Query Interactions in Database Systems

by

Mumtaz Ahmad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

The typical workload in a database system consists of a *mix* of multiple queries of different types, running concurrently and interacting with each other. The same query may have different performance in different mixes. Hence, optimizing performance requires reasoning about *query mixes* and their interactions, rather than considering individual queries or query types. In this dissertation, we demonstrate how queries affect each other when they are executing concurrently in different mixes. We show the significant impact that query interactions can have on the end-to-end workload performance.

A major hurdle in the understanding of query interactions in database systems is that there is a large spectrum of possible causes of interactions. For example, query interactions can happen because of any of the resource-related, data-related or configuration-related dependencies that exist in the system. This variation in underlying causes makes it very difficult to come up with robust analytical performance models to capture and model query interactions.

We present a new approach for modeling performance in the presence of interactions, based on conducting experiments to measure the effect of query interactions and fitting statistical models to the data collected in these experiments to capture the impact of query interactions. The experiments collect samples of the different possible query mixes, and measure the performance metrics of interest for the different queries in these sample mixes. Statistical models such as simple regression and instance-based learning are used to train models from these sample mixes. This approach requires no prior assumptions about the internal workings of the database system or the nature or cause of the interactions, making it portable across systems.

We demonstrate the potential of capturing, modeling, and exploiting query interactions by developing techniques to help in two database performance related tasks: workload scheduling and estimating the completion time of a workload. These are important workload management problems that database administrators have to deal with routinely.

We consider the problem of scheduling a workload of report-generation queries. Our scheduling algorithms employ statistical performance models to schedule appropriate query mixes for the given workload. Our experimental evaluation demonstrates that our interaction-aware scheduling algorithms outperform scheduling policies that are typically used in database systems.

The problem of estimating the completion time of a workload is an important problem, and the state of the art does not offer any systematic solution. Typically database administrators rely on heuristics or observations of past behavior to solve this problem. We propose a more rigorous solution to this problem, based on a workload simulator that employs performance models to simulate the execution of the different mixes that make up a workload. This mix-based simulator provides a systematic tool that can help database administrators in estimating workload completion time. Our experimental evaluation shows that our approach can estimate the workload completion times with a high degree of accuracy.

Overall, this dissertation demonstrates that reasoning about query interactions holds significant potential for realizing performance improvements in database systems. The techniques developed in this work can be viewed as initial steps in this interesting area of research, with lots of potential for future work.

Acknowledgements

I would like to express my gratitude to Dr. Frank Tompa for providing me the opportunity to pursue my research interest in the area of database systems. I am thankful to my academic supervisor, Dr. Ashraf Aboulnaga, for his guidance and financial support during my PhD studies. His encouragement and help have been instrumental in completing this work.

I am grateful to Dr. Shivnath Babu for his collaboration in this work. I would like to thank Dr. Kamesh Munagala for his help in developing the linear programming formulation of the query scheduling problem. I would like to acknowledge Songyun Duan for his help with the work on evaluating sampling techniques.

I would like to thank the members of my thesis committee: Dr. Ugur Cetintemel, Dr. Wojciech Golab, Dr. Tamer Özu, Dr. Ken Salem, and Dr. Frank Tompa. I would like to express my appreciation to the Cheriton School of Computer Science staff and in particular Margaret Towell for all the help during these years.

Finally I would like to thank my family for their patience and prayers.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Database management systems are widely used in software applications in diverse areas. Every day, users come across business, industrial, and financial applications that rely on underlying database management systems. It is almost impossible to think of any application area that does not rely on some kind of system for storage and retrieval of data and records. Thus, software applications for our banking, financial, health care and retail sectors are all dependent on these database management systems.

The performance of software applications that rely on database systems depends on the performance of these database systems. Therefore, studying and developing techniques for performance improvement of database systems is an important area of interest for computer science research. Achieving good performance in a database system requires careful administration and tuning, which requires significant effort on the part of a database administrator. Helping database administrators is also an important research area that has received much attention during the past 20 years, resulting in the development of techniques to facilitate the automated administration of database systems. Database administrators have increasingly sophisticated tools that drastically reduce the man-hours needed to administer the systems in an efficient manner.

In this dissertation, we present work that contributes to this ongoing effort to develop improved techniques for automated administration of database systems. We focus on the study of workload management issues in database systems. In particular, we focus on the observation that during the typical execution of a workload, a database system, at any point of time, is executing a *mix* of queries of different types. The queries in the mix run concurrently with each other and *interact* 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 interactions within, rather than considering queries or query types in isolation.

Consider the simple case of query scheduling. Scheduling policies like shortest job first (SJF) or shortest remaining time first (SRTF) have been proposed in the research literature. The next query is scheduled on the basis of its running time. However, this approach completely ignores the fact that the running time of the query, which is employed as the basis for scheduling, is the running time of the query when the query is running alone in the system. Depending on the nature of the query, its running time will be affected by the currently executing mix of queries in the system. Thus, for example, if the current mix of queries is heavily I/O bound, a short I/O-bound query may fare worse than a long CPU-bound query if it is scheduled to run at that time on the basis of its running time when it is run alone in the system. Reasoning about query mixes and having a better understanding of their effect on the system would enable us to better choose a query for scheduling from the pool of available queries. Furthermore, understanding these interactions can help in providing better solutions for other performance problems in addition to query scheduling.

A query Q_1 that runs concurrently with another query Q_2 can impact Q_2's performance in different ways, either negatively or positively. For example, the resource demands of Q_1 and Q_2 can interfere with each other, with the interference happening at one or more of different physical resources like CPU, L1 or L2 cache, memory, and I/O. Moreover, the queries can interfere at internal resources inside the database system such as latches, locks, and buffer pools. In such cases, the concurrent presence of more instances of Q_1 will degrade the performance of Q_2. On the other hand, queries running concurrently in a mix may positively affect each other. For example, Q_1 may bring data or index blocks into the buffer pool 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 Q_2 were to run without Q_1 running with it.

We illustrate the impact of query interactions in query mixes using 60 instances of TPC-H [103] queries running on a 10GB TPC-H database in the IBM DB2 database management system [38]. (The details of our experimental setup are given in Chapter 3.) Figure 1.1 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 among the *query mixes* scheduled by DB2 for each workload. All other aspects of the system, including hardware resources, configuration settings, and physical design, are kept the same across the executions of the three workloads.

The difference in scheduled mixes among the three workloads causes a 2.1 hour (63%) difference in completion time between the "worst workload" W_1 and the "best workload"

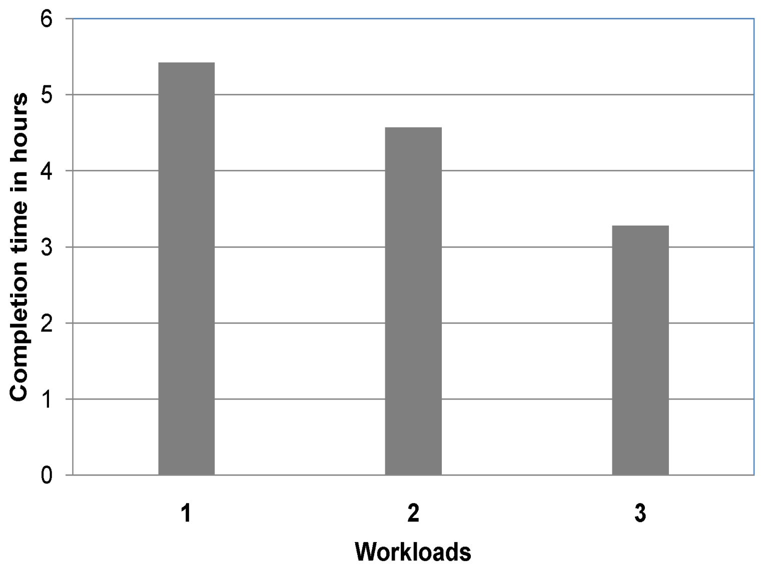


Figure 1.1: Completion times of three different workloads that consist of the same set of 60 TPC-H queries

W_3. It is important to note that each workload runs the same set of queries under identical system configurations. In workload W_1, which runs in 5.4 hours, queries that compete for resources get executed concurrently, resulting in negative interactions. Workload W_3 runs the same set of queries in just 3.3 hours. As mentioned earlier, the workload queries and system settings are exactly same and this difference in runtimes is solely a result of the different query interactions that come into play in different mixes that get scheduled in the different workloads. In workloads W_2 and W_3, the interactions are less negative and occasionally positive, with queries that help each other executing concurrently.

Figures 1.2 and 1.3 visually illustrate how query interactions impact workload execution. Consider a database system with a *multi-programming level (MPL)* of 2. That is, at most 2 queries will be scheduled to run concurrently in the system at any point of time. Suppose the system has to run four queries Q_1, Q_2, Q_3, and Q_4 which are currently queued up waiting to be scheduled. Figures 1.2 and 1.3 show two different schedules A and B. Each bar in these figures shows the time interval where a query runs in the database system. Queries Q_1 and Q_2 are run first in both schedules, and Q_2 completes before Q_1.

When Q_2 finishes, Q_3 gets scheduled in Schedule A, whereas Q_4 gets scheduled in Schedule B. Suppose Q_1 and Q_3 have strong negative interaction, whereas Q_1 and Q_4

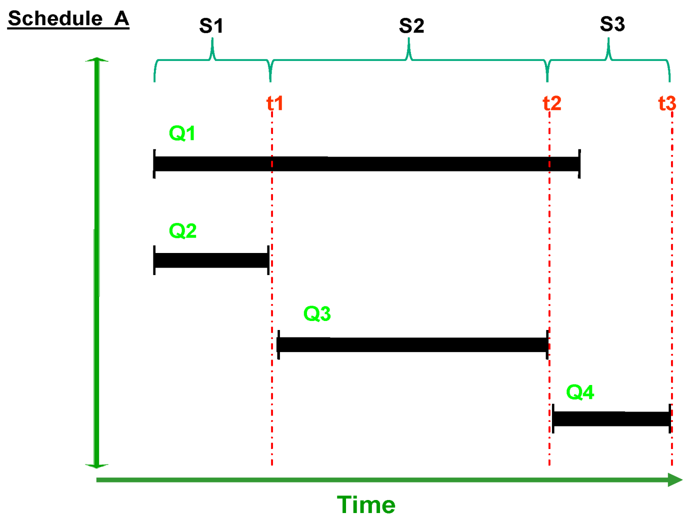


Figure 1.2: Schedule A: Negative interaction between Q_1 and Q_3

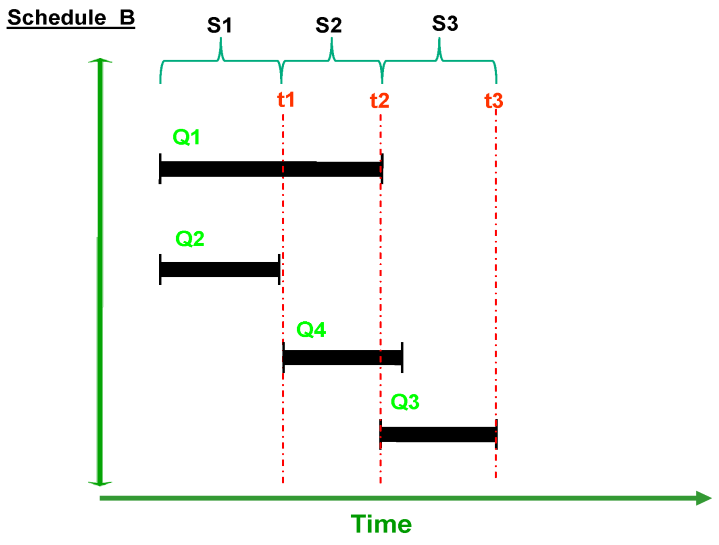


Figure 1.3: Schedule B: Positive interaction between Q_1 and Q_4

have positive interaction. Since Q_1 and Q_3 run concurrently in Schedule A but not in Schedule B, Q_1 will take much longer to complete in A than in B, which is what we see in

Figure 1.2. The difference in overall completion time between Schedules A and B is caused solely by the different interactions in the query mixes that execute in the system.

If the 60 TPC-H queries from Figure 1.1 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. We need to develop techniques that can estimate the impact of different queries on each other when they run concurrently with each other. At the same time, we need to develop algorithms that can make use of this estimation of interaction and solve the performance problem at hand (e.g. query scheduling problem in this case).

A major hurdle in making database systems interaction-aware is in finding effective ways to capture and model query interactions. As we mentioned 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 and operating system configuration may not happen when the same system runs on a different configuration.

In a database system, it is the job of the query optimizer to come up with the most efficient query execution plan for a given query. Query optimizers rely on analytical cost models to come up with best query plans. Would it be possible to reason about concurrently executing query mixes using the analytical cost models used by database query optimizers to estimate the cost of query execution plans? We believe that 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 these conventional cost models to capture interactions in query mixes, we would 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 dissertation, we explore approaches to capture and model the query interactions in concurrently executing query mixes. As discussed above, the infeasibility of analytical cost models poses a big challenge. We overcome this challenge by employing *experiment-driven performance modeling*. The experiment-driven performance modeling approach consists of two steps. In the first step, experiments are run to collect training data. These experiments consist of running a set of judiciously chosen query mixes, and measuring the performance of different queries in these mixes. In the second step, after the training data is collected, statistical modeling approaches are used to fit a model to this data.

In addition to presenting experiment-driven performance modeling approaches to reason about query interactions in concurrently executing query mixes, our work proposes interaction-aware end-to-end solutions to two problems in the area of database workload management: scheduling batches of report-generation queries in database systems and predicting the completion time of a batch workload of report-generation queries. Report-generation workloads are a common type of workload in 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, and with the emergence of frontline data warehouses [19, 49, 106] our techniques become more relevant. Frontline data warehouses seek to provide service level agreements that are similar to transactional systems. This makes workload management techniques like the ones that we propose even more important, since these techniques would be needed to guarantee that the service level agreements are met. Furthermore, frontline data warehouses aim to develop techniques that achieve low latency even for enterprise sized data warehouses. Having this low latency means that the cost of experiment-driven modeling does not become prohibitive with the increase in the size of the data warehouse. Thus, even for large databases, the training data can be collected at relatively low cost.

The end-to-end solutions that we present demonstrate that query interaction can be effectively modeled and exploited. The successful application of the techniques developed in our work to two important problems validates the effectiveness of our approach and algorithms, and at the same time it demonstrates the significant potential for performance improvement in database systems that can be realized by reasoning about query interactions in concurrently executing query mixes. Next we present the summary of our contributions and a roadmap for the remainder of this dissertation.

1.1 Summary of Contributions and Roadmap

Literature Survey: In Chapter 2 we present a survey of the literature from related areas that touch upon the different aspects of our work.

Query interactions: We conduct a detailed experimental study using TPC-H queries on the DB2 database system to observe the impact of query interactions in concurrently executing query mixes. Chapter 3 presents the results of this experimental study illustrating the significant impact that query interactions can have, thereby motivating the need to reason about query mixes. The query mixes presented are composed of query instances belonging to TPC-H. TPC-H defines 22 query templates where each template can be instantiated with different parameter values to generate hundreds of distinct query instances. Typically such templates would define the query types that make up the mixes. Chapter 3 also proposes techniques that can be employed by the database administrator to automatically identify the query types in an application.

Experiment-driven modeling: After demonstrating that query interactions in concurrently executing query mixes can have a great impact on performance, the challenge is to come up with ways to better reason about and understand these interactions. In particular, we need models that can capture these interactions and predict the performance of queries in different mixes. As discussed before, analytical models may not be the best suited for this purpose. In our work, 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 query types are affected by running them in a mix instead of in isolation. Third, having collected a representative set of samples, we employ statistical learning techniques to fit a model to the observed samples. Thus, our approach requires no prior assumptions about the internal workings of the database system or the nature or cause of query interactions, making it portable across systems. Chapter 4 presents our approach for planning experiments and statistical modeling to capture the impact of query interactions.

Workload scheduling: A core contribution of this dissertation is represented in end-to-end solutions to two real-world problems that demonstrate how our approach enables interactions to be modeled and exploited in database systems. The first problem we consider is that of scheduling report-generation queries in database systems. As discussed in our motivating example in Figure 1.1, scheduling the correct query mixes can have a huge impact on the performance of the workload. Typical scheduling policies like first come first serve (FCFS) and shortest job first (SJF) are not interaction-aware and may perform sub-optimally when there are significant interactions among queries. Chapter 5 describes our novel interaction-aware scheduling algorithms that schedule the appropriate query mixes for a given query workload, with the goal of minimizing the completion time of the workload.

Predicting completion time of workloads: The second problem that we consider is answering a simple but very important question : "How long will this workload take to complete?". This problem is important for database administrators in many workload management contexts. The state of the art does not provide a database administrator with any tools that predict the completion time of a workload, so database administrators typically rely on heuristics and past behavior. Here, again, we demonstrate that reasoning about query mixes enables us to come up with a systematic approach that can help database administrators in estimating the workload completion time. We develop a novel mix-based workload simulator that simulates the execution of the different query mixes to estimate the total completion time of the workload. Chapter 6 describes our approach for predicting the completion time of a given query workload.

Conclusion and future work: We conclude in Chapter 7 and present directions for future work. For example, the emergence of cloud computing and server consolidation gives rise to many interesting research challenges in the context of workload management. We believe that the ideas from our work can be quite useful in these areas.

Chapter 2

Literature Survey

There is very little work that deals directly, in a general way, with the performance of concurrently executing query mixes and the interactions within these mixes. In this chapter, we present a survey of the literature from several related areas that touch upon the various aspects of our research.

2.1 Transaction Mix Models

Some papers have employed the concept of *transaction mixes* [60, 97, 111, 112]. The transaction mix models proposed in these works have been used for performance prediction, capacity planning, and detecting anomalies in performance. These works define a transaction mix as transactions of different types running during a time interval or monitoring window and they do not consider which transactions ran concurrently with which other transactions. Further, these works reason about transactions where a transaction may consist of more than one query. The workloads considered are typically multi-tier transactional applications. In general, an assumption is made that the number of transaction types is a fixed small number that is known a priori.

In [60], the authors propose the use of transaction mix models based on linear regression for performance anomaly detection in multi-tier enterprise applications. It is assumed that the system is well provisioned and queuing delays at resources are negligible. During a fixed time interval, the total number of transactions of each type is monitored alongside their response times. The paper claims that this model of aggregate response times can retrospectively explain the performance of various enterprise applications that the authors consider. An anomaly is detected in the system using the simple idea that if the observed time of various transaction types during an observation interval is in agreement with the time suggested by the model, then it is considered normal behavior. Otherwise, if there is a disagreement, the behavior is deemed as anomalous. This work is significantly extended in [97].

In addition to retrospectively explaining the performance of an application and highlighting anomalous behavior, the work in [97] also aims to predict the application level response times given a future workload. The paper also focusses on predicting the impact of application consolidation on transaction response times. The basic transaction mix model is the same as that in [60]. In [60], one of the assumptions is that queuing delays are negligible. In [97], this assumption is relaxed, and queuing delays at resources are also included in the extended model. In case of server consolidation, system performance is predicted by concatenating the transaction mix models of each application running in isolation. In order to show the improvement achieved by reasoning about transactions mixes, the results are also compared to a scalar variant of the basic model that only uses the total number of transactions in the observed interval instead of reasoning about the mixes. The work assumes, like its predecessor, that (a) the total number of different types of transactions are fixed and limited, (b) the interactions among transactions are negligible and can be ignored, and (c) that there is not much contention for the resources except the congestion that is being modeled by queuing delays. Since even with a small number of transaction types, the number of possible mixes and hence the model space can be very large, the authors claim that non-stationarity of mixes (i.e. the mixes running during an observation interval keep changing from interval to interval) helps them to sample the number of mixes required for model calibration, without employing invasive system or application instrumentation or controlled sampling.

In [112], the authors discuss resource provisioning and capacity planning for multi-tier applications and use the TPC-W benchmark [104] for experimentation. In multi-tier applications, different tiers may run on different machines and the workloads are generally characterized at the level of sessions, with transactions having strong inter-transaction dependencies. The authors simulate transaction based behavior where the next transaction is selected based on a certain probability. A regression-based transaction mix model is proposed to approximate the CPU demand of client transactions on given hardware for each tier. Once the approximate demand for CPU is known, this approximation is used to parameterize an analytical model of a simple network of queues, where each queue represents a tier. This analytical model dynamically evaluates the required system resources in changing workload conditions. The work shows that for diverse workloads with a changing transaction mix over time, the CPU demand can be modeled effectively using the transaction mix model. The authors use the TPC-W benchmark for their experimentation and study the accuracy and efficiency of their proposed models for the three different types of traffic mix that are defined by TPC-W (shopping, browsing, and ordering). In [111], the authors extend their work done in [112]. In particular, the work in [111] considers workloads where the number of transactions are much higher than what we see in typical synthetic workloads. The contribution of the paper is to illustrate how the approach can be extended to production workloads with a large set of transaction types. The idea is to extract a set of popular core transactions and then apply regression modeling. The concept of core transactions is based on the observation that during a typical run, the distribution of transactions is such that a core subset of transactions account for the majority of the time, e.g., it is observed that the top 20 transaction types are responsible for 93.6% of the client accesses in the trace collected from two different application servers that provide customized client access to HP OpenView [80]. Thus it is only these transactions that are considered for transaction mix models.

As discussed earlier, these works 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 . In this dissertation, 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.

2.2 Optimizing the Performance of Concurrent Query Mixes

There has been some previous work on optimizing the performance of concurrently executing queries. Work on this topic falls into two categories: work on multi-query optimization (e.g., [85]) and work on sharing scans in the buffer pool (e.g., [79]). 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 the diverse kinds of positive as well as negative query interactions regardless of the known or unknown underlying cause. Our approach does not require a priori information about the database system internals or the hardware and operating system environment.

Since the time of publishing the work on query mixes and query interactions that makes up this thesis [10, 11, 12, 13, 15, 16], there have been several papers that build upon our work and adopt our model of query mix.

The work in [102] employs sampling and modeling ideas from our work and focuses on the problem of admission control in multi-tier web applications. In particular, the authors employ reasoning about query mixes to decide which requests a system should process and which it should reject in order to minimize the client timeouts. When a query arrives, an expected execution time is computed using the model that takes into account the currently executing query mix. If the expected execution time does not exceed the timeout threshold, the query is accepted, otherwise it is rejected.

In [40], the authors focus on predicting the response latency of concurrently executing query types in BI workloads. They propose the use of average latency of a logical I/O operation as the metric of choice for query latency indication. This metric is called *buffer access latency (BAL)*. A linear regression model, *BAL to latency (B2L)* relates the end-to-end query latency with the BAL metric. In order to employ this B2L model to make predictions, a model for estimating the average BAL value under contention, *BALs to concurrent BAL (B2cB)*, is proposed that incorporates the observations based on pairwise interactions between query templates. Once B2cB for a query type in a query mix is estimated, it is straightforward to compute the end-to-end latency by using B2L. In order to learn both these models, training data is collected using Latin Hypercube Sampling (LHS) along with directly sampling of query mixes consisting of pairwise interactions. In order to predict the query latency in changing mixes, the authors propose approaches that are very much inspired by our workload simulator presented in Chapter 6

The problem of predicting the response time of an individual query in a given query mix is also addressed in [93]. However, that work does not consider the execution in changing mixes. The focus in that work is to learn models on-line such that the models are constantly updated in response to new training data. The new training data needs to be taken into account as there are changes in the number of query templates or the underlying resource configurations. The work shows that, by using Bayesian learning approaches, the models can be efficiently adapted to changes in query types or resource configurations in an on-line manner.

2.3 Workload Management in Database Systems

In Chapter 5 we consider the problem of scheduling report-generation workloads in database systems. There is a wealth of literature on job scheduling from an operations research or industrial engineering perspective (e.g., see [31]). Scheduling in database systems has been studied in the context of concurrency control, where the focus is on minimizing lock contention [55, 58], in real-time database systems (RTDBMS) [2, 3, 4, 5, 54, 57, 82], and for providing prioritization and QoS in transactional workloads in general-purpose database systems [68, 69, 91].

Real-time database systems are different from general-purpose database systems in the sense that they have intrinsic timing constraints. Each transaction is associated with time-dependent deadlines and the overall goal, in general, is to minimize missed deadlines. Works on scheduling in real-time database systems study how scheduling around critical resources (CPU, locks, and I/O) helps meet the desired goals. In [2, 3, 4, 5] the authors conclude that for real-time database systems, CPU is the most important resource because the transactions only acquire other resources after they have the CPU. For the scheduling policy, the authors study priority inheritance and preemptive prioritization. In [57] and [82], the focus is on differentiating classes of requests in an RTDBMS. In [57] the focus is on real time main memory databases.

Translating transaction-level priorities into priorities on resource usage (e.g., CPU and locks) has been studied in the context of general-purpose database systems in [27, 68, 69]. The optimal buffer space requirement for each query is estimated in [87] and used to ensure that memory consumption of scheduled queries does not exceed the available memory. In contrast to all these works on scheduling, reasoning about query mixes is central to our approach. Ignoring query interactions in a mix can result in suboptimal scheduling decisions.

Scheduling techniques are also important in web servers. As system load increases, the importance of effective scheduling increases because load-shedding may have to be employed to keep systems responsive under high load. The work in [90] proposes using shortest remaining time first (SRTF) scheduling to avoid dropping requests in web servers when the system is under high load. While our scheduling algorithms share the goal of avoiding overload, our approach is to take query interactions into account to make better scheduling decisions. For example, we show in Chapter 5 that shortest job first, the non-preemptive version of SRTF scheduling, is often suboptimal.

Workload management is an area of growing importance in database systems. In addition to scheduling, work in this area includes techniques for admission control and setting the multi-programming level (MPL) of the database system. The multi-programming level of the system represents the number of queries that execute concurrently in the system at any time. In this dissertation we do not focus on the problem of admission control directly, as our scheduling algorithms do not reject or queue the queries. However, admission control can be considered a form of scheduling, so we present a review of related work.

One of the first works to discuss the problem of adaptive tuning of MPL in database systems is [51]. The authors outline the factors contributing to overload conditions and suggest the use of feedback control mechanisms to adjust the concurrency level in the system. The proposed approach continuously monitors transaction throughput over fixed-size time intervals and a feedback loop is employed to adjust the MPL. The MPL is adjusted according to increases or decrease in the transaction throughput during the last time interval.

Most of the focus in the area of workload management has been on transactional workloads. In transactional workloads an important concern is data contention thrashing, that is, performance degradation because of excessive lock conflicts. Therefore, many proposed techniques perform load control based on some locking performance metric [28, 43, 44, 58, 72, 73, 99, 100]. Thus, for instance, in [72] a metric called the *conflict rate* is used. This metric records the ratio of the number of locks held by all transactions to the number of locks held by active (non-blocked) transactions. If this ratio increases beyond a certain threshold, the admission of new transactions is suspended until the ratio drop below the threshold. The approach is further extended in [73], where the authors propose that advance knowledge of certain properties of a transaction can be used to estimate the effect on the conflict rate if the transaction were admitted, and admission decisions could be based on the estimated rather than the currently observed value of the conflict rate. Adaptive load control has also been studied for multi-tier applications and internet servers [41, 56, 107].

In [92], the authors experimentally investigate how low we can set the MPL to facilitate effective scheduling. They experimentally demonstrate that a low MPL, compared to the total number of clients, is sufficient to achieve near-optimal throughput and mean response time for a range of different workloads and transaction types. A feedback control loop is used to tune the MPL. The feedback control loop is augmented with queueing theoretic models that seek to capture basic properties of the relationship between system throughput and response time and the MPL. These models predict a lower bound on the MPL, and this lower bound provides the control loop with a good starting value. The control loop then tries to optimize MPL starting with this value. In this dissertation, we focus on external scheduling for any given MPL, so our work is complementary to work on setting the MPL.

DB2 Query Patroller (QP) [83] is a query management system to control the admission and flow of queries into a database system. It is able to classify queries such that small high-priority queries may run promptly instead of waiting behind large low-priority queries. An administrator can set resource usage policies at the system and user level and can make sure that these policies are not being violated by controlling the concurrency level. DB2 Workload Manager (WLM) [109] represents the evolution of workload management in DB2. In addition to features provided by QP, it is able to manipulate resource allocations among different service classes of queries being admitted to improve their response times.

As mentioned above, the report-generation workloads that we consider in this dissertation consist of complex analytic queries that are typical in BI workloads. These workloads are very different from transactional workloads. For transactional workloads, there are many different approaches for setting the MPL dynamically. We discussed some of these works above. However, these approaches do not work well for BI workloads. In general, setting the MPL dynamically for BI workloads is non-trivial, which is why these workloads are typically run using a statically pre-tuned MPL. Most of the works on MPL tuning for transactional workloads employ some mechanism of feedback. In case of BI workloads this is not particularly effective for many reasons. There can be orders of magnitude difference in throughput between transactional workloads and BI workloads. In transactional workloads the throughput can be thousands of transactions per second, while in BI workloads it may be tens or hundreds of queries per hour. Furthermore, there can be a lot more variance in the completion time of different query types in BI workloads. For example, in our experiments with TPC-H queries on a 1GB database, we have queries with completion times that range from 1 second to more than 10 seconds. Similarly, in our experiments with a 10GB TPC-H database, we have queries with completion times that range from 100 seconds to more than 500 seconds. Long running queries make it very difficult to select a proper control interval to implement feedback mechanisms, and shifts in the workload between long- and short-running queries can make convergence problematic.

