University of Waterloo Technical Report CS-2010-22

Interaction-Aware Scheduling of Report Generation Workloads

Mumtaz Ahmad · Ashraf Aboulnaga · Shivnath Babu · Kamesh Munagala

Abstract The typical workload in a database system consists of a mix of multiple queries of different types that run concurrently. Interactions among the different queries in a query mix can have a significant impact on database performance. Hence, optimizing database performance requires reasoning about query mixes rather than considering queries individually. Current database systems lack the ability to do such reasoning. We propose a new approach based on planning experiments and statistical modeling to capture the impact of query interactions. Our approach requires no prior assumptions about the internal workings of the database system or the nature and cause of query interactions; making it portable across systems.

To demonstrate the potential of modeling and exploiting query interactions, we have developed a novel interaction-aware query scheduler for report-generation workloads. Our scheduler, called *QShuffler*, uses two query scheduling algorithms that leverage models of query interactions. The first algorithm is optimized for workloads where queries are submitted in large batches. The second algorithm targets workloads where queries arrive continuously, and scheduling decisions have to be made on-line. We report an experimental evaluation of QShuffler using TPC-H workloads running on IBM DB2. The evaluation shows that QShuffler, by modeling and exploiting query interactions, can consistently outperform (up to 4x) query schedulers in current database systems.

Keywords Business Intelligence · Report generation · Query interactions · Scheduling · Experiment-driven performance modeling · Workload management

M. Ahmad
D.R. Cheriton School of Computer Science, University of Waterloo
E-mail: m4ahmad@uwaterloo.ca

A. Aboulnaga
D.R. Cheriton School of Computer Science, University of Waterloo
E-mail: ashraf@cs.uwaterloo.ca

S. Babu
Department of Computer Science, Duke University
E-mail: shivnath@cs.duke.edu

K. Munagala
Department of Computer Science, Duke University
E-mail: kamesh@cs.duke.edu

1 Introduction

The typical workload in a database system at any point in time is a mix of queries of different types running concurrently and interacting with each other. The interactions among concurrent queries can have a significant impact on database performance. Hence, optimizing database performance requires reasoning about *query mixes* and their interactions, rather than considering queries or query types in isolation.

A query Q_1 that runs concurrently with another query Q_2 can impact Q_2's performance in different ways, either positively or negatively. For example, Q_1 may bring data or index blocks into the buffer cache that are useful for the concurrently running Q_2. In this scenario, the increased cache hits will make Q_2 complete much faster than if it were to run without Q_1 running side by side. Alternatively, Q_1 and Q_2 could interfere with each other on hardware resources such as CPU, memory, or L2 cache, or on internal database system resources such as latches or locks. In this scenario, the concurrent presence of Q_1 will increase the completion time of Q_2, possibly by a large amount.

We illustrate the impact of query interactions in query mixes using 60 instances of TPC-H queries running on a 10GB IBM DB2 database[1]. Figure 1 shows the respective completion times of three different workloads composed of these 60 queries. The total *set* of queries in all three workloads is the same. However, we change the arrival order of the queries across the three workloads, causing differences

[1] Details of our experimental setting are provided in Section 9.

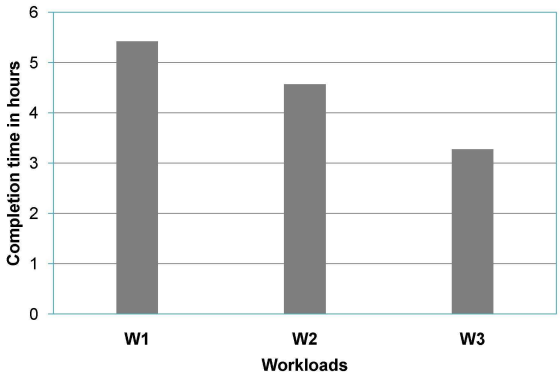

Fig. 1 Completion times of three different workloads that consist of the same batch of 60 TPC-H queries

among the *query mixes* scheduled by DB2 for each workload. All other aspects of the system, including hardware resources, DB2 configuration parameters, physical design, and multi-programming level, are kept the same across all executions of the three workloads. The *multi-programming level (MPL)* is the number of queries that execute concurrently in the system, and it is set to 10 in these experiments.

The figure shows that the difference in scheduled mixes among the three workloads causes a 2.1 hour (63%) difference in completion time between the best and worst performing workloads. It is important to note that each workload runs the same batch of queries under an identical system configuration. In workload W_1, which takes 5.4 hours, queries that compete for resources get executed concurrently, resulting in negative interactions and poor performance. In workloads W_2 and W_3, the interactions are less negative and occasionally positive, with queries that help each other executing concurrently. This results in best performing workload, W_3, running the same query batch in just 3.3 hours.

If the 60 TPC-H queries from Figure 1 were submitted to the database system as a single batch, then we would like the system to run the queries as per workload W_3 in the figure. However, the system has to be *interaction-aware* to be able to choose this schedule over, say, workload W_1. A major hurdle in making database systems interaction-aware is in finding effective ways to capture and model query interactions. As we hinted earlier, there is a large spectrum of possible causes for interactions that includes resource-related, data-related, and configuration-related dependencies. Interactions are often 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, interactions that occur when a database system runs on one hardware/OS configuration may not happen when the same system runs on a different configuration.

Would it be possible to capture interactions using the analytical cost models used by database query optimizers to cost query plans? Unfortunately, the answer is no. The cost models used in almost all database systems today work on a per query plan basis; so they cannot estimate the overall behavior of multiple concurrent queries. For the conventional cost models to capture query interactions, we will need to extend them to model 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 query 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 interactions. Second, we propose a proactive *experiment-driven* approach to tease out the significant interactions that can appear in a query workload. This approach is based on running a small set of carefully-chosen query mixes, and measuring how the average completion times of various queries are affected by running them in a mix instead of in isolation. We show that most significant interactions can be captured in practice by *sampling* very few mixes.

While the experiment-driven approach requires advance knowledge of the different query types, and has to be repeated for each new database and hardware/OS setting (e.g. when MPL or indexes are changed in the database, or new memory is added to the system), it has two important advantages: (i) it is effective irrespective of the true cause of interactions because the effect of any interactions will be captured in the monitoring data, and (ii) it supports incremental update as query workloads evolve over time, as well as on-line maintenance based on monitoring data available when query mixes run in the production setting.

QShuffler: A core contribution of this paper is an end-to-end solution that demonstrates how our approach enables interactions to be modeled and exploited in order to gain huge performance improvements in database systems. The problem we consider is that of scheduling report-generation queries in database systems. Report-generation workloads are a very common type of workload in modern Business Intelligence (BI) settings. These workloads are critical for operational and strategic planning, so it is important to run them efficiently. Report-generation workloads continue to rise in importance with the emergence of frontline data warehouses that seek to monetize data collected about customer interactions and application usage [1,2].

In particular, we have developed a query scheduler, called *QShuffler* (for *Query Shuffler*), that focuses on the throughput-oriented workloads encountered in report-generation systems. There is a fixed number of report types that a user can request in such systems, but the reports requested during any given period may vary. 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 requested reports (i.e., to maximize throughput). The response time of individual queries is not important as long as all the reporting queries are completed within a desired time window. This is a common scenario in BI systems like Cognos [3] and Business Objects [4].

Concretely, the goal of QShuffler is to schedule appropriate query mixes for a given query workload W in order 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, so they can produce suboptimal schedules when significant inter-query interactions exist. QShuffler's interaction-aware scheduling gives performance improvements up to 4x over the 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 taking query interactions into account, QShuffler's performance gains come from two novel algorithms for scheduling queries. The algorithms cater respectively to two common scenarios found in report generation. The first scenario involves queries being submitted to the database system in large batches. QShuffler's *batch scheduling algorithm* is designed for this scenario. 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. The algorithm continues to perform well even as modeling accuracy decreases, although the optimality guarantee does not hold any more.

The second scenario involves client applications or workflows submitting queries in small batches, often one at a time. As soon as queries are processed and the results returned, new queries are submitted by the clients until report generation is complete. Since queries keep arriving continuously at the database system, scheduling decisions have to be made on-line (albeit with some limited lookahead). QShuffler's *on-line scheduling algorithm* is designed for this scenario. While the on-line algorithm uses a conventional priority-based scheduling approach—because of the need to keep scheduling overhead low—the technique for computing priorities is interaction-aware and novel. This algorithm needs a measure of the cost that a query mix incurs while running on the system. Our results show, perhaps surprisingly, that resource utilization metrics (e.g., CPU or I/O utilization) are not good metrics to use for scheduling. These metrics may be useful if the objective is to monitor and control system resource utilization. However, our results show

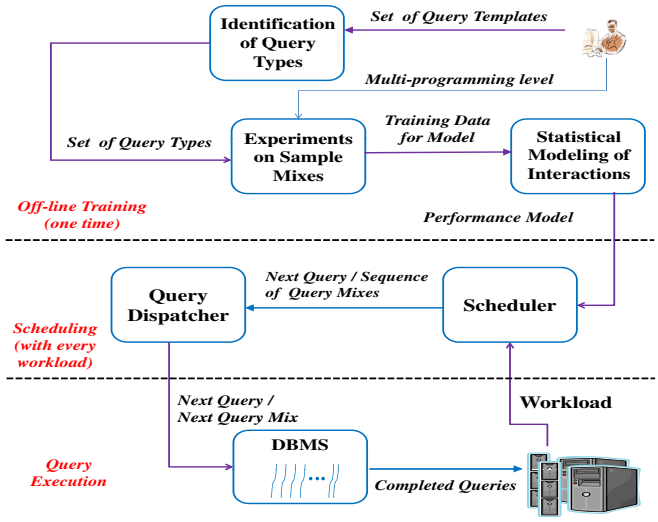

Fig. 2 Solution framework

that these metrics are quite unrepresentative in quantifying the completion time of queries in different mixes. We develop a new metric, called *normalized run-time overhead (NRO)*, to address this problem. The *NRO* metric is a measure of the run-time overhead that queries of different types incur when they run concurrently with other queries in a mix as compared to running alone in the system.

QShuffler has different components, shown in Figure 2:

- **Identifying query types:** An off-line step is used to cluster query instances from the report-generating applications into a set of distinct *query types*. The query mixes considered by QShuffler are composed of query instances belonging to these types.
- **Sampling:** A set of carefully-chosen query mixes is run in order to generate training data for the modeling step.
- **Modeling:** The training data is used to learn statistical models that can estimate the performance of queries in any given mix.
- **Scheduling:** The scheduler uses the performance models and the appropriate scheduling algorithm—batch or on-line—to decide when to schedule each submitted query. The query dispatcher runs the queries on the database system as per this schedule.

These different components are discussed in the rest of the paper, in which we make the following contributions:

- **Query interactions:** Section 3 considers a number of query mixes to show the significant impact that query interactions can have, thereby motivating why database systems need to be interaction-aware.
- **Interaction-aware batch scheduling:** Section 4 describes QShuffler's algorithm for scheduling queries in large batches.
- **Interaction-aware on-line scheduling:** Section 5 describes QShuffler's on-line scheduling algorithm. This

algorithm uses the NRO metric to cost query mixes, which is described in Section 5.1. Section 6 presents on-line scheduling algorithms that are based on query optimizer cost estimates as an alternative to QShuffler, and we compare against these algorithms in our experiments.
- **Experiment-driven modeling:** Section 7 presents QShuffler's novel approach to capture the impact of query interactions, based on conducting experiments and statistical modeling.
- **Handling skew and identifying query types:** Section 8 considers challenges that skewed datasets pose to QShuffler, and proposes a novel solution that leverages the database query optimizer. A significant advantage of this solution is that it gives a natural way to identify the query types that QShuffler needs to consider.
- **Empirical evaluation:** We have prototyped all algorithms described in the paper. Section 9 presents an end-to-end empirical evaluation of QShuffler using TPC-H queries running on IBM DB2. The results show up to 4x improvements over the interaction-unaware scheduling algorithms used in database systems today.

2 Related Work

To the best of our knowledge, our work is the first to consider the spectrum of positive and negative interactions possible among concurrently-executing queries in query mixes. This paper significantly extends an earlier paper [5] and a poster [6]. In [5], we demonstrated the impact of query interactions, and introduced the experiment-driven approach to modeling as well as the batch algorithm for scheduling. In [6] we gave a high level description of the on-line scheduling problem. This paper builds on [5] and [6] by presenting the details of the on-line scheduling algorithm, the NRO cost metric for query mixes, and techniques to identify query types as well as handle skewed data. A recent paper discussed applications of query-interaction modeling in the field of database testing [7].

Prominent categories of prior work on optimizing concurrent execution of queries include work on multi-query optimization (e.g., [8]) and work on sharing scans in the buffer pool (e.g., [9]). Both these categories of work try to induce positive interactions among concurrent queries based on detailed knowledge of database system internals. However, the types of interactions considered are fairly restricted. In contrast, our work focuses on capturing all different kinds of positive as well as negative query interactions regardless of the known or unknown underlying cause. Our approach does not require information a priori about the database internals or the hardware/OS environment. More recently, in [10], pairwise synergies among queries—the influence of a query on another—are learnt and used to predict the performance of queries. The authors show the effectiveness of their approach in a database simulation framework.

