multi-programming level for external scheduling. In *ICDE*, 2006.

[31] Skewed TPC-D data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[32] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys*, 2007.

[33] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *USITS*, 2003.

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

[35] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware*, 2007.

[36] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC*, 2007.