Niu et al. [75, 76, 77, 78] present a general framework for workload adaptation that can handle scheduling for time-varying workloads. These works describe a query scheduler that handles different service classes for different query types. The query scheduler uses a utility function that aims to meet different service level objectives for the different service classes. The query scheduler uses the query optimizer cost estimates (known as *timerons* in 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. An important assumption in these papers is that there is a global system saturation point for query optimizer cost for all query types. However, in Chapter 5 we show that the optimizer cost estimates can be misleading indicators of the actual performance of different queries.

In [70], the authors propose a batch workload manager for BI systems that does admission control based on the estimated memory requirement of the BI queries. The main objective of the work is to avoid thrashing and overload in the system. The authors argue that typically it is memory contention that causes thrashing. They measure the peak memory requirement of each query type in the workload and then use it to prioritize the queries. The queries with highest memory requirements get the highest priority and vice versa. These priorities are used to order the queries, and queries are admitted into the system such that the queries with highest priority are at the head of the admission queue. The queries are thus admitted according the order of their priorities such that the combined memory requirement of the concurrent queries does not exceed the total available memory of the system. As a query finishes and more memory becomes available, the next query can be admitted if adding this query to the system does not exceed the total available memory of the system. This work completely ignores the interaction among queries. In Chapter 5 we show that resource consumption metrics, such as the memory requirement of queries, can be quite misleading in determining the performance of queries in mixes.

2.4 Machine Learning and Experiment-driven Performance Modeling

Traditionally, database systems have relied on analytical models for reasoning about performance. In particular, query optimizers are well known for using analytical cost models to evaluate different query execution plans. However, as the complexity of systems keeps increasing, machine learning and experiment-driven performance modeling are gaining wide acceptance as methodologies to build robust performance models. In our work we use experiment-driven performance modeling to capture and model query interactions in query mixes, constructing models that predict the performance of queries in different mixes. In this regard, there are many related works that adopt a similar approach in their study of DBMS performance.

A very relevant set of work in this area is [37, 45, 46, 64]. In [46], the authors use machine learning techniques to predict performance metrics for database queries. They use Kernel Canonical Correlation Analysis (KCCA) [23] for making these predictions. In the training phase, KCCA develops a model of multivariate correlations between a dataset of query plan features and a dataset of runtime performance features. This model can then be used to predict the runtime performance characteristics of a given query by observing its query plan features. The authors demonstrate that this approach of machine learning enables performance predictions for individual query types with less than 20% error in over 85% of the test cases. Their work, however, exclusively focuses on single query types and does not consider interactions among queries in query mixes. In [45] the authors extend the approach presented in [46] to predict the performance of MapReduce jobs in Hadoop [18]. In [37, 64], the authors discuss this approach of predicting runtime performance features in the context of a general framework of workload management where the effectiveness of different workload management policies is evaluated using a simulator for the database engine.

In [50], a decision tree is used to predict ranges of query execution times. The decision tree is constructed by taking into account query plan features and load features of the system. This decision tree is then used to predict the execution time range of a new query. In contrast to our work, no explicit features of query mixes and interactions are taken into account and the multi-programming level (MPL) of the system is the only feature used to represent the overall load on the system. IBM's LEO learning optimizer [98] uses machine learning to improve the quality of cost estimates by the query optimizer. The execution of the queries is monitored to get actual cost statistics that are compared to the optimizer's estimates. The adjustments are learnt and can be used during future optimizations of queries. In [110], the authors use machine learning to produce a self-tuning cost model for XML queries. Similarly, machine learning is employed for database provisioning in [48, 96].

Experiment-driven performance modeling has also been used for tuning database and system resource configuration parameters [22, 39, 95]. In [39], the authors propose a tool that helps in finding better configuration parameter settings for a database system by planning experiments corresponding to different parameter value settings. In [95], the authors propose techniques to better tune the CPU and memory allocations for database workloads running inside virtual machines. In [7], experiment-driven performance modeling is used in cloud computing to help better scheduling of MapReduce jobs on a cluster of machines.

In these works as well as ours, there is a need to run sampling experiments to collect training data that is later on used to train statistical models. An important concern is that "where and when" these experiments should be run. This concern is not unique to these approaches; it is a problem that a database administrator faces routinely when evaluating or implementing any change to the production system. The problem increases in complexity as the application stack becomes more layered and complex.

Depending on the system setup, an administrator can try to handle this problem in many different ways. The experiments can be done on the system before it goes into production. If the database is already in production use, the maintenance windows can be used for running experiments. Similarly, an administrator can use test systems or even a standby system (put in place to handle failovers) for this purpose. In [39], an important contribution of the authors is that they propose an *experiment executor* that exploits the under-utilized resources to run the experiments. The resource set includes all the production, test, and standby systems, and the user can specify policies for each resource that determine when that resource can be used for experiments.

Commercial vendors are also realizing the importance of this issue and supplementing their products with appropriate tools to help an administrator in running experiments. For example, Oracle 11g provides an infrastructure called Real Application Testing [24]. The Real Application Testing framework allows the user to evaluate the impact of changes to the database system before applying these changes to the production system. It has a component called Database Replay, which enables the user to capture real workloads running on a production system with minimum impact. During the specified capture window, it captures all the database requests, including all the information about transactional concurrency and system load. Having captured the workload, the database administrator can run it on a test system exactly as it ran on the production system.

In our work we simply propose running the sampling experiments on the production system when it is not in use. However, we note that any of the above frameworks can be adapted to run our experiments. A detailed study of this problem of where and when to run the experiments is complementary to the work presented in this dissertation and beyond the scope of our work. We always assume that we have available capacity to run the experiments to collect the required training data for different query mixes, although we are always mindful of the fact that there is a "budget" of query mixes that can be run during sampling experiments.

In the next chapter we present the results of our detailed experimental study that illustrate the significant impact that query interactions can have in query mixes.

Chapter 3

An Experimental Study of Query Interactions

In Chapter 1, we presented an example to illustrate the effect of query interactions on database workloads. We showed that there can be a significant difference in the performance of different workloads due to different query interactions in the execution of these workloads. In order to drill down into the performance of query mixes, we conducted a detailed experimental study to observe the impact of query interactions in concurrently executing query mixes [10]. In this chapter, we present several examples of query interactions observed in this experimental study. A related question that we discuss in this chapter is how query types can be defined.

3.1 Granularity of Modeling Query Interactions

In this dissertation we focus on studying interactions in a database system at the granularity of query types that we identify by using query templates. A range of other options exist. A simpler scheme that models interactions at a coarser granularity than ours is to categorize queries based on their resource consumption. For example, we can categorize queries into CPU or I/O bound so that two or more concurrent CPU (I/O) bound queries would be considered to interact negatively. A more complex scheme that models query interactions at a finer granularity than ours involves capturing interactions at the level of different phases of execution of physical operators in a query execution plan like Hash Join and Sort.

Modeling interactions at a finer granularity has the potential of being more accurate. However, fitting a fine granularity model requires much more training data and much more fine grained measurements than fitting a coarse granularity model. For example, if we want to capture interactions between different physical operators in a query execution plan, we need to conduct experiments in which we observe and measure interactions not only between whole queries but also between individual operators in these queries.

Similarly there can be multiple choices for performance features of interest. The performance metric that we use in this dissertation is the *average completion time of different query types.* An alternative would be, for example, to observe system and database resource consumption features. Completion time is an intuitive and universal metric that is oblivious to the cause of interactions. Our choice of modeling granularity and performance metric works well for the workload management problems that we consider in our work. In our experimental study we also collected data about system resource consumption in different query mixes. We present this resource consumption data as well to show that looking at the resource consumption of queries alone is not sufficient to capture query interactions.

3.2 Experimental Setup

We conducted our experimental study using queries from the TPC-H decision support benchmark with two scale factors, 1 and 10 [103]. The scale factor 1 generates a dataset of 1GB and scale factor 10 generates a dataset of 10GB. The database system we use is DB2 version 8.1, and we ran our experiments on a machine with dual 3.4GHz Intel Xeon CPUs and 4.0GB of RAM running Windows Server 2003. The buffer pool size of the database was set to 400MB for the 1GB database, and to 2.4GB for the 10GB database. The TPC-H benchmark has 22 query types that are instantiated with different parameter values, and we use query mixes consisting of different numbers of instances of different TPC-H query types, with different parameter values in each query instance as required by the TPC-H specification. The standard TPC-H specification provides two programs for generating data and queries, DBGEN and QGEN. We use these programs to generate our datasets and query instances. In order to ensure that we have a good set of indexes we used the DB2 Design Advisor [113] to recommend indexes. Similarly, we ran the DB2 Configuration Advisor [65] to make sure that the configuration parameters are well tuned. The size of the databases with all indexes was 2.4GB and 23GB for scale factors 1 and 10, respectively. We refer to the scale factor 1 database as the *1GB database* and the scale factor 10 database as the *10GB database.*

Query Type	runtime t_j (sec)
$Q1$	10.07
$Q2$	1.95
$Q3$	3.41
$Q4$	2.09
$Q5$	2.59
$Q6$	4.77
$Q7$	5.76
$Q8$	4.15
$Q9$	9.66
$Q10$	2.65
$Q11$	0.98
$Q12$	2.38
$Q13$	6.12
$Q14$	0.42
$Q16$	1.74
$Q17$	0.03
$Q18$	7.12
$Q19$	0.34
$Q20$	4.48
$Q21$	7.30
$Q22$	1.07

Table 3.1: Runtime, t_j, of different TPC-H query types on a 1GB database

Let $Q_1, Q_2, \ldots, Q_{22}$ be the TPC-H query types. Table 3.1 shows the average completion time of the TPC-H query types on a 1GB database when they run alone in the system, which we denote by t_j. The completion time (or runtime) of a query instance is the time elapsed between when the query starts and when it finishes. Table 3.2 shows the completion time of these queries on a 10GB database. We omit Q_{15} since it creates and drops a view, which is not supported by our prototype implementation. Each completion time in the two tables represents the average completion time of 10 instances of the particular query type. There is very little variance in completion times for a specific query type, which is expected since TPC-H uses uniform distributions for data and query parameters.

We now consider multiple queries running concurrently. Let the multi-programming level (MPL) of the database be denoted by M. A set of queries that execute concurrently in the system is referred to as a *query mix*. Formally, query mix m_i can be represented as a

Query Type	runtime t_j (sec)
$Q1$	294.61
$Q2$	97.25
$Q3$	247.95
$Q4$	170.90
$Q5$	136.04
$Q6$	346.63
$Q7$	102.06
$Q8$	387.72
$Q9$	578.61
$Q10$	353.89
$Q11$	14.58
$Q12$	483.17
$Q13$	101.27
$Q14$	5.56
$Q16$	26.59
$Q17$	43.23
$Q18$	554.56
$Q19$	222.31
$Q20$	273.08
$Q21$	570.37
$Q22$	29.17

Table 3.2: Runtime, t_j, of different TPC-H query types on a 10GB database

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 $\sum_{j=1}^{T} N_{ij} = M$. The symbol T denotes the number of query types. We denote the average completion time of queries of type Q_j in mix m_i by A_{ij}. The average completion time is computed by adding the runtimes of all N_{ij} instances of Q_j in mix m_i, and then dividing the sum by N_{ij}. Table 3.3 summarizes our notation used throughout the dissertation.

To sample a query mix, we use the following procedure. All queries in the mix $m_i = \langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$ are started at the same time t_0. The different query types have different runtimes and they do not finish at the same time. So in order to make sure that we are always running the same query mix m_i, as soon as an instance of Q_j finishes, we start another instance of Q_j. This continues until the longest running query instance of the initial set of instances (started at t_0) finishes at time t_1. The completion time of the mix is $t_1 - t_0$. This process is illustrated in Figure 3.1.

Symbol	Description
M	Multi-programming level
T	Number of query types
Q_j	Query type j
q_j	An instance of query type j
t_j	Average completion time of a Q_j query when running alone
$m_i = \langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$	A query mix, m_i, with N_{ij} instances of each query type j
A_{ij}	Average completion time of a Q_j query when running in mix m_i

Table 3.3: Notation used in this dissertation

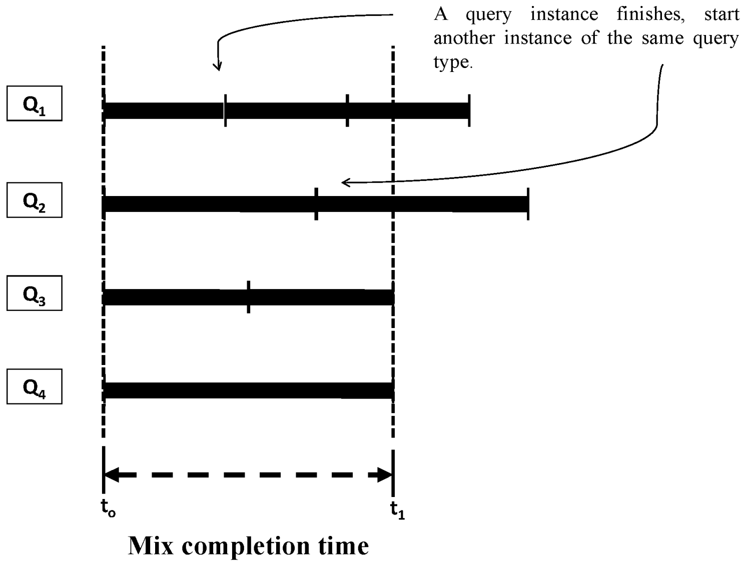

A query instance finishes, start another instance of the same query type.
Q1
Q2
Q3
Q4
t0
t1
Mix completion time

Figure 3.1: Sampling a query mix

	$Q1$		$Q7$		$Q9$		$Q13$		$Q18$		$Q21$	
Mix	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}
m_1	1	1093.14	1	578.36	3	1190.15	0	0.0	0	0.0	0	0.0
m_2	1	1794.97	1	1261.39	2	2638.62	1	432.12	0	0.0	0	0.0
m_3	1	1186.74	1	663.97	0	0.0	3	311.53	0	0.0	0	0.0

Table 3.4: A_{ij} (in seconds) for different query types in query mixes on a 10GB database

In the beginning of an experimental run we start by running some random query mixes. This is our warm up time for sample collection to ensure that we take our observations on a warm buffer pool. In general, the completion time of a mix is minimally affected by its preceding mix. As we mentioned above, the standard TPC-H specifications provides two programs for generating data and queries, DBGEN and QGEN. These standard programs ensure that generated data values follow a uniform distribution. Similarly, when queries are instantiated, the different instances of the same query type uniformly touch different data ranges in the tables that they access. Thus, we find that there is negligible variation in the runtimes of different instances of the same query type and the runtime of a mix is also minimally affected by the mixes that precede it.

3.3 Effect of Query Interactions on Query Completion Time

In this section we present several examples of the impact of interactions in a query mix on the completion time of a given query type in this mix. 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.*

We start with a simple example of query interactions. Table 3.4 shows three mixes consisting of 6 long-running query types on the 10GB database. Each mix in this table, and also in Tables 3.5– 3.9 was run five times, and the measurements shown are averages of these five runs. In all cases, the variance in completion time for all query types was less than 4% of the mean. The high variability in A_{ij} illustrates the effect of query interactions. The behavior of queries changes from mix to mix depending on the interaction among queries. In all these mixes, A_{ij} values are much higher than the corresponding t_j values shown in Table 3.2. Thus, all queries are impacted negatively in these mixes. Consider the average

Mix	$Q1$		$Q7$		$Q9$		$Q13$		$Q18$		$Q21$	
	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_4	8	114.40	2	45.76	1	193.16	4	71.12	4	111.87	1	55.38
m_5	2	109.88	6	88.13	3	191.48	5	61.23	3	114.21	1	159.95
m_6	11	143.90	8	144.60	3	211.20	2	97.80	2	149.80	4	127.50
m_7	2	361.70	8	298.60	1	476.0	18	121.20	0	0.0	1	231.20

Table 3.5: A_{ij} (in seconds) for different query types in query mixes on a 1GB database

runtime of Q_1 and Q_7 in the first two mixes. Both mixes have $M = 5$, and both have one instance each of Q_1 and Q_7, but there is an increase in A_{ij} in m_2 for all query types. In particular, the runtime of Q_7 is more than twice its time in m_1. One may be tempted to think that this is just because of the characteristics of Q_{13} which was introduced in m_2. The next mix m_3 shows that this is not true. In this mix, both Q_1 and Q_7 actually have better performance than in m_2, even when we increase the number of instances of Q_{13} from one in m_2 to three in m_3.

The effect of query interactions in a 1GB database can be seen in Table 3.5. The table shows different mixes of the same query types as in Table 3.4. These six query types have the longest runtime among all TPC-H query types when run alone on a 1GB database. Consider the average runtime of Q_{21} in the two mixes m_4 and m_5. Both these mixes have $M = 20$, yet A_{ij} for Q_{21} in m_5 is almost three times that for m_4. Similarly, consider the average runtime of Q_7 in the two mixes m_6 and m_7. We run the same number of instances of Q_7 in both mixes and $M = 30$ for both mixes, but A_{ij} for Q_7 in m_7 is almost twice the A_{ij} for Q_7 in m_6.

Next, we present interesting cases of *positive interactions* using the same six query types as before on a 10GB database. Mix m_8 in Table 3.6 presents an example of positive interaction for Q_7. The average runtime of Q_7 in this mix, A_{ij}, is 72.7 seconds, while the runtime of Q_7 when it is run alone in the system is 102.06 seconds (Table 3.2). Thus, Q_7 *benefits* from being run in this mix: an instance of Q_7, on average, 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_8 raises the following question: would Q_7's performance be even better if it were run in a mix that predominantly has instances of Q_7 (e.g., because of possibly increased buffer cache hits)? To answer this, consider the average runtime of Q_7 in mix m_9 which has four concurrent instances of Q_7. Notice that Q_7's runtime in m_9 is much worse than its runtime in m_8, and also worse than Q_7's runtime when it runs alone in the system (i.e., Q_7's interactions are negative in m_9).

	*Q*1		*Q*7		*Q*9		*Q*13		*Q*18		*Q*21	
Mix	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}
m_8	1	1897.40	2	72.70	5	2919.30	0	0.0	2	1904.10	0	0.0
m_9	0	0.0	4	264.47	0	0.0	0	0.0	1	3413.66	0	0.0
m_{10}	4	538.0	0	0.0	0	0.0	0	0.0	1	539.30	0	0.0
m_{10a}	4	537.98	0	0.0	0	0.0	0	0.0	1	541.39	0	0.0
m_{10b}	4	542.94	0	0.0	0	0.0	0	0.0	1	538.26	0	0.0

Table 3.6: A_{ij} (in seconds) for different query types in query mixes on a 10GB database

mix	*Q*9 (N_{ij})	*Q*13 (N_{ij})	*Q*21 (N_{ij})	*Q*21 (A_{ij})
m_{11}	0	4	1	4188.20
m_{12}	1	3	1	5463.80
m_{13}	2	2	1	3476.10
m_{14}	3	1	1	3581.70
m_{15}	4	0	1	2782.40

Table 3.7: A_{ij} (in seconds) for Q_{21} on a 10GB database

Mix m_{10} presents another example of positive interaction, this time for Q_{18}. The average runtime of Q_{18} in this mix is 539.3 seconds, compared to a runtime of 554.56 seconds when it is running alone. Thus, Q_{18} benefits from being run with 4 instances of Q_1. Mixes m_{10a} and m_{10b} show repeated runs of mix m_{10} with different instances of the same query types. The results for all variants of m_{10} are similar, indicating that this positive impact of query interactions that we see in these mixes is consistent and robust across different instances.

Next, we demonstrate that query interactions can be fairly complex, with small changes in the query mix sometimes having a huge impact on performance that may be difficult to predict. Table 3.7 shows more examples of interactions from the 10GB TPC-H database where we focus on three-way interactions of TPC-H query Q_{21} in the presence of queries of type Q_9 and Q_{13}. In all examples we have $M = 5$ and one instance of Q_{21}. The first row of the table shows the instance of Q_{21} running with 4 instances of Q_{13}. In the subsequent rows, we replace instances of Q_{13} with instances of Q_9. The completion time of Q_{21} first increases with the introduction of an instance of Q_9, then it decreases and increases alternatively as we keep increasing the number of instance of Q_9 and decreasing the number of instances of Q_{13}. The worst runtime of Q_{21} is almost twice the best runtime. It can be observed from Table 3.7 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 3.2 because Q_9 is by far the more heavyweight query when each query is considered individually.

mix	*Q*13 (N_{ij})	*Q*9 (N_{ij})	*Q*13 (A_{ij})	*Q*9 (A_{ij})	CPU Utilization(%)	Millisec/Disk Transfer
m_{16}	0	5	0.0	919 .0	24.11	25.30
m_{17}	1	4	356.0	1547.16	27.43	25.0
m_{18}	2	3	422.59	2079.72	19.68	22.08
m_{19}	3	2	388.74	2508.33	4.97	9.8
m_{20}	4	1	289.1	3762.55	53.15	26.0
m_{21}	5	0	224.42	0.0	86.06	16.55

Table 3.8: Resource consumption for different mixes of Q_{13} and Q_9 on a 10GB database

mix	*Q*13 (N_{ij})	*Q*21 (N_{ij})	*Q*13 (A_{ij})	*Q*21 (A_{ij})	CPU Utilization(%)	Millisec/Disk Transfer
m_{22}	0	5	0.0	1300.735	5.79	7.2
m_{23}	1	4	372.05	2196.81	9.30	11.4
m_{24}	2	3	436.62	2283.41	14.38	12.8
m_{25}	3	2	322.71	2576.06	30.68	17.7
m_{26}	4	1	206.88	4188.2	59.36	21.5
m_{27}	5	0	224.42	0.0	86.06	16.55

Table 3.9: Resource consumption for different mixes of Q_{13} and Q_{21} on a 10GB database

In this section, we have presented several examples that illustrate that query runtimes are significantly impacted by interactions in query mixes. Thus, in order to accurately predict the query runtimes, we need to take into account the impact of other concurrently executing queries. Next, we discuss the effect of query interactions on resource consumption.

3.4 Effect of Query Interactions on Resource Consumption

We present some observations about resource consumption in different query mixes. Here, again, we see that traditional approaches that profile the resource consumption of individual queries and workloads while ignoring interactions may not be useful. When queries run concurrently, resource utilization and performance bottlenecks can shift considerably from one mix to another. In Tables 3.8 and 3.9 we show the resource consumption of different mixes with two-way query interaction and $M = 5$ on the 10GB database. The tables report average CPU utilization (in %) and average milliseconds per disk transfer for the different mixes (a measure of I/O consumption). Milliseconds per disk transfer is a direct measure of disk response time including the queueing time, so it captures the effect of varying load.

In both tables we run Q_{13} with one other query type and observe the resource consumption of the mixes. The tables present some interesting observations. It is obvious that resource consumption is different for different mixes based on query types. However, what is interesting is that, even when we have two given query types, just replacing an instance of one query type with an instance of the other can significantly change resource consumption, validating our assertion about the significance of interactions. Consider mixes m_{16} to m_{21} which all consist of instances of query types Q_{13} and Q_9. In m_{19}, the CPU utilization drops significantly and even the disk transfer time is reduced as compared to mixes m_{16} to m_{18}. The CPU utilization then rises considerably again in mix m_{20}. Similar observations are made for mixes m_{22} to m_{27}, where the CPU utilization keeps rising from mix to mix. However the disk transfer time first rises from mixes m_{22} to m_{26} and then drops in mix m_{27}. All these changes happen by just changing one query instance from one mix to the next. Also, observe the complex pattern of resource consumption and how it changes from mix to mix due to the nature of query interactions. It is clear from these tables that considering query mixes is important for answering questions about resource consumption. Another interesting observation from these tables is that there is little correlation between resource consumption and query runtime. These observations lead us to the conclusion that monitoring resource consumption is not a good way to predict performance and interactions in query mixes. Instead, a better approach is to directly monitor and model the completion time of different query types in different query mixes, as we do in this dissertation.

3.5 Identifying Query Types in the System

So far we have assumed that the query types $Q_1, \ldots, Q_T$ are given. The query types could be identified by the DBA or they could be tagged by the application. This section presents some novel techniques 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*. In this dissertation, we use the TPC-H benchmark where query templates are already given to us. However, query templates can be extracted from query logs by writing a simple query template extractor.

A typical query template 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., [74]). 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 [47, 67].

