Fair, Effective, Efficient and Differentiated Scheduling in an Enterprise Data Warehouse

Chetan Gupta, Abhay Mehta, Song Wang, Umesh Dayal
Hewlett-Packard Labs
firstname.lastname@hp.com except for songw@hp.com

ABSTRACT

A typical online Business Intelligence (BI) workload consists of a combination of short, less intensive queries, along with long, resource intensive queries. As such, the longest queries in a typical BI workload may take several orders of magnitude more time to execute, compared with the shortest queries in the workload. This makes it challenging to design a good Mixed Workload Scheduler (MWS). In this paper we first define the design criteria that make a 'good' MWS. We then use these criteria to design rFEED, a MWS that is fair, effective, efficient, and differentiated. We simulate real workloads and compare our rFEED MWS with models of the current best of breed commercial systems. We show that the rFEED MWS works extremely well.

1. INTRODUCTION

Many organizations are creating and deploying Enterprise Data Warehouses (EDW) to serve as the single source of corporate data for business intelligence (BI). Not only are these EDWs expected to scale to enormous data volumes (hundreds of terabytes), but they are also expected to perform well under increasingly mixed and complex workloads, consisting of batch and incremental data loads, batch reports and complex ad hoc queries. A key challenge for an EDW is to manage complex workloads to meet stringent performance objectives. Workload management is the problem of scheduling, admitting and executing queries and allocating resources so as to meet these performance objectives.

1.1 BI Workload Characteristics

A typical distribution of BI queries is shown in Figure 1. In this figure, we have taken one day's worth of queries (about 50,000) from an actual large EDW. The query sizes (execution times) have been binned on the x-axis, and the frequency of each bin is plotted on the y-axis. As can be seen from the distribution, the majority of the queries are small in size (they take a short amount of time to run). A small number of queries are very large in size. The largest query is over 10,000 times larger than the smallest query. This is a classic heavy-tailed distribution [14, 21].

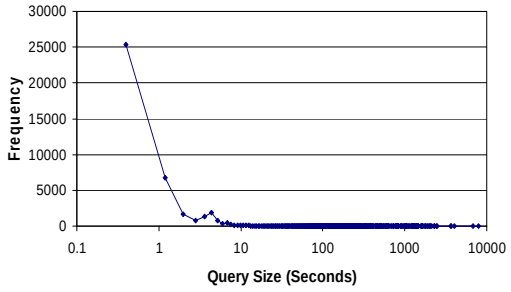

Figure 1: Distribution of Query Sizes

Furthermore, we can understand a typical BI workload by means of the Pareto principle or the 90-10 rule. If we assume that the execution time of a query is an approximation of the amount of load that the query puts on the system, then 10% of the largest queries make up approximately 90% of the load on the system. This can be seen in Figure 2, where the x-axis represents the queries arranged in descending order of size and the y-axis represents the cumulative load placed on the system. To compute the load placed on a system by a single query, we took its execution time (in seconds) and divided it by the sum of the execution times of all the queries in the workload (also in seconds). The workload was defined by all the queries that were run in a specific twenty four hour period.

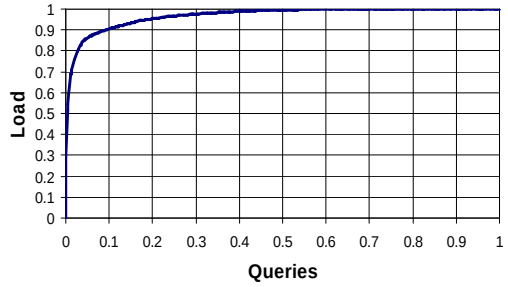

Figure 2: Distribution of Workload Across Queries

These two characteristics of BI workloads are in sharp contrast to traditional OLTP systems which have been the main topic of study in the past, where most of the transactions are

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

of similar sizes and the load is uniformly distributed across the queries. This makes it particularly challenging to design a 'good' Mixed Workload Scheduler (MWS) for the complex BI workloads of tomorrow.

1.2 Desired Properties of an MWS

It is important to clearly define what 'good' means in the context of an MWS. 'Good' can mean that the workload scheduler is *fair* in that no query starves for resources. 'Good' can also mean that the workload scheduler is *effective*, in that it reduces the average execution time of all queries. 'Good' can also mean that the workload scheduler is *efficient*, in that it does not place a large overhead on the system to run. Finally, 'good' can mean that the workload scheduler is *differentiated*, in that it is service level aware and can provide different levels of service to different queries. Together, we call these criteria the FEED criteria for Fair, Effective, Efficient, and Differentiated.

