Dynamic Concurrency Control while Scheduling Query Mixes

Brittany Terese Fasy and Herodotos Herodotou

December 11, 2007
1 Introduction

The demand for multi-user applications has grown significantly with the explosive expansion of the Internet. Concurrent access and modification of data has become the norm, and thus, database systems handle concurrent execution of queries of many different types. Often, the interaction of these queries among themselves can impact the performance of the system significantly. However, standard methods of query optimization used by most commercial database systems (like DB2 or Oracle) are based primarily on evaluating the cost of each query in isolation from the rest.

QShuffler [1], a query scheduler, takes into consideration the interaction of different concurrent queries in order to minimize the completion time of a report generation workload. Given a large set of queries, it selects good query mixes to send to the database system for execution. As queries complete, QShuffler chooses the next query to add to the current mix by selecting the most expensive query that keeps the overall load below a predefined load threshold.

QShuffler assumes that the optimal number \( M \) of concurrent queries is determined in advance, and \( M \) is treated as a constant throughout scheduling. However, keeping \( M \) fixed poses several disadvantages including the potential for system overload, data-contention thrashing, and resource underutilization. We propose improvements to the scheduling algorithm and to the performance model of Qshuffler that allow for a varying number of concurrent queries to be executed in an efficient way.

2 Related Work

Our project was primarily influenced by the QShuffler project discussed in [1]. The goal of that paper was to introduce a query scheduler that is capable of taking query interactions into consideration in order to improve the overall performance of the database.

2.1 QShuffler Overview

QShuffler is a query scheduler that takes query interactions into consideration in order to minimize the completion time of all workloads. It uses a novel cost metric called Normalized Runtime Overhead (NRO) that indirectly represents the load incurred by the system due to queries running in a mix instead of running sequentially. We define the mix \( m_i = \langle N_{i1}, N_{i2}, \ldots, N_{iT} \rangle \), where \( N_{ij} \) represents the number of queries of type \( j \) in mix \( i \) and \( T \) is the number of query types. In order to predict the NRO of a particular mix, QShuffler uses a multi-dimensional linear regression model that is initialized offline with a randomized training set. The domain of this regression was limited to the query mixes with a fixed \( M \).

2.1.1 Cost Metric

QShuffler determines the "cost" of a mix \( i \) by determining \( NRO_i \), the average normalized overhead per query incurred by a mix running in the database. We do this by finding the average overhead per query and then normalizing. We assume that the overhead of query \( j \) running in mix \( i \) will be captured by the ratio of the running time of query \( j \) in mix \( i \), \( A_{ij} \), to the cost of query \( j \) in isolation from the rest, \( t_j \). Thus, we use the following formula in order to determine the normalized runtime overhead of mix \( i \):

\[
NRO_i = \frac{1}{M^2} \sum_{j=1}^{T} N_{ij} \frac{A_{ij}}{t_j}
\]  

(1)

This metric is used by the QShuffler algorithm in order to determine which query to include in the currently running mix.
To demonstrate the NRO metric we will refer to Figure 1 and 2 for some illustrative examples. Figure 1 contains the average run times of each query type used in our experiment, when they are run by themselves in the database. Figure 2 contains four different mixes from the offline training data set with $MPL = 30$. For each mix, the table contains the number of each query types in that mix ($N_{ij}$), the average run time of each query type in that mix ($A_{ij}$) and the NRO of that mix.

<table>
<thead>
<tr>
<th>Mix</th>
<th>Q3</th>
<th>Q8</th>
<th>Q10</th>
<th>Q11</th>
<th>Q14</th>
<th>Q16</th>
<th>Q18</th>
<th>Q19</th>
<th>NRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>30</td>
<td>1807.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1309</td>
</tr>
<tr>
<td>$m_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.372</td>
</tr>
<tr>
<td>$m_3$</td>
<td>17</td>
<td>2610.7</td>
<td>4</td>
<td>2681.9</td>
<td>1</td>
<td>690.2</td>
<td>2</td>
<td>73.0</td>
<td>1</td>
</tr>
<tr>
<td>$m_4$</td>
<td>2</td>
<td>714.7</td>
<td>4</td>
<td>744.7</td>
<td>2</td>
<td>390.8</td>
<td>11</td>
<td>22.3</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2: NRO Value for Different Query Mixes