The TPC-H 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 = ?
```

The one-to-one correspondence between query templates and query types works well for databases where data values have uniform distribution. This is the case with the standard TPC-H specification. However, equating query types to query templates can be a suboptimal choice in the presence of data skew. We study the performance of all instances of a query type Q_j in a mix m_i based on a single number A_{ij}, the average completion time of all instances of Q_j in m_i. This representation works as long as all instances of Q_j in m_i have similar performance. This is the case when the underlying data values have a uniform distribution. 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, in the presence of data skew, 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, our approach is to partition the query template into two or more query types. Thus, in the general case of skewed databases, each query template will correspond to *one or more* query types.

We present a methodology that helps us in automatically determining the best set of query types in the database workload. Our methodology is implemented in a module called the *query template partitioner*, which partitions a template into multiple query types if different instances of that template can have different completion times. We present this module next.

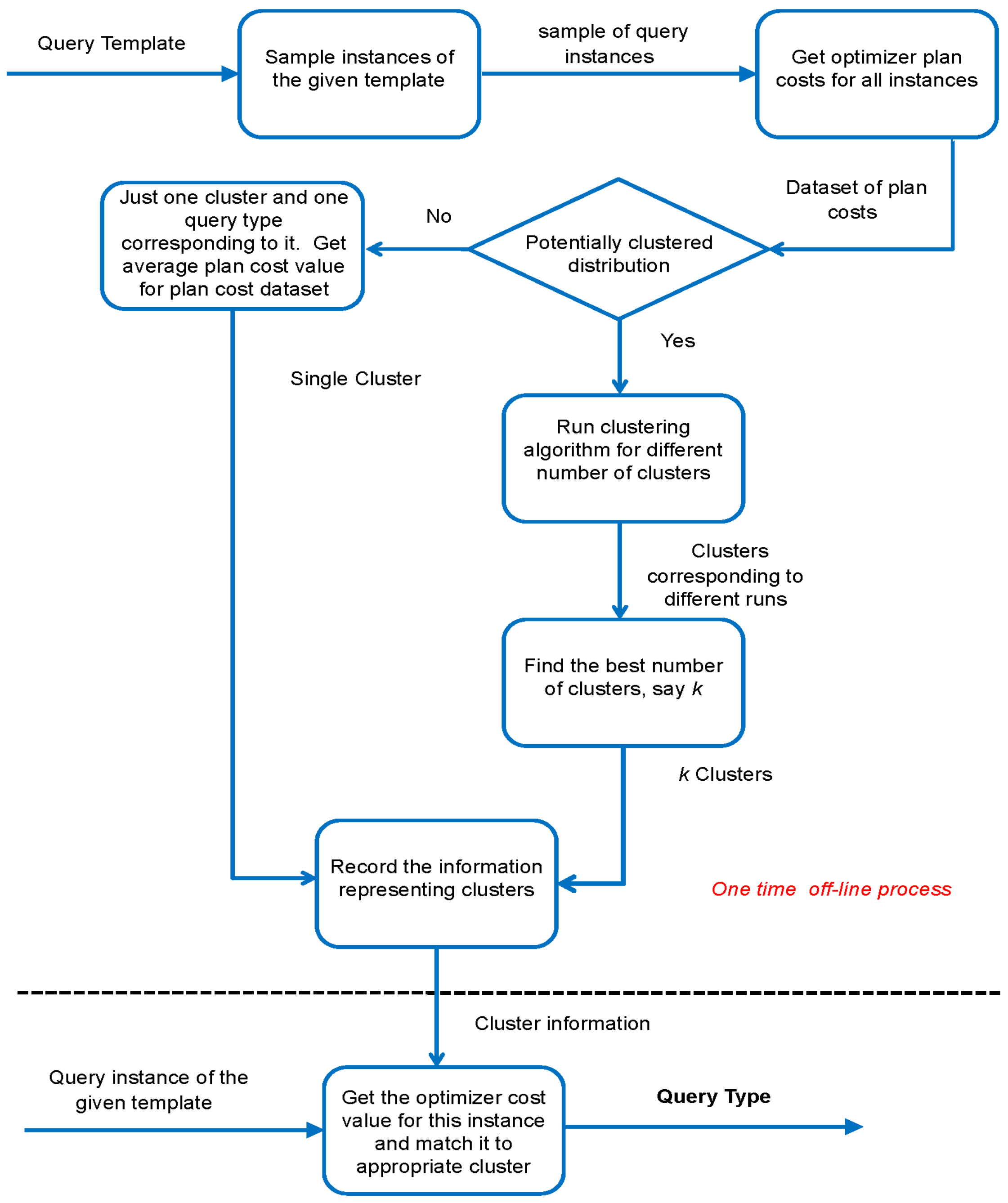
3.5.1 Query Template Partitioner

The query template 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. Thus, we need not reinvent the wheel on that front. The partitioner's work flow, summarized below and in Figure 3.2, leverages the query 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. Let the instances be q_1,...,q_n.
2. For each instance q_i, run the query optimizer to find q_i's execution plan p_i, and p_i's estimated cost c_i. Thus, we get n tuples of the form $\langle q_i, p_i, c_i \rangle$.
3. Consider the distribution of c_i values, and decide whether these values form a single cluster or multiple clusters, as described below. 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 query type of a given instance q of template Q, we use the query optimizer to find q's plan p and the associated cost c. The cluster centroid closest to c defines q's query type.

Note that even though this approach relies on the query optimizer, we do not rely on the optimizer cost estimates to estimate the performance of queries. Moreover, we never execute the queries to determine the query template partitioning or the type of a given input query. What we want is for the optimizer to be able to come up with different cost estimates for different query instances of the same query template if the parameter values in these different instances have an effect on performance due to data skew. That is, we need the optimizer to be able to detect the effect of data skew on performance and reflect this in its cost estimates. Beyond that, we do not use the optimizer cost estimates.

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)* –



Query Template
Sample instances of the given template
sample of query instances
Get optimizer plan costs for all instances
Dataset of plan costs
Potentially clustered distribution
No
Just one cluster and one query type corresponding to it. Get average plan cost value for plan cost dataset
Yes
Single Cluster
Run clustering algorithm for different number of clusters
Clusters corresponding to different runs
Find the best number of clusters, say k
k Clusters
Record the information representing clusters
One time off-line process
Cluster information
Query instance of the given template
Get the optimizer cost value for this instance and match it to appropriate cluster
Query Type

Figure 3.2: Identifying query types from given query template

also known as the *Fano factor* – of the c_i values [32, 42]. The coefficient of dispersion of a distribution is defined as the ratio of the variance to the mean:

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

where $\bar{c}$ denotes the mean of the c_i values. A value of $CoD > 1$ (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 number of clusters, K. 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 the 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* [59] 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 drop in S_K as K is increased further. For the TPC-H queries on skewed data, 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 execution.

Suppose we want to find the type of a query instance q of a given template Q in Case (ii) where the template corresponds to multiple query types. We first use the query optimizer to find q's plan p and associated cost c. This step does not 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 query type corresponding to the nearest centroid is returned as q's query type.

To test the robustness of our approach, we use the skewed TPC-D/H database generator available at [94]. This database generator populates a TPC-H database using skewed random values that are distributed according to a Zipf distribution [101]. This distribution has a parameter z that controls the degree of skew, where $z = 0$ generates a uniform distribution and as z increases, the data becomes more and more skewed. We use a 1GB dataset that was generated using $z = 1$.

We consider TPC-D/H query templates and consider the six templates used in previous experiments (Q_1, Q_7, Q_9, Q_{13}, Q_{18}, and Q_{21}). Our first step is to see how many of these given query templates will be divided into more than one query type. For this we generate 200 instances of each query template with different values for the parameter markers and get their plan cost values by running the DB2 optimizer in its EXPLAIN mode. We then run our coefficient of dispersion test on these plan cost values. The queries show some variance in their plan cost values for different instantiation of the parameters as expected.

Query Type	$Q1$	$Q7$	$Q9\_1$	$Q9\_2$	$Q9\_3$	$Q9\_4$	$Q13$	$Q18$	$Q21$
runtime t_j (sec)	9.99	10.61	3.06	10.61	14.01	24.19	5.94	7.35	6.39

Table 3.10: Runtime, t_j, of queries in a 1GB skewed TPC-H database

Interestingly, however, only the plan costs of instances of Q_9 show enough dispersion to merit subdividing into clusters. We further verify this conclusion by examining the variation in actual runtimes of the various instances of the six query templates. The runtimes of instances of Q_9 do indeed exhibit high variation. We see runtimes varying from 3 seconds to 24 seconds. The runtimes of different instances of each of remaining five query templates also exhibit some variation as parameters vary, but the variation is not high and can be well represented by an average value. Now having decided that Q_9 needs to be divided into further query types, we run K-means clustering on the plan cost dataset of the 200 instances of Q_9. We use Matlab to run K-means clustering, and we vary 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 the silhouette metric as discussed above. The silhouette metric indicated that $K = 4$ gives best clustering quality. Thus, we now have 9 query types in the system: Q_1,Q_7,Q_{13},Q_{18},Q_{21},Q_{9_1},Q_{9_2},Q_{9_3}, and Q_{9_4}. The last 4 query types correspond to the Q_9 template.

Table 3.10 shows the runtime of queries of these types when they run alone in the system in this setting of skewed TPC-D/H database. There is significant difference in the runtime of the query sub-types of Q_9. Table 3.11 shows mixes consisting of these query types with $M = 30$. Once again, we can see widely varying performance of queries from mix to mix depending on the interactions among queries. Thus, having identified the query types in a skewed database, we can study the query mixes and interactions as we did in Tables 3.4– 3.9

The examples in this chapter illustrate that interactions in query mixes impact query runtimes significantly. We cannot accurately predict the performance of a query q unless we are able to model the effect of other queries running concurrently with q. Focusing on individual query types and ignoring interactions and the concurrently executing query mix can lead to inaccurate conclusions about performance. Thus, it is important to develop mix-based reasoning about query workloads to better manage the performance of database systems. This mix-based reasoning is our focus in Chapter 4.

Mix	$Q1$		$Q7$		$Q9\_1$		$Q9\_2$		$Q9\_3$		$Q9\_4$		$Q13$		$Q18$		$Q21$	
	N_{ij}	A_{ij}	N_{ij}	A_{ij}	N_{ij}	A_{ij}	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_{28}	4	197.63	4	272.87	1	132.41	1	205.78	2	317.40	1	435.53	8	90.67	4	183.72	5	211.32
m_{29}	1	244.31	1	207.66	1	1720.53	1	167.67	6	252.89	18	418.13	1	230.77	1	147.17	0	0.0
m_{30}	4	141.62	6	222.61	1	749.38	0	0.0	2	256.03	2	359.03	5	75.86	6	122.44	4	106.18

Table 3.11: A_{ij} (in seconds) for different query types in query mixes on a 1GB skewed TPC-H database

Chapter 4

Experiment-Driven Modeling of Query Interactions

In Chapter 3, we presented several examples of positive and negative query interactions in concurrently executing queries in query mixes. We concluded that mix-based and interaction-aware reasoning can offer us significant benefits in understanding database performance. In this chapter we turn our attention to the question of how to construct interaction-aware performance models for queries in database systems.

Traditionally, analytical formulas have been used by database query optimizers to estimate the execution cost of query plans. However, the cost models used in almost all database systems today work on one query at a time, so they cannot estimate the overall behavior of multiple concurrent queries that are interacting with each other. Developing accurate analytical formulas to estimate the properties of query mixes requires a detailed understanding of all possible causes of inter-query interactions. Interactions can arise from a variety of causes: resource limitations, locking, configuration parameter settings (including misconfigurations), properties of the hardware or the software implementation, correlation or skew in the data, and others. This space of potential causes is large, not fully known ahead of time, and can vary from one database system to another. Thus the task is seemingly impossible.

Since general-purpose analytical formulas are hard to develop for estimating query completion times, we decide to leverage machine learning techniques by employing experiment-driven modeling. Our approach for experiment-driven modeling is based on running a small set of carefully chosen query mixes to collect *training samples* of the form shown in the tables in Chapter 3 (e.g. Table 3.4). Each sample gives a measure of how the average completion time of different queries is affected by running them in a specific mix. The full set of collected samples can be analyzed to identify various interactions (and non-interactions). Having collected the samples, statistical models can be trained from these samples, and then used to estimate completion time. While the experiment-driven approach has to be repeated for each new database, hardware and operating system setting, it is effective irrespective of the underlying cause of interactions because the effect of significant interactions will be captured in the monitoring data collected from experiments.

We treat the problem of generating an interaction-aware performance model for query type Q_j as a problem of training a *regression model*. Concretely, for each query type Q_j, we need an interaction-aware performance model that will take as input the query mix $m_p = \langle N_{p1}, N_{p2}, \ldots, N_{pT} \rangle$ and return an estimate $\hat{A}_{pj}$ of the average completion time of an instance of Q_j in m_p. Thus, we build one regression model for each query type. The model is trained from a set of n samples, where sample $s_i, 1 \leq i \leq n$, has the form $s_i = \langle m_i, A_{ij} \rangle = \langle N_{i1}, \ldots, N_{iT}, A_{ij} \rangle$. Sample s_i denotes an observation that an instance of Q_j, when run in mix m_i, is completed in time A_{ij}. An appropriate type of regression model M_j can be fitted to the n samples $s_1, \ldots, s_n$ to predict the completion time A_{pj} for an instance of Q_j when run in any mix m_p. The model will have the form: $\hat{A}_{pj} = M_j(m_p) = M_j(N_{p1}, \ldots, N_{pT})$. Two key questions arise:

- *Sampling question:* How to efficiently generate a representative set of samples from which to train the model?
- *Modeling question:* What type of regression model gives the best accuracy in estimating query completion times in mixes?

We first consider the sampling question in Section 4.1. The modeling question is considered in Section 4.2.

4.1 Sampling to Collect Training Data

Our goal for sampling is to generate a representative set of samples to train the models presented in Section 4.2. This problem is non-trivial because the space of possible mixes can be very large. If M is the MPL and T is the number of query types, 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 [86]. And the number of possible selections is given by:

$$S(T, M) = \binom{M + T - 1}{M}$$

Even for the simple case of 6 query types, we get $S(6, 20) = 53130$ for $M = 20$, and $S(6, 50) = 3478761$ for $M = 50$. And when we have $T = 22$ and $M = 60$, the number of possible mixes is well above 10^{19}. Thus, as the query types and MPL increase, a simple enumeration of the space by running all possible mixes is an infeasible approach.

We need to develop algorithms that help us to design experiments to collect a subset of mixes from the entire space. On one hand, more experiments bring in more samples, which leads to more accurate models to estimate the performance of different query types in different mixes. On the other hand, experiments add overhead on the system, so we want to minimize the number of experiments needed. In this section, we consider techniques that collect representative samples by executing a small number of selected mixes.

4.1.1 CDR Sampling

A straightforward approach to collect samples is to pick randomly from the space of possible mixes. However, random sampling has some obvious limitations. For instance, a disadvantage of pure random sampling, in particular when there are relatively few samples, is that some query types may not appear at all in these few samples. Also, it is not uncommon to have scenarios in which during a certain activity period all concurrently executing queries are of the same type. Similarly, on the other end of the spectrum, there can be a typical activity period when a query mix consists of instances from all query types present in the system. Thus, if we are picking a small set of random samples, it is highly likely that we may not get even a single sample corresponding to these cases. However, we can augment random sampling with simple heuristics that are based on our observations and domain knowledge. These heuristics can help us to overcome some of the limitations of random sampling by explicitly sampling from certain "interesting" parts of the space.

With the above observations in mind, we developed a simple sampling approach called *corner, diagonal, and random sampling (CDR sampling)*. CDR sampling works as follows.

Input: Number of query types T, number of mixes to sample S, MPL $= M$;

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

2. If $M > T$, we take D samples "diagonally".

a. If $M \bmod T = 0$, we sample a mix with an equal number of occurrences of each query type, i.e., $\langle \frac{M}{T}, \frac{M}{T}, \ldots, \frac{M}{T} \rangle$. If $M \bmod T \neq 0$, then we skip this step.

b. Then, we take a fixed number of 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 randomly across a small range of values in $1, \ldots, \frac{M}{T} - 1$. A sample is collected as follows:

   i. Assign each query type k instances

   ii. Next, a query type Q_i, $1 \leq i \leq T$, is picked randomly and it is randomly assigned n_i, $0 \leq n_i \leq M - (T * k)$, instances.

   iii. Next, another query type Q_j is randomly picked and we randomly assign n_j, $0 \leq n_j \leq M - (n_i + T * k)$, instances. We continue in this fashion until we have $T * k + n_1 + n_2 \cdots + n_T = M$.

3. Finally, we take $S - (D + T)$ samples at random from the full space of mixes. A sample is collected by using a procedure similar to that outlined in steps 2b(ii) and 2b(iii) above:

   a. A query type Q_i, $1 \leq i \leq T$, is picked randomly and it is randomly assigned n_i, $0 \leq n_i \leq M$, instances. Next another query type Q_j is randomly picked and we randomly assign n_j, $0 \leq n_j \leq M - n_i$ instances. We continue in this fashion till we have $n_1 + n_2 \cdots + n_T = M$.

In step 1, we sample the corner points of the space. In step 2 we sample the "diagonal" points of the space for the case when the number of query types is less than the multi-programming level of the system. We call these mixes "diagonal" in the sense that each of theses mixes contain all the query types present in the system. The number of instances of each query types is varied randomly in these diagonal mixes. Finally we sample the remaining mixes by randomly selecting the query types and assigning a random number of instances to each query type.

4.1.2 Interaction Level Aware Latin Hypercube Sampling

CDR sampling is a straightforward approach to complement random sampling with simple heuristics to ensure that corner and diagonal points in the space are covered. But as the dimensionality of the input space (i.e., the number of query types T) increases, random sampling tends to exhibit clustering and poor coverage of the space [88]. The techniques in the family of *space-filling* designs of experiments aim to spread the samples regularly throughout the input space. *Latin Hypercube Sampling (LHS)* is one such technique that provides a space-filling experimental design.

Latin Hypercube Sampling

Latin Hypercube Sampling is a form of stratified sampling. In stratified sampling the entire space is divided into subspaces such that the subspaces are mutually exclusive and collectively exhaustive. The samples are collected to ensure that all subspaces are well represented. Latin Hypercube Sampling performs well in practice, and has been shown to perform better than simple random sampling [53, 88]. It should, however, be noted that the quality of an experimental design can only be evaluated with respect to a certain goal. In our case, the goal is to use the collected samples to train an interaction-aware performance model that can estimate the average completion time of an instance of a query type in a given mix. The lower the estimation error for the trained model, the better the sampling technique. In Section 4.3, we show that our sampling approach based on Latin Hypercube Sampling is indeed better than the CDR approach.

To illustrate LHS we first present a simple case where our objective is to sample n points in two dimensions. These n points are sampled as follows:

1. Divide the axis along each dimension into n equally spaced intervals.
2. Label the intervals along each dimension with integers $\{1, 2, \ldots, n\}$.
3. There are now n^2 cells or subranges in the plane. Each cell can be identified with an ordered pair (i, j). Now randomly pick n cells such that each interval along each dimension appears only once in these n cells. One way to accomplish this can be to randomly permute the set $\{1, 2, \ldots, n\}$ for each dimension and then pick the ordered pair (i, j) by picking the corresponding elements i and j from each set.

Figure 4.1 shows an example where we have two dimensions, and we have to select $n = 5$ points using LHS. In our setting, this will correspond to selecting $n = 5$ mixes for $T = 2$ query types. The two axes T_1 and T_2 in Figure 4.1 denote the number of query instances of each query type in a mix. Latin Hypercube Sampling divides each of these dimensions into 5 equal intervals. The "*" symbols in Figure 4.1 denote the set of mixes that LHS selects.

We can now generalize the above approach. In general, LHS selects n mixes from a space of T query types as follows:

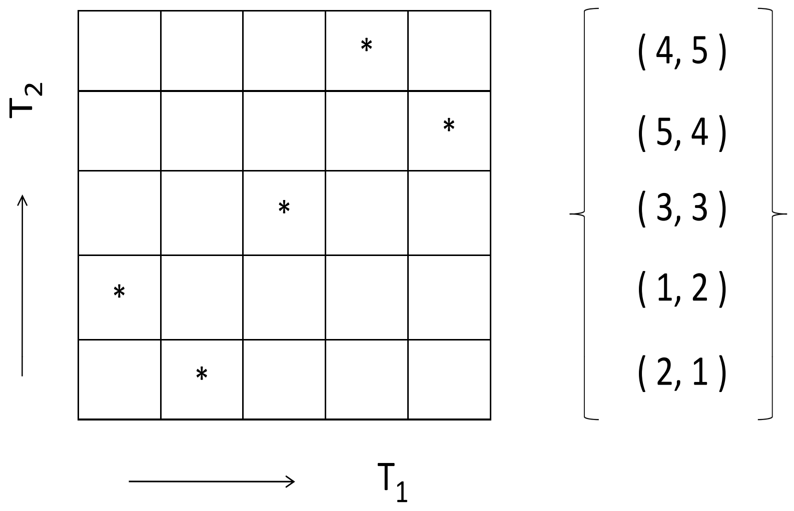


Figure 4.1: Space-filling sampling from the space of possible mixes via Latin Hypercube Sampling (LHS)

1. The range of the number of possible instances of each query type is divided into n equal subranges.
2. n mixes are selected from the space such that each subrange of each query type has one and only one selected mix in it. This can be done, as described above, by randomly permuting the set $1, 2, \ldots, n$ for each dimension and picking the corresponding elements of each permutation for the samples.

Thus, each subrange in each dimension is covered by exactly one selected mix and no two selected mixes have the same subrange in any of the dimensions. Step 2 above guarantees this property of LHS and leads to the selection of a good set of samples that is well spread out in the space.

Interaction Level of Query Mixes

In Section 4.1.1 we augmented random sampling with heuristics to sample corner and diagonal points. Beyond these simple heuristics, we do not assume any prior knowledge about the workloads. However, it is clear that it would be beneficial if the mixes executed during sample collection contain enough information for predicting the performance of unobserved mixes. In this dissertation we are interested in information about query interactions. Thus, if an insight about query interactions can lead to better samples, we can augment our sampling algorithm by incorporating this insight.

One such insight that we observed during our empirical study of query interactions is that the performance of different query types in a mix is greatly influenced by the number of distinct query types present in the mix. We define the *interaction level (IL)* of a mix as the number of distinct query types in this mix. Notice that total number of possible interaction levels in a mix would be $num\_ILs = min(T, M)$.

We demonstrate experimentally that covering all possible interaction levels of mixes in the sample is important for accurate prediction. We want to show that given a set of mixes whose performance we want to predict, sampling from the subspace of mixes with the same interaction levels (ILs) as the given mixes can produce a better prediction model than sampling from the subspace of mixes with interaction levels different from that in the given mixes. In our testing of this hypothesis we use the Gaussian processes (GP) modeling approach, which is described in Section 4.2.2. However, the results are similar for other modeling approaches discussed in Section 4.2. The testing methodology is as follows:

1. Create a set of mixes with interaction levels in the range $Range\_IL_1$ as the test mixes.
2. Create another set of random mixes with ILs in the range $Range\_IL_1$ as the training mixes and learn a Gaussian processes (GP) model GP_1 from them. Evaluate the prediction accuracy of GP_1 on the test mixes.
3. Create another set of random training mixes with ILs in the range $Range\_IL_2$, such that $Range\_IL_2$ is disjoint from $Range\_IL_1$ and learn a GP model GP_2 from the these mixes. Evaluate the prediction accuracy of GP_2 on the test mixes.
4. Compare the accuracy values from Steps 2 and 3. If GP_1 can make more accurate prediction than GP_2, it shows that to build better prediction models for a set of test mixes, we should use mixes with the same interactions levels in our training samples as those found in the test mixes.

The error metric we use to evaluate the prediction accuracy is the *mean relative error (MRE).* If we pick Y test mixes where a_i is the actual performance metric observed for mix m_i, and e_i is the model predicted value of the performance metric, MRE is defined as: $\frac{1}{Y}\sum_{i=1}^{Y}\frac{|e_i - a_i|}{a_i}$.

MRE is commonly used for computing model accuracy. We test the effect of the interaction level of the training sample on performance using mixes containing the six

Train	Test	Q1	Q2	Q3	Q4	Q5	Q6
TR_1	TT_1	0.21	0.28	0.26	0.21	0.2	0.41
TR_1	TT_2	0.62	2.3	1.01	0.79	0.54	1.22
TR_2	TT_1	0.32	0.5	0.45	0.36	0.31	0.69
TR_2	TT_2	0.15	0.25	0.19	0.16	0.15	0.25

Table 4.1: The effect of the interaction levels of the training mixes on prediction error

longest running query types on a 1GB database, shown in Table 3.1. The mixes are sampled for $M = 30$ and are split into two subsets: (i) Set I that contains mixes with ILs in $Range\_IL_1 = [1, 3]$, and (ii) Set II that contains mixes with ILs in $Range\_IL_2 = [4, 6]$. We randomly pick 500 mixes from Set I as training mixes TR_1, and we pick another disjoint 300 mixes from Set I as test mixes TT_1. Similarly from Set II we pick 500 training mixes TR_2 and pick 300 test TT_2.

Table 4.1 shows MRE in different scenarios. For each row, the first column shows the set of training mixes that was used to train Gaussian processes models. The second column shows the set of test mixes. The remaining six columns shows MRE for different query types in test mixes. The main observations are:

i. When the training mixes and test mixes come from the same range of interaction levels, prediction is more accurate (lower error in the table). For instance, the case of using TR_2 to predict TT_2 is much better than the case of using TR_1 to predict TT_2, since in the former case the query interactions present in TT_2 are captured by mixes in TR_2, while in the latter case this is not true. For example, let us look at the prediction errors for query Q_1 in Table 4.1. When $Train = TR_1$ and $Test = TT_1$, the prediction error for Q_1 is 0.21. When $Train = TR_1$ and $Test = TT_2$, the prediction error for Q_1 is a much higher 0.62.

ii. In both cases where training mixes and test mixes are from different ILs, the prediction accuracy is poor. However, using TR_2 to predict TT_1 is relatively better than using TR_1 to predict TT_2. Recall that TR_2 and TT_2 are both from $IL \in [4, 6]$, which partially capture query interactions in mixes from $IL \in [1, 3]$. Therefore the GP model learned from TR_2 covers more query interactions than that of the model learned from TR_1.

The above evaluation demonstrates that it is important to sample mixes that cover different interaction levels to make accurate performance prediction for unseen mixes that will be encountered in real workloads, and whose interaction levels are unknown at sampling time. Thus, we would like to incorporate this heuristic of sampling from the subspaces that cover mixes with different interaction levels in our sampling algorithms. Also, recall that we have a hard constraint that the multi-programming level of the system is M. We adapt LHS to satisfy the condition on MPL and make the sampling algorithm interaction level (IL) aware. These adaptations give us an interaction level aware LHS sampling approach, which we call, *IL_LHS*.

Input: Number of query types T, number of mixes to sample S, MPL = M;

1. We start by running T experiments where we sample the "corner" points of the space, i.e., the mixes $\langle M, 0, \ldots, 0\rangle$, $\langle 0, M, \ldots, 0\rangle$, ..., $\langle 0, 0, \ldots, M\rangle$.
2. For every interaction level k, $2 \leq k \leq num\_ILs$
   i. Generate a matrix MT_k of size $(S * T)$ using the standard LHS design; each row vector represents a mix.
   ii. For each mix (row vector) in MT_k, randomly set $T - k$ out of T values to 0 to make IL equal to k for this mix, and scale the values to make each value an integer such that the sum of these $T - k$ values is equal to M.
   iii. Since our budget is S and we want to cover each IL, the number of mixes that are allowed for $IL = k$ is $\lceil (S - T)/(num\_ILs - 1)\rceil$. We randomly pick this number of mixes from a given MT_k for each k.

Incremental Sampling

An issue with IL_LHS sampling is that if we collect a sample set with a small number of samples, say 100 mixes, and a sample set with a large number of samples, say 200 mixes, the smaller set will not be a subset of the larger set. The IL_LHS algorithm needs to know the total number of samples S prior to the running of the algorithm. If we invoke the algorithm again to add more samples, it would not satisfy the requirements of stratified sampling because the algorithm would try to cover the subspaces that are already covered. To address this issue, we developed an *incremental* version of the IL_LHS algorithm that works as follows:

1. First, select the T samples with IL=1 which correspond to the "corner points of the mix space" of the form $\langle M, 0, 0, ..., 0\rangle$, $\langle 0, M, ..., 0\rangle$, ..., $\langle 0, 0, ..., M\rangle$.

2. Use the IL-aware algorithm to plan $S-T$ training samples to collect from the feasible ILs greater than 1.

3. Generate a sequential list of samples by sampling randomly without replacement from the set of $S-T$ samples. This step maintains a roughly uniform distribution across ILs as more samples are collected

The DBA can specify a large S according to her budget. Then she can collect samples incrementally based on the sequential list from Step 3 above. She can suspend the sampling process any time, and resume it later. This approach of specifying a large set of samples in advance and collecting the samples incrementally can also be used with CDR sampling. If this incremental sampling approach is used, the DBA can decide to stop the sampling at any time if she finds that the accuracy of the models is adequate.

We now have two sampling approaches to collect data for training the models: CDR sampling and IL_LHS. CDR sampling is simpler while IL_LHS gives better coverage of the space of possible query mixes. We expect IL_LHS to result in more accurate performance models, an issue we will study in Section 4.3. But first, we discuss the modeling question and present different modeling approaches that we have considered.

4.2 Regression Models for Query Interactions

There are many candidate regression models in the machine-learning literature, e.g. linear regression, locally weighted linear regression, regression trees, polynomial regression, and neural nets [108]. Such machine-learning techniques have been employed in database tuning before, including for predicting single-query completion times without taking interactions and query mixes into account (e.g., [46]).

Our choice of which model type to use is driven by two key constraints. First, we cannot rely on the features of individual queries alone since we have to capture query interactions. Second, while previous work assumed that many training samples (few 1000s) are available, we expect few training samples per model because of the high cost of sample generation. As a rule of thumb in machine learning, more complex model types can be more accurate if (and only if) trained well, but they need more samples for accurate training. On the other hand, the simplicity of a model does not necessarily imply that it is always going to be less accurate. If the underlying data is relatively simple and we have a good representative sample to train the model, even a simple model can yield good accuracy. We discuss two types of models that we have considered. In Section 4.3, we study the accuracy of the models.

4.2.1 Linear Regression Models

The simplest type of model that we have considered for modeling query completion times is the *linear regression model* (LR). A linear regression model uses the following structure to compute $\hat{A}_{pj}$, the estimate of average completion time for query type $Q_j, 1 \leq j \leq T$ in mix m_p:

$$\hat{A}_{pj} = \sum_{k=1}^{T} \beta_{kj} N_{pk}$$

where N_{pk} denotes the number of instances of query type $Q_k, 1 \leq k \leq T$ in mix p. The β parameters are regression coefficients that will be estimated while learning the model from training data. There is a different set of β parameters for each query type, and they are all learned from the training data, giving us a linear regression equation for each query type. There is no intercept term present in the regression equation. This fits the intuition that when we have zero instances of all the query types, the predicted completion time is zero. The β parameters of the regression models are learnt once, and they are repeatedly used to estimate the completion time of different queries in different mixes.

As mentioned above, the β parameters are estimated while learning the model from training data. Given a training set of S samples of the form $\{A_{ij},\ N_{i1}, \ldots, N_{iT}\}$, for query type Q_j, we have:

$$A_{ij} = \beta_{1j} N_{i1} + \beta_{2j} N_{i2} + \ldots + \beta_{Tj} N_{iT} + \varepsilon_{ij}, \quad i = 1, \ldots, S$$

The above equation can be written in vector form as:

$$A_j = N\beta_j + \varepsilon_j$$

where

$$A_j = \begin{pmatrix} A_{1j} \\ A_{2j} \\ \vdots \\ A_{Sj} \end{pmatrix},\ N = \begin{pmatrix} N_{11}\ N_{12}\ \cdots\ N_{1T} \\ N_{21}\ N_{22}\ \cdots\ N_{2T} \\ \vdots \\ N_{S1}\ N_{S2}\ \cdots\ N_{ST} \end{pmatrix},\ \beta_j = \begin{pmatrix} \beta_{1j} \\ \beta_{2j} \\ \vdots \\ \beta_{Tj} \end{pmatrix},\ \varepsilon_j = \begin{pmatrix} \varepsilon_{1j} \\ \varepsilon_{2j} \\ \vdots \\ \varepsilon_{Sj} \end{pmatrix}$$

The β_j parameters can be estimated using the method of least squares estimation [8, 61, 62]. The ϵ_j terms are error terms that are minimized by the least-square estimates of β_j.

4.2.2 Gaussian Processes