There is a wealth of literature on scheduling (e.g., see [11]). Scheduling in database systems has been studied in the context of concurrency control, where the focus is on minimizing lock contention [12, 13], and in real-time database systems (RTDBMS) [14–16] with the goal of minimizing missed deadlines. These works study how scheduling around critical resources helps meet the desired goals. In [17] and [18], the focus is on differentiating classes of requests in an RTDBMS. Translating transaction-level priorities into prioritization in resource usage (e.g., CPU, locks) has been studied in the context of general-purpose database systems in [19–21]. The optimal buffer space requirement for each query is estimated in [22], and used to ensure that memory consumption of scheduled queries does not exceed available memory. That paper addresses only one resource (buffer space) and does not consider the potentially significant interactions among queries in terms of buffer cache usage. Scheduling under heavy load is studied in [23], which proposed using shortest remaining time first (SRTF) scheduling to avoid dropping requests when the system is under high load. In contrast to all these works, reasoning about query mixes is central to our approach. Ignoring query interactions in a mix can result in suboptimal scheduling decisions. For example, [24] proposes using shortest job first as a scheduling policy. As we show in Section 9, this policy can perform poorly in the presence of query interactions.

Some recent papers have employed the concept of *transaction mixes* in different application areas. These papers use *transaction mix models* for performance prediction, capacity planning, and detecting anomalies in performance [25–28]. However, unlike our work, none of these papers consider interactions caused by the *concurrent* execution of transactions. Furthermore, we address the problem of scheduling, while these papers focus on performance prediction.

These papers define a transaction mix as all the transactions of different types that run during a time interval or monitoring window without considering which of these transactions ran concurrently. This is fundamentally different from our notion of a concurrent query mix. For example, these papers would not distinguish between the following three cases: (a) a monitoring window in which 10 transactions of type T_1 execute concurrently with 10 transactions of type T_2, (b) a monitoring window in which 10 transactions of type T_1 execute concurrently followed by 10 concurrent transactions of type T_2, and (c) a monitoring window in which 5 transactions of type T_1 execute concurrently with 5 transactions of type T_2 and when these transactions finish another 5 transactions of type T_1 execute concurrently with another 5 transactions of type T_2. These three cases will all be considered to have the same transaction mix: 10 transactions of type T_1 and 10 transactions of type T_2. Like our

work, these papers use statistical techniques to learn models to estimate performance metrics for transaction mixes. However, their performance models would not distinguish between the three cases above even though they have very different concurrently executing transactions (even with different MPLs). In this paper, we show that the concurrent execution of different queries in different mixes has a significant effect on performance, and our performance models explicitly take this concurrent execution into account.

Workload management is an area of growing importance in database systems. Work in this area includes techniques for admission control and setting the multi-programming level of the database system [24, 29–31]. The proposed techniques perform load control, reacting in different ways to deviations of workload performance metrics from the desired range. Most of the focus has been on transactional workloads consisting of large numbers of relatively light-weight queries. The report-generation workloads that we consider consist of complex analytic queries and are very different from transactional workloads. In [32], the authors propose a batch workload manager for Business Intelligence (BI) systems that does admission control based on the estimated memory requirement of analytic queries. Our work takes a significant step towards extending workload management in BI systems with effective scheduling techniques. We also compare our scheduling approach to an approach that relies on query optimizer cost estimates for scheduling queries. The optimizer-based approach is an adaptation of techniques proposed by Niu et al. [33, 34]. We describe this optimizer-based approach in Section 6, and we demonstrate in Section 9 that QShuffler outperform this approach.

Experiment-driven performance modeling is gaining wide acceptance as a way to build robust performance models for complex systems. A very relevant work in this area is [35]. Like our work, the authors use statistical learning techniques to effectively predict performance metrics for database queries. However, their work focuses exclusively on single query types and does not consider inter-query interactions in query mixes. Experiment-driven performance modeling has also been used for tuning database configuration parameters [36, 37]. An infrastructure for running experiments in a data center is proposed in [38]. Oracle 11g provides an infrastructure called *test-execute* for running experiments within the database system during maintenance sessions [39]. Like these prior works, we use experiment-driven performance modeling because its effectiveness and robustness in modeling query interactions. Any of the frameworks proposed in the prior works can be used to run the experiments required by QShuffler.

3 Impact of Query Interactions

This section provides a number of real examples of how queries running concurrently in a database system impact each other's performance in negative or positive ways. Consider a database system whose workload comes from a set of report-generation clients. Each query in the workload belongs to one of T *query types* denoted $Q_1, Q_2, \ldots, Q_T$. (Table 1 summarizes the notation used throughout the paper.)

Table 1 Notation used in the paper

Symbol	Description
M	Multiprogramming level
T	Number of query types
Q_j	Query type j
q_j	An instance of 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
R_i	A cost metric for query mix m_i
$\hat{R}_i$	Estimated R_i
θ_R	Cost Threshold
A_{ij}	Average completion time of a Q_j query when running in mix m_i
NRO_i	Normalized run-time overhead metric of query mix m_i
$\widehat{NRO_i}$	Estimated NRO_i
θ_{NRO}	*NRO* target
W	Workload to be scheduled
I_j	Total number of instances of Q_j to be scheduled
n_i	Time for which mix m_i is scheduled in the batch scheduling algorithm
L	Lookahead

As a concrete example, let us consider a report-generation workload generated by the popular TPC-H decision-support benchmark [40]. TPC-H defines 22 query templates where each template can be instantiated with different parameter values to generate hundreds of distinct query instances. We will consider each query template to be a query type, so we have $T = 22$. In reality, the problem of identifying query types is nontrivial. We will address this problem in Section 8 and describe a novel technique for identifying distinct query types in a workload automatically. Until then we will assume for simplicity that the query types have already been identified.

Let the multi-programming level (MPL) of the database system be set to M. Recall that 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 $\langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$, where N_{ij} is the number of instances of query type Q_j in m_i, and $\sum_{j=1}^{T} N_{ij} = M$.

Let t_j denote the average completion time of queries of type Q_j when run alone in the system. The completion time of a query instance is the time elapsed between when the query starts and when it finishes. The average completion time t_j of a query type Q_j is determined by taking an average of the completion times of a number of instances of Q_j (in our case 10) that are run alone in the system at $M = 1$. Let

Table 2 Run time t_j in seconds of 12 different TPC-H query types when they are run alone in the system

Database Size	Q_1	Q_3	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{13}	Q_{18}	Q_{20}	Q_{21}
1GB	10.07	3.41	2.60	4.77	5.76	4.15	9.66	2.65	6.12	7.12	4.48	7.30
10GB	294.61	247.95	136.04	346.63	102.06	387.72	578.61	353.89	101.27	554.56	273.08	570.37

Table 3 N_{ij} and A_{ij} (in seconds) for different query mixes in a TPC-H database

		Q_1		Q_7		Q_9		Q_{13}		Q_{18}		Q_{21}	
Mix	Database Size	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	1GB	11	143.9	8	144.6	3	211.2	2	97.8	2	149.8	4	127.5
m_2	1GB	2	361.7	8	298.6	1	476.0	18	121.2	0	0.0	1	231.2
m_3	10GB	1	1897.4	2	72.7	5	2919.3	0	0.0	2	1904.1	0	0.0
m_4	10GB	0	0.0	10	599.8	0	0.0	0	0.0	0	0.0	0	0.0
m_5	10GB	4	538.0	0	0.0	0	0.0	0	0.0	1	541.4	0	0.0
m_{5a}	10GB	4	538.0	0	0.0	0	0.0	0	0.0	1	539.3	0	0.0
m_{5b}	10GB	4	542.9	0	0.0	0	0.0	0	0.0	1	538.3	0	0.0
m_6	10GB	0	0.0	4	264.5	0	0.0	0	0.0	1	3413.7	0	0.0

A_{ij} denote the average completion time of queries of type Q_j when run in mix m_i. Interactions among queries that run concurrently in mixes can be negative or positive. We say that a query of type Q_j has negative interactions in mix m_i if $A_{ij} > t_j$, i.e., an instance of Q_j is expected to run slower in the mix than when run alone. On the other hand, $A_{ij} < t_j$ indicates positive interactions.

Next, we provide some examples of interactions in query mixes running over TPC-H databases of scale factors 1 and 10 (denoted 1GB and 10GB respectively). Table 2 shows the run times of the 12 longest running TPC-H queries when run alone in the system (i.e., the values of t_j). Tables 3 shows the run times of queries in six different mixes. For each mix, the number of queries in the mix, N_{ij}, and the average run time in seconds, A_{ij}, are shown. Consider the two mixes m_1 and m_2 shown in the table for the 1GB database. The following observations can be made about m_1 and m_2 from the A_{ij} values observed:

1. In both m_1 and m_2, all A_{ij} values are way higher than the corresponding t_j values shown in Table 2. Thus, all queries are impacted negatively in these mixes.
2. Both mixes have the same number ($N_{ij} = 8$) of instances of Q_7 and the same total number of queries ($M = 30$). However, A_{ij} for Q_7 in m_2 is almost twice the A_{ij} for Q_7 in m_1. Thus, instances of Q_7 are expected to run twice as slow in m_2 than in m_1.

Mixes m_1 and m_2 show how the behavior of queries can vary drastically from mix to mix depending on the interactions present[2]. However, the interactions in these mixes are exclusively negative. Mixes m_3 and m_5 in Table 3 display some positive interactions.

[2] All experiments reported in this paper have been repeated multiple times to verify consistency and statistical significance.

Mixes m_3, m_4, m_5 and m_6 in Table 3 run on the 10GB database. Mixes m_3 and m_4 have $M = 10$, and mixes m_5 and m_6 have $M = 5$. Mix m_3 displays positive interactions for Q_7. Recall from Table 2 that the run time of Q_7 when it runs alone in the system is 102.06 seconds. In mix m_3, Q_7 has an average run time of 72.7 seconds. Thus, Q_7 *benefits* from being run in mix m_3: Q_7 runs faster when run concurrently with 1 instance of Q_1, 5 instances of Q_9, and 2 instances of Q_{18} than when it runs alone in the system.

The performance of query Q_7 in mix m_3 raises a natural question: wouldn't Q_7's performance be even better if it were to run in a mix that predominantly has instances of Q_7 (e.g., because of possibly increased buffer cache hits)? In this regard, consider the average run time of Q_7 in mix m_4 which has 10 concurrent instances of Q_7. Notice that Q_7's run time in m_4 is worse than its run time in m_3, and also worse than Q_7's run time when it runs alone in the system (i.e., Q_7's interactions are negative in m_4).

Mix m_5 in Table 3 presents another example of positive interaction, this time for Q_{18}. The average run time of Q_{18} in this mix is 539.3 seconds, compared to a run time of 554.6 seconds when it is run alone. Thus, Q_{18} benefits from being run concurrently with 4 instances of Q_1. Now contrast Q_{18}'s performance in mixes m_5 and m_6. Both mixes have the same total number of queries, and both have one instance of Q_{18}. While Q_{18} benefits from positive interaction in m_5, it suffers drastically in m_6, its run time degrading by more than 6x.

Mixes m_{5a} and m_{5b} show repeated runs of mix m_5 with different instances of the same query types. The results for all variants of m_5 are similar, illustrating the repeatability of our results. We observed the same pattern for other mixes. The standard deviation of the completion times across runs was less than 4% of the mean for all mixes reported here.

Table 4 shows more examples of interactions from the 10GB TPC-H database. Mixes m_7-m_{11} in this table focus on

Table 4 Changes in Q_{21}'s performance as the mix is varied

mix	Q_9 (N_{ij})	Q_{13} (N_{ij})	Q_{21} (N_{ij})	Q_{21} (A_{ij})
m_7	0	4	1	4188.2
m_8	1	3	1	5463.8
m_9	2	2	1	3476.1
m_{10}	3	1	1	3581.7
m_{11}	4	0	1	2782.4

three-way interactions of TPC-H query Q_{21} in the presence of queries of type Q_9 and Q_{13}. All mixes have $M = 5$ and one instance of Q_{21}. As we go down the table from mix m_7 to m_{11}, the number of instances of Q_9 increases while the number of instances of Q_{13} decreases correspondingly. The completion time of Q_{21} first increases sharply and then it decreases sharply. The worst run time of Q_{21} is almost 2x the best run time. Table 4 shows that Q_{21} runs much better in a mix with concurrent instances of Q_9 than with concurrent instances of Q_{13}. This behavior is not what we would have expected from Table 2 because Q_9 is by far the more heavy-weight query when each query is considered individually.

We make the following observations based on the suite of examples presented:

- Interactions in query mixes impact query run times significantly, sometimes by orders of magnitude.
- Interactions are fairly complex in nature. Rules of thumb or simple intuition cannot always explain query behavior in mixes.
- The performance of a query Q cannot be predicted unless we are able to model the effect of other queries running concurrently with Q. Thus, it is important to develop mix-based reasoning of query workloads to better manage the performance of database systems.

The rest of this paper describes an end-to-end solution for query scheduling that takes query interactions into account.

4 Interaction-aware Batch Scheduling

In this section, we describe an interaction-aware scheduling algorithm that enables QShuffler to schedule large batches of queries efficiently. Since most database systems do not preempt queries once they start, we focus on non-preemptive scheduling throughout this paper. There is work on preemptive scheduling of plan operators (e.g., [41]), but our focus is on scheduling entire queries.

The workload to be scheduled, W, comes from a set of clients, e.g., report-generation applications. Each client issues a fixed number of queries where each query belongs to one of the T possible types $Q_1, Q_2, \ldots, Q_T$. Let I_j denote the total number of queries of type Q_j in W. That is, $|W| = \sum_{j=1}^{T} I_j$. The clients place their batch of queries in an arrival queue, and QShuffler schedules queries from this

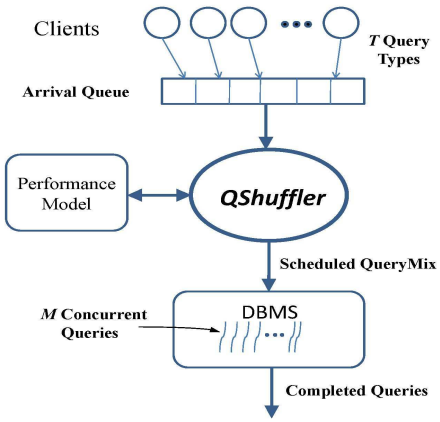

Fig. 3 QShuffler overview

queue. It is assumed that the database system is free to execute the queries queued at the system in any order. Precedence constraints among queries have to be enforced outside the system. Also, it is assumed that only throughput is important, not latency. Figure 3 illustrates the problem setting.