Note that providing different service levels for different types of queries is an important requirement of a good MWS. A higher service level means that queries assigned to that level will experience better average performance characteristics compared with queries assigned to lower service levels. Typically, batch queries have lower service level objectives than interactive queries. Additionally, the service level objectives for an interactive query may depend on a user's role and position in the corporation.

The four different FEED criteria mentioned above are all important in a 'good' MWS. Notice that these four criteria are often at odds with each other. One can design a 'fair' system by executing in a 'first in first out' (FIFO) mode, but this would not be 'effective' because a short query stuck behind a long one might have to wait for a long time. Similarly, one can design an 'effective' system by executing queries in a 'shortest job first' (SJF) scheme, but this might cause some large queries to starve and never get executed. Similarly, one can design a 'differentiated' system by always giving a higher preference to the queries that belong to a higher service level. However, this system might be neither 'fair', nor 'effective'. In this paper we will systematically study the tradeoffs between these different criteria, and design a 'good' MWS for heavy-tailed BI workloads.

1.3 Studying Stretch

Another key differentiator between past research on OLTP systems and our work on BI systems is the perspective that is used to evaluate the system. In OLTP systems, the key metric used to evaluate the performance is the average response time, or equivalently, the flow of a set of transactions [2]. This is often reported as a throughput number (queries per hour). Most OLTP transactions are similar to each other, so an aggregate measure representing the system perspective of performance is sufficient.

On the other hand, in BI systems, it is the user perspective, not the system perspective that is important. In talking to end users and Database Administrators (DBAs), we have concluded that a better metric to use, to evaluate the system from the user perspective is 'stretch', not 'throughput'. The 'stretch' of a query is the ratio of observed processing time to the ideal processing time. Within a service level, 'stretch' captures the idea of fairness from a user perspective: if a user has a small query he/she will expect a quick response, whereas if the query is large he/she might be willing to wait proportionally longer to get results.

If there is a large variation in the execution time of queries as is the case with BI workloads, the order of execution may not have a large effect on the average response time of the queries, but it may have a big impact on the average stretch of the queries. The example shown in Figure 3 helps illustrate this effect (For this illustrative example, we assume a single queue and a single threaded server).

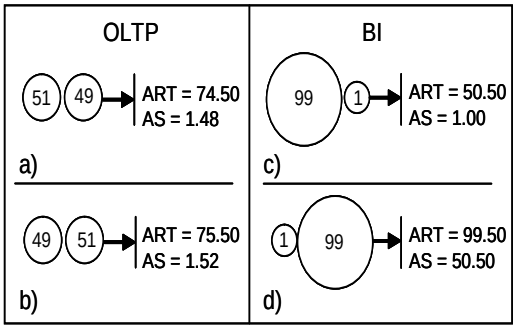

Figure 3: Average Response Time and Average Stretch of Different Workloads

In workload 'OLTP', there are two queries of size 49 and 51. If the shorter query goes in first (Figure 3.a), the average response time (ART) is 74.50 and the Average Stretch (AS) is 1.48. If the longer query goes in first (Figure 3.b) the ART is 75.50 and the AS is 1.52. Since both queries are approximately the same size, the order has only a small effect on both the ART and the AS.

In the second workload 'BI', the two queries are of size 1 and 99. If the short query goes in first (Figure 3.c), the ART for the workload is 50.50. If the long query goes in first (Figure 3.d), the ART roughly doubles to 99.5. Contrast this to the increase in stretch, which is much more pronounced: If the short query goes in first (Fig 3.c), the average stretch is approximately 1. However, if the long query goes in first (Fig 3.d), the average stretch is 50.50. So, although the reordering of queries caused only a factor of 2 increase in average response time (system perspective), it caused a 50 fold increase in the average stretch (user perspective).

Another advantage of using stretch is that typically, minimizing stretch will also result in lower values of response time, since stretch essentially is response time divided by the expected execution time. In our experimental section, we show that our approach works well for response time too.

Most of the past work studied average response time or throughput, which measures the system performance from the system perspective. In this research, we will be focusing on the user perspective (stretch metric), which is extremely relevant in BI systems, where the query distributions are heavy-tailed and where the ratio between the long queries and the short queries are very large.