The first two mixes represent two very simple mixes containing multiple instance of the same query. Mix $m_1$ consists of 30 queries of type Q3 and mix $m_2$ consists of 30 queries of type Q11. We notice that the NRO for $m_1$ is relatively high, thus we suspect that many instance of Q3 running together cause a heavy load in the system. Indeed, the average runtime of all the queries in the mix is 1807.8 seconds, whereas if we had run the 30 queries sequentially it would have taken on average $30 \times 46.04 = 1381.2$ seconds. On the other hand, $m_2$ has a relatively low NRO and hence we expect that queries of type Q11 do not interact with each other and do not load the system. Indeed, the average runtime of the queries in $m_2$ is 20.6 seconds whereas a sequential run of these queries would have taken on average $30 \times 1.85 = 55.5$ seconds.

The last 2 mixes reveal how the NRO can be used to distinguish bad mixes from good mixes. Looking at the NRO we see that $m_3$ loads the system more than $m_4$. This will become obvious as soon as we observe the $A_{ij}$ values. For example, 1 query of type Q16 takes 220.3 seconds to complete in $m_3$ whereas 2 queries of the same type take only 133.4 seconds to complete in $m_4$. The same observation can be done for almost all other query types in the 2 mixes. The scheduler using the NRO metric will avoid bad mixes like $m_3$ and try to schedule good mixes like $m_4$.

### 2.1.2 QShuffler Algorithm

The algorithm begins by initiating the execution of the first $M$ queries that enter the system. Once the first query finishes execution, the scheduler looks in to the arrival queue, which contains a set number of queries (we will call this parameter the lookahead $L$), and then evaluates which one is the "best" query to add into the current mix of queries in the system.

In order to determine the optimal query to add, QShuffler uses two metrics: an estimate for the NRO and the $\theta_{NRO}$, a load threshold which is set a priori. The NRO is calculated using the regression model based on the current mix running in the system. Finding an optimal $\theta_{NRO}$ can be done experimentally by executing QShuffler using different values in order to determine which one consistently executes the workload fastest. For each query in the arrival queue, QShuffler estimates the NRO, then chooses the query whose addition to the currently executing queries creates the mix with the greatest estimated NRO below $\theta_{NRO}$. If no such query exists, then QShuffler chooses the query whose corresponding mix has the smallest estimated NRO.
3 Multi-Programming Level (MPL)

The performance of a database system depends heavily on several configuration parameters that can be set in order to execute queries as efficiently and effectively as possible. One of the most important parameters that most commercial databases provide is the Multi-Programming Level (MPL). The MPL is the maximum number of queries that can be executed concurrently by the system. Database administrators, system administrators or data analysts usually perform the task of customizing the different configurations of the database, including the MPL. Setting the MPL at appropriate levels is a very challenging task and there exists an extensive literature on how to find the optimal MPL level. (See [2], [3], [4], and [5]). However, even if some optimal value for the MPL is found, there are still several disadvantages in using a static value.

The database performance with respect to the chosen MPL level is inherently sensitive to the different types of workloads. For some workloads a specific MPL level might be considered too high, whereas for some other workloads it may be considered too low. Having a high MPL increases the chances for data-contention thrashing. Data-contention thrashing occurs when several transactions are competing for the same resources or for accessing the same data. This usually increases the amount of lock conflicts causing long and frequent delays in the system. Moreover, excessive lock conflicts may also lead to deadlocks, depending on the characteristics of the system. On the other hand, a low MPL leads to under-utilizing the available resources since transactions will be forced to needlessly wait in the arrival queue instead of executing. This might increase the response time for some queries to unacceptable levels.