The objective of QShuffler's batch scheduling algorithm is to schedule the submitted queries as a sequence of query mixes so that the total completion time of W is minimized (which is equivalent to maximizing throughput for this workload). Formally, the completion time of W can be defined as the time elapsed between when the first query in W starts execution and when all the queries in W have finished execution. In a 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. We will show how, 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.

Intuitively, the algorithm works as follows. 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 in W should be run using the selected mixes. Next, we describe how X is picked and how the schedule is chosen from X.

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 $S(T,M) = \binom{M+T-1}{M}$ [42].

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. Section 4.3 discusses how b is chosen.

Linear program to pick a subset of X: QShuffler uses a *linear program (LP)* [43] 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, $1 \leq j \leq T$, the total number of instances of each query type Q_j to be scheduled. The LP contains an unknown variable n_i ($n_i \geq 0$) 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 the following T constraints in the LP:

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

These constraints are derived as follows. Let 1 denote the (normalized) amount of work needed to complete the execution of one instance of Q_j. Thus, the total work required to complete the execution of the I_j instances of Q_j in the input workload is I_j. $N_{ij} \times \frac{1}{A_{ij}}$ denotes the fraction of this work that gets completed per unit time when mix m_i is scheduled. Recall that N_{ij} denotes the number of instances of query type Q_j in m_i, and A_{ij} denotes the average completion time of a query of type Q_j in m_i. When mix m_i runs for one unit of time, each of the N_{ij} query instances of type Q_j in m_i will do a $\frac{1}{A_{ij}}$th fraction of the work required to complete it. (N_{ij} and A_{ij} are constants that depend only on m_i and Q_j. Section 7 shows how A_{ij} values can be estimated for each mix in X.) It follows that $\sum_{i=1}^{|X|} n_i \frac{N_{ij}}{A_{ij}}$ denotes the total work done for Q_j in the chosen schedule. This work must not be less than I_j in a feasible schedule for W. This reasoning explains the T constraints presented in Equation 1.

Working with the constraints in Equation 1, 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:

$$\text{Minimize} \sum_{i=1}^{|X|} n_i$$

We can solve the LP using any LP solver. In QShuffler, we use the highly-efficient CPLEX tool [44]. We give two lemmas to illustrate the properties of the LP solution.

Lemma 1 *The number of nonzero n_i variables in the LP solution is at most T, assuming $T \leq |X|$.*

The above lemma follows from linear-programming theory, where it is known that the number of variables set to nonzero values in the LP solution will not be greater than the number of constraints in the LP [43]. Recall from Equation 1 that our LP has T constraints, one per query type.

Lemma 2 *The LP solution produces a schedule that has the optimal workload completion time among any schedule consisting of mixes from X, provided that instantaneous preemption of queries is possible.*

In the LP solution, some n_i variables will be set to nonzero values and the rest will be zero. It follows from Lemma 1 that at most T (assuming $T \leq |X|$) variables can be nonzero. The mixes with nonzero n_i will be chosen in the optimal schedule. That is, the LP chooses 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 mix, the respective n_i value found by the LP gives the total time for which query instances should be run with that mix. Thus, we can generate a complete schedule from the LP solution. The optimality of the schedule follows from the optimality of the LP.

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 supported by most database systems as it requires instantaneous query suspend and resume features, we need to transform the preemptive schedule generated by the LP to an efficient preemption-free schedule.

4.1 Bound on Degradation from Optimal

We present Theorem 1 and then give a technique to produce a preemption-free schedule S from the LP solution. Theorem 1 uses the following notation: (i) In each mix with nonzero n_i in the LP solution, let a_j be the maximum average completion time among all query of types present in this mix. Let a_{max} be the maximum among all a_j for all mixes. (ii) Let $OPT = \sum_{i=1}^{|X|} n_i$ be the time to completion computed by the LP for the input query workload. That is, OPT is the optimal workload completion time with preemption (Lemma 2).

Theorem 1 *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: We can construct a preemption-free schedule S with the property stated in Theorem 1 as follows:

1. Pick one of the remaining mixes with a nonzero n_i. Suppose we pick mix m_i.
2. Schedule input query instances to run in mix m_i for time n_i.

Table 5 Run time of LP in CPLEX for different values of b with $T = 6$, $M = 60$

b	10	6	5	4	3	2	1
Number of mixes	462	3,003	6,188	15,504	53,130	324,632	8,259,888
LP run time (sec)	0.01	0.06	0.13	0.33	0.95	5.56	179

3. Wait until all scheduled queries finish. Do not schedule any more queries to run in mix 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. The optimal schedule with preemption requires m_i to be scheduled for n_i time then preempted. Since schedule S does not preempt running queries, it will need to wait beyond n_i for all the queries in m_i to complete. The maximum waiting time is the maximum time it takes for any query to complete, which is a_{max}. This will happen if S schedules the longest running query with completion time a_{max} just as time n_i is about to expire (we assume that as queries finish, this will not increase the completion time of any query in the mix). This explains the bound $n_i + a_{max}$. Since there are at most T mixes with nonzero n_i (Lemma 1), S will finish in time at most $OPT + a_{max}T$. □

The assumption in Theorem 1 may not hold in all cases. If it does not hold, we can still produce a good preemption-free schedule using the same approach. We only lose the theoretical bound on the maximum degradation from the optimal preemptive schedule. In practice, this assumption holds in the vast majority of cases. For example, in the 8K mixes that we ran in our sampling experiments (described in Section 7), there were 2.8 million pairs of mixes for which the assumption can be tested. The assumption was violated in only 7% of these mix pairs. Only 0.7% of the mix pairs had a violation that affected the maximum run time a_j, from which we derive a_{max}.

4.2 Robustness of the Chosen Schedule

The LP requires estimates of the A_{ij} values for the mixes in X. Section 7 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. In our implementation of QShuffler, we use this more robust approach to produce non-preemptive schedules.

4.3 Scalability of Linear Programming

Our LP solver can handle a very large number of mixes in the set X in real time. Table 5 shows the run time of the LP solver for different values of b with a number of query types $T = 6$ and an MPL $M = 60$ (the highest MPL in our experiments). It can be seen that 320 thousand variables are processed in less than 6 seconds and 8.26 million variables are processes in less than 3 minutes. Thus, 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 in the chosen schedule.

QShuffler picks X as a subspace of size $S(T, \frac{M}{b}) = \binom{\frac{M}{b}+T-1}{\frac{M}{b}}$ from the full space of possible mixes. Here, the value of parameter b can be user-specified or determined as follows. The user can specify an upper bound on the time allowed to pick the batch schedule. We can then determine the maximum number of variables the LP solver can handle within this limit, and use that to determine the highest feasible value of b. A reasonable default is to set b such that $|X|$ is around 10^5.

In this section we presented our interaction-aware batch scheduling algorithm. Next we present a scheduling algorithm for the scenario when clients submit queries in small batches or one at a time.

5 Interaction-aware On-line Scheduling

For many report generation workloads, queries are submitted to the database system not in large batches, but rather continuously or in small batches. In this section, we present an interaction-aware on-line scheduling algorithm that QShuffler uses for these workloads. The on-line

scheduling algorithm schedules a new query whenever a running query finishes. While making each scheduling decision, the on-line algorithm has to work with a limited *lookahead*, namely, the queries in the arrival queue (recall Figure 3). The online scheduling algorithm exploits the implicit batching of queries made possible by the queue. No assumptions are made about the future workload. In particular, no query is held up by our scheduling algorithm with the hope that other queries arriving in future could have positive interactions with this query. Thus, the challenge is to get the best possible global performance while being limited to local decisions under partial information.

When a query mix m runs on the database system, a *cost* is incurred based on the characteristics of m. There are a number of ways to measure the cost of running a mix. For example, the cost can be measured in terms of the load on resources like CPU, memory, and I/O bandwidth. (We will show shortly that conventional cost metrics are inadequate, and a new metric is needed.) A simple scheduling policy would always pick the next query to schedule as the one that gives the minimum cost among all queries present in the arrival queue. However, this greedy policy can be highly suboptimal. Consider a scenario where there are *light* queries and *heavy* queries in the queue. The greedy scheduler will keep scheduling the light queries until it has no option but to run a mix of heavy queries. This is a highly sub-optimal schedule with very poor performance: when the light queries are scheduled together, the system is *underutilized*, and when the heavy queries are scheduled together the system is *thrashing* [45].

A better, but more conservative, policy in the above scenario will try to keep a mix of light and heavy queries running in the system subject to system capacity and MPL. QShuffler's on-line scheduling algorithm takes such an approach. This algorithm makes decisions to achieve the objective of running the system as close as possible to a *cost threshold*. This conservative policy is aimed at avoiding overload while running query mixes that give good performance in the near term. Intuitively, the system takes on as much work as it can take efficiently in the near term so that it is not stuck with too much work in the far term.

We have designed the interaction-aware on-line scheduling algorithm using a template that can be instantiated with alternative implementations of the following three things:

1. A *cost metric*, R_i, for capturing the cost incurred by a query mix, m_i executing in the database system.
2. A *performance model* to compute $\hat{R}_i$, which is the estimated value of the cost metric R_i incurred by a query mix, m_i.
3. A *cost threshold*, θ_R, that specifies the desired value of $\hat{R}_i$ in the database system as query mixes are run.

Algorithm 1 On-line scheduling algorithm

GetNextQueryToSchedule(m_r: Current mix, AQ: Query arrival queue)

```
1  if (AQ is empty)
2     then return null; ▷ No queries to schedule
3  for i ← 1 to T
4        do
5           Let m_p be the query mix resulting from adding a query
6                 of type Q_i to m_r;
7           R̂_p ← Cost of m_p estimated using performance model;
8           Priority P_i ← 1/|θ_R − R̂_p|;
9           r[i] ← P_i; ▷ Array r stores priority of each query type
10 ▷ Schedule a query instance corresponding to the query type
11 ▷ with highest priority in the arrival queue
12 Sort r in decreasing order of priority;
13 for i ← 1 to T ▷ Traverse r in decreasing order of priority
14       do
15          Let Q_j be the query type corresponding to r[i];
16          if (AQ has an instance of Q_j)
17             then return earliest query in AQ of type Q_j;
```

Algorithm 1 shows the algorithmic template used by the on-line scheduling algorithm. This template provides a generic, low-overhead framework for scheduling that can be implemented within the database system or outside of it (e.g., in the JDBC driver). The template can be instantiated with any definition of the cost metric R_i, a corresponding cost threshold θ_R, as well as a performance model for estimating R_i (i.e., computing $\widehat{R_i}$) for candidate mixes.

When a query finishes, the *GetNextQueryToSchedule* function in Algorithm 1 picks the query to schedule next. The algorithm uses the performance model to answer the following what-if question: "For each query type, what would be the cost that results from adding a query of this type to the currently running query mix?" Each query type is assigned a *priority* based on how close it would keep the system to the desired cost threshold. A query instance belonging to the query type with the highest priority in the arrival queue is scheduled.

The overhead of the scheduling algorithm is a function of the number of query types, T, and not the size of the arrival queue. Thus, the arrival queue can be arbitrarily large without increasing the scheduling overhead. Having more queries in the queue is better for the scheduler since it provides more possible mixes to schedule. Practically, the arrival queue will have a bounded size that gives the scheduler its window into the future. We call the size of the queue the *lookahead*, L. QShuffler's focus is on total completion time of report-generation workloads, so delaying a query in the queue has no penalty (i.e., fairness is not required). Since report-generation workloads are bounded in size, all queries will eventually be scheduled; starvation is not an issue.

The on-line scheduling algorithm of QShuffler instantiates the algorithmic template in Algorithm 1 as follows. For cost metric, R_i, QShuffler uses the *NRO* metric described in Section 5.1. The cost threshold is θ_{NRO}, and Section 5.2 ex-

plains how to set this threshold. Finally, QShuffler employs regression models to compute $\widehat{NRO_i}$, the estimated value of *NRO* for a given mix m_i, and this is discussed in Section 7.

5.1 NRO: A Novel Cost Metric for Query Mixes

The main purpose of defining a cost metric for query mixes is to be able to separate "good" (low cost) query mixes from "bad" (high cost) query mixes while making scheduling decisions. It is tempting to consider cost metrics that are based on the demand placed on important resources while a query mix is running. Example metrics that fall into this category include CPU utilization and I/O bandwidth requirements.

One of our contributions is to show that resource-based cost metrics are inadequate to differentiate between good and bad mixes during scheduling. The intuitive reason is that different query mixes place very different demands on various resources. As a result, there often is no strong correlation between the average running times of queries in mixes and the observed resource consumption. In effect, we are stating that it is not possible to quantify the impact of query interactions by looking at one or more resource-consumption metrics alone.

Instead, our insight is that all different kinds of significant interactions happening in the database system should manifest themselves in the average run time that queries exhibit in a given mix. Thus, we develop a cost metric that relies on overall query execution time, and thereby accounts for all kinds of query interactions.