1.4 Varying BI Workloads

A challenge with the past and current mixed workload schedulers is that they require a significant amount of manual input and tuning, which in turn depends on the expected mix of the workload. This is a big challenge in typical BI environments because the mix of the workload can vary widely and can change continuously. This dynamically changing nature of the workload mix makes it very challenging, if not

impossible for a human to manually tune the mixed workload scheduler.

1.5 Our Solution

We address all of the above challenges by designing a new MWS, called rFEED. rFEED is fair, effective, efficient, and differentiated. rFEED is parameterized and we compute the various parameters for practical BI workloads, so it can run without any tuning or manual intervention. We simulate workloads and compare our rFEED MWS with models of the current best of breed commercial systems. We run experiments and show that the rFEED MWS outperforms these methods. Overall, in terms of the average stretch of queries, rFEED does up to 120 times better than the current best-of-breed models. We make four main contributions in this work:

1. We identify the important design criteria for a good Mixed Workload Scheduler (MWS).
2. We design a MWS based on these criteria and compute the various parameters needed to implement this design for complex, mixed BI workloads.
3. We model the current best-of-breed workload scheduler, and compare them with rFEED.
4. We show that rFEED does not need any manual tuning, and works very well even with dynamically changing, mixed workloads.

In section 2, we discuss related work. In section 3 we identify the important design criteria. Then in section 4 we present the design of our MWS. We first describe our MWS, and then we compute the various parameters needed for the MWS. In section 5, we describe our experiments. We also model current best of breed workload management systems and then we go head to head against the best of breed models, and show that our MWS does extremely well. Finally we summarize and conclude in section 6.

2. RELATED WORK

The scheduling work in academic settings aims towards proving hard bounds for a metric of interest, for example minimizing the average value of stretch. If precise bounds are difficult, approximation results are presented with worst case guarantees. On the other hand, the work in industrial settings in DBMSs aim towards a more hands-on approach where the user specifies various parameters based on experience.

In the academic setting there has been much work in the area of scheduling [6,18]. We are interested in online scheduling, i.e., the query properties are not known in advance. Online scheduling is discussed extensively in [18]. Scheduling is also studied as preemptive and non-preemptive. We take a non-preemptive approach to scheduling since it can be expensive to preempt really small queries that make the bulk of a BI workload.

The scheduling literature focuses on minimizing some metric as a measure of goodness. In our case we are interested in stretch. Two related metrics are flow, which measures the amount of time an item spends in the system and weighted flow, where the flow is weighted with some quantity. Stretch, can be thought of as a special case for weighted flow, where flow is weighted with the processing time. For both these metrics, minimization of the average and the maximum value has been studied. We summarize the work with regards to these metrics below.

It is a well-known result that the Shortest Job First (SJF) minimizes the average value for *flow* and First In First Out (FIFO) minimized the maximum value for flow. The *weighted case* [2,9,10] of flow is known to be NP-hard even on a single machine.

The metric we are interested in is the *stretch* metric. The stretch metric was first analyzed by Bender [4] in the context of scheduling. They proved that no online algorithm can approximate the maximum stretch to within a factor of $O(n^{0.5-\epsilon})$ unless $P = NP$ for the non-preemptive case. Their algorithm though, requires them to know the actual value of max stretch and is not sublinear in complexity. In a further development [5], they describe an algorithm for max stretch that has a much lower complexity then their previous work. Maximum stretch is also considered in Legrand [17] and they provide a heuristic for the online multiprocessor case. Bender et. al. [5] also provide a $1+\epsilon$-polynomial time approximation scheme for average stretch. Muthukrishna et. al. [20] provide some bounds for the average stretch.

Our approach is different from these in two significant ways: (i) We do not wish to just minimize the average or the max stretch, but try to strike a balance between the two, i.e., we don't just want to minimize the average stretch but while doing that we also want to control for the maximum value of stretch; (ii) Also, unlike these past studies we address the issue of different service levels.

In the industrial setting, scheduling has been studied extensively in various scenarios from job shop scheduling to operating systems. Various heuristics like *Most Requests First*, *First Come First Served* and *Longest Wait First* were considered in wireless context by Kalyan [16], in web servers by Friedman [13] and Crovella [12]. Another interesting piece of work is by Bedekar [3] in the context of CDMA. Our approach and our results are different, since we study stretch in the context of DBMS towards meeting the FEED requirements. Query suspend and resume [8] has been used as a internal scheduling strategy in DBMS. Such techniques can be used in conjunction with rFEED.