Another main disadvantage of a static MPL is that it does not react to changes in the query mixes of the current load. For example, suppose that most of the workloads of a system consist of short, read-only queries but every once and a while, some long queries that perform updates arrive in the system. Knowing these facts, a database administrator would probably have to set the MPL at a low, conservative level in order to ensure that the update queries don’t overload the system every time they arrive. However, a higher MPL would have been more appropriate for most of the time. One possible solution to this scenario would be to categorize the workloads into different types and change the MPL every time the workload changes. Such a task would probably be overly complicated, especially when the workloads are very diverse and unpredictable.

A system that is able to dynamically vary the MPL would eradicate the above problems and help to further optimize the performance of the database system. Our query scheduler is able to detect different kinds of interaction between the arriving queries. Thus, if the current workload consists of many queries with bad interactions between them, the scheduler will choose to decrease the MPL level, avoiding a potential system overload. On the other hand, if the scheduler detects that the current workload consists of queries with good interactions between them, it will increase the MPL ensuring a smart utilization of resources. Overall, our query scheduler is able to dynamically adjust the MPL level in order to maximize the performance of the database system while being robust to drastic changes in the workloads.

4 Updating the Algorithm to Allow for Varying MPL

The algorithm for QShuffler-Dynamic Concurrency Control (QShuffler-DCC) generally follows the original algorithm: First, the execution of the first $M$ queries begins. Once the first query finishes execution, the scheduler looks into the arrival queue, and then evaluates whether or not there exists a query that should be added into the current mix of queries in the system.

In order to determine which query is best, QShuffler-DCC uses the estimate for the $NRO$ and $\theta_{NRO}$. For each query in the arrival queue, the algorithm estimates the $NRO$, then chooses the query whose addition to the currently executing queries creates the mix with the greatest estimated $NRO$ below $\theta_{NRO}$. If no such query exists, then no query is added to the mix at this time, thus effectively decreasing the $MPL$. If the
algorithm does find a query to start executing, QShuffler-DCC continually adds queries to the current mix until it can no longer add a query without the load exceeding $\theta_{NRO}$. In this scenario, the MPL is increasing.

5  Linear Regression

Linear Regression is a regression method that can be used to model the relationship between the cost metric ($NRO$) for a specific mix $i$ and the number of queries ($N_{ij}$) of each query type $j$ in that mix. The goal of linear regression is to create the model that best fits the set of training data we have collected in order to accurately predict the $NRO$ values of query mixes we haven’t experimented with. Selecting the appropriate model that fits the data is a very challenging task. The original QShuffler implementation used a linear model as their regression model because of its simplicity and satisfactory results. However, our experiments have shown that a quadratic model fits the data better and provides better outcomes.

5.1  Linear Model

In the original QShuffler paper, the following regression model was used:

$$\hat{NRO}_i = w_0 + \sum_{j=1}^{T} w_j N_{ij}$$  (2)

The $\hat{NRO}_i$ represents the estimate for the $NRO_i$ in mix $i$. The $w_0$ term is called the intercept (or constant) of the regression and the $w_j$ terms are the regression coefficients. These coefficients will be estimated during the offline training of the model, using the data collected as explained in Section 6. The above model is linear in $N_{ij}$ and forms a hyperplane in $(T + 1)$ dimensions.

5.2  Quadratic Model

Let’s consider the following regression model:

$$\hat{NRO}_i = w_0 + \sum_{j=1}^{T} w_j N_{ij} + \sum_{j=1}^{T} v_j N_{ij}^2.$$  (3)

In this model, both the $w_j$ and the $v_j$ terms constitute the regression coefficients that will be estimated during the offline training of the model. The model is quadratic in $N_{ij}$ and forms a hyper-paraboloid in $(T + 1)$ dimensions. It should be noted that a complete quadratic model would also include the interaction terms of the form $N_{ij} \times N_{ik}$ for all possible pairs $(j,k)$. Those terms were not included in the model because of the consequences that follow from the large number of all possible pairs. The number of all pairs is $\binom{T}{2}$ which is in $O(T^2)$. For our experiments, the introduction of the interaction terms would have added 28 more terms. A large number of terms in definitely not desirable because it requires a much larger set of training data to get accurate results. In addition, the extra terms would add a significant overhead in calculating the $\hat{NRO}_i$ during the actual execution of the system.