In linear regression, the entire training data is used to learn the regression parameters in a pre-determined manner and hence a single global model for each query type is used to fit all the training data. Intuitively speaking, this approach can be hit or miss: the prediction is good only if the true structure of the data matches the global linear structure assumed by the model. Also, an underlying assumption is that the structure of the data in the modeling space is relatively simple so that it can be effectively captured by a simple linear model.

However, as we have seen in Chapter 3, query interactions can be very complex, and the performance of queries may change dramatically with small changes in the mix. This motivates us to consider a different and more powerful modeling approach called *instance-based learning* [108]. The basic idea in instance-based learning is to infer the performance of a mix of interest, m_p, from the performance of mixes in the training set that are "close" to m_p. Intuitively, instance-based learning focuses on the *local space* around m_p at prediction time.

A simple type of instance-based learning is *1-nearest neighbor (1-NN)*. When asked to estimate A_{pj}, the average running time of query type Q_j in mix m_p, 1-NN returns the average running time of Q_j in the mix that is nearest to m_p among the training samples according to some distance metric. The *k-nearest neighbor (k-NN)* model is a more robust alternative to 1-NN. A k-NN model finds the top k mixes that are nearest to m_p among the training samples, denoted $m_1, \ldots, m_k$, and returns the estimate $\hat{A}_{pj} = \frac{\sum_{i=1}^{k} A_{ij}}{k}$. However, the density of training samples can differ across different regions of the full space. The parameter k degrades to a coarse and unpredictable tuning knob. In dense regions, k-NN focuses on the local space and in less dense regions it becomes more global.

In this dissertation, we use *Gaussian processes (GP)* from machine learning as our instance-based learning technique. In this section we explain what a Gaussian process is and how it is used for regression.

A univariate Gaussian (Normal) distribution can be specified by its mean and variance. Thus, a random variable $r \in \mathbb{R}$, that has Gaussian distribution with mean m and variance v^2, is written as $r \sim \mathcal{N}(m, v^2)$, and its probability density function is given as:

$$p(r; m, v^2) = \frac{1}{\sqrt{2\pi}v} e^{-\frac{1}{2v^2}(r-m)^2}$$

A multivariate Gaussian distribution generalizes the univariate Gaussian distribution to multiple dimensions and it can be specified by a mean vector and a covariance matrix.

A vector valued random variable $Y \in \mathbb{R}^d$, that has multivariate Gaussian distribution with mean vector $\mu \in \mathbb{R}^d$ and covariance matrix C, is written as $Y \sim \mathcal{N}(\mu, C)$, and its probability density function is given as:

$$p(Y; \mu, C) = \frac{1}{(2\pi)^{d/2}|C|^{1/2}} e^{(-\frac{1}{2}(Y-\mu)^T C^{-1}(Y-\mu))}$$

A Gaussian process is a stochastic process ($\mathcal{Z}_x : x \in \mathcal{X}$, where $\mathcal{X}$ is the index set), i.e., a collection of random variables. The defining characteristic of a Gaussian process is that any finite subset of these random variables has a multivariate Gaussian distribution [84]. A Gaussian process is completely specified by its *mean function* and *covariance function*. For any finite subset of random variables from the Gaussian process, the mean function returns the mean vector for that subset and the covariance function gives us a covariance matrix for that subset.

Before understanding how we can use Gaussian processes for our regression problem, we look at two standard properties of the multivariate Gaussian distribution that we are going to use [105].

P1. Let $Y \in \mathbb{R}^d$ be a multivariate Gaussian random variable such that $Y \sim \mathcal{N}(\mu, C)$. Let $a = (1, 2, \ldots r)$ and $b = (r+1, \ldots d)$, and then partition Y such that

$$Y = \begin{pmatrix} Y_a \\ Y_b \end{pmatrix}, \text{ where } \mu = \begin{pmatrix} \mu_a \\ \mu_b \end{pmatrix} \text{ and } C = \begin{pmatrix} C_{aa} & C_{ab} \\ C_{ba} & C_{bb} \end{pmatrix} \tag{4.1}$$

The distribution of Y_b conditional on $(Y_a = y_a)$ is multivariate Gaussian $(Y_b|Y_a = y_a) \sim \mathcal{N}(\mu_{b|a}, C_{b|a})$, such that

$$\mu_{b|a} = \mu_b + C_{ba}C_{aa}^{-1}(y_a - \mu_a), \text{ and} \tag{4.2}$$

$$C_{b|a} = C_{bb} - C_{ba}C_{aa}^{-1}C_{ab} \tag{4.3}$$

Since Y is a d-dimensional vector, this distribution essentially gives us a distribution on b co-ordinates out of the d co-ordinates of Y when the remaining a are given.

P2. The sum of two independent Gaussian random variables $Y \sim \mathcal{N}(\mu, C)$ and $Z \sim \mathcal{N}(\mu', C')$ is also Gaussian, with $Y + Z \sim \mathcal{N}(\mu + \mu', C + C')$.

In order to understand how Gaussian processes are used for regression, consider the Bayesian formulation of linear regression. Given a set of data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, the probabilistic interpretation of linear regression [25] assumes that the y_i variables are univariate Gaussian random variables such that

$$y_i = w^T x_i + \epsilon_i, \quad i = 1, \ldots, n$$

where it is assumed that ϵ_i are independent and identically distributed (i.i.d.) noise variables such that $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. Similarly, it is assumed that parameters vector w is a multivariate Gaussian with prior distribution $w \sim \mathcal{N}(0, \rho^2 I)$. Given these assumptions, explicit results can be obtained for posterior predictive distribution and inference can be made for new data points. Gaussian processes regression generalizes Bayesian linear regression as follows [84].

Let us assume that we are given a set of observed data points $(x_1, y_1), (x_2, y_2) \ldots (x_n, y_n)$, and we want to make predictions for new data points $(x_{n+1}, y_{n+1}), \ldots, (x_l, y_l)$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.

Our task at hand is to get a posterior predictive distribution on unobserved $y_{n+1}, \ldots, y_l$, given observed $y_1, \ldots y_n$. Let $y_b = (y_{n+1}, y_{n+2}, \ldots, y_l)$ and $y_a = (y_1, y_2, \ldots, y_n)$. Let $(Y_1, Y_2, \ldots, Y_l)$ be random variables that model observed and unobserved y values. Let there be a Gaussian process $\mathcal{Z}_x$ with zero mean function and covariance function $\mathcal{K}$ (the covariance function is also known as the *kernel function*). Gaussian processes regression (GPR) models the random variable Y_i as:

$$Y_i = \mathcal{Z}_{x_i} + \epsilon_i, \quad i = 1, \ldots, l$$

This can be written in vector form as:

$$\mathbf{Y} = \mathcal{Z} + \varepsilon \tag{4.4}$$

where $\varepsilon = (\epsilon_1, \epsilon_2, \ldots, \epsilon_l)$ is a multivariate Gaussian. By the i.i.d. noise assumption we have (recall that σ^2 is the variance of the noise distribution):

$$\varepsilon = \begin{pmatrix} \varepsilon_a \\ \varepsilon_b \end{pmatrix} \sim \mathcal{N}(0, \mathbf{K}_\varepsilon), \text{ where } \mathbf{K}_\varepsilon = \begin{pmatrix} \sigma^2 I & 0 \\ 0 & \sigma^2 I \end{pmatrix}$$

The vector $\mathcal{Z}$ is a multivariate Gaussian by the definition of a Gaussian process such that $\mathcal{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$, where $\mathbf{K}$ is the covariance matrix. Each entry $\mathcal{K}_{ij}$ in matrix $\mathbf{K}$ is defined as $\mathcal{K}_{ij} = \mathcal{K}(x_i, x_j)$ (recall that $\mathcal{K}$ is the covariance function). As in Equation 4.1, $\mathbf{K}$ can be written as:

$$\mathbf{K} = \begin{pmatrix} \mathbf{K}_{aa} & \mathbf{K}_{ab} \\ \mathbf{K}_{ba} & \mathbf{K}_{bb} \end{pmatrix}$$

Thus, by property P2 discussed previously, $\mathbf{Y}$ in Equation 4.4 is a multivariate Gaussian such that $\mathbf{Y} \sim \mathcal{N}(\mathbf{0}, \mathbf{K} + \mathbf{K}_\varepsilon)$, and the matrix $(\mathbf{K} + \mathbf{K}_\varepsilon)$ can be written as:

$$\mathbf{K} + \mathbf{K}_\varepsilon = \begin{pmatrix} \mathbf{K}_{aa} + \sigma^2 I & \mathbf{K}_{ab} \\ \mathbf{K}_{ba} & \mathbf{K}_{bb} + \sigma^2 I \end{pmatrix} \tag{4.5}$$

Thus the problem of regression now boils down to finding posterior predictive distribution $\mathbf{Y}_b | (\mathbf{Y}_a = y_a)$. Property P1 tells us that $\mathbf{Y}_b | (\mathbf{Y}_a = y_a)$ is also a multivariate Gaussian such that $\mathbf{Y}_b | (\mathbf{Y}_a = y_a) \sim \mathcal{N}(\overline{\mu}_{b|a}, \mathbf{K}_{b|a})$. By substituting from Equation 4.5 into Equations 4.2 and 4.3, we get

$$\overline{\mu}_{b|a} = \mathbf{K}_{ba}(\mathbf{K}_{aa} + \sigma^2 I)^{-1} y_a, \text{ and} \tag{4.6}$$

$$\mathbf{K}_{b|a} = (\mathbf{K}_{bb} + \sigma^2 I) - \mathbf{K}_{ba}(\mathbf{K}_{aa} + \sigma^2 I)^{-1}\mathbf{K}_{ab} \tag{4.7}$$

Thus, we have the mean vector and covariance matrix of a posterior predictive distribution. For regression, what we want from this posterior predictive distribution is a point prediction for an unobserved data point $(x^\star, y^\star)$. In practice, the mean value of the conditional (Equation 4.6) corresponding to $x^\star$ is returned as $y^\star$ [84]. It is noteworthy that Gaussian processes provide the user with a powerful regression framework just by using standard properties of the multivariate Gaussian distribution. In order to do regression using Gaussian processes, all that is needed is to specify a covariance function. A covariance function defines similarity between two points and is defined in terms of input space. This function helps us to encode the assumption that the points that are close in input space are likely to have similar output values. There can be many choices for such a covariance function. A typically used function, which is the one we use in this thesis, is the squared exponential or Gaussian kernel function [84]. For $x, x' \in \mathbb{R}^d$ the function is given as:

$$\mathcal{K}(x, x') = e^{-\sum_{i=1}^{d} \frac{(x_i - x_i')^2}{\lambda^2}}, \text{ for some } \lambda > 0$$

and for query mixes m_1 and m_2, the function can be written as:

$$\mathcal{K}(m_1, m_2) = e^{-\sum_{i=1}^{T} \frac{(N_{1i}-N_{2i})^2}{\lambda^2}} \tag{4.8}$$

This covariance function has the desirable properties that when m_1 and m_2 are close, $(N_{1i} - N_{2i}) \approx 0$ and $\mathcal{K}(m_1, m_2) \approx 1$. As the distance between m_1 and m_2 increases, $\mathcal{K}(m_1, m_2)$ approaches 0.

To summarize, Gaussian processes give us a framework for using the properties of a multivariate Gaussian distribution for prediction. We are given a training set of n samples. For a query type Q_j, a sample $s_i, 1 \leq i \leq n$, has the form $s_i = \langle m_i, A_{ij} \rangle = \langle N_{i1}, \ldots, N_{iT}, A_{ij} \rangle$, and we want to predict the completion time $\hat{A}_{pj}$ for any mix m_p. By Property P2 and Equation 4.4, we can think of an $(n+1)$-dimensional multivariate Gaussian $\mathbf{Y} = (A_{1j}, A_{2j}, \ldots, A_{nj}, \hat{A}_{pj})$. The values A_{ij}, where $1 \leq i \leq n$, are known and the value $\hat{A}_{pj}$ needs to be predicted. Property P1 tells us that if we have a d-dimensional multivariate Gaussian, the conditional distribution on b co-ordinates out of the d co-ordinates, when the remaining a are given, is also Gaussian. In our case, we have an $(n+1)$-dimensional multivariate Gaussian $\mathbf{Y}$. Thus, the distribution of $Y_b = \hat{A}_{pj}$ conditional on $Y_a = (A_{1j}, A_{2j}, \ldots, A_{nj})$ is Gaussian with mean and variance given by Equations 4.6 and 4.7. To compute these equations all that is needed is a covariance function as discussed above. In principle, decision theory and loss functions can be employed to return a point prediction for $\hat{A}_{pj}$ by sampling from this conditional distribution [84]. However, in practice, the mean value of the conditional distribution is a good enough estimate that can be computed by simple linear algebraic operations as specified in Equation 4.6, once the covariance function is specified [84]. It can be seen that parts of Equation 4.6 can be precomputed based on the n training samples, and they can be used to predict the required $\hat{A}_{pj}$ for any given test mix m_p.

The Gaussian process modeling approach that we have presented has two hyper-parameters, σ and λ. In this dissertation, we use the Weka machine learning toolkit [108]. The Weka toolkit has default values for the hyper-parameters σ and λ, which we leave unchanged. Other toolkits can use more advanced methods to learn these hyper-parameters based on the training data, but we have found in our experiments that the simple approach adopted by Weka results in accurate models.

In this section we have presented two modeling approaches that we employ to estimate the query completion times in query mixes. Next we study the accuracy of our sampling and modeling approaches.

4.3 Accuracy and Cost of Modeling

In the preceding sections we presented various modeling and sampling approaches. In this section we analyze the performance of these approaches. Our analysis aims to answer two questions: (1) How accurate are our performance models? and (2) How expensive is it to build these models? We expect the more complex models such as Gaussian processes to be more accurate than simple linear regression. We also expect more samples to lead to more accuracy. However, in all realistic scenarios, we will have a limited budget to run experiments to collect samples. Thus, we are interested in studying how the accuracy improves with the number of samples, and how much time is taken to collect these samples.

4.3.1 Accuracy of Modeling

We first concentrate on the accuracy of our models. Our approach is as follows. We consider a training set S of $3T$, $5T$, $7T$, and $10T$ samples collected using the two sampling algorithms that we presented in Section 4.1. Since we have T independent *predictor variables* – the T query types – we specify the sample size in terms of multiples of T. For both IL_LHS and CDR sampling, we use the incremental sampling algorithm presented in Section 4.1.2. We evaluate the accuracy of different models trained using these samples, allowing us to study how the accuracy varies with increasing number of samples. For the test set, we use a set of $20T$ samples where $10T$ samples are collected by each of the two sampling algorithms. This ensures that our test set is not biased to any sampling algorithm.

We present the results from both the 1GB and 10GB TPC-H databases described in Section 3.2, using the same experimental setup. (We use this experimental setup throughout the dissertation.) In both cases, we show the results for Q_1 and Q_{18}. These two queries are long-running queries, and the prediction error for these two queries is representative for all other TPC-H queries except for two queries that have sub-second execution times, namely Q_{17} and Q_{19}, in 1GB database. For these two queries, the relative prediction error may be high, but the absolute error is very small, representing less than a second of runtime.

We start with the 1GB database, and we present the results from the settings when $T = 12$, and we use the 12 longest running query types shown in Table 3.1. We consider the case when $M = 30$.

Figures 4.2– 4.5, show the accuracy of different modeling approaches in predicting the completion time of Q_1 and Q_{18} for different sampling approaches on the 1GB database. The x-axis shows the number of samples in multiples of the number of query types T.

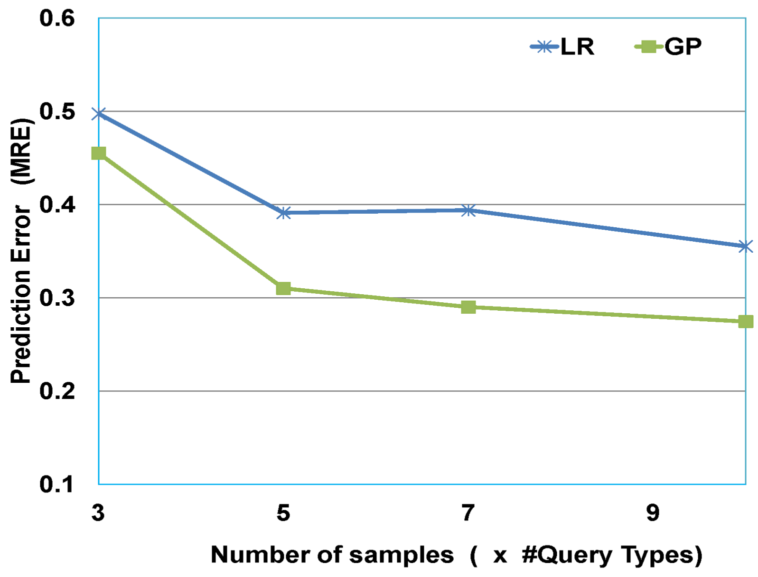


Figure 4.2: Modeling the completion time of Q_1 using CDR sampling on the 1GB database

The y-axis shows the prediction error on the $20T$ test samples measured using the MRE metric.

The figures show that GP is the better modeling approach for both sampling algorithms, and the error is higher for linear regression. We see that the additional sophistication of GP does indeed result in higher accuracy, and that the simplicity of linear regression comes at a cost of lower accuracy. The figures also show that as we increase the number of samples, the accuracy improves in general. However, it is important to realize that the relationship between the number of samples and accuracy in prediction is not always monotonic. Sometimes as we increase the number of samples, there can be a slight decrease in accuracy. This should be expected because we are measuring accuracy against a test set of mixes, and sometimes bringing in more samples may be just adding noise to the models when they are predicting the accuracy of the test set.

Another very important observation is that, in general, the improvement in accuracy is very steep as we increase the number of samples from $3T$. However as we keep collecting more samples this improvement becomes less steep. In most cases, going from $7T$ to $10T$ samples does not significantly increase accuracy. This shows that our modeling approach is robust and can get good accuracy with a reasonably small number of samples. For experiments in later chapters, we typically use $10T$ samples which – based on these results – should provide sufficiently high accuracy.

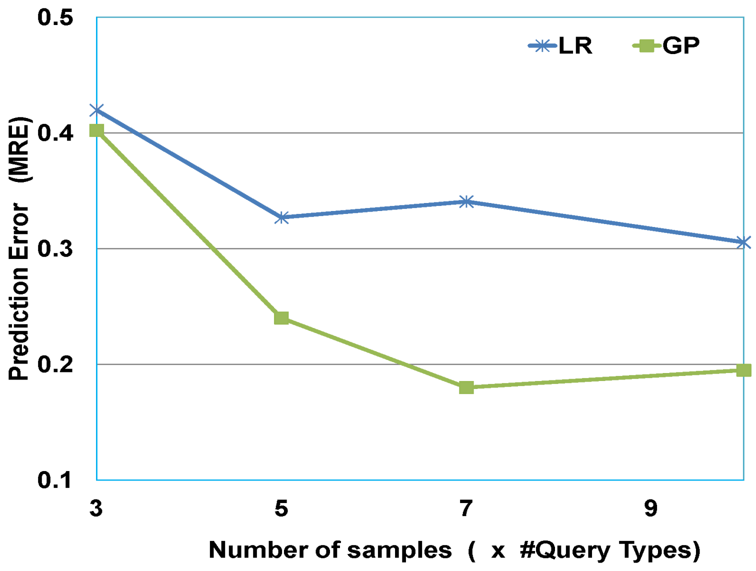


Figure 4.3: Modeling the completion time of Q_1 using IL_LHS sampling on the 1GB database

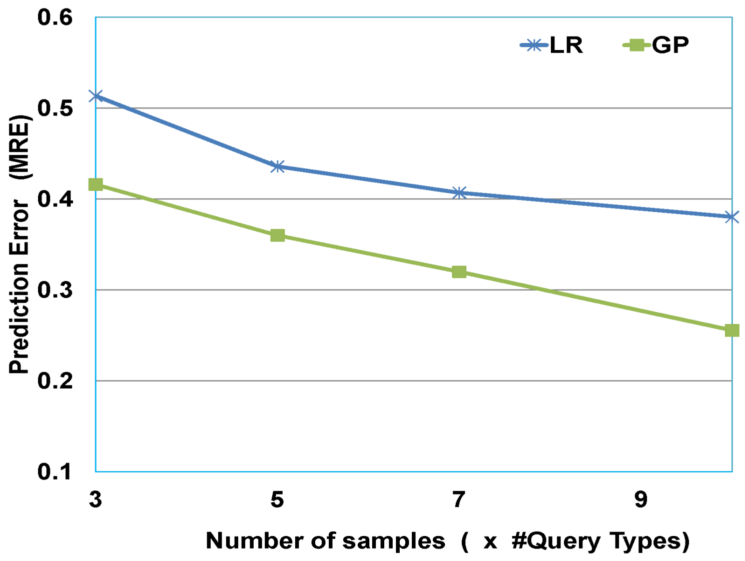


Figure 4.4: Modeling the completion time of Q_{18} using CDR sampling on the 1GB database

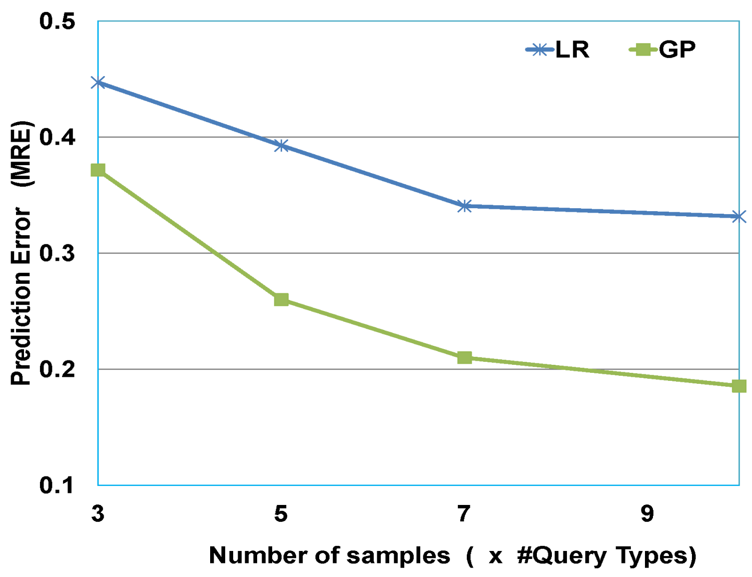


Figure 4.5: Modeling the completion time of Q_{18} using IL_LHS sampling on the 1GB database

Figures 4.6– 4.9 show the accuracy of each modeling approach in predicting the completion time of Q_1 and Q_{18} for different sampling approaches. This is the same data as in Figures 4.2– 4.5, but plotted differently to study the effect of the sampling approach. These figures show that all modeling approaches show better accuracy in case of IL_LHS. As we saw before in modeling, the more sophisticated sampling approach does indeed result in higher accuracy, justifying its extra complexity.

The figures show that for both Q_1 and Q_{18} we are able to get MRE in the range $[0.15 - 0.3]$ with 10T samples and using GP models. This is the typical error range that we observe for most of the settings.

Now we switch to the 10GB case. Figures 4.10– 4.13 show the accuracy of different modeling approaches in predicting the completion time of Q_1 and Q_{18} for different sampling approaches for the 10GB settings where $T = 6$ and $M = 10$. Again, we can observe similar trends to the ones that we observed for the 1GB database.

4.3.2 Cost of Modeling

In Section 4.3.1 we considered the accuracy of our performance models. The second question that we consider is the time taken to sample the data and train the models. For

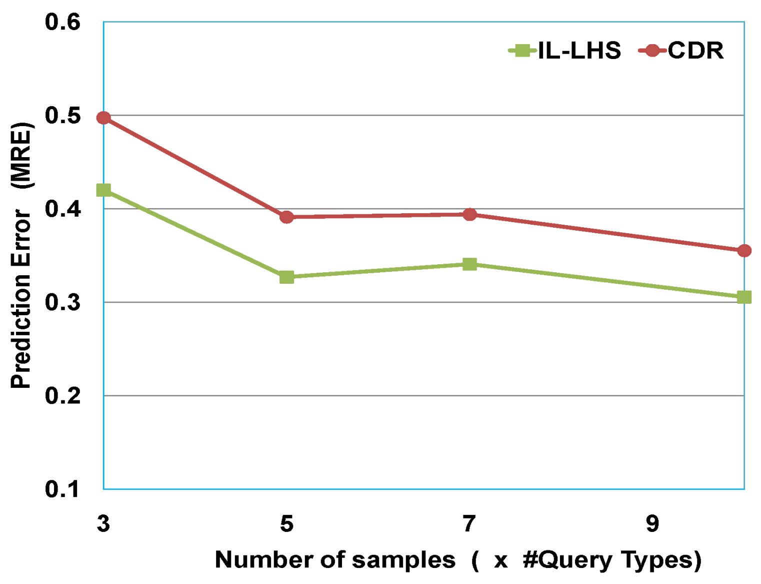


Figure 4.6: Modeling the completion time of Q_1 using linear regression on the 1GB database

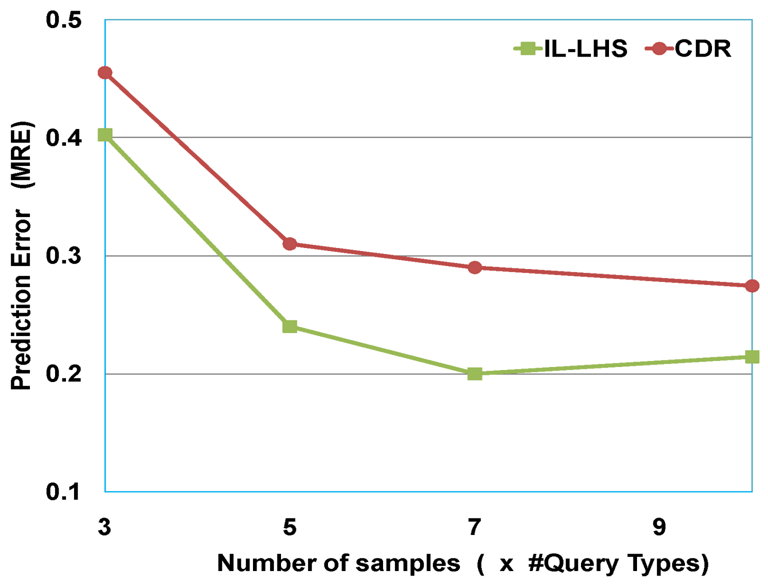


Figure 4.7: Modeling the completion time of Q_1 using Gaussian processes on the 1GB database

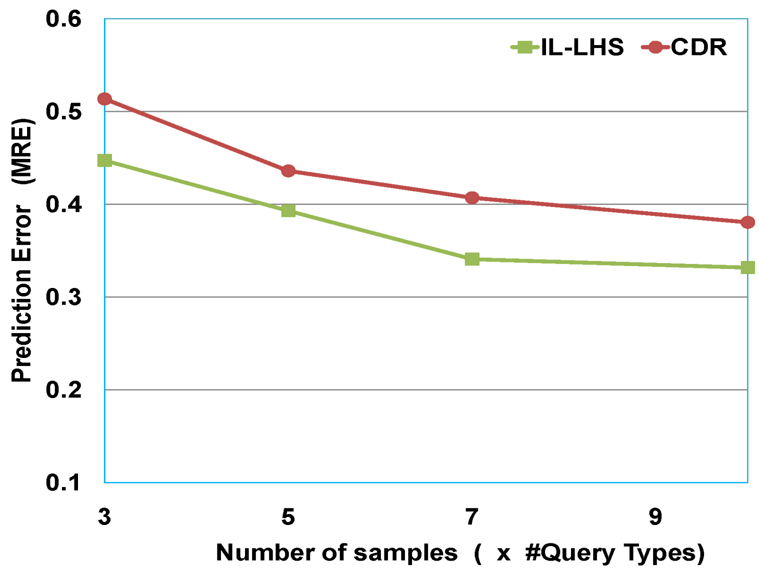


Figure 4.8: Modeling the completion time of Q_{18} using linear regression on the 1GB database

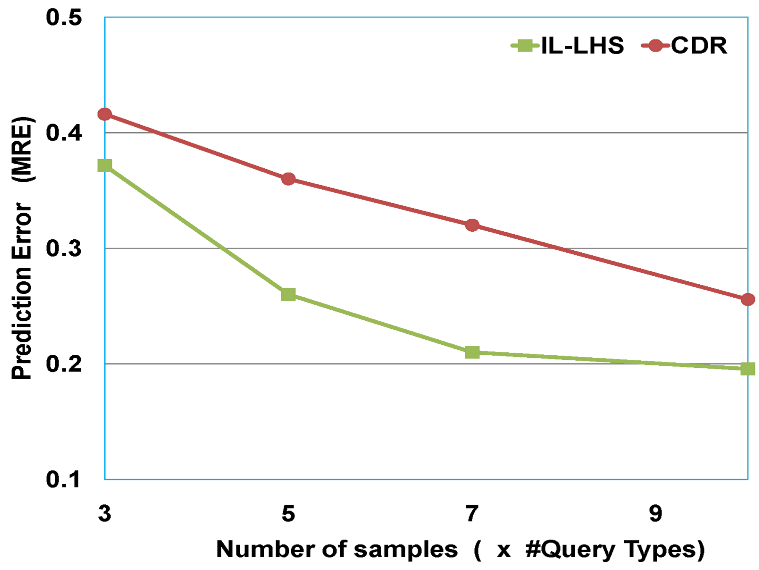


Figure 4.9: Modeling the completion time of Q_{18} using Gaussian processes on the 1GB database

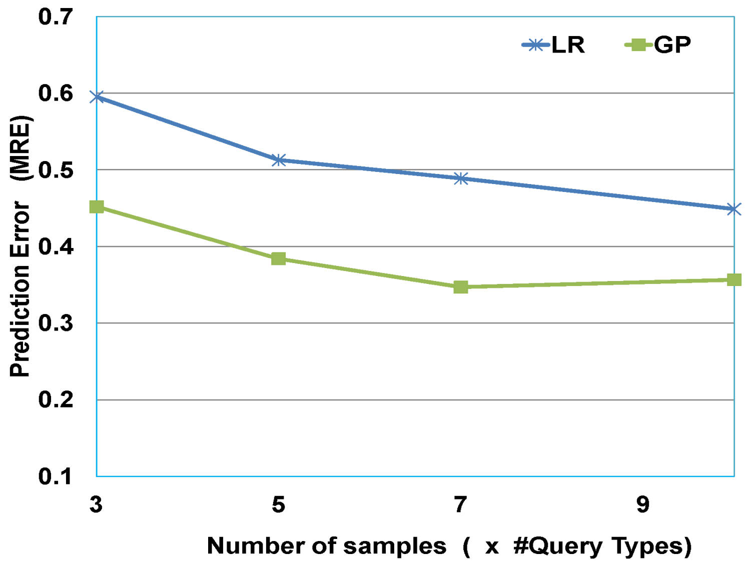


Figure 4.10: Modeling the completion time of Q_1 using CDR sampling on the 10GB database

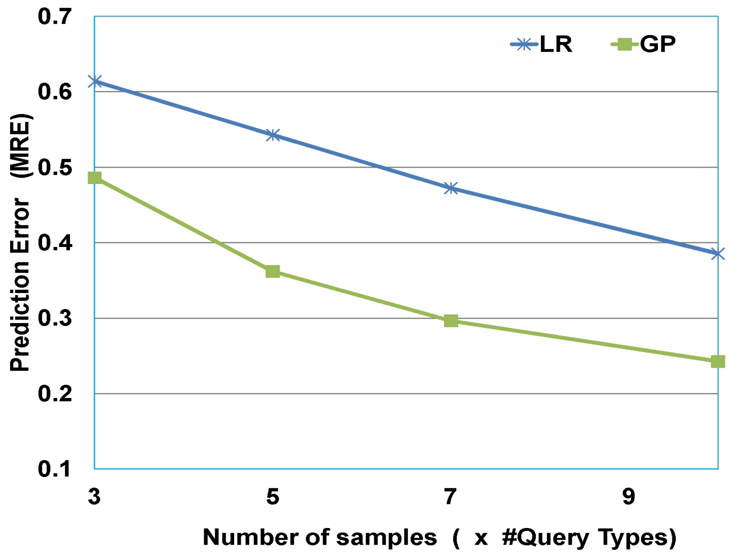


Figure 4.11: Modeling the completion time of Q_1 using IL_LHS on the 10GB database

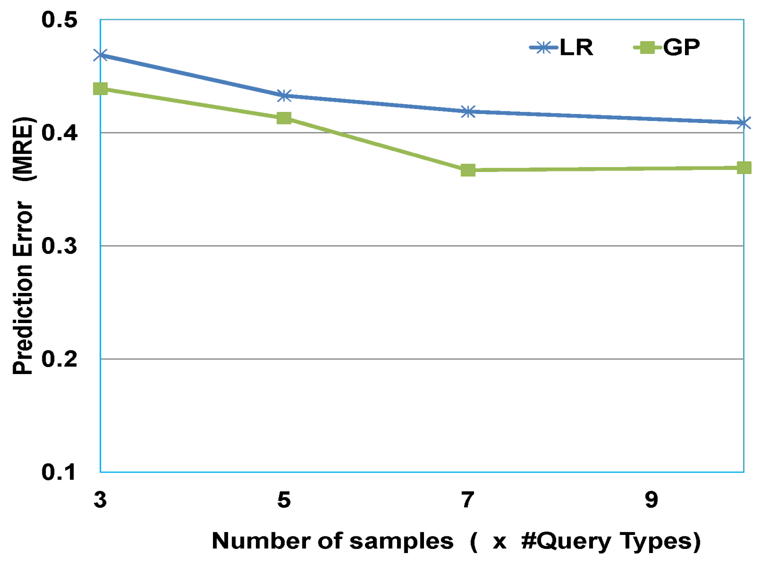


Figure 4.12: Modeling the completion time of Q_{18} using CDR sampling on the 10GB database

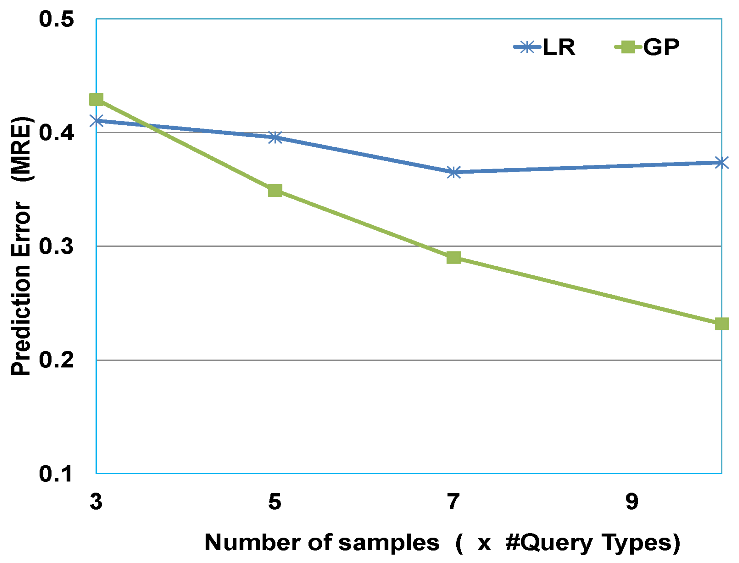


Figure 4.13: Modeling the completion time of Q_{18} using IL_LHS sampling on the 10GB database

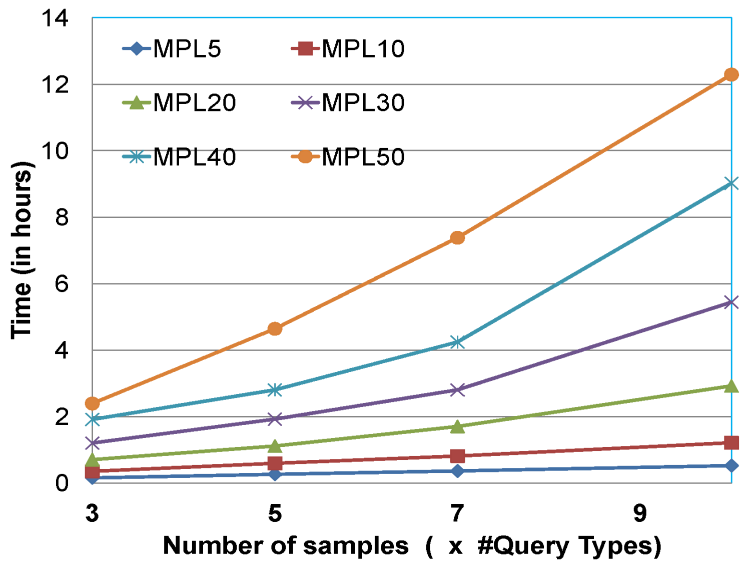


Figure 4.14: Sample collection time for 1GB database with $T = 6$

a given setting of MPL and query types in the system, the time required to collect the samples mainly depends on the number of samples. Figures 4.14– 4.17 show the sampling time for the 1GB and 10GB databases.

The figures show that sampling times increase considerably with MPL. For example, it takes less than an hour to collect samples for $T = 6$ and $M = 5$, while it takes about 12 hours to collect samples for $T = 6$ and $M = 50$. At higher MPLs more queries are running concurrently and there is significant contention for the resources resulting in higher times for the mixes.

The figures also show that as we collect more samples the increase in sampling time is not necessarily linear. This is expected because different mixes have different running times because of different interactions that come into play. Hence, increasing the number of mixes from $2T$ to $4T$ does not mean that the sample collection time will also increase by exactly two fold.

Another interesting observation is that sometimes $T = 6$ takes more time than $T = 12$. For example, it takes about 9.0 hours to collect $10T$ samples for $T = 6$ and $M = 40$, while it takes about 8.5 hours to collect $10T$ samples for $T = 12$ and $M = 40$. When we consider $T = 6$ query types in the system, we consider the 6 longest running query types. Thus, the mixes with $T = 6$, which consist of only the longest running query types may take more

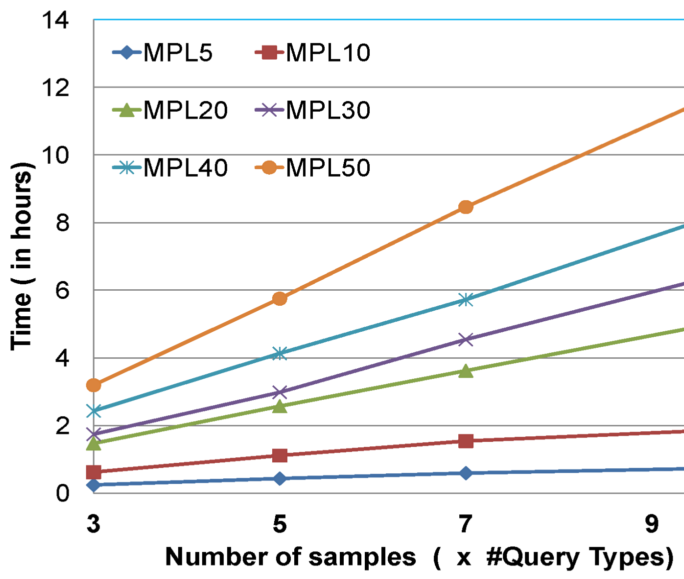


Figure 4.15: Sample collection time for 1GB database with $T = 12$

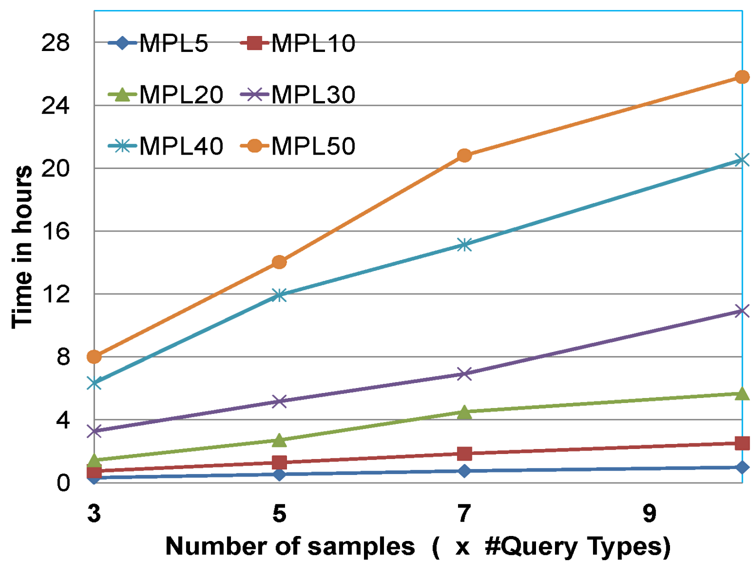


Figure 4.16: Sample collection time for 1GB database with $T = 21$

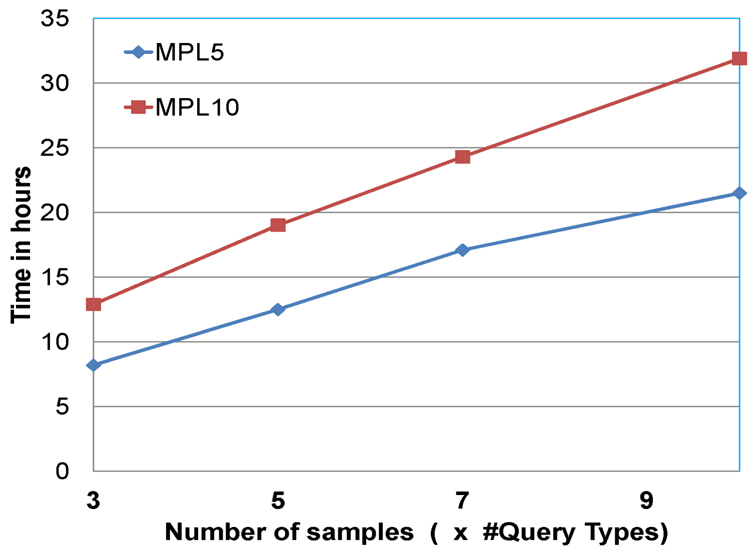


Figure 4.17: Sample collection time for 10GB database with $T = 6$

time than the mixes that also contain some short running queries, which happens as we move to $T = 12$ and $T = 21$.

Next, we consider the model training time and the prediction time for different models. In our implementation we have used the Weka toolkit [108]. We do not expect model training time and prediction time to be a significant factor, but nevertheless we want to verify this assumption. For model training and prediction, the time taken depends on the number of predictor variables, which in our case is the number of query types T. Model training time also depends on the number of samples used for training. Table 4.2 shows for different values of T the time that it takes to train the models and make predictions for a test set consisting of 600 mixes. These numbers are shown for samples collected from the 1GB database, although we emphasize that training and prediction times depend not on the size of the database, but rather on the size of the training set and the number of query types. It is clear from the table that both modeling approaches show training and prediction times that are within reasonable time limits. As T and the number of training samples increase, it can be observed that linear regression is faster, which is expected since it is the simpler model.

T	Number of Training Samples	LR	GP
$T = 6$	60	0.14	0.20
	600	0.18	3.34
$T = 12$	120	0.15	0.29
	600	0.23	3.37
$T = 21$	210	0.17	0.5
	600	0.24	3.4

Table 4.2: Model training and prediction time (in seconds) for different models.

4.3.3 Conclusion

In this chapter we considered the problem of generating interaction-aware performance models. We presented two different approaches to collect samples on which to train the models: CDR sampling and interaction level aware IL_LHS. We presented two different modeling techniques to estimate the query completion times in mixes: linear regression and Gaussian processes.

Our evaluations shows that Gaussian process models trained on samples collected by IL_LHS show the best accuracy for estimating query completion times. This is expected since GP and IL_LHS are the more sophisticated modeling and sampling approaches among the ones that we considered. On the other end of the spectrum, linear regression models trained on samples collected by CDR sampling – the combination of the simpler sampling and modeling approaches – generally show the highest error among all combinations of modeling and sampling approaches. However, the error of LR with CDR sampling is still reasonable and is not much higher than the other approaches. Henceforth we focus on these two combinations: GP with IL_LHS and LR with CDR. These two combinations represent the two ends of the spectrum for accuracy vs simplicity.

In the next chapters we put these modeling and sampling approaches to practical use and develop solutions for the problems of workload scheduling and workload completion time estimation. We will see that the required model accuracy depends on how the model is being used. For scheduling, where the goal is to distinguish good mixes from bad mixes, the simple LR with CDR is adequate. On the other hand, for completion time prediction we need the more accurate GP with IL_LHS. Thus, having different choices for sampling and modeling approaches is beneficial.

Chapter 5

QShuffler: Interaction-Aware Scheduling of Report-Generation Queries

5.1 Introduction

A core contribution of this dissertation is that we develop interaction-aware end-to-end solutions to workload management problems, demonstrating that our approach for modeling query interactions can be exploited in order to gain huge performance improvements in database systems. The first problem we consider is that of scheduling report-generation queries in database systems. As discussed in Chapter 1, report-generation workloads are a common type of workload in modern Business Intelligence (BI) settings and data warehouses [19, 49, 106].

We have developed a query scheduler, called *QShuffler* (for *Query Shuffler*), that focuses on the throughput-oriented workloads encountered in report-generation systems [11, 12, 13]. 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 [30] and Business Objects [26].

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 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 significant performance improvements over these 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.

QShuffler implements two novel interaction-aware 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 optimized for this scenario. This algorithm uses a linear-programming-based formulation of the scheduling problem.

  The linear-programming-based solution requires query preemption. The algorithm continues to perform well when we produce a preemption-free schedule, 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 (with some limited *look-ahead* that is based on observing the queries in the arrival queue). 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. We have seen in Chapter 3, 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 that these metrics are quite unrepresentative in quantifying the completion time of queries in different mixes. We develop a new metric, called *normalized runtime overhead (NRO)*, to address this problem. The *NRO* metric is a measure of the runtime 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.

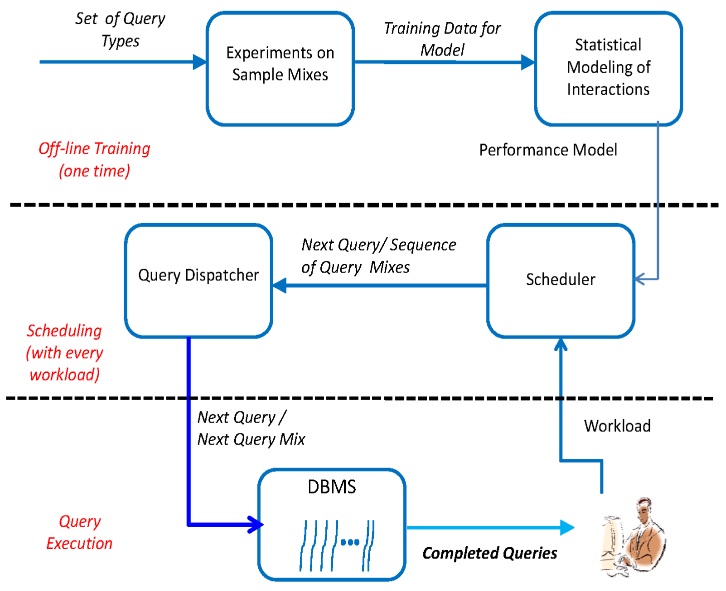

Set of Query Types
Experiments on Sample Mixes
Training Data for Model
Statistical Modeling of Interactions
Off-line Training (one time)
Performance Model
Query Dispatcher
Next Query/ Sequence of Query Mixes
Scheduler
Scheduling (with every workload)
Next Query / Next Query Mix
Workload
DBMS
Query Execution
Completed Queries

Figure 5.1: QShuffler Workflow

Figure 5.1 shows the workflow of our solution approach. Given T query types, the DBA identifies the scenarios where scheduling is needed for workloads consisting of these query types. The query types can be identified by employing the techniques presented in Chapter 3. The DBA generates a list of training mixes to be collected. Depending on the budget and resources, these training samples can be collected all at once or incrementally as described in Chapter 4. The collected samples are used for learning a statistical model that predicts performance in the presence of interactions. Sampling and model building are one-time off-line steps. After that, the learned model is used by the scheduling algorithms. Next we present QShuffler's two scheduling algorithms.

5.2 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 work. There is work on preemptive scheduling of plan operators (e.g., [21]), but our focus is on scheduling entire queries.

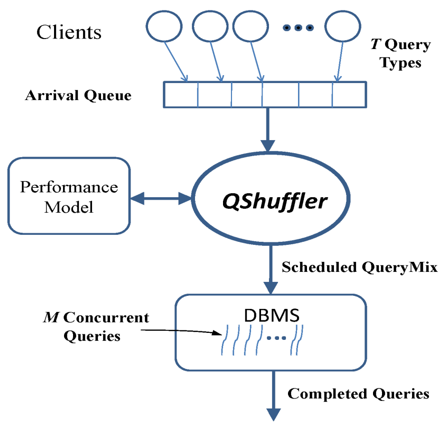


Figure 5.2: QShuffler architecture

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 query types $Q_1, Q_2, \ldots, Q_T$. Let I_j denote the total number of queries of type Q_j in W. Thus, $|W| = \sum_{j=1}^{T} I_j$. The clients place their batch of queries in an arrival queue, and QShuffler schedules queries from this 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 5.2 illustrates the solution architecture.

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, the objective of minimizing total completion time corresponds to producing all the reports requested in a certain period as fast as possible to stay within the available time budget.

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. As discussed in Section 4.1, for an MPL M and number of query types, T, the space of possible mixes is a bounded T-dimensional space, and 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 $S(T, M) = \binom{M+T-1}{M}$ [86].

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 $S(T, \frac{M}{b}) = \binom{\frac{M}{b}+T-1}{\frac{M}{b}}$. This restriction significantly reduces the number of possible mixes that need to be considered, and we can now enumerate all mixes in this subspace. This strategy for enumerating possible mixes can generate mixes with up to M instances of one query type, Q_j. If there is a query type Q_j with fewer than M instances (i.e., $I_j < M$), we prune mixes with more than I_j instances of Q_j from the search space X.

Linear program to pick a subset of X: QShuffler uses a *linear program (LP)* [89] 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 $m_i \in X$. The variable 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:

$$\sum_{i=1}^{|X|} n_i \frac{N_{ij}}{A_{ij}} \geq I_j, \quad \forall j \in \{1, \ldots, T\} \tag{5.1}$$

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 4.2 shows how A_{ij} values can be estimated for each mix in X using our interaction-aware performance models.) 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 5.1.

Working with the constraints in Equation 5.1, 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 [33]. 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 [89]. Recall from Equation 5.1 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 will 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. 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

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 runtime (sec)	0.01	0.06	0.13	0.33	0.95	5.56	179

Table 5.1: Runtime of LP in CPLEX for different values of b with $T = 6$, $M = 60$

need to transform the preemptive schedule generated by the LP to an efficient preemption-free schedule.

Obtaining a Preemption-free Schedule: We present a technique to produce a preemption-free schedule from the LP solution. 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 $n_1, n_2, \ldots, n_T$. The approach described here would still work if fewer than T mixes have nonzero n_i. We partition the total number of instances I_j of query type Q_j among mixes $m_1, m_2, \ldots, m_T$ 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. While this technique does not need query preemption, the generated schedule does not have a provable bound on total completion time. In our implementation of QShuffler, we use this more robust approach to produce non-preemptive schedules from the LP solution.

Scalability of Linear Programming: Modern LP solvers are quite scalable, and can handle large problem sizes with reasonable efficiency. For example, the CPLEX solver that we use can handle a very large number of mixes in the set X in reasonable time. To illustrate the scalability of CPLEX for our problem, Table 5.1 shows the runtime of the CPLEX 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 $320K$ 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.

Next, we present QShuffler's on-line scheduling algorithm, which is suitable for the scenario in which clients submit queries in small batches or one at a time.

5.3 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 5.2). The on-line 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 the 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_i runs on the database system, a *cost* is incurred based on the characteristics of m_i. 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 that these conventional resource-based 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 suboptimal 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* [29].

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, for capturing the cost incurred by a query mix executing in the database system.

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 $i \leftarrow 1$ 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           $\hat{R}_p \leftarrow$ Cost of $m_p$ estimated using performance model;
8           Priority $P_i \leftarrow 1/|\theta_R - \hat{R}_p|$;
9           $r[i] \leftarrow P_i$; ▷ Array $r$ stores the 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 $i \leftarrow 1$ 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 $Q_j$;
```