In the database scenario, scheduling has been considered by Schroeder [22]. Another study that talks about scheduling but in terms of multi-query optimization and operators is that of Sharaf [23]. Like [23] we are interested in the non-preemptive case of the related problem of stretch.

Our scheduling function uses a composite function of processing time and wait time of the queries to be scheduled. A similar idea was used in the context of web scheduling in [11], called alpha-scheduling. Different from their work, we study the stretch of a query. More importantly, we introduce different service levels and suggest specific values for the constants in our rank function.

In the industrial DBMS context, IBM's workload scheduler is called Query Patroller [15]. Different from our proposed approach, in Query Patroller a user (or an administrator) has to specify a number of parameters. They could be: the number of large queries that can be run, the maximum number of queries per user etc. This is done in a mixed workload setting to mitigate the adverse impact of large queries on the smaller queries. Teradata's workload scheduler [7] involves specifying particular time shares for

different users based on service levels and their needs. Their approach also requires manual intervention. Oracle deploys an approach similar to that of the IBM query patroller.

To the best of our knowledge, there is no existing work that studies stretch in the context of a system being both fair and effective (as discussed later, this requirement can be formalized to an l_2 norm) while at the same time providing differentiated service to different service levels. (A shorter version of this paper is slated to be published [19]). We aim to satisfy these requirements while eliminating the need for manual intervention by data base administrator.

3. MIXED WORKLOAD SCHEDULER

As mentioned previously, we will use stretch as the metric of choice to study our scheduling approach.

DEFINITION 1. *The stretch of a query j at time t is:*

$$S_j = \frac{t - a_j}{p_j} \tag{1}$$

And when $t = c_j$, we get the final stretch for a query:

$$S_j = \frac{c_j - a_j}{p_j} = \frac{w_j + p_j}{p_j} \tag{2}$$

Where t is the current time, c_j is the completion time of a query, a_j is the arrival time and p_j is the execution time of a query and w_j is the wait time time of the query.

As we discussed before stretch captures the idea of fairness from a user's perspective, i.e., a user of a short query expects a short response time and a user of a large query expects a large response time.

REMARK 1. *By minimizing the stretch of a query we mean minimizing the final stretch. In our text, we use stretch instead of final stretch unless the meaning is not clear from the context.*

DEFINITION 2. *By the size of query q we mean the processing time p_q of the query.*

REMARK 2. *In this work when we say* processing time*, we mean the time it would take a query to execute if it was run by itself on a DBMS. Prediction of the execution time is a complex problem. In absence of a mechanism for predicting execution time precisely, the optimizer's estimated execution cost can be used as the processing time. In the experimental section, we introduce error in execution time to account for errors in optimizer's cost estimate. Our approach is independent of the method of computing the expected execution time and more complex execution time estimation approaches can be adopted.*

We want to create a mixed workload scheduler (MWS) that follows the FEED properties:

Fair: In an unfair system, queries will have very different values for stretch, i.e., some queries might have a very small value for stretch, whereas others might have a very large value for stretch. A way for obtaining a fairer DBMS is for the MWS to minimize the maximum value of stretch for the queries in the system.

Effective: Effectiveness for a system can be measured by the average value for stretch. A way to make the DBMS more effective then, is for the MWS to minimize the average stretch for queries in the system.

Efficient: In an efficient online system the complexity of the scheduling algorithm should be low otherwise the algorithm might be too expensive to use in a real life scenario. A way for making an MWS more efficient is to make the implementation sub-linear in complexity.

Differentiated: In a differentiated DBMS, different query types and users (based on their importance, needs etc.) should be offered different performance in terms of the average value of stretch, i.e., queries with higher service level requirement should have a lesser stretch than a corresponding query with a lower service level requirement. A way of making an MWS differentiated is to incorporate service level requirements directly into the scheduling function.

4. rFEED MWS DESIGN

As mentioned earlier, the challenge for any good MWS is to achieve these goals simultaneously, since these goals can be at odds with each other. We call our approach rFEED since it meets the FEED criterion through a rank function.