5.3  Model Comparison Using Two Query Types in 2D

During the early stages of our experimentation, we designed some simple experiments to both test the accuracy of the quadratic model and also to get a better insight of the data we collected. We decided to use only two query types in order to be able to visualize our results. We created a workload consisting of all possible mixes for MPL’s, ranging from 1 to 12. The workload contained a total of 90 mixes. We executed the queries in the workload and calculated the $NRO$ for each mix as described in Section 2.1.1. The data we created were of the form $< [N_{i1}, N_{i2}], NRO_i >$, for $i = 1, 2, ..., 90$. These data were used to train both the linear and quadratic model on various MPL’s. The quadratic model fitted the data significantly better than
the linear model. Figure 3a shows the $NRO$ values plotted against the $N_1$ and the best linear fit into the data for $MPL = 12$. Note that the value for $N_2$ is simply $MPL − N_1$. Figure 3b shows the best quadratic fit into the data for $MPL = 12$. It is evident that the quadratic fit represents the data a lot more accurately than the linear fit. Specifically, the coefficient of determination ($R^2$) for the linear model is 0.21 whereas the $R^2$ for the quadratic model is 0.72. Hence, statistically, the actual data has almost no correlation with the linear model but has a strong correlation with the quadratic ones.

![Figure 3: $NRO$ of Two Query Types](image1)

(a) Best Linear Fit  
(b) Best Quadratic Fit

5.4 Model Comparison Using Two Query Types in 3D

The results from the model comparisons in 2D generalize in the case for a varying $MPL$. The above data were also used to train the two models on the entire region. Once more, the quadratic model fitted the data significantly better than the linear model. Figure 4a shows the triangularization of the data points and the best linear fit into the data, represented by a plane. Figure 4b shows the best quadratic fit into the data for the entire region, represented by a paraboloid. Again, it is evident that the quadratic fit represents the data a lot more accurately than the linear fit.

![Figure 4: $NRO$ of Two Query Types, Varying $MPL$](image2)

(a) Best Linear Fit  
(b) Best Quadratic Fit

As the number of query types and the MPL level increases, we discovered that the gap between the best linear fit and the best quadratic fit actually decreases. This fact justifies the use of a linear model in the
original implementation of QShuffler. However, the quadratic model still outperforms the linear model in our experiments. Further discussion and experimental results are presented on Section 8.1.

6 Sampling Method

In order to train the regression models, a training phase is needed. On average, it took about one hour for a query mix to complete when running until all queries have finished execution. So, after one hour we can find one training point. Thus, training is expensive since it requires that the mixes running are the only queries currently running in the database. Since there are 8 variables, the number of each type of query in the mix, the ”rule of thumb” requires at least $8 \times 10 = 80$ data points for training\cite{7}. Using the smart sampling technique described next, the training time for a $MPL = 30$ and $T = 8$ was 5 days, producing 120 training data. The length of this training makes clear the fact that it would be infeasible to actually calculate the cost of each query mix in the database, of which there are $\binom{M+T-1}{M}$.

6.1 Sampling over a Fixed $MPL$

One could make an argument that simply randomly sampling over the space of mixes would be sufficient to gather the training data needed. There are two issues of relying only on random sampling. First, the space of all possible mixes is very large, so it is not certain that random sampling will uniformly cover this domain. Second, there are particular interactions that we want to be sure to capture in our model.

Of the 120 samples, eight were corners, where exactly one query type was running in the database with a concurrency of 30. This is important not only for training the model, but also because the ratio of runtime of a query in these types of mixes to the runtime of the same query in isolation is a metric we used to classify a query as heavyweight or lightweight (see Section 7.2.2 for a full explanation). The next 35 samples capture interactions between pairs and triples of different queries. We randomly selected 14 of these pairs and 21 triples to include in the sample. The next datum used was one of the diagonals. We define a diagonal to be a mix such that there are an equal (or about equal) number of queries of each type running in the system. Finally, the remaining 76 samples were randomly generated from the sample space.