2. A *performance model* to compute $\hat{R}_i$, which is the estimated value of the cost metric R 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 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, a corresponding cost threshold θ_R, as well as a performance model for estimating R (i.e., computing $\hat{R}_i$) 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, and 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, QShuffler uses the *NRO* metric described in Section 5.3.1. The cost threshold is θ_{NRO}, and Section 5.3.2 explains how to set this threshold. Finally, QShuffler employs statistical models to compute $N\hat{R}O_i$, the estimated value of *NRO* for a given mix m_i, as we have discussed in Section 4.2.

5.3.1 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 completion 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 runtime 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 Runtime Overhead (NRO).* The NRO metric is a measure of the runtime 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 3.2: t_j denotes the average runtime of a query of type Q_j when it runs alone in the system, and A_{ij} denotes its average runtime when run in the query mix m_i. We define the runtime 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 runtime 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 runtime 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 runtime overhead for mix m_i is:

$$\begin{aligned} RO_i &= \frac{N_{i1} \times \frac{A_{i1}}{t_1} + N_{i2} \times \frac{A_{i2}}{t_2} + \cdots + N_{iT} \times \frac{A_{iT}}{t_T}}{N_{i1} + N_{i2} + \cdots + N_{iT}} \\ &= \frac{1}{M} \sum_{j=1}^{T} \left(N_{ij} \times \frac{A_{ij}}{t_j} \right) \end{aligned}$$

The value of RO_i represents the total runtime 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 $\frac{\text{mix\_runtime}}{\sum N_{ij} \times t_j}$ that are based directly on the total runtime of the mix are less robust: the effect of a single query that suffers a large increase in runtime 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 runtime.

- Finally, *NRO* values correlate well with our intuitive separation of good mixes from bad ones.

Next, we use example mixes of TPC-H queries to illustrate the effectiveness 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 5.2 shows several mixes of the 6 longest-running query types on a TPC-H 1GB database on IBM DB2. The individual runtimes, t_j, for each of these 6 query types are shown in Table 3.1. For each mix, Table 5.2 shows the query frequencies, N_{ij}, and the average runtime 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 $M = 30$.

The first two mixes in Table 5.2, m_1 and m_2, are simple mixes consisting of multiple instances of one query type running concurrently. We see that *NRO* is small for m_1, which suggests that Q_1 queries do not interfere with each other. (Recall that lower values of *NRO* are better.) The 30 Q_1 queries finish in 163.7 seconds in m_1, 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 3.1. Thus, m_1's low *NRO* value matches the fact that m_1 is a good mix.

On the other hand, m_2 has a much higher *NRO* than m_1, which suggests that m_2 is a worse mix. Indeed, the 30 Q_7 queries take 331.8 seconds to finish in m_2, compared to just $30 \times 5.76 = 172.8$ seconds for running these 30 instances sequentially (5.76 seconds obtained from Table 3.1). Also, notice that the Q_7 queries in m_2 take much longer to run than the Q_1 queries in m_2 despite the fact that Q_1 is almost 2x slower than Q_7 when run alone in the system (see Table 3.1). The *NRO* cost metric captures this effect appropriately

	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_1	30	163.7	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0.542	6.6	99.9
m_2	0	0.0	30	331.8	0	0.0	0	0.0	0	0.0	0	0.0	1.920	113.6	69.6
m_3	11	143.9	8	144.6	3	211.2	2	97.8	2	149.8	4	127.5	0.630	54.5	95.8
m_4	2	361.7	8	298.6	1	476.0	18	121.2	0	0.0	1	231.2	1.026	78.3	80.9
m_5	0	0	0	0	2	463.5	25	135.0	2	304.3	1	385.4	0.873	91.6	75.7
m_6	0	0	1	206.6	20	184.9	9	113.4	0	0	0	0	0.651	98.0	76.9

Table 5.2: Values of different cost metrics for selected query mixes running on a TPC-H 1GB database.

because *NRO* is not biased towards long-running queries. Table 5.2 also shows that the disk consumption of m_1 is lower than m_2, while its CPU utilization is higher than that of m_2.

Mixes m_3 and m_4 further illustrate how, unlike resource-based cost metrics, *NRO* can differentiate good mixes from bad ones. *NRO* tells us that m_4 is costlier than m_3, 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_3 and avoid mixes like m_4. As before, the resource consumption metrics are not useful for distinguishing between the performance of these two mixes. Mix m_4 places a higher load on disk than m_3, but m_3 places a higher load on CPU than m_4. Furthermore, the resource consumption levels of these two mixes are lower than the individual highs in Table 5.2.

The final two mixes, m_5 and m_6, clearly demonstrate the inadequacy of the given resource-based cost metrics. These two mixes are quite similar to each other if we consider resource-based metrics alone. On the other hand, *NRO* tells us that m_5 is costlier than m_6. 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_5 is worse than in m_6.

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 bad ones, while *NRO* can distinguish good mixes from bad ones.

5.3.2 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 may be underutilized if only light queries are scheduled, 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 administrator similar to what is required by popular tools like physical design advisors [9, 113]. 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 5.2 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, …, 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 runtime of the respective mixes m_1, m_2, …, m_T in S_R. Recall from Section 5.2 that the LP which computes S_R also gives the time for which each mix will run in S_R.

- $N\hat{R}O_1$, $N\hat{R}O_2$, …, $N\hat{R}O_T$, 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:

$$\theta_{NRO} = \frac{n_1 \times N\hat{R}O_1 + n_2 \times N\hat{R}O_2 + \ldots + n_T \times N\hat{R}O_T}{n_1 + n_2 + \ldots + n_T}$$

Intuitively, this approach aims to set θ_{NRO} such that the schedule generated by the online 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.

5.4 Scheduling Based on Query Optimizer Cost Estimates

In this dissertation 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 have proposed capturing the impact of query interactions by experimentally 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 our implementation of a query scheduler that uses query optimizer cost estimates, and we experimentally compare against this scheduler in Section 5.5.

Our optimizer cost based query scheduler is based on the work of Niu et al. [75, 76, 77, 78] (in particular, the query scheduler described in detail in [76, 77]). 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 determined by running an experiment 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 work in [75, 76, 77, 78] includes a general framework for workload adaptation that can handle scheduling for time-varying workloads. In addition, the scheduler described in these works can also handle different service classes for different query types, and it can perform admission control separately for the different service classes by defining a separate timeron threshold for each service class. 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.

In this dissertation, 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 "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*.

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.

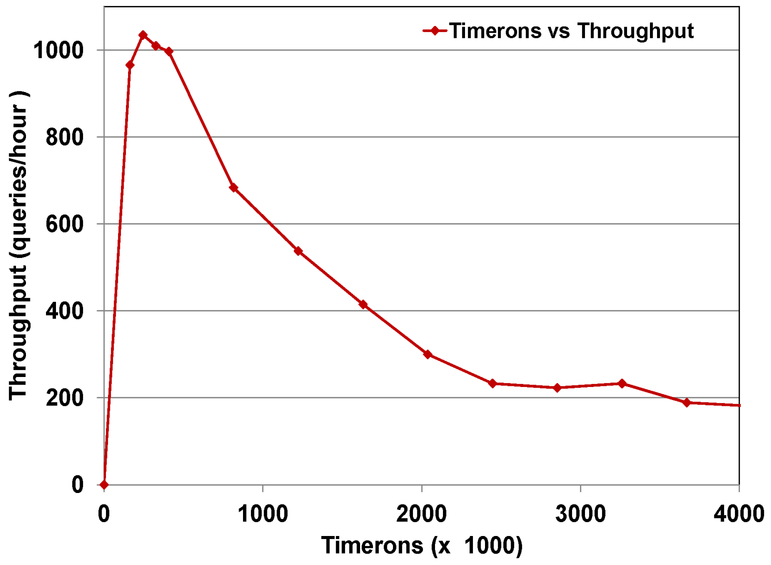


Figure 5.3: Throughput vs timerons for Q_{13} in the 1GB database

Consider Q_{13} and Q_{21} on the 1GB TPC-H 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 5.3 and 5.4 show throughput (measured based on an actual experiment) 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.