Our new cost metric for a query mix is called *Normalized Run-time Overhead (NRO)*. *NRO* is a measure of the run-time overhead that queries of different types incur when they run concurrently with other queries in a mix as compared to running alone in the system. Recall the following notation introduced in Section 3: t_j denotes the average run time of a query of type Q_j when it runs alone in the system, and A_{ij} denotes its average run time when run in the query mix m_i. We define the run-time overhead for the query type Q_j in mix m_i as $\frac{A_{ij}}{t_j}$. Note that this definition captures all kinds of interactions for this query type, including negative (where $A_{ij} > t_j$) and positive (where $A_{ij} < t_j$) interactions.

The next step is to generalize the definition of run-time overhead from a single query type to an entire query mix. Consider a query mix m_i with T query types and an MPL of M, with N_{ij} denoting the number of query instances of type Q_j in m_i. We define the overall run-time overhead for the T query types in the mix as the weighted average of their individual overheads. Here, the weight associated with query type Q_j is the fraction of queries of this type in the mix. Thus, the run-time overhead for mix m_i is:

$$RO_i = \frac{N_{i1} \times \frac{A_{i1}}{t_1} + N_{i2} \times \frac{A_{i2}}{t_2} + \cdots + N_{iT} \times \frac{A_{iT}}{t_T}}{N_{i1} + N_{i2} + \cdots + N_{iT}}$$

$$= \frac{1}{M} \sum_{j=1}^{T} \left(N_{ij} \times \frac{A_{ij}}{t_j} \right)$$

The value of RO_i represents the total run-time overhead for the query mix m_i with MPL M. To be able to use the same metric to measure overhead for mixes of different sizes (i.e., different MPLs), we define our cost metric NRO_i as the normalized overhead computed per query processed. That is, we divide RO_i by the MPL M to get NRO_i. This normalization captures the fact that incurring an overhead of, say, 5 while processing 20 concurrent queries is better than incurring an overhead of 5 while processing 10 concurrent queries. Thus:

$$NRO_i = \frac{RO_i}{M} = \frac{1}{M^2} \sum_{j=1}^{T} \left(N_{ij} \times \frac{A_{ij}}{t_j} \right)$$

We developed the *NRO* metric after considering several other cost metrics, none of which have the following desirable properties of *NRO*:

- *NRO* is not overly sensitive to the effect of a small number of long running queries in the mix. On the other hand, metrics like $\frac{\text{mix\_run\_time}}{\sum N_{ij} \times t_j}$ that are based directly on the total run time of the mix are less robust: the effect of a single query that suffers a large increase in run time in the mix will dominate even if none of the other queries in the mix suffer any degradation.
- At the same time, *NRO* does not average out overheads per query type so much that it cannot distinguish between good and bad mixes. Without careful averaging as done in *NRO*, significant overheads incurred by multiple individual query types can get lost in the overall average run time.
- Finally, *NRO* values correlate well with our intuitive separation of good mixes from the bad ones.

Next, we use example mixes of TPC-H queries to illustrate the usefulness of *NRO*. We show that while *NRO* is able to distinguish between good and bad mixes, resource-based metrics can fail to make this distinction.

Table 6 shows several mixes of the 6 longest-running query types on a TPC-H 1GB database on IBM DB2. The individual run times, t_j, for each of these 6 query types are shown in Table 2. For each mix, Table 6 shows the query frequencies, N_{ij}, and the average run time in seconds, A_{ij}, for each query type. The table also shows the values of three candidate cost metrics for each mix: (i) *NRO*, (ii) average number of disk transfers per second (a measure of disk consumption), and (iii) average CPU utilization. All mixes in the table have an MPL of 30.

The first two mixes in Table 6, m_{12} and m_{13}, are simple mixes consisting of multiple instances of one query type running concurrently. We see that *NRO* is small for m_{12}, which suggests that Q_1 queries do not interfere with each

Table 6 Values of different cost metrics for selected query mixes running on a TPC-H 1GB database

	Q_1		Q_7		Q_9		Q_{13}		Q_{18}		Q_{21}		Cost Metrics		
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}	NRO_i	*Disk (tps)*	*CPU (%)*
m_{12}	30	163.7	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0.542	4.6	99.9
m_{13}	0	0.0	30	331.8	0	0.0	0	0.0	0	0.0	0	0.0	1.920	269.8	69.6
m_{14}	11	143.9	8	144.6	3	211.2	2	97.8	2	149.8	4	127.5	0.630	195.1	95.8
m_{15}	2	361.7	8	298.6	1	476.0	18	121.2	0	0.0	1	231.2	1.026	270.8	80.9
m_{16}	0	0	0	0	2	463.5	25	135.0	2	304.3	1	385.4	0.873	306.9	75.7
m_{17}	0	0	1	206.6	20	184.9	9	113.4	0	0	0	0	0.651	307.6	76.9

other. (Recall that lower values of *NRO* are better.) The 30 Q_1 queries finish in 163.7 seconds in m_{12}, while it would take $30 \times 10.07 = 302.1$ seconds if we were to run these 30 instances sequentially. Here, 10.07 seconds is the t_j value of Q_1 for the TPC-H 1GB database from Table 2. Thus, m_{12}'s low *NRO* value matches the fact that m_{12} is a good mix.

On the other hand, m_{13} has a much higher *NRO* than m_{12}, which suggests that m_{13} is a worse mix. Indeed, the 30 Q_7 queries take 331.8 seconds to finish in m_{13}, compared to just $30 \times 5.76 = 172.8$ seconds for running these 30 instances sequentially (5.76 obtained from Table 2). Also, notice that the Q_7 queries in m_{13} take much longer to run than the Q_1 queries in m_{13} despite the fact that Q_1 is almost 2x slower than Q_7 when run alone in the system (see Table 2). The *NRO* cost metric captures this effect appropriately because *NRO* is not biased towards long-running queries.

Table 6 shows that the disk and CPU consumption of m_{13} is lower than that of some other mixes; which means that m_{13} is not placing an overly high load on system resources as compared to other mixes in the table. On the other hand, the CPU consumption of m_{12} is significantly higher than that of any other mix; so it may seem that m_{12} is overloading CPU. Thus, using CPU consumption as the cost metric would lead us to believe that m_{12} is worse than m_{13}, which is an incorrect conclusion as we can see from the query run times.

Mixes m_{14} and m_{15} further illustrate how, unlike resource-based cost metrics, *NRO* can differentiate good mixes from bad ones. *NRO* tells us that m_{15} is costlier than m_{14}, which we can indeed see by comparing A_{ij} values between the two mixes. An algorithm that uses the *NRO* metric will schedule query mixes like m_{14} and avoid mixes like m_{15}. As before, the resource consumption metrics are not useful for distinguishing between the performance of these two mixes. Mix m_{15} places a higher load on disk than m_{14}, but m_{14} places a higher load on CPU than m_{15}. Furthermore, the resource consumption levels of these two mixes are lower than the individual highs in Table 6.

The final two mixes, m_{16} and m_{17}, clearly demonstrate the inadequacy of resource-based cost metrics. These two mixes are virtually indistinguishable if we consider resource-based metrics alone. On the other hand, *NRO* tells us that m_{16} is costlier than m_{17}. We can validate this observation qualitatively by considering the average completion times of Q_9 and Q_{13}, which are the two query types common between the mixes. For both query types, performance in m_{16} is much worse than in m_{17}.

We have observed effects similar to those described here for a variety of different workloads and while using various resource consumption metrics such as CPU queue length, disk queue length, and bytes transferred per second: the level of consumption of a single resource or of a combination of multiple resources cannot consistently distinguish good mixes from the bad ones, while *NRO* can distinguish good mixes from bad ones.

5.2 Setting the Cost Threshold

The cost threshold θ_{NRO} is an important tuning parameter in the on-line scheduling algorithm. The setting of θ_{NRO} exposes a tradeoff that we will illustrate using our earlier example of a workload that consists of light and heavy queries. If θ_{NRO} is set low, the low-cost mixes composed almost exclusively of light queries will have priority over all other mixes during scheduling. This situation can have two undesirable consequences: (i) resources will be underutilized, and (ii) the heavy queries will queue up and ultimately force a situation where high-cost query mixes have to be run for long-periods.

The above problem cannot be solved by increasing θ_{NRO} arbitrarily because a high θ_{NRO} will tend to favor high-cost, and hence poorly-performing, query mixes over better ones. There is some optimal value for θ_{NRO} that depends on the (unknown) future query workload. We have developed a solution to pick a robust setting of θ_{NRO} that leverages the desirable properties of QShuffler's batch scheduling algorithm. Our solution uses the following three steps:

1. Choose a representative workload W_R
2. Run the batch scheduling algorithm on W_R to generate the corresponding batch schedule S_R
3. Compute θ_{NRO} as a weighted average of the *NRO* values of the mixes chosen in S_R

We will describe each of these steps in turn.

Choosing a Representative Workload W_R: The representative workload W_R can be specified by the database adminis-

trator similar to what is required by popular tools like physical design advisors [46,47]. QShuffler can automate this process partially because report-generation workloads tend to repeat themselves with a high degree of regularity. For example, the same set of reports may be generated every night or every weekend. In these situations, simple hints from administrators that give the time period of the workload cycle are enough to capture a representative workload.

If the variability in the workload is too high to be captured in a single representative workload, then we can collect different workloads for different time periods (e.g., every hour). QShuffler then relies on the practical heuristic that the recent past is a good predictor of the near future, and uses the workload from the last time period as the representative workload for the next time period.

Running the Batch Scheduling Algorithm: Next, the batch schedule S_R for the representative workload W_R is determined by running the batch algorithm from Section 4 on W_R. Only the schedule is computed; the workload is not actually run. Recall that the batch algorithm chooses a good set of mixes to schedule based on statistical models to estimate query completion time.

Picking θ_{NRO}: After the batch scheduling algorithm we have:

- $m_1, m_2, \ldots, m_T$, which represent the T query mixes comprising the batch schedule S_R for W_R (recall Lemma 1). S_R is a good approximation of the optimal schedule for W_R.
- n_1, n_2, …, n_T, which represent the run time of the respective mixes m_1, m_2, …, m_T in S_R. Recall from Section 4 that the LP which computes S_R also gives the time that each mix will run for in S_R.
- $NRO_1, NRO_2, \ldots, NRO_T$, which represent the *NRO* values of the respective mixes m_1, m_2, …, m_T in S_R. *NRO* values are computed using the performance model.

We set θ_{NRO} to the weighted average of the *NRO* values of the mixes in the batch schedule S_R. Here, the weight of each *NRO* value is the fraction of time for which its corresponding mix will run in S_R. That is:

$$\theta_{NRO} = \frac{n_1 \times NRO_1 + n_2 \times NRO_2 + \ldots + n_T \times NRO_T}{n_1 + n_2 + \ldots + n_T}$$

Intuitively, this approach aims to set θ_{NRO} such that the schedule generated by the on-line algorithm will be close to the best batch schedule for the representative workload. Steps 1-3 will be run once to set θ_{NRO} if a single representative workload can be identified. The value of θ_{NRO} will be periodically recomputed if the predicted workload is different for different time periods. The process of recomputing θ_{NRO} for a new workload is very efficient because the bottleneck is in computing the new batch schedule, which can be finished within seconds (recall Section 4.3).

6 Scheduling Based on Query Optimizer Cost Estimates

In this paper we argue that the analytical cost models used by database query optimizers may not be the best choice to reason about query interactions. Instead, we propose capturing the impact of query interactions by measuring how they affect the average completion time of different query types. We would like to compare our proposed approach for scheduling to an approach that is based on query optimizer cost estimates. In this section, we describe a query scheduler that uses query optimizer cost estimates, and we experimentally compare against this scheduler in Section 9.

The scheduler that uses query optimizer cost estimates is based on the work of Niu et al. [33,34,48,49] (in particular, the query scheduler described in detail in [33,48]). The query scheduler in these works uses the query optimizer cost estimates (termed as *timerons* in IBM DB2) to measure the cost of the queries executing in the system. The scheduler uses a *timeron threshold* to define the capacity of the system. The scheduler admits a query if admitting it will not increase the total optimizer cost (in timerons) of all queries executing in the system beyond the timeron threshold. The timeron threshold is defined using an experiment that is conducted off-line before scheduling starts. In this experiment, a varying number of queries is executed concurrently, and the throughput of the system is plotted against the total timerons to determine the timeron value that results in peak throughput. This timeron value is used as the timeron threshold. The scheduler can also handle different service classes for different query types, and can perform admission control separately for the different service classes by defining a separate timeron threshold for each service class.

The work in [33,34,48,49] also includes a general framework for workload adaptation that can handle scheduling for time-varying workloads. In addition, the different service classes can have different service level objectives, and the scheduler dynamically adjusts the timeron threshold for each service class based on a utility function that quantifies how well the different service level classes presently meet their service level objectives. In this paper, we do not use the framework for workload adaptation since our workloads are not time varying so we should be able to statically determine the timeron threshold. Our performance objective is maximizing overall throughput without distinguishing between different query types, so we do not define different service classes with explicit service level objectives. Moreover, we do not rely on a utility function since the "utility" we are maximizing is simply throughput.