6.2 Sampling over a Varying $MPL$

When sampling over the domain of mixes with a bounded $MPL$, we considered the fact that the optimal $MPL$ was set by a database administrator in order to maximize the throughput of the system, without significantly affecting the average runtime of queries. Thus, we used the $MPL$ set by the administrator as the median allowed $MPL$. We sampled over the space of mixes with this optimal $MPL$ as described in Section 6.1. In addition, we sampled over the minimum and the maximum allotted $MPL$ in a similar way, but in a more scarce manner. Finally, we sampled over the entire domain of mixes of varying $MPL$ (omitting those mixes with an $MPL$ that was explicitly sampled).

7 Experiments

We now will describes the framework in which we implemented QShuffler.

7.1 Environment

This section describes the machine-specific information on which we ran our experiments.
7.1.1 Machines

Our experiments were run on a machine that is part of a research cluster at Duke University with dual 2.0GHz Intel® Zeon™ CPU’s, 2.0GB of RAM memory, running Linux Version 2.6.18-xenU (Debian 4.1.1-21).

7.1.2 Database System

The database server we used was DB2 v9.1.0.2. We used the TPC Benchmark™ H (TPC-H) database with a scale factor of 1GB. TPC-H is a decision support benchmark that is comprised by a set of queries that try to simulate business queries. The TPC-H tool "DBGEN" was used to generate all the data in our database. We used the DB2 Design Advisor to produce indexes for the TPC-H workload in order to ensure a proficient physical design but we used the default value for all other configuration parameters.

7.1.3 Scheduler Implementation

QShuffler is currently implemented as a stand-alone java application, separate from the database. The queries were placed in the arrival queue of the scheduler until they were chosen to be sent to the database. A set of threads were responsible for coordinating the concurrent execution of the queries. The number of active threads at any point in time represents the number of queries running in the system and hence it enforces the MPL level. The default value for the MPL in DB2 is 200, so the database never rejects or delays any of the queries sent by QShuffler.

7.2 Experimental Setup

In order to implement a query scheduler, we had to have queries and workloads that define the order in which the queries enter the system. This section describes how we were able to create these.

7.2.1 Query Generation

For our experiments we used 8 TPC-H query types from the 22 possible choices, shown in Figure 1. The TPC-H tool "QGEN" was used to generate all instances of the queries with different parameter values.

7.2.2 Workload Generation

We performed several experiments with several sizes. We focused on workloads with size 100 and 200 queries. In each workload, we used the same number of each query type. We also varied the sequence of the queries in those workloads because scheduling is sensitive to the arrival order of the queries. We employed two major schemes to do this. In the first schema, we arranged the query types in the order in which they are specified in the TPC-H benchmark, that is, in numerical order. Then, we went through this list in a Round Robin manner choosing $p$ queries of each query type and placing them in the arrival query of the scheduler. The parameter $p$ represents how many queries of each type will arrive together for scheduling creating imbalance in the system. We used $p = 5, 10$ and $25$ for our workloads with 200 queries and $p = 25$ for our workloads with 100 queries. In the second schema, we ordered the query types in a completely random way in order to test the handling of highly varying and unpredictable workloads. In addition, we also generated some more specialized workloads to test the variation of the MPL. We created some workloads with only heavyweight queries - that is queries we had identified as having bad interactions between them - and some with lightweight queries - that is queries we had identified as having good interactions between them. We also created a workload where the first half consisted of heavyweight queries and the second half consisted of lightweight queries and a workload with the opposite arrangement.

7.3 Experimental Parameters

There are several variables which we had control over in the code. These parameters are described below.
7.3.1 Multi-Programming Level (MPL)
The MPL represents the number of queries executed concurrently into the system. We decided to vary the MPL within a range relatively close to the optimal value of 30, determined by the paper [4]. Thus, we decided to set the maximum MPL to be 35 and the minimum to be 24. The main reason for this decision was training from a bigger region requires exponentially more sampling data points.