We have an external scheduling mechanism that decides on which query is to be let into the DBMS. (Single queue can serve multiple servers.) At all times we maintain a *single queue* L of queries outside the DBMS. Every query $q \in L$ is assigned a rank R_q. At the time of execution the query with the highest rank is admitted into the DBMS for execution. Precisely:

DEFINITION 3. *Let R_q be a rank assigned to a query. Then at any time, the rFEED scheduling scheme first admits for execution the query q with the highest rank R_q in the queue of queries L.*

Unlike traditional designs, we maintain a single queue of queries ordered on the rank function. Having multiple queues can lead to the local optimization of some metric (for example stretch or flow) for every queue. For global optimality we would require to consider all the queries in the various queues. Since, global optimality is the desired outcome, we are better off having a single queue. Note that while the single queue determines the order in which queries are admitted into the DBMS, the execution can have multiple queues for a parallel DBMS.

REMARK 3. *In the discussion below we assume without any loss of generality that the minimum possible query size is 1. This can be done by dividing the processing time of each query with that of the smallest query. We will use ψ for the processing time of the query with the largest processing time.*

4.1 Rank Function

We denote the rank of a query q with R_q and we aim to achieve the FEED goals simultaneously through the use of a rank function.

4.1.1 Effectiveness

Our first aim is to make our scheduling effective, i.e., we want to minimize the total value for stretch for a set of queries (which is the same as minimizing the average value).

Consider two queries, with processing times p_1 and p_2 and wait times w_1 and w_2 respectively. We need to decide their order such that the total stretch is minimized. Let the order be: first p_1 and then p_2. Then the total stretch S would be:

$$\frac{p_1 + w_1}{p_1} + \frac{p_2 + p_1 + w_2}{p_2} \implies 2 + \frac{w_1}{p_1} + \frac{w_2}{p_2} + \frac{p_1}{p_2} \tag{3}$$

In Equation 3, only the last quantity depends on the order and hence to minimize the above quantity, p_1 should be less than p_2. This gives us our first rule: give a higher rank to the query with the lesser execution time.

To give the highest rank to the shortest query we take the inverse of the processing time. Then the rank becomes:

$$R_q = \frac{1}{p_q} \tag{4}$$

Where, p_q is the processing time for a query q.

Equation 4 is nothing but Shortest Job First or SJF. (As discussed in the previous section, SJF is known to be provably optimal for minimizing total flow).

4.1.2 Fairness

We want our scheduling scheme to be fair. This means that a larger query, if it has waited for some time, should have a higher rank (and have precedence) compared with a similar sized query that has just arrived. Hence, to discuss fairness we study queries with the same processing time but different waiting times. Mathematically, fairness can be thought of as an attempt to minimize the maximum stretch for any query.

Consider a set of queries with waiting times $w_1, w_2, \ldots, w_n$ and the processing time equal to 1. If they are executed in the order, $1, 2, \ldots, n$, then their stretches would be: $1 + w_1, 2 + w_2, \ldots, n + w_n$. To minimize the maximum value of stretch, we need to execute the query with the highest wait w_{max} first, since if it is kept waiting the highest stretch would become w_{max} plus some positive quantity.

With this insight, we add a component to Equation 4 for waiting time, i.e., the higher the wait time, the higher the rank should be.

$$R_q = \frac{1}{p_q} + Kw_q \tag{5}$$

Where, w_q is the waiting time for a query and K is some constant. For $K > 0$, the longer the query waits, the higher its rank becomes. We add the waiting time and not perform some other operation since we want our equation to be linear in waiting time with a constant slope for efficiency. We discuss this in detail in a section 4.1.4.

Later in the text, we discuss how to assign the value of K such that the two competing goals of fairness and effectiveness can be balanced.

4.1.3 Differentiation

Our third goal is to provide differentiated service levels to queries based on their QoS requirement, i.e., we want the query with higher service levels to have a higher rank. This can simply be achieved by multiplying the inverse of processing time in Equation 5 by the service level requirement to get:

$$R_q = \frac{\Delta_q}{p_q} + Kw_q \tag{6}$$

Where, $\Delta_q \geq 1$ is the normalized service level for a query q.

The number of distinct values of Δ_q is equal to the number of distinct service levels and every query belonging to the same service level will have the same Δ_q. For the queries at the lowest service level the value of Δ_q is 1 and a higher service level queries will get higher Δ_q values.

Equation 6 indicates that, for two queries arriving at the same time and having the same processing time, the query with the higher Δ_q will have a higher rank. The effect of multiplying by the normalized service level is to shift the rank of queries of different service levels by Δ. We explain with the help of an example.