Despite these differences, we still find that the work of Niu et al. represents a useful comparison point because it enables us to answer the following two questions: (1) Can we use query optimizer cost estimates as our cost metric for scheduling? and (2) Can we effectively use different service classes to distinguish between "lightweight" query types and

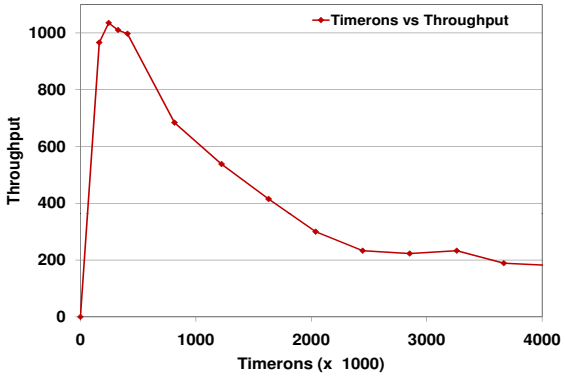

Fig. 4 Throughput vs. timerons for Q_{13} in the 1GB database

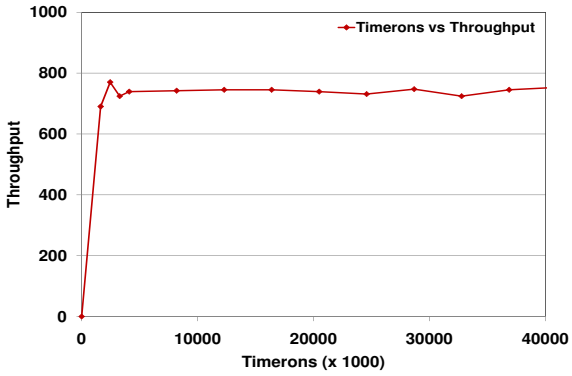

Fig. 5 Throughput vs. timerons for Q_{21} in the 1GB database

"heavyweight" query types as measured by their query optimizer costs? A scheduler with these two service classes may be able to indirectly capture query interactions. In particular, such a scheduler may be able to avoid scheduling too many heavyweight queries concurrently.

To answer these questions, we adapt the scheduler of Niu et al. to our setting. The adapted scheduler works as follows: (1) before the workload runs, define a timeron threshold for the system, (2) (optionally) divide the query types into lightweight and heavyweight based on their optimizer cost estimates and divide the timeron threshold between these two service classes, (3) when the workload runs, admit the next query only if this will not increase the total timeron cost of all queries in the system (or in this query's service class) beyond the timeron threshold. We refer to this scheduler as the *optimizer-based scheduler*.

A fundamental difference between the optimizer-based scheduler and our batch and on-line schedulers is that the optimizer-based scheduler allows the MPL to vary within the execution of one workload, while our schedulers operate at a fixed MPL for a given workload execution. Using a fixed MPL is the most common approach used for BI workloads [32], and our schedulers can handle any MPL chosen by the DBA. We show experimentally in Section 9 that our schedulers work well across a wide range of MPL values, but we always use one MPL for each workload execution and this MPL does not change while the workload is running. On the other hand, the optimizer-based scheduler allows MPL to vary subject to the timeron threshold(s).

The first step of the optimizer-based scheduler is to define the timeron threshold that represents system capacity. In our setting, we know the query types a priori but we need to handle different workloads that consist of queries of these types. The query types have widely varying estimated and actual execution costs, which makes finding the timeron threshold difficult. Finding the timeron threshold requires finding the system saturation point at which throughput peaks, but the system saturation point depends on the workload. We illustrate this with a concrete example.

Consider Q_{13} and Q_{21} in Table 2. In our 1GB database, the estimated cost of Q_{13} is 81,507 timerons and that of Q_{21} is 819,324 timerons. Thus, the estimated cost of Q_{21} is more than 10 times that of Q_{13}, while the actual completion time of Q_{21} is only 1.2 seconds more than that of Q_{13}. Figures 4 and 5 show throughput vs. timerons for Q_{13} and Q_{21}, respectively, as the number of concurrently executing queries increases. The system saturation point for Q_{13} is 244,521 timerons, while the saturation point for Q_{21} is 2,457,973 timerons. Using these two query types results in timeron thresholds that differ by an order of magnitude. Furthermore, if there exists a global timeron threshold for all query types, then for every instance of Q_{21} we should be able to admit 10 instances of Q_{13}. However, the figures clearly show that the throughput of Q_{13} drops quickly as we add more queries beyond its saturation point, while for Q_{21} we can keep adding instances without a significant drop in throughput. This example clearly illustrates that optimizer cost estimates can be misleading indicators of the actual performance and resource consumption of different queries. The example also shows that there is no straightforward way to find a system saturation point that works for different workloads even if we know the query types. We cannot plot a throughput vs. timerons graph without having a specific workload, and in our setting we do not have a specific workload that is known a priori.

To find the system saturation point and the corresponding timeron threshold in our setting, we propose the following methodology. We create a workload consisting of an equal number of queries of each query type (say, 10 queries of each type) and we randomly shuffle these queries. We run this workload at different MPLs and find the workload run with the best throughput (i.e., the lowest total completion time). This workload run corresponds to the system saturation point, and we use it to define the timeron threshold.

Defining the timeron threshold requires averaging the timeron values throughout the workload run, which itself is not straightforward. We propose two approaches for averaging the timeron values to obtain the timeron threshold.

Both approaches rely on tracking the query mixes that executed in the workload run and the total timeron cost for each query mix. The first approach is to simply average the timeron costs of these query mixes, which gives a timeron threshold $Thr_{mix-averaged}$ defined as follows:

$$Thr_{mix-averaged} = \frac{\sum_{i=1}^{k} opt_i}{k}$$

where k is the total number of mixes that executed in the workload run and opt_i is the total timeron cost of mix i. The second approach is to use a time-weighted average of the timeron costs, which gives a timeron threshold $Thr_{time-weighted}$ defined as follows:

$$Thr_{time-weighted} = \frac{\sum_{i=1}^{k} (l_i \times opt_i)}{\sum_{i=1}^{k} l_i}$$

where l_i is the time in seconds for which mix i ran. We found that these two threshold values were close to each other and gave similar scheduling results, with $Thr_{time-weighted}$ performing slightly better, so we use $Thr_{time-weighted}$.

To use service classes in the optimizer-based scheduler, we define two service classes, one for lightweight query types and one for heavyweight query types. We manually place each query type in one of the two service classes based on its optimizer cost. In our experiments we found that there is a large difference between the optimizer costs of lightweight and heavyweight query types, so there was no ambiguity in assigning query types to service classes (details in Section 9). Instead of evaluating a specific algorithm to find the best way of dividing the timeron threshold between these two service classes, we conducted experiments in which we varied the fraction of the timeron threshold given to each service class across a wide range of values. In all these experiments, using one service class was superior to using two service classes that distinguish between lightweight and heavyweight queries. Thus, we conclude that using two service classes does not improve the performance of the optimizer-based scheduler, so we did not try to find a "best" way for dividing the timeron threshold between the two service classes.

7 Experiment-driven Modeling

The on-line and batch scheduling algorithms pick a query mix m for scheduling based on the estimated properties of m. The on-line algorithm has to estimate mix m's *NRO*, and the batch algorithm has to estimate the average completion time (A_{ij}) of each query type in m.

One approach is to develop analytical formulas to estimate *NRO* and A_{ij} for mixes. Historically, analytical formulas have been used successfully by database query optimizers to estimate the execution cost of query plans. 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.

While general-purpose analytical formulas are hard to develop, a robust and effective alternative exists: using *statistical modeling* based on actual observations of query interactions. Our approach for statistical modeling 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 6. Each sample gives a measure of how the average completion time of different query types is affected by running them in a specific mix. The set of collected samples can be used to identify various interactions, and statistical models can be trained from the collected samples to estimate *NRO* and A_{ij}.

This approach is not sensitive to the causes of interactions because the effect of all interactions will show up in the samples. The rest of this section gives the full details of our experiment-driven modeling approach. The effectiveness of this approach is shown empirically in Section 9.

Sampling: There are two parts to our experiment-driven modeling approach: (1) sampling, and (2) statistical modeling. For the statistical model to accurately reflect query interactions, it is important to identify a representative set of samples. Each sample is collected by identifying a query mix m_i and scheduling an *experiment* where the selected query mix is run to observe the average query completion times (A_{ij} values) of the queries it contains. The value NRO_i is computed for mix m_i.

A straightforward approach to select samples (and hence experiments) is to sample randomly from the space of possible query mixes. A disadvantage of pure random sampling is that some query types may not appear at all in the samples, especially if there are few samples. Furthermore, important parts of the space (e.g., the corners) may not be covered. To selectively cover different parts of the space of mixes with a small number of samples and ensure that query interactions are adequately represented, we developed a new sampling approach called *corner, diagonal, and random (CDR) sampling*. CDR sampling collects samples at MPL M as follows:

1. We start by running T experiments where we sample the "corner" points of the space, i.e., the mixes $\langle M,0,\ldots,0\rangle$, $\langle 0,M,\ldots,0\rangle$, $\ldots$, $\langle 0,0,\ldots,M\rangle$.
2. 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 ran-

dom 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 the range of values in $1, \ldots, \frac{M}{T} - 1$.

3. Finally, we take some samples completely at random (like random sampling) from the full space of mixes.

Once we have a set of samples, we can fit statistical models to these samples to predict A_{ij} and NRO. The models compute an estimate for A_{ij} or NRO ($\widehat{A_{ij}}$ or $\widehat{NRO}$) for mix m_i as a function of $N_{i1}, N_{i2}, \ldots, N_{iT}$, the number of queries of each type in the mix.

In our empirical evaluation, we measure how many samples are needed to produce fairly-accurate models for estimating A_{ij} and NRO_i values. We will show that these values can be estimated with reasonable accuracy from a small number of 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 algorithms performs well as long as they 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} and NRO_i values with high absolute 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.

Statistical Models: We consider two types of models: *linear models* and *regression trees*. A linear model uses the following structure to compute $\widehat{A_{ij}}$, the estimate of A_{ij} for mix i and query type j, and $\widehat{NRO_i}$, the estimate of NRO_i for mix i respectively :

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

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

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

Note that linear regression is among the simplest types of statistical models and hence it is very easy to construct with many popular software tools (e.g., Excel). However, its simplicity means that it may not be the most accurate model, which sometimes leads research works that focus on the prediction accuracy of different models (independent of a specific application) to eschew it in favor of more complex models (e.g., [35]). In our paper, the performance model has to be accurate enough to distinguish good mixes from bad mixes. One of our contributions is to show that this can be done effectively with linear regression without the need to resort to more complex statistical models.

Regression trees are piecewise regression models [50, 51]. 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 $\widehat{A_{ij}}$ and $\widehat{NRO_i}$ 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 CART [51]).

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 models incrementally.

8 Identifying Query Types Automatically

So far we have assumed that the query types $Q_1, \ldots, Q_T$ are given. In a production BI setting, the query types could be identified by the DBA or they could be tagged by the application. This section presents some novel techniques that we developed to simplify the task of choosing query types.

One straightforward technique is to have one-to-one correspondence between *query templates* and query types, i.e., each distinct template forms a query type. A query template consists of SQL text along with possible *parameter markers*. The TPC-H decision support benchmark defines T=22 distinct query templates. The following template, derived from TPC-H, is an example query template with one parameter marker which is represented by the symbol "?". Different value settings of the parameter marker give rise to different instances of this query template.

```
Select * From lineitem as l, orders as o,
     supplier as s, nation as n
Where l.l_orderkey = o.o_orderkey and
      l.l_suppkey = s.s_suppkey and
      s.s_nationkey = n.n_nationkey and
      n.n_name = ?
```

However, equating query types to query templates can be a suboptimal choice in the presence of data skew. Both the online and batch scheduling algorithms represent the performance of all instances of a query type Q_j in a mix m_i based

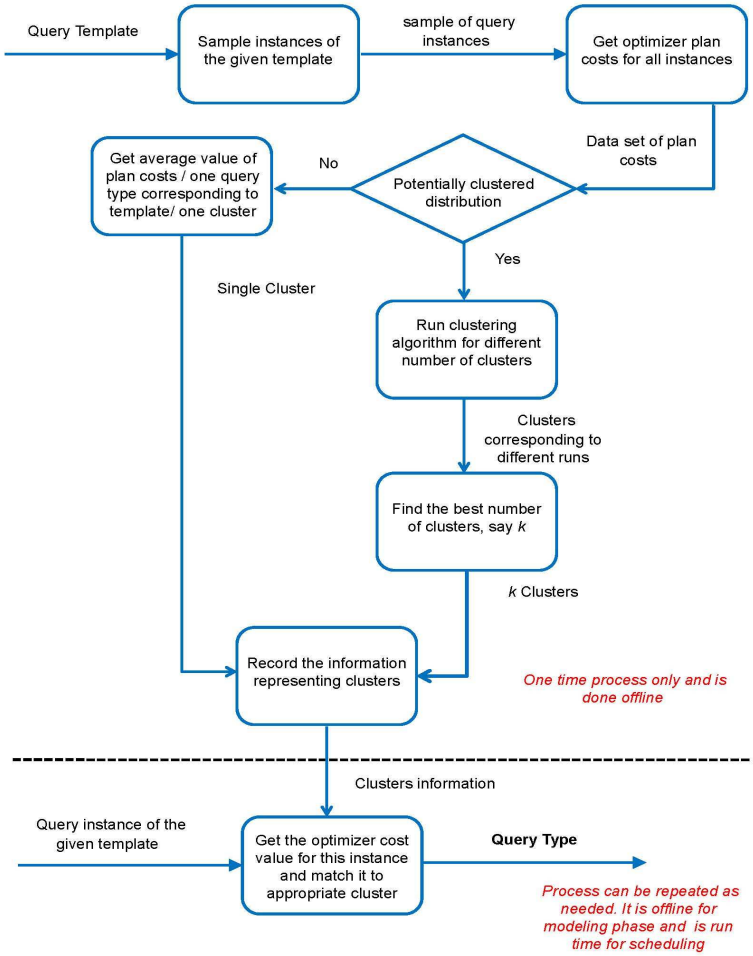

Fig. 6 Identifying query types from a given query template

on a single number A_{ij}. A_{ij} denotes the average completion time of instances of Q_j in m_i, so this representation works as long as all instances of Q_j have similar performance in m_i. The presence of data skew can cause two Q_j instances q_1 and q_2 with different values of the parameter marker(s) to perform differently. For example, an instance of the above TPC-H template for n_name="USA" could take much longer to run than say for n_name="Mexico". To deal with such behavior, we have to further partition the query template into two or more query types.