Notwithstanding the above, we still want to evaluate the effectiveness of optimizer-based scheduling. We therefore need to find the system saturation point. To find the system

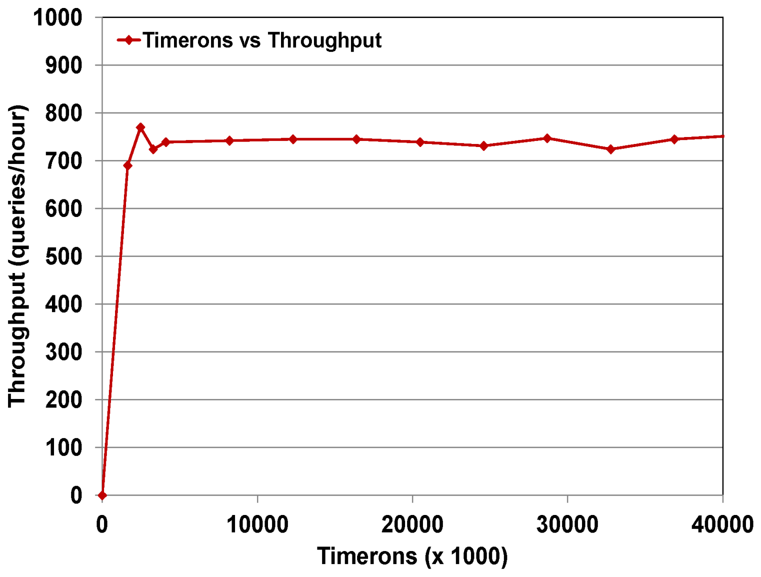


Figure 5.4: Throughput vs timerons for Q_{21} in the 1GB database

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 5.5). 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.

5.5 Experimental Evaluation

5.5.1 Experimental Setup

Our machine setup is the same as the setup described in Section 3.2. Other details are as follows.

Query Types: To study the different aspects of our algorithms, we use the 12 longest running TPC-H query types on the 1GB database, shown in Table 3.1, with different parameter values for each instantiation. The parameter values are chosen according to the TPC-H rules. These 12 query types represent our base case settings. When we use other settings to study the sensitivity and robustness of our approach, the details are specified for the corresponding experiments.

Arrival order and workloads: As we demonstrate in our motivating example in Chapter 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 test QShuffler under various settings, 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 $Q_3, \ldots$, up to 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 B randomly generated instances of each query type in the arrival queue until all queries are in the queue. The parameter B specifies the degree of skew in the arrival order of the workload. As B increases, more queries of the same type arrive together. For the 1GB database, we use $IQ = 10$ and $B = 5$, 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 $B = 2$, 5, and 10. We use 10 instances of each query type. We limit the workload sizes for the 10GB database due to the long runtimes 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 the QShuffler batch scheduling algorithm, which requires the entire workload to be known in advance. The other four algorithms do not require the full workload to be known in advance, and we assume that the scheduler can see only the next L queries in the arrival queue. The value L is the lookahead value discussed earlier in the chapter. The algorithms are: first come first serve (FCFS), which is insensitive to L; shortest job first (SJF), which schedules the shortest query available in the next L queries using the runtimes in Tables 3.1 and 3.2; our on-line scheduling algorithm; and the optimizer-based scheduler (Section 5.5.7).

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 QShuffler on-line algorithm, SJF, and FCFS. We measure the performance of each of these algorithm in terms of *slowdown compared to batch schedule*, defined as:

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

5.5.2 Choice of Sampling and Modeling Technique

In Chapter 4 we identified two sampling and model training choices: Gaussian processes models trained on samples collected by the IL-aware LHS algorithm (*GP with IL_LHS*) and

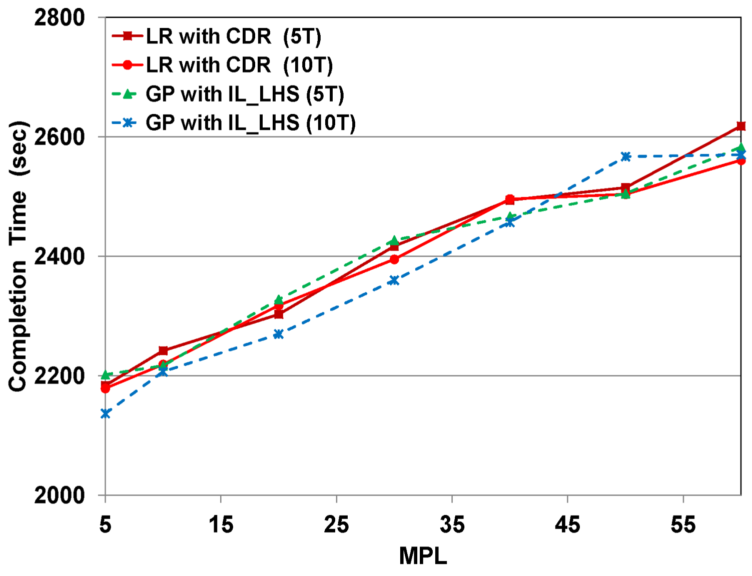


Figure 5.5: Batch schedule execution time for different modeling approaches

linear regression models trained on samples collected by CDR sampling (*LR with CDR*). We first discuss some observations on the performance of our scheduling algorithms with both these approaches. Then we discuss the results of our scheduling algorithms in detail.

Figure 5.5 shows the total completion time of the schedules chosen by our batch scheduling algorithm for different MPLs for a workload consisting of the longest running 12 query types on the 1GB database. For each sampling and modeling approach we consider models built based on $5T$ and $10T$ samples. The figure shows that for $5T$ samples both approaches result in very similar performance. For LR with CDR we do not see much improvement in performance when the number of samples is increased from $5T$ to $10T$. In case of GP with IL_LHS we do observe that there is an improvement in performance as we collect more samples. However, it should be noted that the difference between the best and the worst performance for different choices of the modeling approach and the number of samples is not more than 2–3%.

Next, we turn our attention to the performance of the QShuffler on-line algorithm. We still consider the case when $T = 12$, $M = 30$, and the database is 1GB. As mentioned above we use the batch scheduling algorithm to evaluate the performance of on-line algorithms. Figure 5.6 shows the performance of the QShuffler on-line scheduling algorithm. Once again we show the performance of the on-line algorithms for both model training approaches, and for each approach we consider $5T$ and $10T$ samples. In the figure the batch algorithm

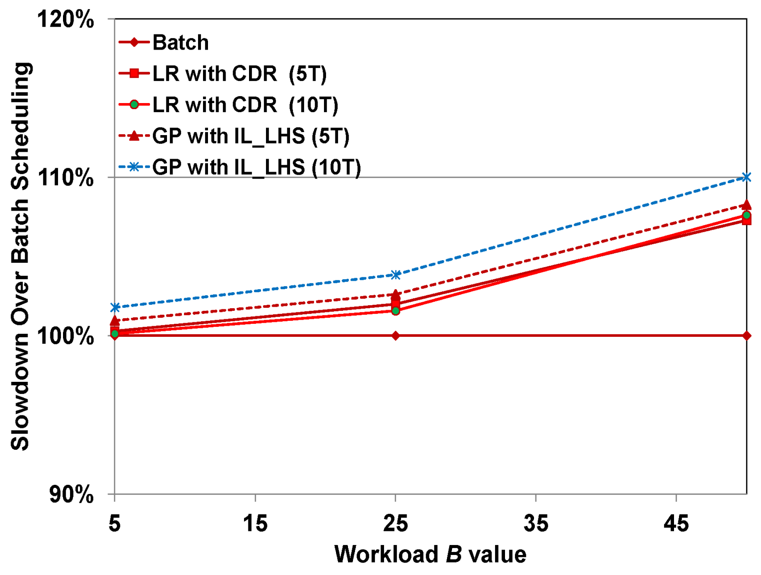


Figure 5.6: QShuffler on-line algorithm for different modeling approaches

uses LR with CDR with $5T$ samples for modeling. The figure shows that we get very similar performance with LR with CDR for both the $5T$ and $10T$ cases. In the case of GP with IL_LHS we observe that its performance is worse than the performance achieved by employing LR with CDR. The reason for this is the high computational and memory requirements of the GP algorithms. This is not a problem when using GP in an off-line manner. However, in the on-line scheduling algorithm, the time to consult the GP model is on the critical path of workload execution, so the overhead of the model has a direct effect on performance. This overhead would be acceptable if the GP model results in significantly better scheduling decisions, but it does not.

One may expect to get significantly better scheduling performance by employing more accurate and sophisticated modeling and sampling techniques. However, these experimental results demonstrate an important point about the scheduling problem: the scheduling algorithm does not need models that provide high accuracy in the absolute sense. As long as the modeling accuracy is good enough to distinguish good mixes from bad ones, the models are good enough for the scheduling algorithms. There are other situations where the models need to be as accurate as possible. For example, in Chapter 6 when we study predicting the completion time of BI workloads, we will see that we need accurate models and hence we will use GP with IL_LHS. However, in this chapter the extra complexity of GP with IL_LHS is not justified. Therefore, we use LR with CDR modeling with $5T$ samples for the remainder of this chapter.

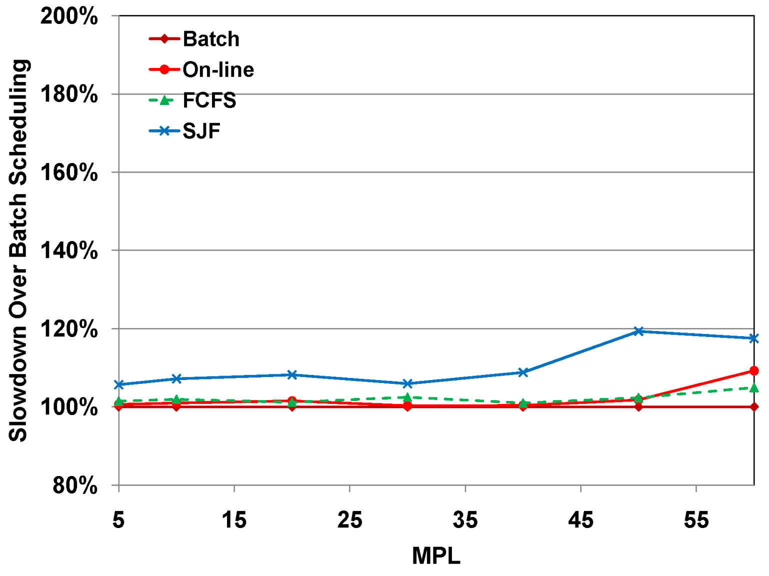


Figure 5.7: Scheduling for $B = 5$

5.5.3 Scheduler Effectiveness

Figures 5.7, 5.8, and 5.9 show the performance for different MPLs of our on-line scheduling algorithm, FCFS, and SJF for $B = 5$, 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 lookahead is $L = 60$. The methodology in Section 5.3.2 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 B increases. FCFS is the scheduling algorithm used by all database systems that we are aware of, and these experiments show that varying 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. At $B = 5$, 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 B increases, and the arrival order starts

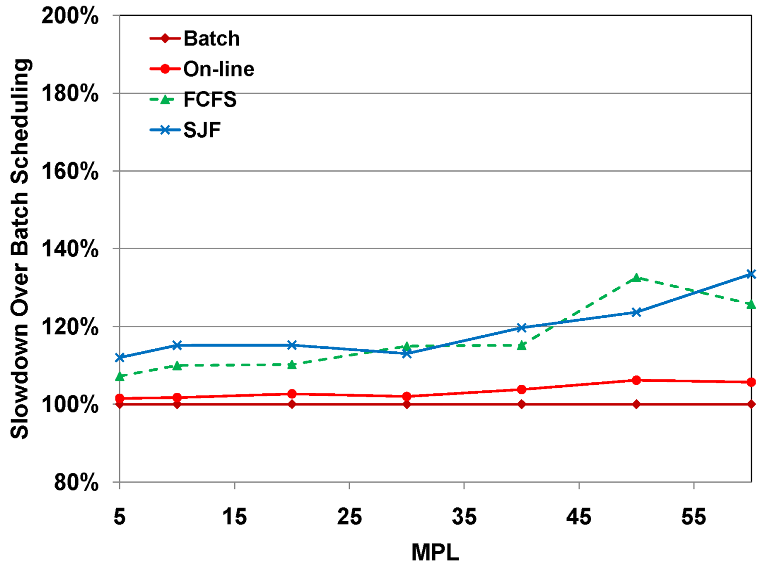


Figure 5.8: Scheduling for $B = 25$

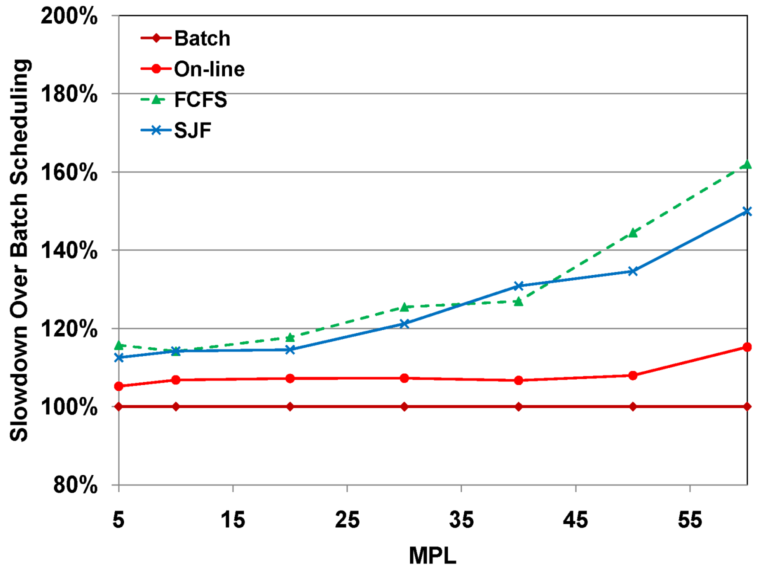


Figure 5.9: Scheduling for $B = 50$

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 $B = 50$ 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.

5.5.4 Sensitivity to θ_{NRO}

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 high value of the *NRO* metric 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 5.10 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.

5.5.5 Scalability and Robustness of the On-line Algorithm

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 queries in the TPC-H benchmark as shown in Table 3.1. The workload consists of 60 queries of each type, for a total of 1220 queries.

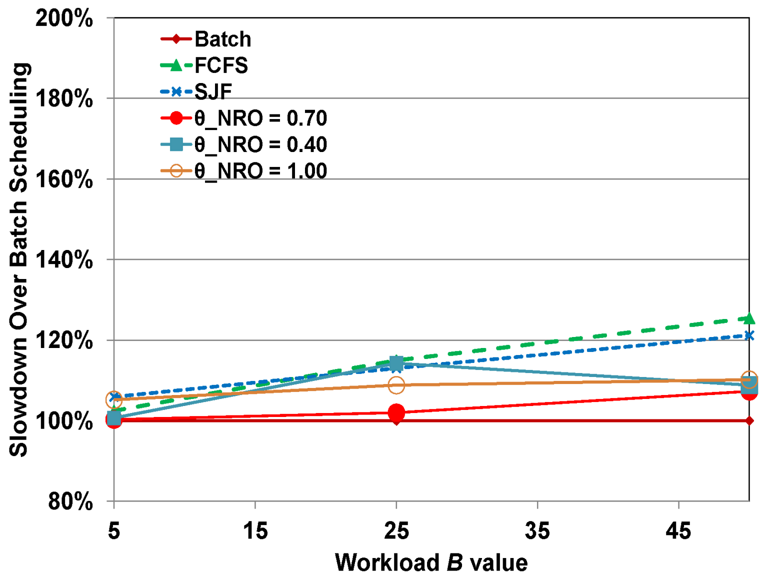


Figure 5.10: Scheduling for different values of θ_{NRO} ($M = 30$)

Figure 5.11 shows the performance of the different scheduling algorithms for this workload for MPL 30 and varying B. The figure shows that, as in previous experiments, the on-line scheduling algorithm performs better than FCFS and SJF.

To test QShuffler for larger database sizes, we use the 10GB TPC-H database. Since the hardware is unchanged from the 1GB case, the queries place a much higher load on the system and have much higher runtimes in the 10GB case. Therefore, we experiment with only the 6 longest running query types from Table 3.2. The workload consists of 10 queries of each type for a total of 60 queries, the lookahead is set to $L = 10$, and MPL is set to 10. The value of θ_{NRO} was computed based on this workload to be 0.33. The arrival order of the queries is determined based on the parameter B, and we use $B = 2, 5$, and 10, since we have only 10 queries of each type.

Figure 5.12 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 $B = 10$. 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 in the 10GB case. This results in SJF and FCFS both improving

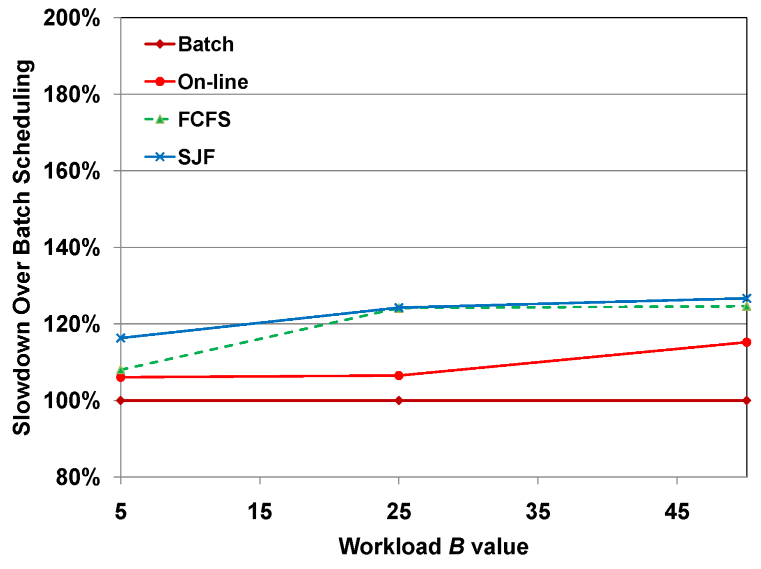


Figure 5.11: Scheduling for $T = 21$

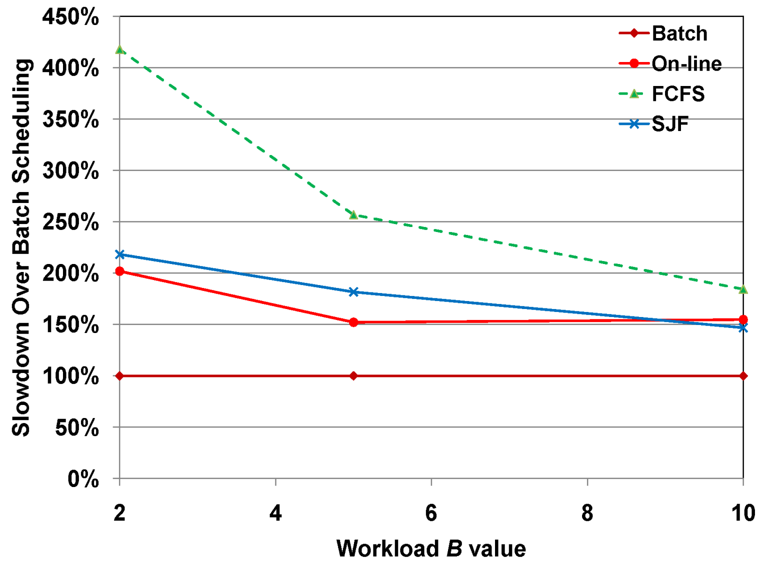


Figure 5.12: Scheduling for 10GB database

as the arrival patterns become more skewed, since both algorithms would schedule mixes consisting of queries of the same type when this happens.

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 B tend to present this kind of mixes, so there is more opportunity for performance improvement over FCFS at higher values of B. 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 B. Our scheduler is able to take advantage of the scheduling opportunities in both these cases.

5.5.6 Scheduling for Skewed Data Distribution

In this section we present the results for robustness of our scheduler in case of databases with skewed data distributions. Our approach for dealing with skewed data distributions was presented in Section 3.5. For this experiment we use the 9 query templates on a 1GB skewed database shown in Table 3.10. Figure 5.13 shows our scheduling results for workloads consisting of query instances (with different parameter values) of these query templates.

The figure shows three workloads with $B = 5$, 25, and 50, and $M = 30$. The figure shows that QShuffler is consistently better than FCFS and SJF for the different workloads for all values of B. For the sake of comparison, in Figure 5.14 we also show the case where we continue to group all instances of the Q_9 query template together into one query type, i.e., $K = 1$. The results shows that our approach for handling skew improves the performance of both batch and on-line scheduling algorithms in Figure 5.13. FCFS behaves the same whether $K = 1$ or $K = 4$. In Figure 5.13 we can see that the gap between FCFS and the QShuffler algorithms increases for all workloads. For example, for the workload with $B = 25$, the slowdown over batch scheduling for FCFS is 140% in Figure 5.14, and it is 145% in Figure 5.13. 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 5.14.

5.5.7 Comparison to Optimizer-based Scheduler

In this section, we compare the performance of QShuffler against the optimizer-based scheduler presented in Section 5.4. For this comparison, we use the 10GB database and

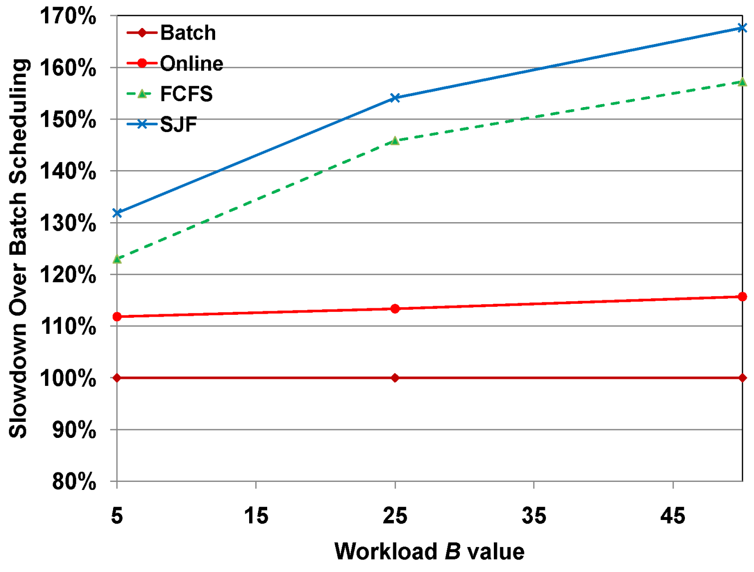


Figure 5.13: Scheduling for skewed data with $K = 4$ ($M = 30$)

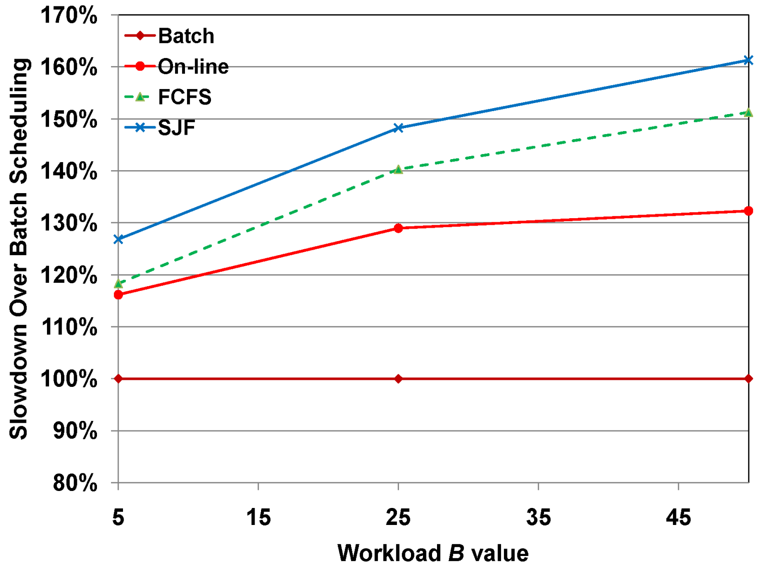


Figure 5.14: Scheduling for skewed data with $K = 1$ ($M = 30$)

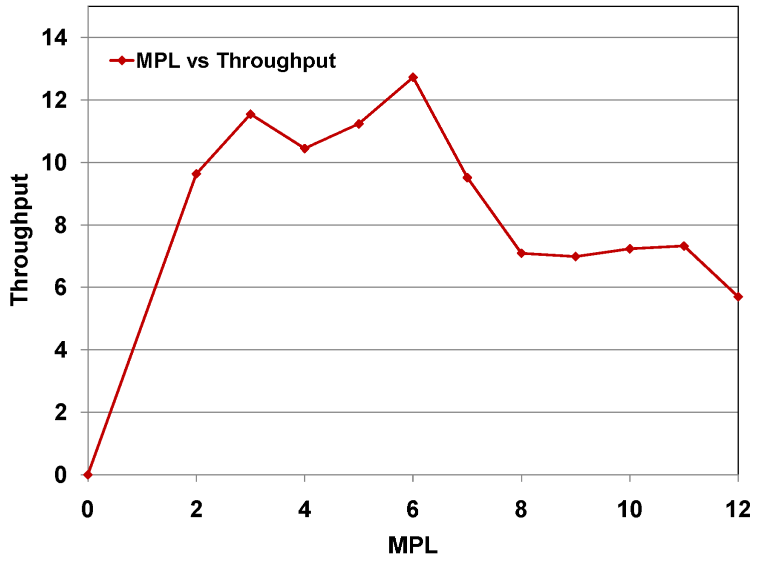


Figure 5.15: Throughput vs MPL for W_R

the three workloads used in Figure 5.12 (60 queries with different B values). The first step in using the optimizer-based scheduler is to determine the timeron threshold as outlined in Section 5.4. For this, we construct a workload consisting of the same 60 queries used in Figure 5.12, with the arrival order randomly shuffled. We call this workload W_R. Figure 5.15 shows the throughput vs MPL graph for W_R. The best throughput is obtained at MPL $M = 6$, and this corresponds to a timeron threshold $Thr_{time-weighted} = 17, 3353, 73.87$ timerons.

Figure 5.16 shows the performance of the optimizer-based scheduler using this timeron threshold for W_R and the three workloads in Figure 5.12. 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 5.16 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 $M = 10$ (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.

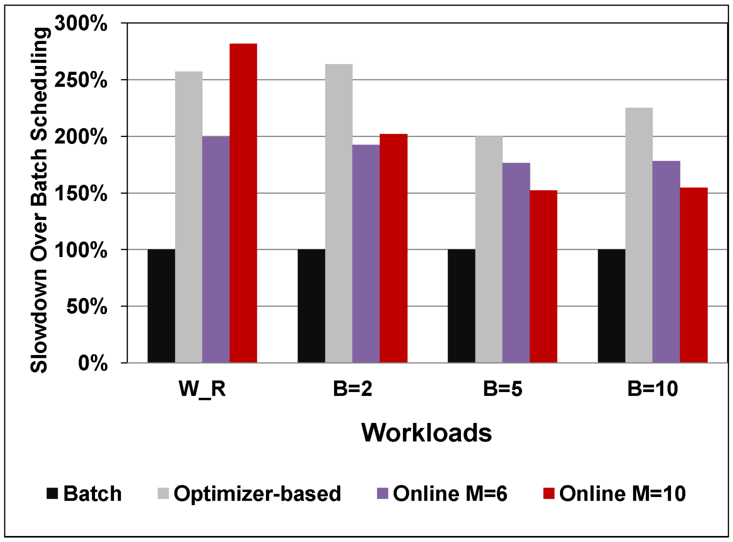


Figure 5.16: Optimizer-based scheduling

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

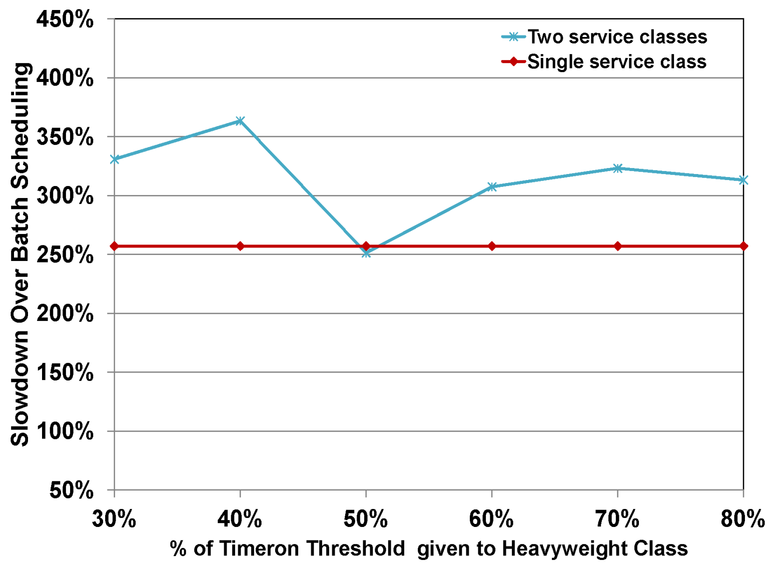


Figure 5.17: Optimizer-based scheduling of workload W_R using two service classes

the best chance of finding a good schedule.