EXAMPLE 1. *Imagine that the there are two sets of queries with average processing times of 1 and 10 respectively and all of them arrive at the same time. If they have the same Δ_q, the smaller queries on average will be executed first. If the smaller queries have $\Delta_q = 1$ and larger queries have $\Delta_q = 10$,then the two sets will be executed interchangeably. If the smaller queries have $\Delta_q = 1$ and larger queries have $\Delta_q = 100$, the larger queries will be executed first.*

There is no need to manually compute the Δ_q corresponding to the service level. Later on in the text we explain how to obtain a value of Δ_q given a Quality of Service (QOS) requirement.

4.1.4 Efficiency

In an online setting, the scheduling algorithm should have sub-linear complexity. In our context, we cannot afford to sort the queue of queries to obtain the highest rank query every time a new query is to be executed. (For example, on a normal day our customer EDW received more than 67000 queries. Such a large number of queries can easily result in a very long waiting queue).

To avoid this, a *priority queue*, which is a well known data structure, can be used. If implemented as a heap, it has $O(logn)$ complexity for both insertion and updating of the queue.

In rFEED, we use the rank function to insert queries into the priority queue. Every time a new query q comes in, we insert the new query in the order of its rank R_q. To insert a query in order, for the new query q, we compute its rank R_q and also of those queries with which it is compared during insertion. For a new query the waiting time is zero, i.e., $t = a_q$. Then, $R_q = \frac{\Delta_q}{p_q}$, and for the existing queries, the rank of each query can be computed as $R_q = \frac{\Delta_q}{p_q} + Kw_q$, where w is the waiting time for that query.

Typically, in a priority queue, the priority of an inserted element does not change after insertion. In our case, since the rank is a function of time, the rank of every query changes with time. However, since the rank is linear in time and changes at the same rate K for all the queries, the relative order of queries does not change, i.e., if query a is ahead of query b at the time of insertion, it will continue to remain so. Similarly, if a query a is behind query b at the time of insertion, it will continue to remain so. In other words, the queue maintains the order of queries once the queries are inserted.

This means that, as time progresses we do not need to update the rank of every query but only of those queries with which the query to be inserted is to be compared. As with priority queues in general, this gives us $O(logn)$ complexity.

4.1.5 Discussion

The idea behind the rank function is to look at both the processing time and wait time in a single equation so that while the smaller queries are executed quickly, the larger queries do not have to wait infinitely to get executed. This is done so that the minimization of the average value for stretch does not lead to a large increase in the maximum

value of stretch. For rFEED to be efficient, the fact that our rank equation is linear in wait time gives us nice properties, in that it can be converted to a sub-linear algorithm using priority queues. Finally, multiplying the rank function with Δ_q has the effect of giving a higher rank to queries coming from higher service levels.

4.2 Computing K for Δ_q

Based on our value of K, our rank function generalizes some well known scheduling functions. If $K = 0$, then the algorithm becomes Shortest Job First (SJF). Having a non-zero K ensures that ranks of the larger queries increases and at some point their ranks starts becoming greater then new smaller queries. For values of K greater than some constant, the algorithm becomes First In First Out (FIFO), i.e., queries are scheduled without regard for their processing times.

For the analysis below, recall that the shortest possible waiting time and smallest query size is 1 and the size of the largest query is ψ. For ease of illustration, in the discussion below we assume that all queries have the same normalized service level.

Our scheme will behave as a FIFO scheme when the largest query with waiting time of 1 has a higher rank than a newly arrived smallest query. In that case, using Equation 6:

$$\frac{1}{\psi} + K \times 1 > \frac{1}{1} + K \times 0 \implies K > 1 - \frac{1}{\psi} \tag{7}$$

So for all values of $K > 1 - \frac{1}{\psi}$, our scheme will behave as a FIFO queue.

Equation 7 includes the largest query size, which is unknown in advance. In practice a reasonable estimation based on experience is used. In our experiments we have shown that the precise value is not necessary but an approximation is sufficient.

In the related works section we mentioned that SJF is considered optimal for minimizing the total flow and FIFO minimizes the maximum flow. In other words, $K = 0$ would give us an effective algorithm and $K = 1 - \frac{1}{\psi}$ will give us a fair scheme. So, if we use a value of K such that $0 < K < 1 - \frac{1}{\psi}$, we should get a scheme that is fair and effective.

