QShuffler: Getting the Query Mix Right

Mumtaz Ahmad [1], Ashraf Aboulnaga [1], Shivnath Babu [2], Kamesh Munagala [2]

[1]*David R. Cheriton School of Computer Science, University of Waterloo*
{m4ahmad, ashraf}@cs.uwaterloo.ca

[2]*Department of Computer Science, Duke University*
{shivnath, kamesh}@cs.duke.edu

Abstract— **The typical workload in a database system consists of a mixture of multiple queries of different types, running concurrently and interacting with each other. Hence, optimizing performance requires reasoning about query mixes and their interactions, rather than considering individual queries or query types. In this paper, we use such a reasoning approach to develop a query scheduler. We treat the database system as a black box and experimentally build a model to estimate the performance of different query mixes. Our scheduler uses this model to decide which query mixes to schedule, with the goal of maximizing throughput. We experimentally demonstrate the effectiveness of our scheduler using queries from the TPC-H benchmark on DB2.**

I. Introduction

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

Surprisingly, little work deals with optimizing performance for concurrently-running and time-varying query mixes where significant inter-query interactions may exist. Different query types running concurrently in a mix may have different and possibly interfering resource demands. This interference can happen at different resources, both internal and external to the database system (e.g., buffer pool, shared data structures, latches, working memory, CPU, disk, etc.) and can vary with time. Current database systems cannot isolate the resource demands of different queries, so they must deal with the effect of query interaction. It is therefore important to understand and reason about performance in terms of query mixes with interacting queries. This issue has not been adequately addressed by the database research community, and we believe that it presents a significant opportunity for improving performance and robustness, and may require revisiting many research issues such as scheduling, query optimization, database physical design, and statistics collection and maintenance. In all these areas, decisions that are optimal for individual queries or query types may be highly suboptimal for query mixes.

In this paper, we study the performance of query mixes, using as a concrete vehicle for our study of the problem of scheduling queries in a database system while taking into account the interaction among queries in a mix. Our scheduler, called *QShuffler* (for *Query Shuffler*), focuses on *throughput-oriented* workloads like the report generation workloads of business intelligence systems such as Cognos and Hyperion. For these workloads, the objective is to complete a certain number of queries (i.e., produce a certain number of reports) as efficiently as possible within an available time window. Hence, the goal of QShuffler is to schedule the appropriate query mixes for a given workload W to minimize W's *total completion time*. Conventional schedulers (e.g., shortest job first) rely on the characteristics of individual queries, and can produce suboptimal schedules when significant inter-query interactions exist. QShuffler makes scheduling decisions based on query mixes, so it produces better schedules. QShuffler has two main components: (1) modeling the performance of query mixes, and (2) scheduling based on the performance model. We describe these two components in the next two sections.

II. Modeling Query Mix Performance

When reasoning about the performance of different query mixes, whether for scheduling or other tuning tasks, we need a *model* that characterizes the performance of the different possible mixes. Some recent work attempts to model the performance of transaction mixes, but without taking concurrent execution into account [1]. One of the main features of our work is that we take concurrent execution into account when modeling performance. For example, in our scheduler we use the performance model to identify mixes in which there is significant negative interaction among the queries so that we can avoid scheduling them. Other performance tuning tasks will require modeling other performance characteristics, but it is always important to capture query interaction.

Modeling the performance of query mixes analytically is difficult because the model would need to capture the many possible complex interactions among queries in any mix from the very large space of possible mixes. Therefore, a design decision we made was to avoid analytically modeling the complex internal behavior of a database system and the complex query interactions. Instead, we treat the database system as a black box, and build models experimentally as an off-line process. The black-box approach makes our modeling oblivious to the specific cause of query interaction and allows us to use the same modeling methodology for any performance metric and any database system.

The off-line model-building process involves (1) running a sample of the possible query mixes and measuring the performance parameters of interest from each sample run, and then (2) fitting a model to the measurements. This requires a priori knowledge of the different query types so that we can sample the space of possible mixes. It also requires that all queries of a particular type have the same effect on performance so that our models can be accurate. For example, if the data is uniformly distributed, we could define the query types to be the different query templates that appear in the client applications, with different instances of the query type corresponding to instantiating the template with different parameter values. If data skew results in queries with the same template having different performance depending on the values of parameters, we could define different query types for different sets of parameter values.