7.3.2 Threshold NRO ($\theta_{NRO}$)
This parameter represents the threshold NRO value used by QShuffler during scheduling. QShuffler selects the query that will induce the highest NRO value to the current mix below that threshold as described in Section 2.1.2. The value of $\theta_{NRO}$ depends on the database and the expected query workload, thus we set it based on experimentation. We run several experiments with different values for the $\theta_{NRO}$ and chose to use the value 0.85. Experiments with this value exhibited the lowest total completion time even though the variance between the other results was very low.

7.3.3 Lookahead ($L$)
This value refers to the size of the arrival query of the scheduler. It simulates the arrival of queries in a real system. Thus, the scheduler can only choose from $L$ possible queries which correspond to less than $L$ query types. For our experiments, we set $L = 12$ in order to limit the number of query types the scheduler had to choose from.

7.3.4 Number of Query Types ($T$)
The TPC-H benchmark offers 22 different query templates that could be used for our experiments. At the initial stages of our experimentations we performed several experiments using workloads containing queries from different types in isolation and workloads containing different mixes. We chose a representative sample of 8 query types that included short queries, long queries, queries with bad interactions between them and queries with good interactions between them.

7.4 Performance Metric
We compare the performance of the old QShuffler with our improved version by comparing the Total Completion Time of the workloads. Since the workloads are fixed, minimizing the total completion time is equivalent to maximizing the throughput.

8 Experimental Results
We now present our results using the performance metric specified in Section 7.4.

8.1 Linear Vs. Quadratic Regression Model
The first set of experiments we performed was aiming to compare the performance of the linear regression model against the quadratic regression model. We developed a new version of QShuffler that used the quadratic model instead of the linear model. Initially, we collected several training data using the sampling methods described in Section 6 and used the exact same data to estimate the regression coefficients of both models. The coefficient of determination ($R^2$) for the linear model is 0.568 whereas the $R^2$ for the quadratic model is 0.658. Statistically, the actual data have a higher correlation with the quadratic model rather than the linear model, even though the difference might not appear to be very large.

We performed 4 series of experiments with workloads of size 200. The difference between the 4 workloads lied in the ordering of the queries within them. The 3 workloads had a round robin arrangement with
parameter \( p = 5, 10 \) and 25, and the fourth workload had a random arrangement. Round robin and random workload generation are explained in detail in Section 7.2.2. For all experiments we kept the \( MPL \) constant, allowing for 30 queries to execute concurrently in the system. We also set the \( \text{lookahead} \) buffer to 12. Thus, both the schedulers had up to 12 queries in their arrival queue to choose from. Finally, we used \( \theta_{NRO} = 0.85 \) for all experiments. Therefore, the only difference between the two schedulers was the regression model they were using.

The experimental results we observed are summarized in Figure 5. The horizontal axis lists the 4 kinds of workloads used - random (Ran), round robin (RR) with parameter \( p = 5, 10 \) and 25. Note that the size of all workloads is 200. The vertical axis represents the total completion time of all the queries in the workload measured in seconds. In all experiments, the scheduler with the quadratic model performed better than the one with the linear model. The quadratic model exhibited the same behavior we observed in the 2D and 3D experimental results described in Section 5.3. It is possible that including the interaction terms in the quadratic formula or even using a higher degree polynomial, we further improve the performance of the system. However, in doing so we increase the risk of over-fitting the training data. In addition, it is possible that the overhead imposed to the system from the more complicated models will counteract the scheduling improvements.

8.2 Variable \( MPL \)

The main set of experiments we performed focused on dynamically varying the \( MPL \) value in the system. In other words, the scheduler had the flexibility to increase or decrease the number of queries running concurrently in the system, in order to maximize the performance of the system. The scheduler was not given complete freedom for the reasons discussed in Section 7.3.1. We decided that a reasonable region of the \( MPL \) for the scheduler to choose from was between 24 and 35. Of course this region can be readily expanded as long as the regression model is trained appropriately. We developed a different version of QShuffler that is able to vary the \( MPL \). In addition, we decided to deviate from the linear regression model used by the
original QShuffler since we saw that a quadratic model provides a better fit to the data.