Now we show how to compute K to achieve our goal of fairness and effectiveness in the context of stretch. For the rFEED scheme, query b supersedes another query a iff, $R_b > R_a$, i.e. From Equation 5 (Assuming $\Delta_a = \Delta_b = 1$):

$$\frac{1}{p_a} + Kw_a < \frac{1}{p_b} + Kw_b \implies K < \frac{p_a - p_b}{p_b p_a} \frac{1}{w_a - w_b} \tag{8}$$

We first look at effectiveness. Imagine that the query of the largest size ψ arrives at time $t_q = t$. For effectiveness, we want any query that arrives at time $t + \delta_t$ (for ease of argument we assume $\delta_t \geq 1$) to have a higher rank than the largest query. In Equation 8, assuming query a is the one with the size ψ and query b arrives at time $t + \delta_t$, we get (since query b arrives later $w_a - w_b = \delta_t$):

$$K < \frac{\psi - p_b}{\delta_t \psi p_b} \tag{9}$$

In Equation 9, as the value of p_b increases, the value of K decreases. To ensure that all queries are able to overtake the largest query with size ψ, we need the lower bound for K, which we can obtain with $p_b = \psi - 1$, the size of the second smallest query. With this we get the first bound for K:

$$K < \frac{1}{\delta_t \psi(\psi - 1)} \tag{10}$$

In Equation 10, as the value of K decreases the system becomes more and more unfair. For example, in Equation 10, if we were to stipulate that any query b, even if it arrives at time $t + 2$, should be able to overtake query a of size ψ arriving at time t, i.e. $\delta_t = 2$. We get: $K < \frac{1}{2\psi(\psi - \epsilon)}$. This means that as $\delta_t \to \infty, K \to 0$.

This follows the intuition that as K tends to zero the system behaves more and more like a SJF system, which though being highly effective is also very unfair. To obtain a fair behavior while maintaining effectiveness, we equate K to its upper bound, which can be obtained by $\delta_t = 1$. Hence:

$$K < \frac{1}{\psi(\psi - 1)} \approx \frac{1}{\psi^2} \tag{11}$$

Equation 11 means that we take K to be the inverse of the square of the largest query size, where query sizes have been normalized by the size of the smallest query.

Since $\psi > 1$, $\frac{1}{\psi(\psi-1)}$ is always greater than $1 - \frac{1}{\psi}$, hence form Equation 7 rFEED MWS does not behave as a FIFO and since $K \neq 0$, rFEED never behaves as an SJF.

REMARK 4. *In this discussion ψ, is assumed to be the largest query size historically. In practice (as we show in the experiments), it is not required that ψ be known precisely, but an approximation would suffice.*

4.2.1 Avoiding Starvation

A problem with scheduling algorithms can be starvation, i.e., the query never gets to execute. Shortest Job First (SJF) which is known to be optimal for the non-preemptive case of average flow time can suffer from this problem in an online scenario as some long running job might never get a turn to execute. Our rFEED never causes starvation. The rank of a newly arrived query is $\frac{\Delta_q}{p}$. This will be largest for the query with the smallest processing time, $p = 1$, and the largest value of $\Delta_q = \Delta_{max}$ in which case the rank will be Δ_{max}. The longest time a query may wait before its rank becomes Δ_{max} is:

$$\frac{\Delta_q}{p} + Kw = \Delta_{max} \implies w = \frac{1}{K}(\Delta_{max} - \frac{\Delta_q}{p}) \tag{12}$$

The largest value for waiting time w from Equation 12 is $\frac{1}{K}(\Delta_{max} - \frac{\Delta_q}{\psi})$, where ψ is the size of the largest query. After waiting for this long a query q will necessarily have a higher rank than any incoming query and will be the first one to get executed.

4.3 Mapping Service Levels

Now, we show how service levels can be mapped to Quality of Service (QoS) requirements, i.e., we map Δ_q in Equation 6 to QoS.

Service level requirements can come in various forms but very often they are specified in terms of system performance, i.e., a query a from a higher service level should receive r times the system resources as compared to a query b from a lower service level. This directly translates to query performance, i.e., for two same sized queries a and b, query a from a higher service level will have a stretch that is $\frac{1}{r}$ times the stretch of query b from a lower service level. In the discussion below, we discuss how to achieve this type of service

level requirement, i.e., what should the value of Δ_q be, for a query to achieve a particular QoS.