Our methodology to determine the best set of query types automatically consists of two modules: (i) a *query template extractor*, which extracts distinct query templates and the distributions of parameter marker values from database query logs; and (ii) a *query template partitioner*, which partitions a template into multiple query types if different instances of that template can differ significantly in performance. We describe each of these in turn.

Query Template Extractor: The extractor parses database query logs to extract all distinct templates corresponding to queries that executed in the system over a given period of time. This process involves log parsing, identifying parameter values in the query text and replacing them with parameter markers, and using a canonical representation of query templates to facilitate string-based comparisons between extracted templates. Along with the distinct query templates, the extractor also returns the distribution of values seen for each parameter marker.

Counterparts of the query template extractor exist for almost all database systems (e.g., [52]). Query templates are used routinely for purposes like: (i) avoiding the overhead of query optimization for templates seen frequently, and (ii) enforcing the use of manually-tuned plans for important queries that repeat. There has also been work on extracting query templates directly from the source code of database applications rather than parsing logs postmortem [53,54].

Query Template Partitioner: The partitioner takes a query template Q (possibly output by the template extractor) with parameter markers as input. The output returned is a partitioning of Q into one or more query types such that query instances of the same type have similar performance. Modern query optimizers already account for skew in data values while choosing query plans and estimating plan cost (see, for example, [55] and the references therein). Thus, we need

not reinvent the wheel on that front. The partitioner's workflow, summarized below and in Figure 6, leverages the optimizer to generate the query types for a given template Q.

1. Generate a large number, say n=1000, of query instances from Q by instantiating each of Q's parameter markers with values sampled from the corresponding distributions. Let the instances be $q_1,\ldots,q_n$.
2. For each instance q_i, run the query optimizer in what-if mode to find q_i's execution plan ρ_i, and ρ_i's estimated cost c_i. Thus, we get n tuples of the form $\langle q_i, \rho_i, c_i \rangle$.
3. Consider the one-dimensional distribution of c_i values, and decide whether these values naturally form a single cluster or multiple clusters. If there are multiple clusters, then find the best clustering of the values. The centroids of the clusters define the partitioning of Q into one or more query types.
4. To determine the type of a given instance q of Q, we use the query optimizer to find q's plan ρ and associated cost c. The centroid closest to c defines q's type.

Step 3 is the most nontrivial step in the partitioner's workflow. The first decision in Step 3 involves determining whether the plan cost values c_i, $1 \leq i \leq n$, naturally form one cluster or more. To make this decision, we compute the *Coefficient of Dispersion (CoD)*—also known as the *Fano factor*—of the c_i values [56,57]. Coefficient of dispersion is defined as the ratio of variance to mean:

$$CoD = \frac{variance}{mean} = \frac{\sum_{i=1}^{n}(c_i - \bar{c})^2}{n \times \bar{c}}$$

$\bar{c}$ denotes the mean of the c_i values. $CoD > 1$ (over dispersion) is a common rule of thumb used by statisticians to determine that the data is best represented by more than one cluster. We follow this guideline to determine whether to run the clustering algorithm or not on the dataset of c_i values.

There is an abundance of literature on clustering. In our study of plan cost datasets, we have found that the simple *K-means* clustering algorithm works very well. K-means can be described as a partitioning algorithm that partitions the given dataset into a user-specified K number of clusters. Each cluster is represented by its centroid. K-means starts by choosing K initial centroids. Then, it proceeds in an iterative manner assigning data points to clusters so as to minimize the sum of distances from each data point to the centroid of the cluster to which the point is assigned. The data points may switch clusters during the iterations, and the centroids are recomputed until the sum cannot be minimized anymore. The clustering result may be dependent on the initial values chosen for the K centroids. This problem is addressed in practice by repeating the clustering algorithm a few times; and then choosing the cluster partitioning that gives the minimum total sum of distances of points to their cluster centroids.

A critical component of the clustering process is to pick the best value of K automatically. Luckily, this problem has been well studied in the K-means literature. To determine the best value of K, we adopt the *silhouette coefficient* [58] as a metric of the quality of a given clustering of the c_i values. Let $C_1,\ldots,C_K$ denote the K clusters produced by running K-means on the c_i values. For a data point $c \in C_j$, let a_c represent the average distance of c to all the points in cluster C_j. Let b_c represent the minimum over the average distances of c to the points in cluster C_i, $1 \leq i \leq K$, $i \neq j$. That is, b_c is the average distance of c to the closest cluster other than the cluster that c belongs to. The silhouette coefficient of c is defined as:

$$s_c = \frac{b_c - a_c}{max(a_c, b_c)}$$

Intuitively, $-1 \leq s_c \leq 1$ measures how close c is to the points placed in the same cluster as c, compared to points placed in other clusters. That is, s_c compares the intra-cluster distance with the smallest inter-cluster distance from c's perspective. Values of s_c close to the maximum value of 1—which indicates that the inter-cluster distance dominates the intra-cluster distance—denote a good clustering of c.

The notion of silhouette coefficient can be extended to the overall clustering by averaging s_c across all the clusters $C_1,\ldots,C_K$. We define S_K, the silhouette coefficient of the clustering produced by running K-means to produce K clusters, as follows:

$$S_K = \frac{1}{K}\sum_{j=1}^{K}\frac{\sum_{c \in C_j} s_c}{|C_j|}$$

Our goal is to identify the value of K that maximizes S_K for the given set of c_i values. We run K-means with larger and larger values of $K \geq 2$ until we see a consistent drop in S_K as K is increased further. For the TPC-H queries, the largest value of S_K was produced in the $2 \leq K \leq 5$ range.

The output of Step 3 for a given query template Q is either: (i) a validation that Q can be treated as a single distinct query type, or (ii) a partitioning of Q into K query types identified by the cluster centroids produced by running the K-means algorithm with the K value with maximum S_K. Note that Steps 1-3 are done off-line. These steps are not on the critical path of query scheduling.

Suppose we want to find the type of a query instance q of a given template Q in Case (ii). We first use the query optimizer to find q's plan ρ and associated cost c. This step does not usually involve additional overhead since it can be piggybacked with the regular query optimization process of finding the plan to execute q. We then find the cluster to which the plan cost c belongs by finding the nearest neighbor to c among Q's cluster centroids. The type corresponding to the nearest centroid is returned as q's query type. With this mechanism in place to identify query types, no change is required to our modeling or scheduling techniques.

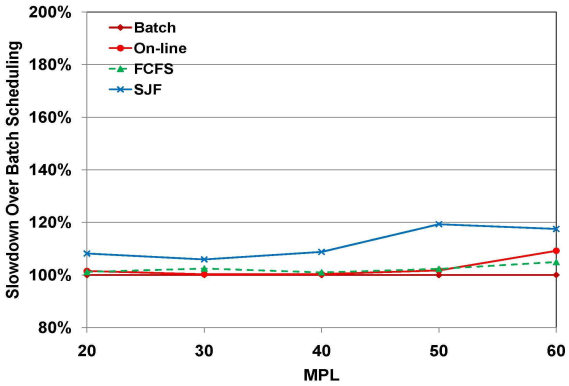

Fig. 7 Scheduling for $p = 5$

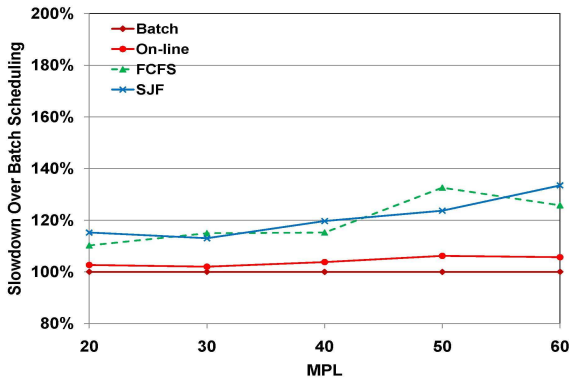

Fig. 8 Scheduling for $p = 25$

9 Experiments

9.1 Experimental Setup

Machine and database: Our experiment were run on a machine with dual 3.4GHz Intel Xeon CPUs and 4GB 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 otherwise noted, we always use the 1GB database with the standard TPC-H data generator that generates uniform data. The exceptions to this are Sections 9.3 and 9.5, in which we use a 10GB database, and Section 9.4 in which we use a data set with a skewed data distribution. The buffer pool size was set to 400MB for the 1GB database, and 2.4GB for the 10GB database. We used the DB2 Design Advisor to recommend indexes for the TPC-H workload. We ran the DB2 Configuration Advisor to ensure that the configuration parameters are well tuned. In our experiments, we vary MPL, M, from 20 to 60. The default MPL for DB2 (the *number of agents*) is 200.

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

Arrival order: As we demonstrate in our motivating example in Section 1, the arrival order of workload queries is important since it determines the query mixes that the system encounters and hence the total completion time of the workload. Thus, to stress test QShuffler, we systematically vary the arrival order of workload queries according to the following strategy. We arrange the query types in our workload in the order in which they are specified in the TPC-H benchmark (i.e.Q_1 first, then $Q_3, \ldots, Q_{21}$). As an initialization step, we go through the list of queries and place IQ instances of each query type in the arrival queue. This ensures that the system has a balanced initial workload. 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. As p increases, more queries of the same type arrive together. For the 1GB database, we use $IQ = 10$ and $p = 5$, 25, and 50. We use a pool of 60 instances of each query type to construct our workloads. For the 10GB database, we use $IQ = 0$ and $p = 2$, 5, and 10. We use 10 instances of each query type. We limit the workload sizes for the 10GB database due to the long run times of queries on this database, e.g., a workload consisting of 60 queries can take more than 5 hours.

Scheduling algorithms: We experimented with five different scheduling algorithms. The first is our batch scheduling algorithm, which requires the entire workload to be known in advance. The four other algorithms do not require the workload to be known in advance, and we assume that the scheduler can see the next L queries in the arrival queue. The algorithms are: First Come First Served (FCFS), which is insensitive to L; Shortest Job First (SJF), which schedules the shortest query available in the next L queries using the run times in Table 2; our on-line scheduling algorithm; and the optimizer-based scheduler (Section 9.5).

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. The batch scheduling algorithm chooses a schedule by taking the entire workload into consideration, so we use it to judge the quality of the on-line, SJF, FCFS algorithms. We measure the performance of each of these algorithm in terms of *slowdown compared to batch schedule*, defined as:

$$\frac{\text{completion time of on-line/FCFS/SJF schedule}}{\text{completion time of batch schedule}} \times 100\%$$

9.2 Scheduler Effectiveness

Figures 7, 8, and 9 show the performance for different MPLs of our on-line scheduling algorithm, FCFS, and SJF for p = 5, 25, and 50, respectively. The workload consists of 60 instances of each of the 12 longest running TPC-H query types on a 1GB database, for a total of 720 queries. The

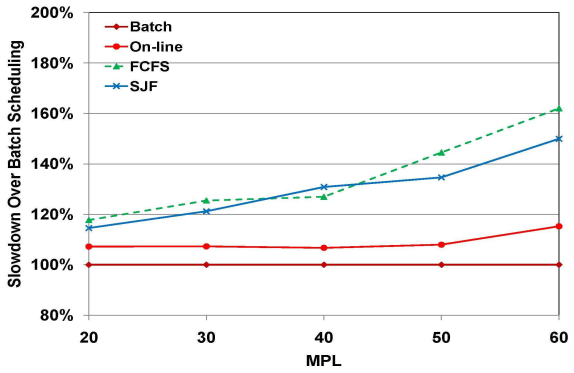

Fig. 9 Scheduling for $p = 50$

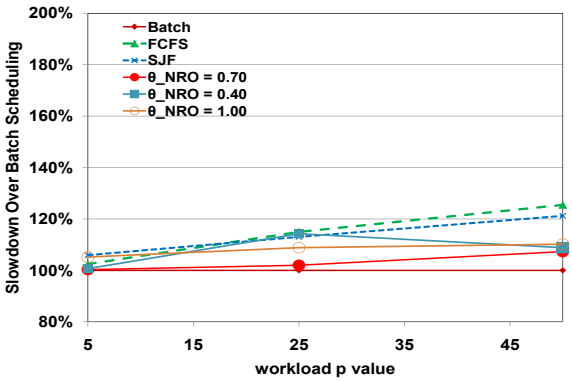

Fig. 10 Scheduling for different values of θ_{NRO} ($M = 30$)

lookahead is $L = 60$. The methodology in Section 5.2 results in a value of θ_{NRO} between 0.63 and 0.7 in all these cases. Therefore, we use $\theta_{NRO} = 0.7$ for all our experiments unless otherwise stated. The figures show that the batch and on-line scheduling algorithms of QShuffler are significantly better than FCFS and SJF. The on-line algorithm performs worse than the batch algorithm, as expected, but the difference between them is low.

The figures clearly demonstrate the benefit of interaction-aware scheduling. The performance gap between the QShuffler algorithms and the other two algorithms increases as p increases. FCFS is the scheduling algorithm used by all database systems that we are aware of, and these experiments show that its sensitivity to the arrival order can significantly degrade its performance. When we examine the NRO and A_{ij} values from our sampling data for this $1GB$ database, we find that the heterogenous mixes (i.e., containing many different query types) are among the best mixes. Thus at $p = 5$, the arrival order is almost approaching round robin, and not too many queries of the same type can arrive together, so FCFS is able to keep up with QShuffler. But as p increases, and the arrival order starts sending "bad" mixes, the performance of FCFS starts degrading significantly. Further, 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 and SJF), we note that for $p = 50$ the performance gain of the on-line scheduler over FCFS is up to 40%. This gain comes "for free" simply by scheduling the queries in the correct way.