The challenge in our off-line modeling process is that the space of possible query mixes that we are modeling is very large. If we have T query types and M queries in each mix, we have a T-dimensional space of possible mixes whose total size is given by $S(T, M) = \binom{M+T-1}{M}$ [2]. We can only sample a small fraction of this space. Thus, we need to be careful in our choice of performance metric and model structure (linear, quadratic, regression tree, etc.) so that we can build accurate performance models based on a small number of sample query mixes. For QShuffler, we have developed such a metric, and we describe it with other aspects of our scheduler next.

III. Scheduling with Query Mixes

Problem Setting: We are given a database system with a fixed multi-programming level (MPL), M, which may have been set using techniques such as [3]. We are also given a fixed number, T, of possible query types. The database client applications place their queries in an arrival queue, and QShuffler schedules queries from this queue. Each client application issues a finite number of queries so the workload, W, has a total size that is bounded. The objective of QShuffler is to schedule a sequence of query mixes so that the total completion time of W is minimized (i.e., throughput is maximized).

Algorithm: QShuffler schedules queries from the arrival queue at the same time that clients are placing queries in this queue. Whenever a running query finishes, the scheduling algorithm looks for the best query to schedule from the arrival queue; it makes scheduling decisions to achieve the objective of running the system *as close as possible to capacity without exceeding acceptable capacity*. The system takes on as much work as it can in the near term so that it is not stuck with too much work in the far term. Implementing this framework requires: (1) A *load metric*, R_i, for measuring the load placed on the system by a query mix. This metric should capture all resource interference between queries, and it should measure the load placed on all resources. (2) A *performance model*, $\hat{R}_i$, to estimate the load metric R_i given a query mix. (3) A *load threshold*, θ_R, that specifies the maximum allowable value of $\hat{R}_i$. This is our measure of system capacity. When a query finishes, the algorithm uses the performance model to ask a what-if question about the effect that adding each query type to the currently running query mix would have on $\hat{R}_i$. The algorithm then schedules the query type from the arrival queue that makes $\hat{R}_i$ as high as possible but less than θ_R. This implements the heuristic of keeping the system as close as possible to capacity. If all query types make $\hat{R}_i > \theta_R$, we schedule the query type with the lowest $\hat{R}_i$ in the arrival queue.

Cost Metric: Queries may interact with each other in complex and time varying ways, and they can interfere at different resources. We observe that all these interactions should manifest themselves in the average response time that a query type exhibits in a given mix. Thus, to be able to measure load in a robust manner, we develop a load metric that relies on *overall query execution time*. Our metric, which we call NRO for *Normalized Run time Overhead*, is a measure of how much run time overhead the queries of different types incur because they are running with other queries in a query mix. Let the average run time of a query of type j if it is running alone in the system be t_j, and let its average run time if it is running in query mix m_i be A_{ij}. Due to query interaction, A_{ij} will be greater than t_j. Thus, the run time overhead to the query due to running in a mix is A_{ij}/t_j. To generalize this to a query mix, m_i, with T query types and a total of M queries, let N_{ij} be the number of queries of type j in the mix. We define the overall run time overhead for the T query types in the mix as the *weighted average* of their individual overheads, where the weight of query type j is the fraction of queries of this type in the mix. Thus, the overhead for mix m_i is

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

or

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

This value represents the total overhead for the query mix. To be able to use the same metric to measure overhead for mixes of different sizes (i.e., different M), we define our load metric, NRO_i, as the *normalized* overhead, computed per query served. That is, we divide the above value by M. Thus,

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

This captures the fact that incurring an overhead of, say, 5 while serving 20 concurrent queries is better than incurring an overhead of 5 while serving 10 queries. We use a load threshold $\theta_{NRO} = 0.7$, which we determined experimentally.

Model for NRO: We have found that a *multi-dimensional linear model* works well for estimating our NRO metric. Thus, we estimate NRO as

$$\widehat{NRO}_i = \sum_{j=1}^{T} \beta_j N_{ij}$$

We use multi-dimensional linear regression based on around 200 sample query mixes to fit the β_j's of the model.