Figure 5.17 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 5.16. 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 experimental error). 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 5.4: (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.

In summary, this chapter presented QShuffler, a throughput oriented scheduler for BI report-generation workloads. QShuffler's batch and on-line scheduling algorithms make interaction-aware scheduling decisions for the workloads. We experimentally validated the effectiveness of our scheduling approach using a BI benchmark on a real database system. We showed that QShuffler can provide significant improvement in performance over the default FCFS scheduler used by database systems , and over other scheduling algorithms such as SJF and scheduling based on query optimizer cost estimates. Next, in Chapter 6 we move away from scheduling and turn our attention to another workload management problem. We present interaction-aware techniques for predicting the completion time of a workload.

Chapter 6

Predicting the Completion Time of Business Intelligence Workloads

6.1 Introduction

In Chapter 5 we considered the problem of scheduling appropriate query mixes for a given query workload in order to minimize the workload's total completion time. In this chapter we consider another important workload management problem: *estimating the completion time of a given batch of queries.* The state of the art does not provide a database administrator (DBA) with any systematic tools that can predict the completion time of a given batch of BI queries. Typically database administrators rely on heuristics and their past experiences in this regard.

Figure 6.1 illustrates the problem setting for this question of estimating the completion time of a workload consisting of batch of queries. A database system has to process a batch of queries q_1, q_2, …, q_n. The multi-programming level of the system is set to M. Whenever a query q finishes among the M currently running queries, a new query q' from the batch will be scheduled in its place based on a given scheduling policy. Given the query batch, scheduling policy, and MPL, can we predict (ahead of time) how long the database system will take to process the entire batch of queries?

Automated tools to answer this question with good accuracy and efficiency can help in several workload management tasks:

- DBAs may need to plan the execution of different report-generation workloads to fit within available time windows.

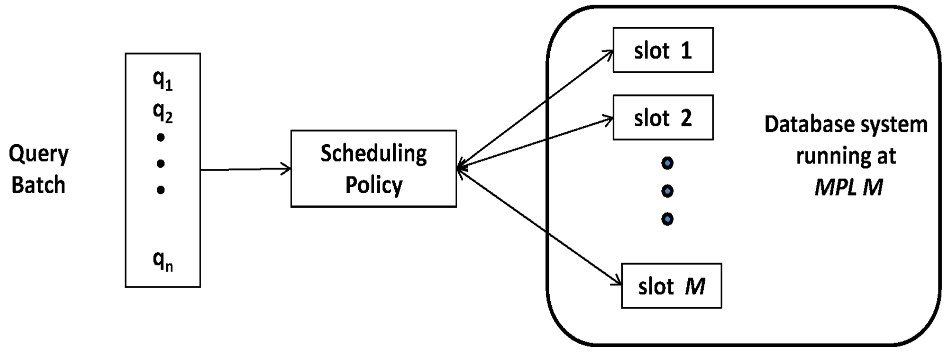


Figure 6.1: Problem setting for predicting workload completion times

- Accurate prediction can be used to give data analysts continuous feedback on the progress of running workloads.
- Such tools form *what-if modules* to determine which scheduling policy to use for a workload or the resources needed to complete a high-priority workload within a given deadline [37, 63].
- Estimating workload completion time can be used as a what-if module to partition a query workload across multiple database instances in a parallel database system.

As mentioned above, there are no research or industrial-strength automated tools for predicting batch query workload completion times in a general way. In this dissertation, we address this limitation and present an interaction-aware solution for predicting workload completion times [15, 16]. The defining feature of our solution is that it treats query batch workloads as a sequence of query mixes that execute, while accounting for the query interactions that arise in these mixes.

6.2 Anatomy of an Interaction-aware Predictor

We begin with an overview of our interaction-aware predictor of batch workload completion times. The predictor comprises a *simulator* that can simulate the execution of query mixes in a given database system. The simulator performs this task using interaction-aware performance models that can estimate the running time of queries executing with other queries in a mix.

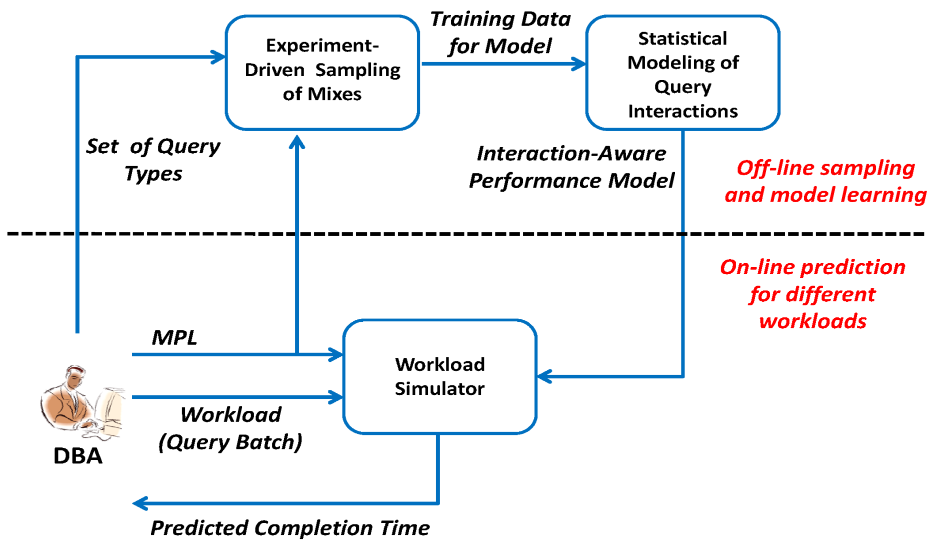

Experiment-Driven Sampling of Mixes
Training Data for Model
Statistical Modeling of Query Interactions
Set of Query Types
Interaction-Aware Performance Model
Off-line sampling and model learning
On-line prediction for different workloads
MPL
Workload Simulator
Workload (Query Batch)
DBA
Predicted Completion Time

Figure 6.2: Workflow of predicting workload completion times

Figure 6.2 shows the overall workflow of the predictor which consists of a predominantly off-line learning component and an on-line prediction component. The workflow is invoked by a database administrator when she identifies a context where batch workloads are executed repeatedly, and predictions of workload completion times can be useful.

Sampling and Modeling (Off-line Phase): We have explained our approach for sampling and modeling query mixes in Chapter 4. The database administrator uses the query types T and multi-programming level M to generate the set of samples and learn a model from these samples as discussed in Chapter 4.

Simulating Workload Execution (On-line Phase): The simulator uses a recurrence relation in conjunction with interaction-aware performance models to simulate the execution of the workload as a sequence of query mixes. This approach overcomes a major disadvantage of conventional analytical simulation where domain experts have to spend many hours implementing the simulator, only for it to become inaccurate when database internals are modified. Few works on database tuning have harnessed the power and flexibility of simulation (a notable exception is [81]). Another noteworthy feature of our simulator is that it incorporates the simulation of the scheduling policy as a pluggable component. While we used this feature to support two common scheduling policies, First Come First Serve (FCFS) and Shortest Job First (SJF), other scheduling policies can be supported if needed.

6.3 Workload Simulator

In this section, we present the workload simulator that estimates the completion time of a given workload. The simulator is given the following input:

- A workload, W, that can come from one or more clients. The workload consists of a batch of queries belonging to the T possible query types. The workload is a batch workload that is known in advance, and all the $|W|$ queries are placed in the arrival queue of the simulator.
- The MPL of the system, M.
- A scheduling policy for determining which queries to schedule next. By default, the simulator uses FCFS, but it can also use other policies such as SJF.

We consider the execution of the workload as a sequence of mixes of M queries each, as shown in Figure 6.3. These mixes, which we call *workload phases*, change when one query finishes and another query starts. The goal of the simulator is to simulate the exection of the workload phases and the transition between them, and to estimate how long each phase will take. The predicted workload completion time is the total time taken by all phases.

We start with an initial mix $\langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$ consisting of the first M queries scheduled from the arrival queue. This is workload phase 1. When one of these M queries finishes and another one starts, we transition to a new mix and a new workload phase. The transition between phases continues until all queries are executed, for a total of $|W| - M + 1$ phases.

When we transition from one phase to the next, the currently executing query mix changes, and this results in different query interactions coming into play. The newly starting query will affect the performance of the currently executing queries and its performance will be affected by these queries. Thus, the simulator needs to track the performance of the executing queries in the different mixes. To do this, the simulator uses an interaction-aware performance model. The performance model estimates, for any mix $\langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle$, the average completion time of the different query types in this mix, denoted by $\langle \hat{A}_{i1}, \hat{A}_{i2}, \ldots, \hat{A}_{iT} \rangle$.

The simulator tracks the *fraction of total work completed* by each query in each phase. Consider a query instance, q_j, of type Q_j. This query instance will start with the start of some workload phase, and this workload phase would be *query phase 1* for this query

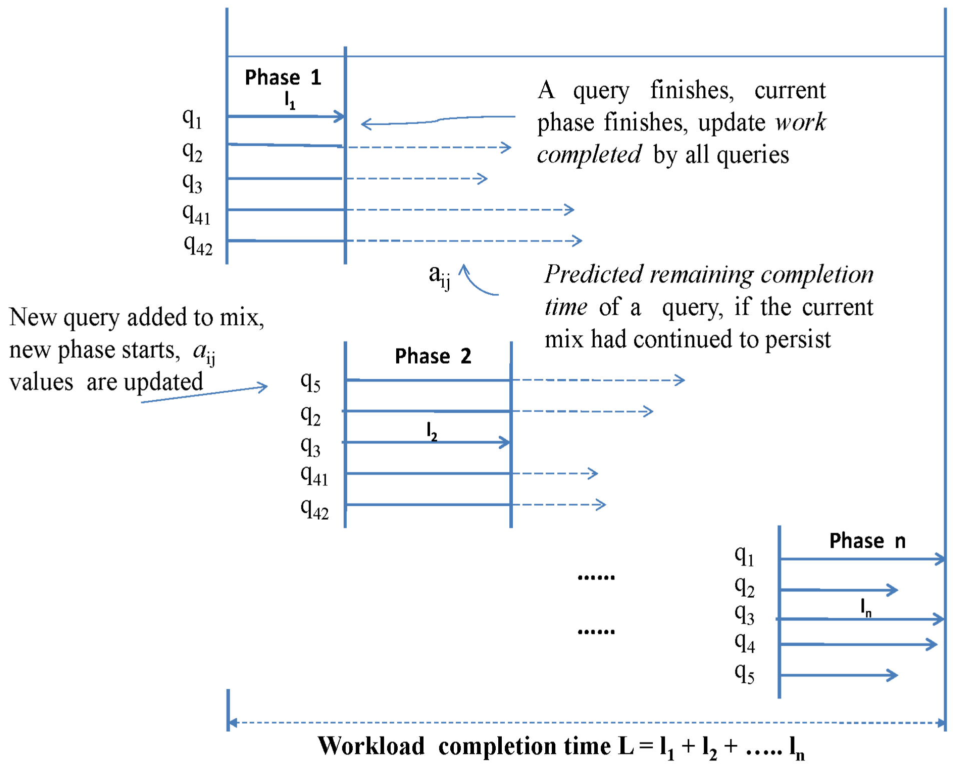

Phase 1
l1
q1
q2
q3
q41
q42
A query finishes, current phase finishes, update work completed by all queries
aij
Predicted remaining completion time of a query, if the current mix had continued to persist
New query added to mix, new phase starts, aij values are updated
Phase 2
q5
q2
q3
q41
q42
l2
......
......
Phase n
q1
q2
q3
q4
q5
ln
Workload completion time L = l1 + l2 + ... ln

Figure 6.3: Simulating the execution of a workload as a sequence of mixes

instance. The query will execute through different workload phases until it completes all the work it needs to perform. Let wc_{ij} be the fraction of q_j's work completed in its query phases 1 to i. When q_j starts, its $wc_{ij} = 0$, and q_j is done when its $wc_{ij} = 1$. We define the following recurrence relation to keep track of wc_{ij} through the different query phases:

$$\begin{aligned} wc_{0j} &= 0 \\ wc_{ij} &= wc_{(i-1)j} + (1 - wc_{(i-1)j}) * f_{ij} \end{aligned} \tag{6.1}$$

The fraction of q_j's work completed up to query phase $i-1$ is $wc_{(i-1)j}$, and the remaining work after phase $i-1$ is $(1-wc_{(i-1)j})$. The fraction of this remaining work that is completed during query phase i is f_{ij}, defined as:

$$f_{ij} = \frac{l_i}{a_{ij}}$$

where l_i is the length of phase i and a_{ij} is the *predicted remaining completion time* of query instance q_j when it executes in the query mix of phase i. If q_j continues executing in this mix, it would finish in time a_{ij}. Since phase i will end in time l_i, q_j will only complete l_i/a_{ij} of its remaining work in this phase.

To estimate a_{ij}, the simulator uses the performance model to obtain the estimated completion time of q_j in the mix of phase i, $\hat{A}_{ij}$. This is time required for q_j to execute from start to finish in this mix. Since q_j has already completed $wc_{(i-1)j}$ of its work, the simulator multiplies the estimate $\hat{A}_{ij}$ by $(1 - wc_{(i-1)j})$. The performance model returns one estimate of completion time for each query type. Thus, if the simulator starts two instances of a given query type at the same time with the start of some workload phase, they will also finish at the same time. In practice, the two instances are unlikely to finish at the same time since there is some variance in completion time for query instances of the same query type (e.g., due to internal scheduling at various resources). To model this variance in the simulator, we perturb the estimate of a_{ij} by a small random amount ϵ between -0.1% and $+0.1\%$ of the estimate $\hat{A}_{ij}$. This leads to the following formula for estimating a_{ij}:

$$a_{ij} = (1 + \epsilon)(1 - wc_{(i-1)j}) * \hat{A}_{ij} \tag{6.2}$$

To estimate l_i, the length of phase i, we observe that phase i will continue until one of the running queries finishes, at which point the simulator will transition to phase $i + 1$. Thus, phase i will end at the earliest time a query finishes. That is,

$$l_i = \min_{j=1\ldots M}(a_{ij}) \tag{6.3}$$

At the end of phase i, the simulator needs to update the state of the simulation to reflect the end of phase i and the start of phase $i + 1$.

End of phase i: To finish phase i, the simulator uses Equation 6.1 to update the work completed for all queries that are running in this phase.

After this update, a query that has $wc_{ij} = 1$ is finished and removed from the mix. The next query in the workload will take its place to start phase $i + 1$.

Start of phase $i + 1$: When phase $i + 1$ starts, the running query mix has changed, so the simulator needs to recompute the estimated remaining time $a_{(i+1)j}$ for all queries in the mix using Equation 6.2. After computing $a_{(i+1)j}$ for all M query instances in the mix, the simulator can estimate the length of phase $i + 1$ using Equation 6.3. The simulator then finishes phase $i + 1$ and starts phase $i + 2$, and this continues until all the workload queries are executed.

The simulator has a special case for handling the last phase of the workload (phase $(|W| - M + 1)$). At the start of this phase, there are M running queries, but as these queries finish, they will not be replaced by other queries. The simulator makes a simplifying assumption and estimates the length of this phase $l_{|W|-M+1}$ as the maximum a_{ij} of all queries in this phase, ignoring the change in query mix as queries finish. This simplifying assumption enables the simulator to use the same performance model (with the same MPL) for all workload phases without affecting prediction accuracy.

The simulator estimates the completion time for the whole workload, L_W, as the total length of all the workload phases:

$$L_W = \sum_{i=1}^{|W|-M+1} l_i$$

6.4 Experimental Evaluation

6.4.1 Experimental Setup

Our machine and workload setup is same setup that we used before, described in Section 3.2 and Section 5.5. A workload run in our experimental setting can be defined by the following 5 parameters: (1) database size, (2) number of query types T, (3)arrival order skew parameter, B , (4) scheduling policy, and (5) MPL M.

Query Types: We consider all the 22 TPC-H query types except for Q_{15} (recall that Q_{15} creates a view and this is not supported by our prototype implementation). In addition to query batches that include all the T=21 query types, we also considered batches containing instances of only the top 12 and the top 6 longest-running query types (when queries run alone in the system) as shown in Tables 3.1 and 3.2. Thus, our batch workloads have $T = 6$, $T = 12$, or $T = 21$.

Scheduling policies: We consider the FCFS and SJF scheduling policies, as used in Chapter 5. The lookahead L for SJF is the same as used in experiments in Section 5.5 ($L = 60$ for 1GB and $L = 10$ for 10GB).

Arrival order skew: As in Chapter 5, we create different workloads by varying the parameters that control the arrival order skew: the initialization parameter IQ and the arrival order skew B. To construct a query batch, we first go through the given list of query types and add IQ instances of each type. Going through the list again in a round-robin fashion, we keep adding B instances of each query type to the batch until all the queries are added. For FCFS, as B increases, more queries of the same type are scheduled together.

Workloads: For the 1GB database, we use $IQ = 10$ and $B = 5$, 25, and 50. We use 60 instances of each query type. Thus, batches with $T = 21$ query types consist of 1260 query instances, batches with $T = 12$ query types consist of 720 query instances, and batches with $T = 6$ query types consist of 360 query instances. Query instances are generated by instantiating TPC-H query templates with different parameter values chosen as per TPC-H rules. For these workloads, we use MPL $M \in \{5, 10, 20, 30, 40, 50, 60\}$.

For the 10GB database, we use $IQ = 0$ and $B = 2$, 5, and 10. We use $T = 6$ and 10 instances of each query type. When running these workloads, we use MPL $M \in \{5, 10\}$. We limit the workload sizes and MPL for the 10GB database due to the long runtimes of queries on this database. Recall from Figure 1.1 that 60-query workloads on the 10GB database can take more than 5 hours to complete.

Evaluation Methodology: By systematically varying the choice of query types, size of query batch, and scheduling policy, the parameter B, and MPL M, we generated 142 distinct and varied workloads. We run these workloads with a warm buffer pool and measure their completion times. These workloads have actual completion times ranging from 30 minutes to more than 5 hours. For each workload W, we compare W's actual completion time *act* with W's completion time predicted by our technique, *pred*. Our error metric is the *relative prediction error*, which is defined as:

$$Rel = \frac{|pred - act|}{act} \times 100\%$$

6.4.2 Choice of Sampling and Modeling Technique

In Chapter 4 we settled on two sampling and model training choices: Gaussian processes models trained on samples collected by the IL-aware LHS algorithm (GP with IL_LHS) and linear regression models trained on samples collected by CDR sampling (LR with CDR). In Chapter 5, we found that LR with CDR was good enough to distinguish good mixes from bad ones for the purpose of scheduling. The problem of predicting the completion times of workloads, however, is more challenging than scheduling and it requires more accuracy in predicting query completion times. Therefore, our chosen approach is GP with IL_LHS. Nonetheless, we did experimentally evaluate LR with CDR as well, and the results are shown in Figure 6.4. The figure shows the cumulative frequency distribution of the relative prediction error in all the 142 workloads that we use in our experiments for both GP with IL_LHS and LR with CDR. (For the cumulative frequency plots that we show in this section, lines towards the top left of the graph indicate better prediction accuracy.) It can be seen that error is quite high for LR with CDR. The error is more than 100% for

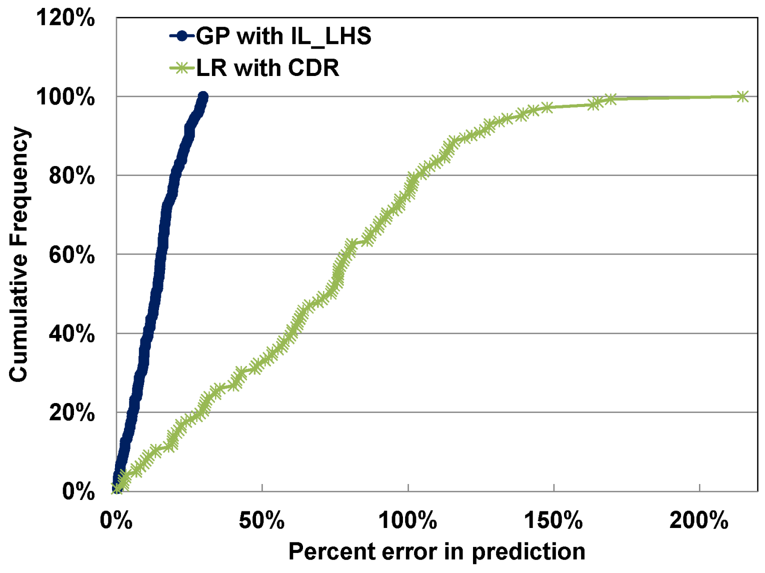


Figure 6.4: Prediction error for different sampling and modeling approaches

20% of the cases, and it reaches 200% for some cases. Overall about 50% of the time the prediction errors are more than 30%. On the other hand, the GP with IL_LHS approach show remarkable accuracy and this is the best choice for predicting workload completion times. Thus, for the remaining experiments we use GP with IL_LHS.

6.4.3 Overall Prediction Accuracy

Figure 6.5 shows two cases for the GP with IL_LHS approach. In one case, we use models trained on $10T$ samples and in the other case we use models trained on $5T$ samples. Thus, for T=21 TPC-H query types, in the case of $5T$ and $10T$, we sample no more than 105 and 210 query mixes, respectively, to train the models.

From Figure 6.5, we can see that in case of $5T$ samples about 70% of the time the prediction errors are less than 20%. If the DBA has a larger sampling budget and is willing to collect up to $10T$ samples, then the overall accuracy improves further to the point where about 80% of the time the prediction errors are less than 20% . These end-to-end results show that our sampling, modeling, and workload simulation algorithms result in accurate and robust predictions across a wide range of workloads. The DBA can now make highly accurate predictions for the future workloads in her database by collecting a small number of samples just once (which can be done along with initial system setup and tuning).

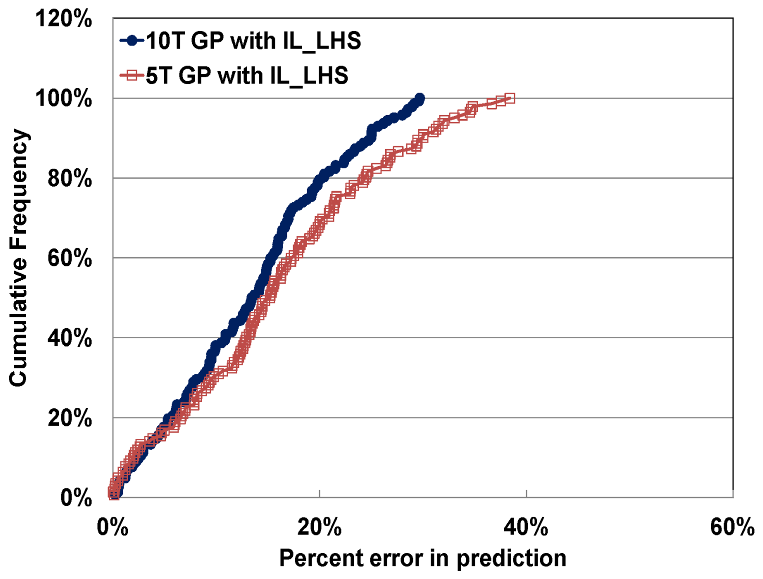


Figure 6.5: Prediction error for $5T$ and $10T$ samples

6.4.4 Incremental Sampling

Our empirical observations indicate that $5T$ training samples are good enough for most cases. Depending on the situation, a DBA may have a lower or larger sampling budget. She may not be sure, e.g., whether less than $5T$ training samples would be good enough or whether she should invest in the time to collect up to $10T$ samples. To address such dilemma the DBAs can use our incremental sampling algorithm presented in Section 4.1.2.

Figure 6.6 shows the results of our incremental sampling algorithm. The plots with the suffix "-inc" show how the prediction error drops as incremental sampling brings in $3T$, $5T$, and $7T$ samples over time. From the figure, we can see that even in case of $3T$ samples about 40% of the time the errors are less than 20%, and about 60% of the time the errors are less than 30%. These accuracies may be enough for the workloads that a DBA is seeing, and she needs to go no further. The effectiveness of the approach is clear in that this case needs only 63 samples for 21 TPC-H query types (as opposed to 210 samples for $10T$). With $5T$ incremental samples, we get errors lower than 30% almost 90% of the time. As more samples are collected incrementally, the figure shows that the results for $7T$ are very close to $10T$, so one may not need to go all the way to $10T$.

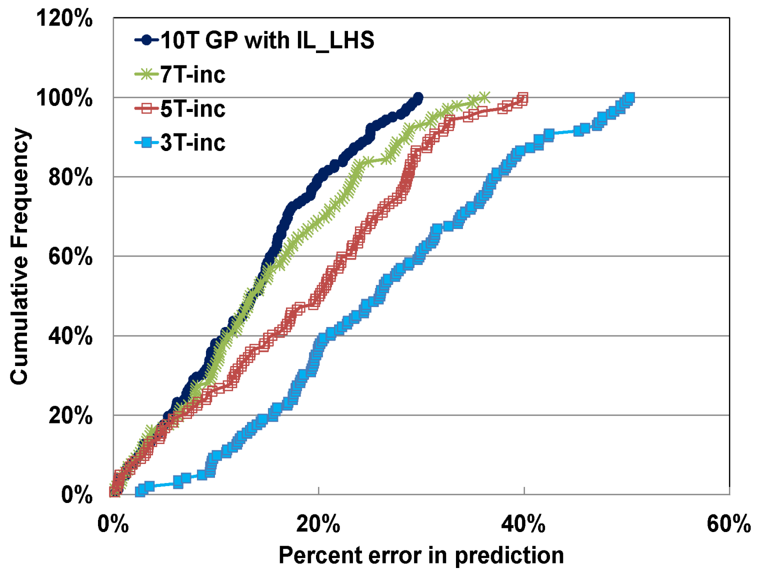


Figure 6.6: Prediction error for incremental sampling

6.4.5 Time Needed for Sampling and Prediction

The time required to obtain the predictions shown in the previous experiments consists of the off-line time to collect samples and build the model, and the on-line time used by the simulator. The running time of the simulator is minimal: for the largest workload consisting of 1260 queries, the simulator takes less than 10 seconds overall. For workloads with fewer queries, the running time of the simulator is typically a fraction of a second. Table 6.1 shows the training time for a representative subset of prediction scenarios. The first column is the prediction scenario identified by the DBA. The second column shows the absolute training time: the time needed by our incremental sampling algorithm to collect samples and learn a model that gives $\leq$20% error for $\geq$80% of the time. Absolute training times do not tell the full story. The third column shows relative training time: the ratio of the absolute training time to the average time to run a workload corresponding to that scenario in our experiments. Notice that the training time is equal to the time to run a very small number of workloads. Our solution offers three important advantages:

- Once the models are trained, they can be reused to give predictions for any number of workloads that match the prediction scenario.
- The relative training time drops as the number of query instances $|W|$ in the batch workload increases, since absolute training time is independent of $|W|$. Once we

Prediction scenario (given by DBA)	Training time	Ratio of training time to avg workload runtime
M=5, T=21, 1GB	0.6 hours	0.9 (1260-query workloads)
M=30, T=21, 1GB	4.3 hours	3.9 (1260-query workloads)
M=10, T=6, 10GB	15 hours	4 (60-query workloads)

Table 6.1: Training time of representative prediction scenarios

train the models, we can give predictions for workloads of any size. In practice, batch workloads can be large in size. For the 10GB case, a typical 60-query workload can take more than 5 hours to run, and a 1260-query workload can take more than 100 hours. In comparison, our training time is 15 hours.

- Incremental sampling gives the DBA the flexibility to collect more samples as and when resources are available. The training overhead can be spread out over time, and there is no need to allocate a contiguous block of training time.

6.4.6 Robustness and Scalability of Predictions

Next we drill down into the accuracy data presented in Figure 6.5 to study different aspects of the performance of our techniques for predicting workload completion times. In all these figures, we use $10T$ training samples. The conclusions that we draw in the following discussion are applicable for all sample sizes.

Effect of MPL: The first question we ask is what effect does MPL have on prediction errors? Recall that to study the accuracy of our approach in different settings, we vary MPL from 5 to 60. We have found that most of the cases where our prediction error is high are cases where the MPL is high. Figure 6.7 shows the cumulative frequency distribution of error for the 142 workloads that we use partitioned by MPL. The figure shows that in case of lower MPLs $(5, 10, 20, 30)$, around 85% of the predictions have error less than 20%. On the other hand, for higher MPLs, around 70% of the predictions have error less than 20%.