The figures also 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.

Our methodology for setting θ_{NRO} requires using the batch scheduling algorithm on a representative workload. If the DBA wrongly assumes that the workload is going to consists of only those query types that have severe negative interaction, then most of the mixes that are proposed by the batch scheduling algorithm would have a higher value of NRO and θ_{NRO} is going to be higher than required. On the other hand, if the DBA uses a workload that consists of only good mixes, then θ_{NRO} is going to be lower than required. Choosing the representative workload for setting θ_{NRO} is important, but the DBA does not need to run or know the exact ordering of queries for this workload; a rough estimate of the number of queries of each type in the workload is sufficient. Moreover, we observed that the on-line scheduling algorithm is quite robust to small variations in θ_{NRO}. To study the effect of large variations in θ_{NRO}, Figure 10 shows the performance of different workloads for $M = 30$ when we set θ_{NRO} unexpectedly low or high corresponding to cases where the DBA did not choose an accurate representative workload. We can see that the on-line algorithm is still either better or at least comparable to FCFS and SJF.

9.3 Scalability and Robustness of QShuffler

We study the scalability of QShuffler in two dimensions: query types T and database size. As T increases, the space of possible query mixes increases, which affects both model building and scheduling. To test QShuffler for higher T, we use a workload comprised of 21 of the 22 queries in the TPC-H benchmark. We do not use Q_{15} 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. For comparison, we show the performance of an algorithm called *on-line incremental* that applies our on-line scheduling using a coarse-grained model that captures only the 12 longest of the 21 TPC-H query types. The remaining 9 query types are lumped together into a "catch-all" query type, i.e., these short query types are indistinguishable from one another for scheduling.

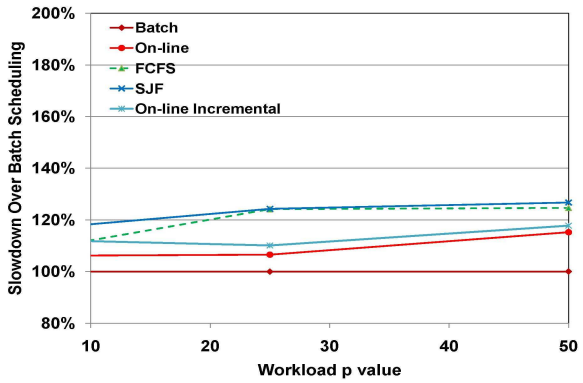

Fig. 11 Scheduling for $T = 21$ ($M = 30$)

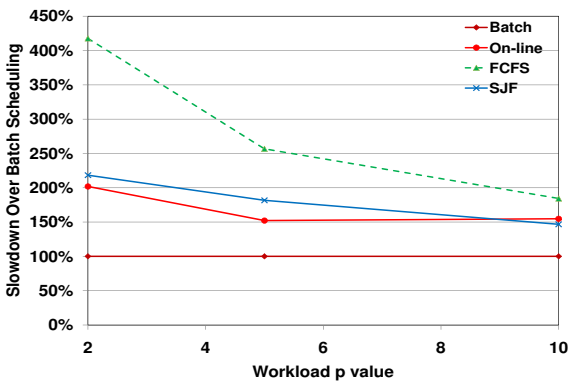

Fig. 12 Scheduling for $10GB$ database ($M = 10$)

Figure 11 shows the performance of the different scheduling algorithms for this workload for MPL 30 and varying p. The figure shows that, as in previous experiments, the on-line scheduling algorithm performs better than FCFS and SJF. The figure also shows that the on-line incremental algorithm performs very close to the on-line algorithm that uses a fine-grained model with all 21 query types. Thus, the quality of the schedule chosen by QShuffler remains good even with the coarse-grained model, which motivates an incremental approach to model building that focuses on the higher-impact query types first.

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 workload consists of 10 queries of each type for a total of 60 queries, the lookahead $L = 10$, and MPL is set to 10. θ_{NRO} was computed based on this workload to be 0.33. The arrival order of the queries is determined based on the parameter p, and we use $p = 2, 5$, and 10, since we have only 10 queries of each type.

Figure 12 shows the performance of the different scheduling algorithms for this workload. The completion time for the batch scheduling algorithm in this case is 1.78 hours and it is significantly better than any other algorithm, e.g., beating FCFS (7.43 hours) by a factor of 4.2. This shows the potential of interaction-aware scheduling. The figure also shows that the on-line scheduling algorithm consistently performs better than FCFS, and better than SJF except when $p = 10$. 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. This results in SJF and FCFS both improving as the arrival patterns become more skewed, since both algorithms would schedule queries of the same type in this case.

The experiments in this section show that our scheduling algorithms are able to exploit different scheduling opportunities in different scenarios. In case of the 1GB database, the mixes containing one query type are bad mixes. The workloads with higher values of p tend to present these kind of mixes, so there is more opportunity for performance improvement over FCFS at higher values of p. On the other hand, for the 10GB database, the mixes containing many different query types are bad mixes, so there is much more opportunity for performance improvement over FCFS at lower values of p. Our scheduler is able to take advantage of the scheduling opportunities in both these cases.

9.4 Scheduling for Skewed Databases

In this section we present the results for robustness of our scheduler in case of skewed data sets. Our approach for dealing with skewed data distributions, presented in Section 8, is to divide each query template into more than one query types. To test the robustness of our approach, we use the skewed TPC-D/H database generator available at [59]. This database generator populates a TPC-D/H database using skewed random values that are distributed according to a Zipf distribution. This distribution has a parameter z that controls the degree of skew, where that $z = 0$ 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 $z = 1$.

We consider the same 6 long-running TPC-H query templates from Table 2 as before. These are the templates with the longest run times on a uniformly distributed database. Our first step is to see how many of these given query templates will be divided into more than one query types. For this we generate 200 instances of each query template by randomly varying the parameter markers and get their plan cost values by running the DB2 optimizer in its "show plan" mode (called the EXPLAIN mode in DB2). Then we run our coefficient of dispersion test on these plan cost values. All the queries show some variance in their plan cost values for different instantiation of the parameters, as expected. Interestingly, however, only the plan cost of instances of Q_9 show enough clustered distribution to merit further analysis. The value of the coefficient of dispersion for this query

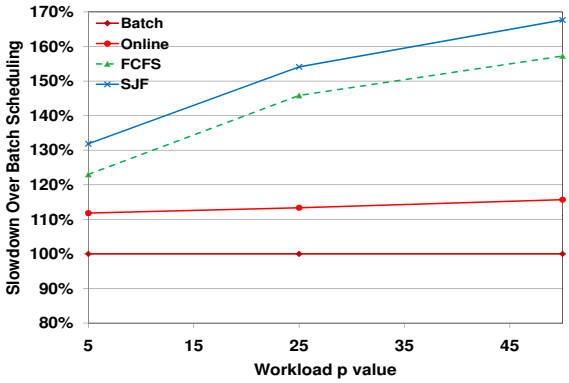

Fig. 13 Scheduling for skewed data with $K = 4$ ($M = 30$)

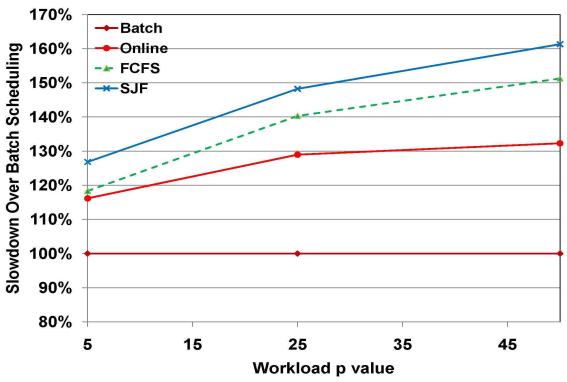

Fig. 14 Scheduling for skewed data with $K = 1$ ($M = 30$)

template is > 2, and the maximum value of the coefficient of dispersion for the other 5 query templates is < 0.5. We further verify this by examining the actual run times of the different query instances. We find that, indeed, Q_9 has a high variability in run time. The actual run times of different instances of each of remaining 5 query templates show much lower variation in their run times as parameters vary, and are well represented by an average value.

Having decided that Q_9 needs to be divided into further query types, we run the K-means algorithm on the plan costs of the 200 instances of Q_9, varying K from 2 to 10. Next, we need to find the best value of K, and that will be the number of query types corresponding to the template under consideration. For this we use *silhouette* metric as discussed in Section 8, which leads to a choice of $K = 4$.

Figure 13 shows our scheduling results for workloads consisting of 60 instances (with different parameter values) of the 6 longest running query templates we used before for a total of 360 queries. The Q_9 queries are split into 4 query types as described above. 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. For the sake of comparison, in Figure 14 we also show the case when we continue to group all instances of Q_9 query template together into one query type, i.e., $K = 1$. The results shows that our approach for handing skew improves the performance of both batch and on-line scheduling algorithms in Figure 13. FCFS behaves the same whether $K = 1$ or $K = 4$. In Figure 13 we can see that the gap between FCFS and the QShuffler algorithms increases for all workloads. For example, for the workload with $p = 25$, the slowdown over batch scheduling for FCFS is 140% in Figure 14, and it is 145% in Figure 13. The improvement of on-line scheduling is much more pronounced. We can also observe that even when no process for automatically identifying query types in the presence of skew is employed ($K = 1$), our basic approach is still robust enough to improve the performance over FCFS and SJF in Figure 14.

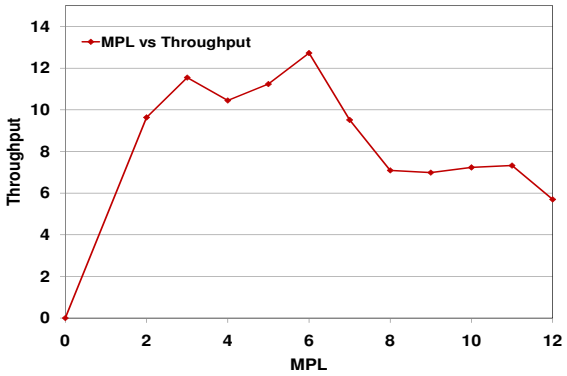

Fig. 15 Throughput vs. MPL for W_R

9.5 Comparison to Optimizer-based Scheduler

In this section, we compare the performance of QShuffler against the optimizer-based scheduler presented in Section 6. For this comparison, we use the 10GB database and the three workloads used in Figure 12 (60 queries with different p values). The first step in using the optimizer-based scheduler is to define the timeron threshold. For this, we construct a workload consisting of the same 60 queries used in Figure 12, with the arrival order randomly shuffled. We call this workload W_R. Figure 15 shows the throughput vs. MPL graph for W_R. The best throughput is obtained at MPL $M = 6$, and corresponds to a timeron threshold $Thr_{time-weighted} = 17,3353,73.87$.

Figure 16 shows the performance of the optimizer-based scheduler using this timeron threshold for W_R and the three workloads in Figure 12. In this figure, we use one service class for all query types (i.e., we do not distinguish between lightweight and heavyweight queries). The MPL varies throughout the workload run when using the optimizer-based scheduler. For example, when we run the optimizer-based scheduler for W_R, the MPL varies from 3 to 12. Figure 16 also shows the performance of the QShuffler on-line scheduling algorithm for MPLs $M = 6$ (the MPL corresponding to the best throughput for W_R) and

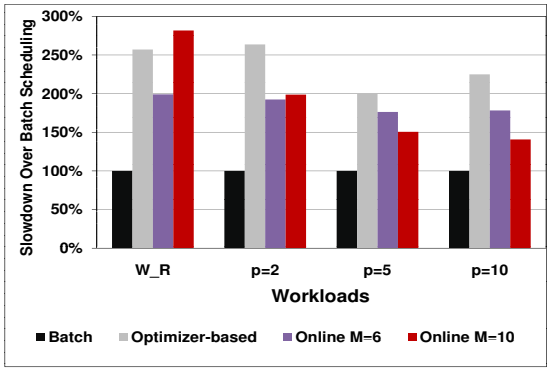

Fig. 16 Optimizer-based scheduling

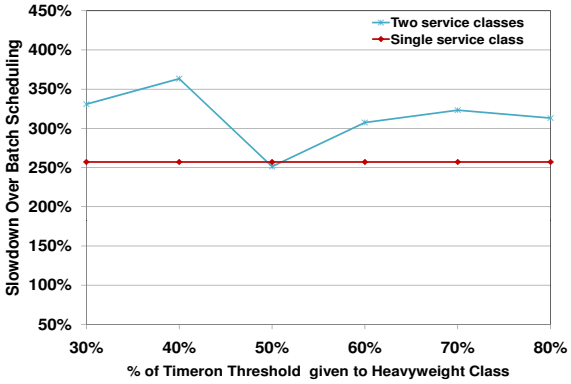

Fig. 17 Scheduling workload W_R using two service classes

$M = 10$ (the MPL used in previous experiments). The figure clearly shows that QShuffler outperforms the optimizer-based scheduler. This demonstrates the importance of considering query interactions in scheduling and modeling actual query completion times rather than relying on query optimizer cost estimates.