We performed 3 series of experiments with workloads of size 100 and 200. The first batch of experiments consisted of workloads with a Round Robin (RR) arrangement with parameter $p = 5, 10$ and $25$. As the different sets of queries rotate through the system, the active workload changes in a periodic fashion. Figure 6 shows the patterns in the changes of $MPL$ that we observed. Round Robin causes the periodic changes in the $MPL$. From the graph we can see the repeated pattern of "heavyweight" queries arriving to the database, followed by "lightweight" queries. The terms "heavyweight" and "lightweight" queries refer to queries with bad and good interactions respectively. We should also note that the $MPL$ varies with a higher frequency for the RR5 workload compared to the RR10 workload as expected.

For the second batch of experiments we constructed more specialized workloads in order to explore the effects of significantly different workloads and test the robustness of the new QShuffler. We created some workloads containing only heavyweight queries and some containing only lightweight queries. We also constructed workloads in which the first half contained only heavyweight queries and the second half only lightweight queries, and vice versa. Figure 7 shows the changes in the $MPL$ for the latter 2 workloads. In the "LH" case (Lightweight queries first and then Heavyweight) we notice that the $MPL$ climbs up to the maximum allowed value of the $MPL$. As soon as the heavyweight queries arrive, the $MPL$ gradually drops down to the minimum allowed value and stays there for the remainder of the workload execution. In the "HL" case we saw the reverse pattern, that is, the $MPL$ first decreased and then somewhere in the middle of the experiment in gradually increased to the maximum value. Towards the end we noticed that the $MPL$ decreased again. This happened because there were still some heavyweight queries left in the arrival queue of the scheduler. When the first lightweight queries arrived in the queue, they were chosen over the remaining heavyweight queries, thus the heavyweight queries were essentially left behind in the queue until the end. This is a prime example of starvation. However, QShuffler concentrates on report generation workloads in which starvation is not considered an issue.

Figure 6: Varying $MPL$ with Round Robin Workloads
The final batch of experiments consisted of workloads generated with random sequences of queries. In this case, we observed some oscillations of the MPL but we didn’t notice any obvious patterns. In all three batches of experiments we recorded improvements over the overall completion time compared to the original implementation of QShuffler. The experimental results we observed are summarized in Figure 8.

Figure 7: Varying MPL with Specialized Workloads

Figure 8: Workload Completion Times
The horizontal axis lists 5 kinds of 200 sized workloads tested in our experiments - random, "HL" workload, "LH" workload and Round Robin (RR) with parameter $p = 5$ and 25. The vertical axis represents the total completion time of all the queries in the workload measured in seconds. We observed that the scheduler performs better in all cases but seems to offer significant improvement over workloads that vary a lot over time.

9 Future Work

Throughout the course of this project, we have discovered several areas which will require further investigation. For QShuffler-DCC, the threshold $\theta_{NRO}$ should probably reflect the number of queries that are currently running in the database, and so should change as the MPL is changing.

9.1 Varying Theta NRO

One of the issues with implementing QShuffler is that the threshold $\theta_{NRO}$ must be set apriori. Without an understanding of what this value actually means, finding this value requires some guess and check work for the database administrator, or whomever is implementing QShuffler. The average value for $\overline{NRO}$ is a potential candidate for the threshold; this idea was investigated and empiracally verified.