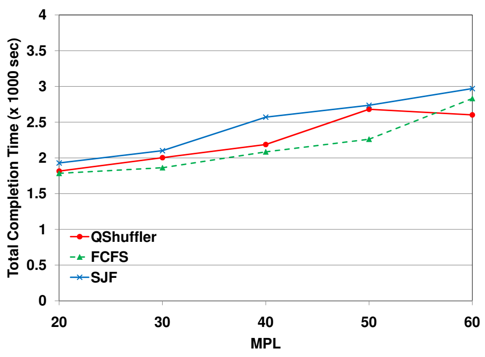

Fig. 1. Scheduling for $p = 5$

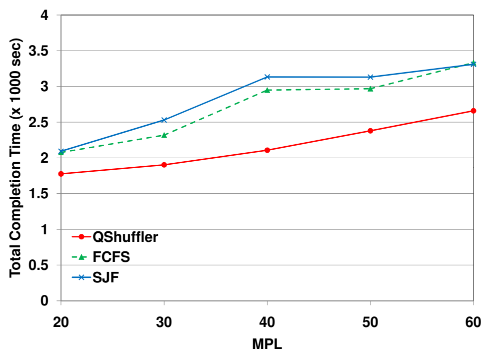

Fig. 2. Scheduling for $p = 25$

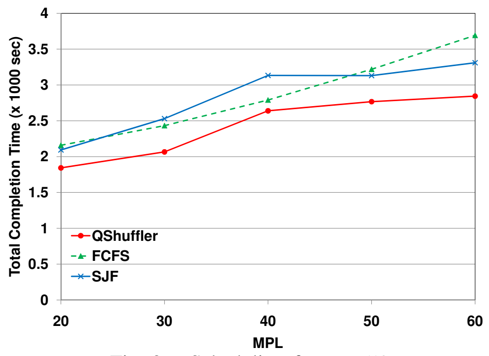

Fig. 3. Scheduling for $p = 50$

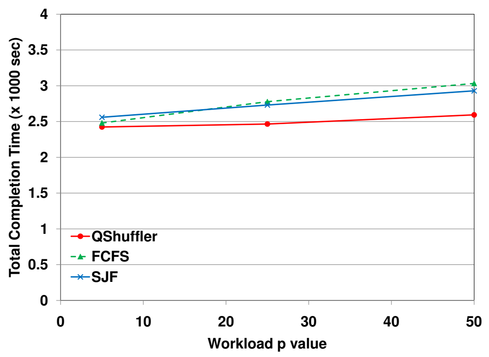

Fig. 4. Scheduling for $T = 12$

IV. Experiments

Our experiments use a machine with dual 3.4GHz Intel Xeon CPUs and 4GB of RAM running Windows Server 2003. The database system we use is DB2 V8.1. We use the TPC-H benchmark with scale factor 1GB. We set the buffer pool size of the database to 400MB. QShuffler is implemented in Java as a stand-alone scheduler, separate from the database server.

For our first experiment, we use a workload consisting of 60 instances of each of the 6 longest running TPC-H queries, for a total of 360 queries. We generate the workload by placing p instances of each query type (with different parameter values) in the arrival queue until all queries are in the queue, with $p =$ 5, 25, or 50. As p increases, the imbalance in the workload increases. At any point in time, the scheduler can only see the next L queries in the queue. This "lookahead" is set to $L = 60$. We compare our scheduling algorithm to two other algorithms: First Come First Served (FCFS), which is the simplest scheduling algorithm, and Shortest Job First (SJF), which is known to be optimal when ignoring query interactions and has been used in other works (e.g., [4]). Our performance metric is the *total completion time* of the workload.

Figures 1, 2, and 3 show the performance of the different schedulers for varying MPL for p = 5, 25, and 50, respectively. The figures show that SJF is the worst policy for all cases since it ignores query interaction. QShuffler is better than FCFS for $p = 25$ and 50, but not for $p = 5$. For $p = 5$, the arrival order is almost approaching round robin, so not too many queries with an adverse effect on the mix can arrive together and FCFS works well. Figures 2 and 3 show that for more unbalanced workloads, QShuffler improves significantly over FCFS.

Our second experiment tests our scheduling algorithm with a workload consisting of 60 instances of each of the 12 longest running TPC-H queries ($T = 12$). Figure 4 shows the performance of the different schedulers on this workload for MPL $M = 30$ and varying p. QShuffler still performs better than the other algorithms, which indicates that the quality of QShuffler is not affected by the number of query types.

V. Conclusion

In this paper, we argue that it is important to make performance related decisions in database systems based on *query mixes* not individual queries. We present QShuffler, a scheduler that reasons about query mixes. QShuffler treats the database system as a black box and experimentally models its performance for different mixes. It then uses the performance model to decide on the query mixes to schedule to maximize throughput. We experimentally demonstrate the effectiveness of QShuffler using TPC-H queries running on DB2.

Acknowledgments

Ahmad and Aboulnaga were supported by grants from NSERC. Babu was supported by an NSF CAREER award and grants from IBM. Munagala was supported by NSF award CNS-0540347.

References

[1] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *Proc. EuroSys Conf.*, 2007.

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

[3] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman, "How to determine a good multi-programming level for external scheduling," in *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2006.

[4] S. Elnikety, E. Nahum, J. Tracey, and W. 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.