Next, we turn our attention to using different service classes for lightweight and heavyweight queries. First, we divide the 6 query types used in this experiment into lightweight and heavyweight according to their query optimizer cost estimates. This classification was easy for these query types since there is a clear separation in cost between the query types with low estimated cost and those with high estimated cost. Denote the highest estimated cost of any query type by Opt_{max}. Of the 6 query types, 3 have estimated costs in the range $[0.82-1]Opt_{max}$, and we place these in the "heavyweight" service class. The remaining 3 query types all have estimated costs in the range $[0.12-0.38]Opt_{max}$, and we place them in the "lightweight" service class. This classification is unambiguous since there is a small range of costs within a class and a large distance between the classes.

After defining the two service classes, the next task is to divide the timeron threshold between these two classes. Recall that the optimizer-based algorithm schedules queries from different service classes using different timeron thresholds. Before embarking on a search for the best algorithm to perform this division, we wanted to experimentally study how well such an algorithm can be expected to perform. We varied the fraction of the timeron threshold given to the heavyweight class from 30% to 80%, with the rest of the timeron budget going to the lightweight class. We ran workload W_R at each of these settings using the optimizer-based scheduler with two service classes. We use W_R as the workload in this experiment since it is the workload used to determine the timeron threshold. Using the same workload to determine the timeron threshold and for scheduling gives the scheduler the best chance of finding a good schedule.

Figure 17 shows the slowdown compared to the QShuffler batch scheduler of the optimizer-based scheduler using two service classes on W_R at each division of the timeron threshold. For comparison, the figure also shows the slowdown of the optimizer-based scheduler with one service class from Figure 16. The figure shows that using one service class always outperforms using two service classes (except for the 50% point in which two service classes is better by a small margin that is well within the range of measurement noise). Thus, no matter what algorithm is used to divide the timeron threshold between the two service classes, using one service class is going to be better.

The experiments in this section provide answers to the two questions posed in Section 6: (1) QShuffler outperforms scheduling based on query optimizer estimates, and (2) this does not change if different service classes are used to distinguish between lightweight and heavyweight queries.

9.6 Cost and Accuracy of Modeling

Since performance modeling is an essential part of our techniques, we focus in our final experiment on: (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, and we use our samples as described in Section 7 to build performance models for NRO and query completion times, A_{ij}.

Mean Relative Error (MRE): We compute the accuracy of a performance model as follows. We pick S test samples at random from the full space of samples, and compute the model-predicted value of performance p_{est} for each test sample. 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. MRE is commonly used for computing model accuracy.

Figure 18 shows, for different types of models for estimating NRO, the MRE on the test samples vs. the number of samples used for model learning. The figures show data for linear models and regression trees (CART [51]). 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).

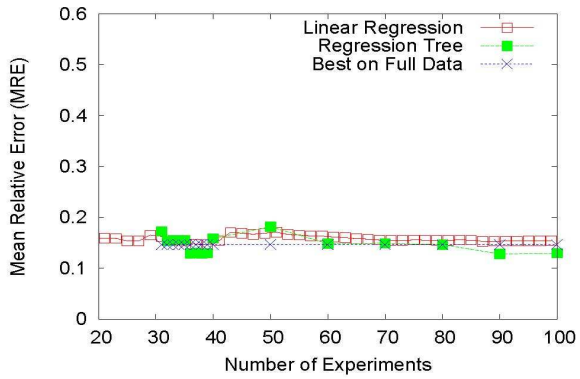

Fig. 18 Accuracy in modeling *NRO*

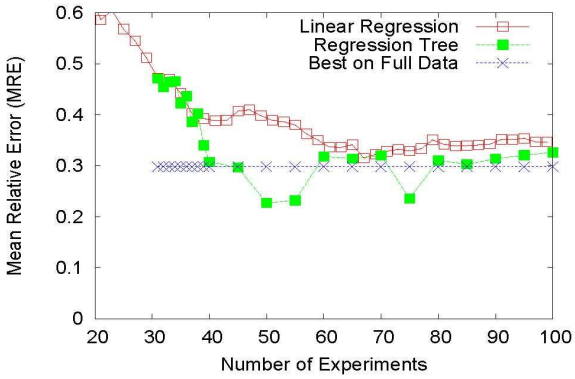

Fig. 19 Accuracy in modeling the completion time of Q_1

From the figure, we can see that: (1) MRE quickly converges to a value of around 10-20% with a small number of training samples (20-40), (2) simple linear models, which we use in QShuffler, are not drastically off the accuracy of the more complex regression tree models. This ease of modeling is one of the desirable features of *NRO*.

Next, we turn our attention to modeling query completion times, which is required for our batch scheduler. Figures 19 and 20 show the accuracy of the models in predicting the completion times of two long-running TPC-H queries. The figures show that modeling query completion times is more difficult than modeling *NRO* since the MRE values are higher. However, MRE still converges quickly so we still need only (40-60) samples for a good model.

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 on just 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 modeling A_{ij} and NRO_i 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.

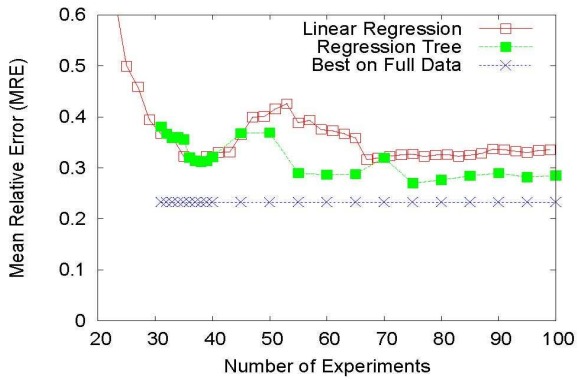

Fig. 20 Accuracy in modeling the completion time of Q_{21}

Statistical models need far fewer samples to separate the good mixes from the bad ones than what they need to predict *NRO* and A_{ij} with high accuracy. 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.

10 Conclusion

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 experiment-driven modeling approach for capturing interactions in query mixes, since analytical modeling of these interactions is not feasible. We present QShuffler, a throughput oriented scheduler for BI report generation workloads. QShuffler's batch scheduling algorithm determines the best schedule in cases where all the workload queries (or large batches of queries) are known in advance. QShuffler also employs an on-line scheduling algorithm for cases when the workload queries are not known in advance and arrive in bursts. The on-line algorithm uses a robust cost metric to guide its choices. 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 provide up to a four-fold improvement in performance over the default FCFS scheduler used by database systems. We also show that QShuffler can effectively handle data skew and that it outperforms a scheduler based on query-optimizer cost estimates.

References

1. Aster data systems. Http://www.asterdata.com/
2. Greenplum. Http://www.greenplum.com/
3. Cognos. Http://www.cognos.com/
4. Business objects. Http://www.businessobjects.com/

5. Ahmad, M., Aboulnaga, A., Babu, S., Munagala, K.: Modeling and exploiting query interactions in database systems. In: CIKM (2008)
6. Ahmad, M., Aboulnaga, A., Babu, S., Munagala, K.: QShuffler: Getting the query mix right. In: ICDE (2008). (poster)
7. Ahmad, M., Aboulnaga, A., Babu, S.: Query interactions in database workloads. In: DBTest Workshop (2009)
8. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. SIGMOD Rec. **29**(2), 249–260 (2000)
9. O'Gorman, K., El Abbadi, A., Agrawal, D.: Multiple query optimization in middleware using query teamwork. Software - Practice and Experience **35**(4) (2005)
10. Albuitiu, M.C., Kemper, A.: Synergy-based workload management. In: PhD Workshop, VLDB (2009)
11. Conway, R.H., Maxwell, W.L., Miller, L.W.: Theory of scheduling. Addison-Wesley (1967)
12. Ibaraki, T., Kameda, T., Katoh, N.: Cautious transaction schedulers for database concurrency control. IEEE Trans. Software Engineering **14**(7), 997–1009 (1988)
13. Katoh, N., Ibaraki, T., Kameda, T.: Cautious transaction schedulers with admission control. TODS **10**(2), 205–229 (1985)
14. Abbott, R., Garcia-Molina, H.: Scheduling real-time transactions. SIGMOD Rec. **17**(1), 71–81 (1988)
15. Abbott, R., Garcia-Molina, H.: Scheduling real-time transactions with disk resident data. In: VLDB (1989)
16. Abbott, R.K., Garcia-Molina, H.: Scheduling real-time transactions: a performance evaluation. TODS **17**(3), 513–560 (1992)
17. Kang, K.D., Son, S.H., Stankovic, J.A.: Service differentiation in real-time main memory databases. Proc. IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (2002)
18. Pang, H., Carey, M.J., Livny, M.: Multiclass query scheduling in real-time database systems. TKDE **7**(4), 533–551 (1995)
19. Carey, M.J., Jauhari, R., Livny, M.: Priority in DBMS resource scheduling. In: VLDB (1989)
20. McWherter, D.T., Schroeder, B., Ailamaki, A., Harchol-Balter, M.: Priority mechanisms for OLTP and transactional web applications. In: ICDE (2004)
21. McWherter, D.T., Schroeder, B., Ailamaki, A., Harchol-Balter, M.: Improving preemptive prioritization via statistical characterization of OLTP locking. In: ICDE (2005)
22. Sacco, G.M., Schkolnick, M.: Buffer management in relational database systems. TODS **11**(4), 473–498 (1986)
23. Schroeder, B., Harchol-Balter, M.: Web servers under overload: How scheduling can help. ACM Trans. Internet Technology **6**(1), 20–52 (2006)
24. Elnikety, S., Nahum, E., Tracey, J., Zwaenepoel, W.: A method for transparent admission control and request scheduling in e-commerce web sites. In: WWW (2004)
25. Kelly, T.: Detecting performance anomalies in global applications. In: Proc. Workshop on Real, Large Distributed Systems (2005)
26. Stewart, C., Kelly, T., Zhang, A.: Exploiting nonstationarity for performance prediction. In: EuroSys (2007)
27. Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: ICAC (2007)
28. Zhang, Q., Cherkasova, L., Mathews, G., Greene, W., Smirni, E.: R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In: Middleware (2007)
29. Heiss, H.U., Wagner, R.: Adaptive load control in transaction processing systems. In: VLDB (1991)
30. Schroeder, B., Harchol-Balter, M., Iyengar, A., Nahum, E., Wierman, A.: How to determine a good multi-programming level for external scheduling. In: ICDE (2006)
31. Mönkeberg, A., Weikum, G.: Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In: VLDB (1992)
32. Mehta, A., Gupta, C., Dayal, U.: BI Batch Manager: A system for managing batch workloads on enterprise data warehouses. In: EDBT (2008)
33. Niu, B., Martin, P., Powley, W., Bird, P., Horman, R.: Adapting mixed workloads to meet SLOs in autonomic DBMSs. In: SMDB Workshop, ICDE (2007)
34. Niu, B., Martin, P., Powley, W.: Towards autonomic workload management in DBMSs. J. Database Manag. **20**(3), 1–17 (2009)
35. Ganapathi, A., Kuno, H., Dayal, U., Wiener, J., Fox, A., Jordan, M., Patterson, D.: Predicting multiple metrics for queries: Better decisions enabled by machine learning. In: ICDE (2009)
36. Babu, S., Borisov, N., Duan, S., Herodotou, H., Thummala, V.: Automated experiment-driven management of (database) systems. In: HotOS Workshop (2009)
37. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with iTuned. In: VLDB (2009)
38. Zheng, W., Bianchini, R., Janakiraman, G.J., Santos, J.R., Turner, Y.: JustRunIt: Experiment-based management of virtualized data centers. In: Proc. USENIX Annual Technical Conference (2009)
39. Belknap, P., Dageville, B., Dias, K., Yagoub, K.: Self-tuning for SQL performance in Oracle database 11g. In: SMDB Workshop, ICDE (2009)
40. Transaction processing perfromance council (TPC). Http://www.tpc.org/
41. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. VLDB Journal **13**(4) (2004)
42. Ryser, H.J.: Combinatorial Mathematics. The Mathematical Association of America (1963)
43. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1998)
44. CPLEX. Http://www.ilog.com/products/cplex/
45. Coady, Y., Cox, R., Detreville, J., Druschel, P., Hellerstein, J., Hume, A., Keeton, K., Nguyen, T., Small, C., Stein, L., Warfield, A.: Falling off the cliff: When systems go nonlinear. In: HotOS Workshop (2005)
46. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., Fadden, S.: DB2 design advisor: integrated automatic physical database design. In: VLDB (2004)
47. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: VLDB (2000)
48. Niu, B., Martin, P., Powley, W., Horman, R., Bird, P.: Workload adaptation in autonomic DBMSs. In: CASCON (2006)
49. Niu, B., Shi, J.: Scalable workload adaptation for mixed workload. In: Infoscale Conf. (2009)
50. Loh, W.Y.: Regression trees with unbiased variable selection and interaction detection. Statistica Sinica **12**, 361–386 (2002)
51. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, second edn. Morgan Kaufmann (2005)
52. MySQL slow query log parser. Http://code.google.com/p/mysql-slow-query-log-parser/
53. Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B.M., Mowry, T.C., Olston, C., Tomasic, A.: Scalable query result caching for web applications. PVLDB **1**(1), 550–561 (2008)
54. Manjhi, A., Gibbons, P.B., Ailamaki, A., Garrod, C., Maggs, B.M., Mowry, T.C., Olston, C., Tomasic, A., Yu, H.: Invalidation clues for database scalability services. In: ICDE (2007)
55. Ioannidis, Y.: The history of histograms (abridged). In: VLDB (2003)
56. Fano, U.: On the theory of ionization yield of radiations in different substances. Physical Review **70**, 44–52 (1946)
57. Cox, D.R., Lewis, P.A.: Statistical Analysis of Series of Events. Chapman & Hall (1966)
58. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley and Sons, Inc (1990)
59. Skewed TPC-D data generator. Ftp://ftp.research.microsoft.com/users/viveknar/ TPCDSkew/