This increase in prediction error is due to two factors. At high MPLs, there is more load on the system. Thus, the variance in query completion times is higher, making prediction harder. The second factor that leads to reduced accuracy at higher MPLs is that the space of possible query mixes (from which we have to sample) grows substantially with increasing MPL. As discussed in Section 4.1, even for the simple case of 6 query types, when we increase MPL from $M = 20$ to $M = 50$ there is a 65-fold increase in the size of

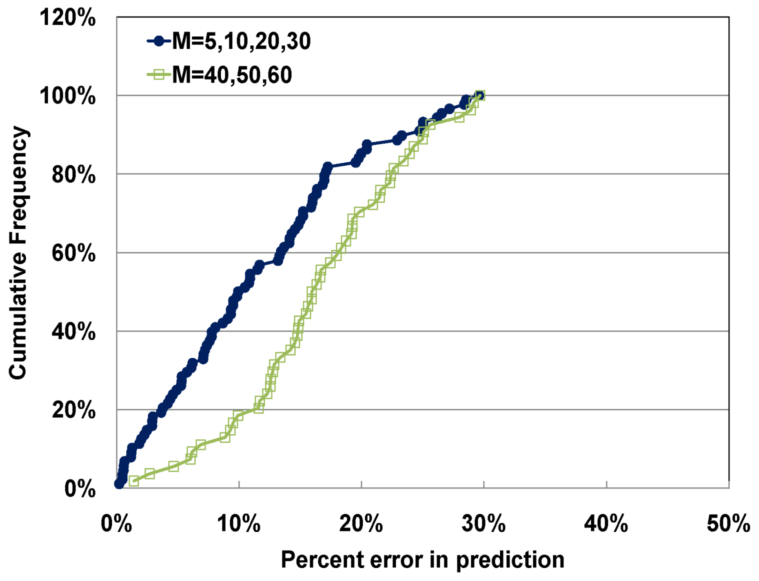


Figure 6.7: Prediction error across different MPL settings

the space to be sampled and modeled. However, in our experiments, we do not increase the number of samples, always collecting the same number of samples for every MPL for a given database size. Despite the high system load and explosion in the modeling space, we still maintain good prediction accuracy, which demonstrates the robustness of our approach.

Sensitivity to workload parameters and scheduling policy: We now turn our attention to varying different parameters that can affect prediction accuracy, such as the number of query types, scheduling policy, and workload size. Our focus is on studying the change in error distribution as these settings vary. Results are shown across all MPLs. It should now be apparent that higher errors generally correspond to higher MPLs.

Since our workload simulator makes predictions based on changing query mixes, we want to see how well it performs when the number of query types changes. When there are more query types to schedule, the simulator will encounter more distinct query mixes, so there is a higher likelihood of error in its prediction of workload completion times. To study the effect of the number of query types on accuracy, Figure 6.8 shows the error distribution for three types of workloads: (i) 1GB workloads that consist of 1260 queries picked from $T = 21$ distinct query types, (ii) 1GB workloads that consist of 720 queries picked from $T = 12$ distinct query types, and (iii) 1GB workloads that consist of 360 queries picked from $T = 6$ distinct query types and 10GB workloads that consist of 60 queries picked from $T = 6$ distinct query types.

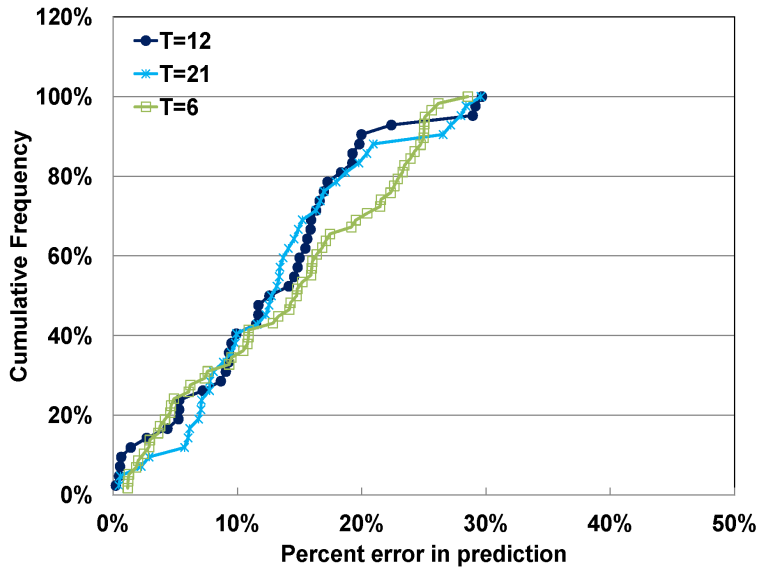


Figure 6.8: Prediction error when varying the number of query types

Figure 6.8 clearly shows the robustness of our approach when the number of query types is increased. In fact, we see slightly better accuracy when the number of query types is higher. Increasing the number of query types increases the sampling space. However, recall that when we use fewer query types, we use the longest-running query types in the system. When the workload consists solely of long-running queries that significantly interact with each other, in particular at high MPLs, there is more variance as compared to a workload where these long-running queries are separated by more predictable short-running queries.

Figure 6.9 shows the error distribution of the workloads that run on a 10GB database. These are long-running workloads, typically taking more than 4 hours to finish. The figure shows that even when we limit ourselves to only $5T$ samples, the prediction accuracy remains good, which shows that our approach is robust in the case of long-running queries even with a small number of samples.

Figure 6.10 shows prediction errors for two scheduling policies, FCFS and SJF. Prediction accuracy is high for both policies, illustrating that our approach is robust to variations in the scheduling policy. Both FCFS and SJF are straightforward to implement. However, for more advanced scheduling algorithms, the scheduler can itself be a simulator. As long as it is possible to know/simulate what the next query is, the prediction simulator can use this information and work in a seamless manner.

Sensitivity to data skew: Figure 6.11 shows the error distributions for workloads on a

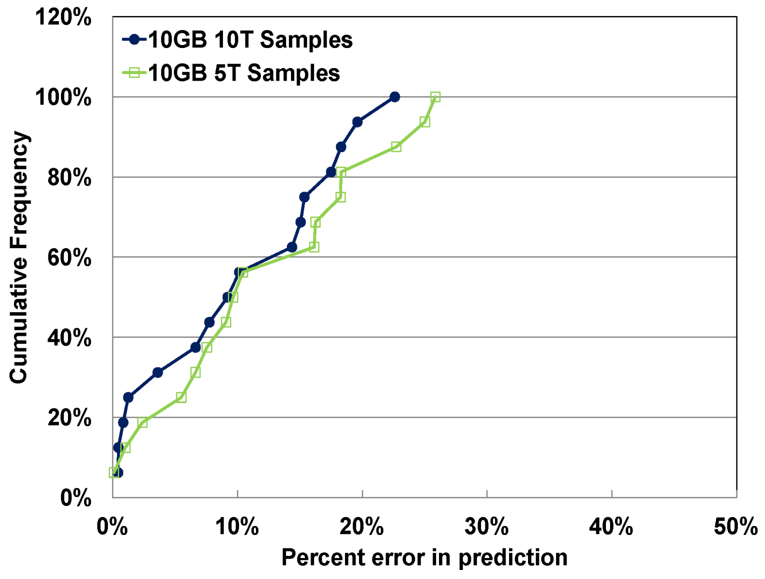


Figure 6.9: Prediction error for workloads on the 10GB database

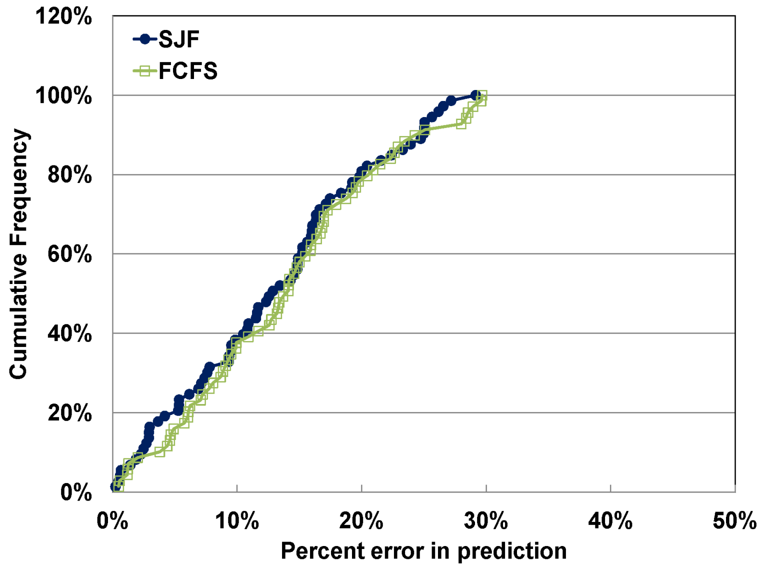


Figure 6.10: Prediction error for different scheduling policies

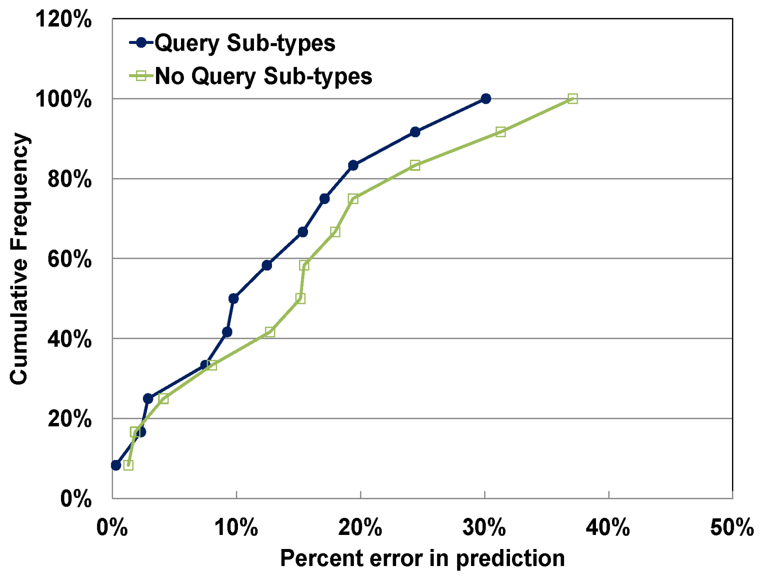


Figure 6.11: Prediction error for skew-aware and default approaches

TPC-H database with a skewed data distribution and 360 query instances comprising the 6 longest-running TPC-H query types. We use the skewed TPC-D/H database generator [94] to generate a 1GB TPC-H dataset with $z = 1$. Our experimental setup is the same as in Section 3.5. Our goal is to study how dividing query types into sub-types based on data skew will affect prediction accuracy. Recall from Section 3.5 that only Q_9 is divided into query sub-types. The figure shows the error for the skew-aware algorithm that partitions Q_9 into query sub-types, and for the skew-oblivious algorithm that does not partition Q_9. Both algorithms have good accuracy, but the skew-aware algorithm is more accurate, illustrating that query partitioning helps to improve the performance further.

Overall, we see that the combination of our sampling, modeling, and workload simulation algorithms result in accurate and robust predictions across a wide range of workload settings. Using our technique, a DBA can identify scenarios where she needs to repeatedly make predictions for batch workloads. She can use our sampling and modeling approach to train the interaction-aware models, and then use the workload simulator to accurately predict the completion times of different workloads.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation we have investigated the notion of *query interactions*. We have focussed on the insight that concurrently executing queries in a database system interact with each other, and these interactions can have a significant impact on query performance. We have argued that reasoning about query interactions can help in developing better techniques for workload management in database systems.

We have presented a detailed experimental study in Chapter 3 to demonstrate the impact of the query interactions among concurrently executing queries. The queries are represented using the abstraction of query types, and we propose techniques that can be employed to identify the query types in an automatic manner. We have chosen to study query interactions by measuring their impact on query runtimes. Query runtime is an intuitive metric because any potential cause of interactions is eventually reflected in the query runtime. Our experimental study illustrates that interactions in query mixes impact query run times significantly. Ignoring these interactions leads to inaccurate understanding of query performance.

Thus, we motivated the need for developing a mix-based reasoning about query workloads to better understand the performance of a database system. In particular we need performance models that can capture query interactions and predict the performance of queries in different mixes. We argued that analytical models are not suited for this purpose, and we proposed experiment-driven modeling as a way to capture query interactions. The experiment-driven modeling approach consists of running a small set of training query mixes and measuring the average completion time of the queries in these mixes. After that machine learning is employed to train models on this sample data. This approach requires no assumptions about the nature of query interactions or the internals of the database system. We discussed this approach in detail in Chapter 4, where we presented two different approaches to collect training mixes: CDR sampling and interaction level aware Latin Hypercube Sampling(IL_LHS). We considered two different modeling techniques to estimate the query completion times in query mixes: linear regression and Gaussian processes. Linear regression and CDR sampling is the simplest combination of modeling and sampling approaches, while Gaussian processes and IL_LHS is the most sophisticated combination that we considered.

We have presented solutions to two workload management problems that take advantage of our approach to capture and model query interactions The first problem we considered is that of scheduling report-generation query workloads in database systems. We presented QShuffler, a throughput oriented scheduler for BI report generation workloads, in Chapter 5. QShuffler employs batch and on-line scheduling algorithms, which correspond to two different scenarios. The batch scheduling algorithm is optimized for the scenario when queries are submitted in large batches. The on-line scheduling algorithm deals with the scenario when the user is submitting queries one at a time or in smaller batches. These algorithms make use of the interaction-aware performance models for estimating query completion times. Our experimental evaluation shows that QShuffler significantly improves performance over FCFS and SJF schedulers.

The second problem that we considered is predicting the completion time of batch BI workloads. Here, again, we demonstrate that reasoning about query mixes enables us to come up with a systematic approach that can help a database administrator in estimating completion time of a workload. In Chapter 6, we presented a mix-based workload simulator that estimates the completion time of a workload and uses the interaction-aware performance models that we propose. Our experiments show that the combination of our sampling, modeling, and workload simulation algorithms result in accurate and robust predictions across a wide range of workload settings.

7.2 Future Work

In this dissertation, when modeling query interactions, we rely on two coarse-grained features to represent a query mix, namely query types and the number of instances of each query type. A very interesting question is what other features of a query type or query instance can be relevant for predicting query completion time. For example, in [46] the authors use query plan features for predicting multiple performance metrics for database queries, although that work does not consider query interactions. Nevertheless, the query optimizer uses a sophisticated cost model to distinguish between candidate plans, and all commercial database systems now include some self-managing features in their optimizers and cost models. For example, some cost models are augmented by taking into account the current state of the system, which effectively means that the plans chosen for queries in a workload can depend upon what other queries are running concurrently. This means that query plan features may provide useful information for modeling interactions, and choosing the correct features is an interesting area for further research. For example, the authors of [40], while employing many ideas presented in this dissertation, propose using a logical I/O based metric to predict the execution performance of the queries in changing query mixes in a workload. In [17], the authors propose using both plan and operator level features to build models for query performance prediction. In [66], the authors use operator level features to model resource consumption for database queries. In this dissertation, we have focussed on black box modeling and our work shows that the models obtained from this approach are accurate enough to help us in important workload management tasks. However, we believe that developing models of query interactions that extend black box modeling with features from query semantics and query execution plans can be an interesting area of future work.

In this dissertation and other related works [40, 93, 102], the models of interactions are learned for a given system configuration, and they are updated as the system evolves. An interesting future direction is to explore whether the models can be learned in a general way, such that if the data or system changes, the models still remain accurate. If we can model interactions in a general way, we may be able to use the models to choose better system design. For example, if we learn about positive and negative interactions in a general way, we may be able to use this information to improve database physical design so that the physical design of a database induces positive interactions and avoids negative interactions. In [79], an effort is made to induce positive interactions by sharing scans in the buffer pool. We believe that there may be an opportunity to extend similar ideas to aspects of database physical design like indexes, partitioning, and materialized views.

With the emergence of cloud computing and software as a service (SaaS), database consolidation, multi-tenancy, and virtualization are gaining wide acceptance as means to reduce the cost and complexity of managing database systems. This new trend also poses many challenges for understanding and predicting system performance. The consolidated databases in a multi-tenant settings share resources and compete with each other for these resources. These interactions on shared resources can be quite complex and raise many interesting research questions. For example, can a new workload be added to an existing machine that is already running a mix of workloads without some resource becoming a bottleneck? If a machine is overloaded, which workloads should be moved to other machines in order to restore performance to the desired level? Should replicas be created for a database to scale-out the workload? Where should these replicas be placed? Recent research efforts have focussed on formulating these challenges from different perspectives and propose solutions for them [1, 6, 20, 34, 35, 36, 52, 71, 93, 95]. We believe that the ideas presented in this dissertation can be extended to some of the research problems in these areas. For example, in [14] we presented some preliminary results of our investigation of applying machine learning techniques to determine how resource metrics of consolidated systems are impacted by workload mixes and their interactions.

In general we are starting to see researchers in the database community showing more enthusiasm for experiment-driven performance modeling. We hope that the ideas presented in this dissertation will serve as useful tools to solve challenging problems in the area of database systems tuning, performance modeling, and workload management.

References

[1] Daniel J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009. 117

[2] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, 1988. 13

[3] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989. 13

[4] Robert K. Abbott and Hector Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *Proc. IEEE Real-Time Systems Symposium*, 1990. 13

[5] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems (TODS)*, 17(3):513–560, 1992. 13

[6] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar F. Minhas, Peter Kokosielis, and Sunil Kamath. Deploying database appliances in the cloud. *IEEE Data Eng. Bull.*, 32(1):13–20, 2009. 117

[7] Ashraf Aboulnaga, Ziyu Wang, and Zi Ye Zhang. Packing the most onto your cloud. In *Proc. Int. Workshop on Cloud Data Management (CloudDb)*, 2009. 17

[8] F. S. Acton. *Analysis of Straight-Line Data*. Dover, 1966. 46

[9] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2000. 78

[10] Mumtaz Ahmad, Ashraf Aboulnaga, and Shivnath Babu. Query interactions in database workloads. In *Proc. Int. Workshop on Testing Database Systems (DBTest)*, 2009. 12, 19

[11] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and exploiting query interactions in database systems. In *Proc. ACM Conf. on Information and Knowledge Management (CIKM)*, 2008. 12, 64

[12] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Qshuffler: Getting the query mix right. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2008. 12, 64

[13] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal*, 20(4):589–615, 2011. 12, 64

[14] Mumtaz Ahmad and Ivan T. Bowman. Predicting system performance for multi-tenant database workloads. In *Proc. Int. Workshop on Testing Database Systems (DBTest)*, 2011. 117

[15] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Interaction-aware prediction of business intelligence workload completion times. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2010. 12, 99

[16] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, 2011. 12, 99

[17] Mert Akdere, Ugur Cetintemel, Eli Upfal, and Stan Zdonik. Learning-based query performance modeling and prediction. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2012. 116

[18] Apache hadoop. http://hadoop.apache.org/. 17

[19] Aster data systems. http://www.asterdata.com/. 6, 64

[20] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2008. 117

[21] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4):333 – 353, 2004. 66

[22] Shivnath Babu, Nedyalko Borisov, Songyun Duan, Herodotos Herodotou, and Vamsidhar Thummala. Automated experiment-driven management of (database) systems. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, 2009. 17

[23] Francis R. Bach and Michael I. Jordan. Kernel independent component analysis. *Journal of Machine Learning Research*, 3:1–48, March 2003. 17

[24] Peter Belknap, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Leonidas Galanis, Shantanu Joshi, Jonathan Klein, Stratos Papadomanolakis, Uri Shaft, Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, Graham Wood, Khaled Yagoub, and Hailing Yu. Oracle real application testing. In *Proc. Int. Workshop on Testing Database Systems (DBTest)*, 2008. 18

[25] George E.P. Box and George C. Tiao. *Bayesian Inference in Statistical Analysis.* Wiley, 1973. 49

[26] Business Objects. http://www.businessobjects.com/. 64

[27] Michael J. Carey, Rajiv Jauhari, and Miron Livny. Priority in DBMS resource scheduling. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989. 13

[28] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. Load control for locking: The 'half-and-half' approach. In *Proc. ACM Symposium on Principles of Database Systems*, 1990. 14

[29] Yvonne Coady, Russ Cox, John Detreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, and Andrew Warfield. Falling off the cliff: When systems go nonlinear. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, 2005. 71

[30] Cognos. http://www.cognos.com/. 64

[31] Richard H. Conway, William L. Maxwell, and Miller Louis W. *Theory of scheduling.* Addison-Wesley, 1967. 13

[32] David R. Cox and Peter A. Lewis. *Statistical Analysis of Series of Events.* Chapman & Hall, 1966. 32

[33] CPLEX. http://www.ilog.com/products/cplex/. 69

[34] Carlo Curino, Evan Jones, Samuel Madden, and Hari Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2011. 117

[35] Carlo Curino, Evan Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2011. 117

[36] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. *CoRR*, abs/1008.3751, 2010. 117

[37] Umeshwar Dayal, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Archana Ganapathi, and Stefan Krompass. Managing operational business intelligence workloads. *ACM SIGOPS Operating Systems Review*, 43(1):92–98, 2009. 17, 99

[38] IBM DB2. http://www-01.ibm.com/software/data/db2/. 2

[39] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2009. 17, 18

[40] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2011. 12, 116

[41] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proc. Int. Conf. on World Wide Web (WWW)*, 2004. 14

[42] Ugo Fano. On the theory of ionization yield of radiations in different substances. *Physical Review*, (1-2):44–52, 1946. 32

[43] Peter A. Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems (TODS)*, 10(1):1–28, 1985. 14

[44] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Wait depth limited concurrency control. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1991. 14

[45] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy H. Katz, and David A. Patterson. Statistics-driven workload modeling for the cloud. In *Workshops Proc. Int. Conf. on Data Engineering*, 2010. 17

[46] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2009. 17, 45, 115

[47] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, and Anthony Tomasic. Scalable query result caching for web applications. *Proc. of the VLDB Endowment (PVLDB)*, 1(1):550–561, 2008. 29

[48] Saeed Ghanbari, Gokul Soundararajan, Jin Chen, and Cristiana Amza. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, 2007. 17

[49] Greenplum. http://www.greenplum.com/. 6, 64

[50] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. PQR: Predicting query execution times for autonomous workload management. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, 2008. 17

[51] Hans-Ulrich Heiss and Roger Wagner. Adaptive load control in transaction processing systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1991. 14

[52] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. of the VLDB Endowment (PVLDB)*, 4(11):1111–1122, 2011. 117

[53] Charles R. Hicks and Kenneth V. Turner. *Fundamental Concepts in the Design of Experiments.* Oxford University Press, 1999. 40

[54] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Don Towsley. On using priority inheritance in real-time databases. In *Proc. IEEE Real-Time Systems Symposium*, 1991. 13

[55] Toshihide Ibaraki, Tiko Kameda, and Naoki Katoh. Cautious transaction schedulers for database concurrency control. *IEEE Transactions on Software Engineering*, 14(7):997–1009, 1988. 13

[56] Abhinav Kamra, Vishal Misra, and Eric M. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered websites. In *Proc. Int. Workshop on Quality of Service (IWQOS)*, 2004. 14

[57] Kyoung-Don Kang, Sang H. Son, and John A. Stankovic. Service differentiation in real-time main memory databases. In *Proc. IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing*, 2002. 13

[58] Naoki Katoh, Toshihide Ibaraki, and Tiko Kameda. Cautious transaction schedulers with admission control. *ACM Transactions on Database Systems (TODS)*, 10(2):205–229, 1985. 13, 14

[59] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley, 1990. 32

[60] Terence Kelly. Detecting performance anomalies in global applications. In *Proc. Workshop on Real, Large Distributed Systems*, 2005. 9, 10

[61] John F. Kenney and E. S. Keeping. *Mathematics of Statistics, Part 1.* 3rd edition. 46

[62] John F. Kenney and E. S. Keeping. *Mathematics of Statistics, Part 2.* 2nd edition. 46

[63] Stefan Krompass, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing long-running queries. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, 2009. 99

[64] Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Kevin Wilkinson, Archana Ganapathi, and Stefan Krompass. Managing dynamic mixed workloads for operational business intelligence. In *Proc. Intl. Workshop on Databases in Networked Information Systems (DNIS)*, 2010. 17

[65] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. Automatic database configuration for DB2 universal database: Compressing years of performance expertise into seconds of execution. In *Proc. Conf. on Database Systems for Business, Technology and Web (BTW)*, 2003. 20

[66] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proc. of the VLDB Endowment (PVLDB)*, 5(11):1555–1566, 2012. 116

[67] Amit Manjhi, Phillip B. Gibbons, Anastassia Ailamaki, Charles Garrod, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, Anthony Tomasic, and Haifeng Yu. Invalidation clues for database scalability services. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2007. 29

[68] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2004. 13

[69] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Improving preemptive prioritization via statistical characterization of OLTP locking. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2005. 13

[70] Abhay Mehta, Chetan Gupta, and Umeshwar Dayal. BI Batch Manager: A system for managing batch workloads on enterprise data warehouses. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, 2008. 16

[71] Umar F. Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *Proc. of the VLDB Endowment (PVLDB)*, 4(11):738–748, 2011. 117

[72] Axel Mönkeberg and Gerhard Weikum. Conflict-driven load control for the avoidance of data-contention thrashing. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1991. 14

[73] Axel Mönkeberg and Gerhard Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1992. 14

[74] MySQL slow query log parser. http://code.google.com/p/mysql-slow-query-log-parser/. 29

[75] Baoning Niu, Patrick Martin, and Wendy Powley. Towards autonomic workload management in DBMSs. *Journal of Database Management*, 20(3):1–17, 2009. 15, 79, 80

[76] Baoning Niu, Patrick Martin, Wendy Powley, Paul Bird, and Randy Horman. Adapting mixed workloads to meet SLOs in autonomic DBMSs. In *Workshops Proc. Int. Conf. on Data Engineering*, 2007. 15, 79, 80

[77] Baoning Niu, Patrick Martin, Wendy Powley, Randy Horman, and Paul Bird. Workload adaptation in autonomic DBMSs. In *Proc. Conf. of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2006. 15, 79, 80

[78] Baoning Niu and Jian Shi. Scalable workload adaptation for mixed workload. In *Int. Conf. on Scalable Information Systems (Infoscale)*. 15, 79, 80

[79] Kevin O'Gorman, Amr El Abbadi, and Divyakant Agrawal. Multiple query optimization in middleware using query teamwork. *Software - Practice and Experience*, 35(4):361–391, 2005. 12, 116

[80] HP service manager software. http://www.managementsoftware.hp.com/. 11

[81] Oguzhan Ozmen, Kenneth Salem, Mustafa Uysal, and M. Hossein Sheikh Attar. Storage workload estimation for database management systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2007. 100

[82] HweeHwa Pang, Michael J. Carey, and Miron Livny. Multiclass query scheduling in real-time database systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):533–551, 1995. 13

[83] Query patroller: IBM DB2 Query Patroller adminsistration guide. http://www.ibm.com/software/data/db2/ querypatroller/. 15

[84] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. 48, 49, 50, 51

[85] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2008. 12

[86] Herbert J. Ryser. *Combinatorial Mathematics*. The Mathematical Association of America, 1963. 37, 68

[87] Giovanni M. Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)*, 11(4):473–498, 1986. 13

[88] Thomas J. Santner, Brian J. Williams, and William Notz. *The Design and Analysis of Computer Experiments*. Springer-Verlag, 2003. 39, 40

[89] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998. 68, 69

[90] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology (TOIT)*, 6(1):20–52, 2006. 13

[91] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, and Erich Nahum. Achieving class-based QoS for transactional workloads. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2006. 13

[92] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to determine a good multi-programming level for external scheduling. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2006. 14

[93] Muhammad B. Sheikh, Umar F. Minhas, Omar Z. Khan, Ashraf Aboulnaga, Pascal Poupart, and David J. Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, 2011. 12, 116, 117

[94] Skewed TPC-D data generator. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/. 33, 113

[95] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2008. 17, 117

[96] Gokul Soundararajan and Cristiana Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, 2006. 17

[97] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *European Conf. on Computer systems (EuroSys)*, 2007. 9, 10

[98] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2001. 17

[99] Alexander Thomasian. Performance limits of two-phase locking. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1991. 14

[100] Alexander Thomasian. Thrashing in two-phase locking revisited. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1992. 14

[101] E. C. Titchmarsh. *The Theory of the Riemann Zeta Function.* Oxford Science Publications, Clarendon Press, Oxford, second edition, 1986. 33

[102] Sean Tozer, Tim Brecht, and Ashraf Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2010. 12, 116

[103] Transaction Processing Perfromance Council (TPC). http://www.tpc.org/tpch. 2, 20

[104] Transaction Processing Perfromance Council (TPC). http://www.tpc.org/tpcw. 10

[105] Richard V. Mises. *Mathematical Theory of Probability and Statistics.* Academic Press, 1964. 48

[106] Vertica. http://www.vertica.com/. 6, 64

[107] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proc. USENIX Symposium on Internet Technologies and Systems*, 2003. 14

[108] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, third edition, 2011. 45, 47, 51, 62

[109] DB2 workload manager. http://www.ibm.com/software/data/db2/. 15

[110] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing XML queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2005. 17

[111] Qi Zhang, Ludmila Cherkasova, Guy Mathews, Wayne Greene, and Evgenia Smirni. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Proc. ACM/IFIP/USENIX Int. Conf. on Middleware*, 2007. 9, 11

[112] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proc. Int. Conf. on Autonomic Computing (ICAC)*, 2007. 9, 10, 11

[113] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-arellano, and Scott Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004. 20, 78