We wish to find the average estimated value for the $NRO$. Recall that the $NRO$ is estimated for mix $i$ by: $\overline{NRO}_i = w_0 + \sum_{j=1}^{T} w_j N_{ij}$. The domain $\overline{NRO}_i$ is the space of all possible mixes of $T$ query types running in a database with a fixed $MPL$:

$$R_T(M) = \left\{ m_i \ni \sum_j N_{ij} = M \text{ and } N_{ij} \geq 0, \forall j \right\}. \quad (4)$$

In order to calculate the average value of the function, we take take the integral:

$$\int \cdots \int_{R_T(30)} \overline{NRO} dm \quad (5)$$

and divide it by the volume of the domain. We calculate the volume of the domain $R_T(M)$ by the following formula:

$$V(R_T(M)) = \frac{M^{T-1}}{(T-1)!}. \quad (6)$$

This formula is easily verified by induction. The general formula for the average value of $\overline{NRO}_i$ is as follows:

$$\text{avg} \left( \overline{NRO} \right) = w_0 + \frac{M}{T} \sum_{i=1}^{T} w_i \quad (7)$$

Another model that we could use is a quadratic model (Section ??):

$$\overline{NRO}_i = w_0 + \sum_{j=1}^{T} w_j * N_{ij} + \sum_{j=1}^{T} v_j * N_{ij}^2.$$

Using this model, the average value becomes:

$$\text{avg} \left( \overline{NRO} \right) = w_0 + \frac{M}{T} \sum_{i=1}^{T} w_i + \frac{M^2}{T(T+1)} \sum_{i=1}^{T} v_i \quad (8)$$
For fixed $MPL = 30$ and $T = 8$, we found the average $\hat{NRO}$ to be 0.87 and using the quadratic model, we found the average $\hat{NRO}$ to be 0.85, which corresponds to the value that we have experimentally found to be optimal.

10 Conclusion

QShuffler is a query scheduler that is capable of taking query interactions into consideration in order to improve the overall performance of the database. Using a multi-dimensional linear regression model, the scheduler estimates the cost of all queries in the arrival queue before determining which one to execute. We proposed an improvement to the performance model used by the QShuffler online scheduling algorithm that will allow for a varying number of concurrent queries to be executed. Using smart sampling, we were able to overcome the challenge imposed by increasing the dimensions of our mix space.

We also investigated an algorithm that allowed for a varying $MPL$. All of the experimental results we obtained demonstrated that a varying $MPL$ could help to improve overall completion times of a report generation workload. One issue that arose while running our experiments was that a linear fit no longer seemed appropriate. Even though the results obtained were satisfactory to the developers, a linear fit does not accurately estimate the $NRO$, especially when we allow the $MPL$ to vary. Thus, we decided to implement a quadratic model in order to more closely capture the cost of mixes. The results we got were consistently better than a linear model.

References


# A Tables

The following table defines the variables and acronyms used throughout this paper:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{ij}$</td>
<td>Average Runtime of Query $j$ in Mix $i$</td>
<td>Section 2.1.1</td>
</tr>
<tr>
<td>$L$</td>
<td>Lookahead. The number of queries in the Arrival Queue.</td>
<td>Section 7.3.3</td>
</tr>
<tr>
<td>$m_i$</td>
<td>Mix $i = &lt;N_{i1},\ldots,N_{iT}&gt;$.</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>$M,MPL$</td>
<td>Multi-Programming Level. For our Experiments</td>
<td>Section 7.3.1</td>
</tr>
<tr>
<td>$N_{ij}$</td>
<td>Number of Queries of Type $j$ Running in Mix $i$.</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>$NRO_i$</td>
<td>Normalized Runtime Overhead of Mix $i$.</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>$\hat{NRO}_i$</td>
<td>Estimated Normalized Runtime Overhead for Mix $i$.</td>
<td>Section 5.1</td>
</tr>
<tr>
<td>$R_T(M)$</td>
<td>Space of Mixes with $T$ Query Types and fixed $M$</td>
<td>Equation 4</td>
</tr>
<tr>
<td>$T$</td>
<td>Number of Query Types. For our experiments, $T = 8$</td>
<td>Section 2.1</td>
</tr>
<tr>
<td>$t_j$</td>
<td>Average Runtime of Query $j$ Running in Isolation</td>
<td>Section 2.1.1</td>
</tr>
<tr>
<td>$\theta_{NRO}$</td>
<td>The threshold. Desired maximal $NRO$.</td>
<td>Section 2.1.2</td>
</tr>
</tbody>
</table>