We take the case of two normalized service levels 1 and Δ. Let there be N_1 queries whose normalized service level is 1 and N_Δ queries whose normalized service level is Δ. Let's consider a query with execution time slightly greater than $p+\epsilon$, where ϵ is an arbitrary small number. Having a normalized service level $\Delta_q = \Delta$ means that the rank of a newly arrived query of size p from $\Delta_q = 1$ is the same as the rank of a newly arrived query of size $p\Delta$ from $\Delta_q = \Delta$.

The ratio r of stretches of a query of size $p+\epsilon$ for the two different service levels will give us a way to map Δ to service level requirements. (In the analysis below all terms with subscript Δ mean that they have $\Delta_q = \Delta$ and with subscript 1 mean that they have $\Delta_q = 1$).

From Equation 1 we know that the stretch requires us to know the waiting times and the processing times. We base our analysis on the assumption that all queries arrive at the same time. Then, the waiting time of a query is the sum of the execution times of all the queries that would be executed before it based on the rank function. For example, assume there are three queries $\{q_1, q_2, q_3\}$ with execution times $\{1, 2, 3\}$ respectively, at the same service level. According to the rank function their order of execution is $\{q_1, q_2, q_3\}$. Then the wait time for q_1 is 0, for q_2 is 1, and for q_3 it is $1+2=3$. Once the waiting time is computed, computing the stretch is straightforward.

Now, we consider a query of size $p+\epsilon$. For a query with $\Delta_q = 1$, the total waiting time for a query of size $p+\epsilon$ will be the sum of the following two quantities:

1. The total execution time of all queries with $\Delta_q = 1$, which have a processing time less than or equal to p. Let this number be $T_{1,p}$.
2. The total execution time of all queries with $\Delta_q = \Delta$, which have a processing time less than or equal to $p\Delta$.

   To compute this, we look at the rank function. A query with $\Delta_q = \Delta$ is executed before query with $\Delta_q = 1$ if $R_\Delta > R_1$. Now, since the waiting time at the time of insertion is zero for all queries, this means: $\frac{\Delta}{p_\Delta} > \frac{1}{p_1} \implies p_\Delta < p_1\Delta$.

   This means that all queries with normalized service level Δ with size up to and including $p\Delta$ will be executed before the query of size $p+\epsilon$ from the service level with normalized service level 1. Let the total execution time of these queries be $T_{\Delta,p\Delta}$

Similarly, for a query with $\Delta_q = \Delta$, the total waiting time will be the sum of the following two quantities:

1. The total execution time of all queries with $\Delta_q = \Delta$, which have a processing time less than or equal to p. As before, let this number be $T_{\Delta,p}$.
2. The total execution time of all queries with $\Delta_q = 1$, which have a processing time less than or equal to $\frac{p}{\Delta}$.

   A query with $\Delta_q = 1$ is executed before query with $\Delta_q = \Delta$ if $R_1 > R_\Delta$. Now, since the waiting time at the time of insertion is zero for all queries, this means: $\frac{1}{p_1} > \frac{\Delta}{p_\Delta} \implies p_1 < \frac{p_1}{\Delta}$.

   This means that all queries with normalized service level Δ with size up to and including $\frac{p}{\Delta}$ will be executed before the query of size $p+\epsilon$ from the service level with normalized service level 1. Let the total execution time of these queries be $T_{\Delta,\frac{p}{\Delta}}$.

The ratio r of the stretches for a query of size $p+\epsilon$ with normalized service level Δ to that of a query of size $p+\epsilon$ with normalized service level 1 is ($\epsilon \to 0$):

$$r = \frac{T_{1,p} + T_{\Delta,p} + p}{T_{\Delta,p} + T_{\Delta,\frac{p}{\Delta}} + p} \tag{13}$$

It is important to note that the quantity $T_{\Delta,\frac{p}{\Delta}} = 0$ for all $\frac{p}{\Delta} < 1$, i.e., no queries exist whose size is smaller than 1.

The distributions of the query can be modeled using some distribution and a closed form for Δ can be obtained. For instance, for Pareto distribution (with Minimum Value 1 and Pareto Index 1) using the same analysis as above we obtain (For the full derivation please refer to the Appendix):

$$\Delta = e^{\frac{(r-1)(N_1 \ln p + N_\Delta \ln p + p)}{N_\Delta + rN_1}} \tag{14}$$

A similar computation can be done for the ratio of waiting times, to get a more simplified expression. We illustrate that with an example below:

EXAMPLE 2. *The QoS might be of the form: "For an average size query, a query from a higher service level should have a wait time of less than r as compared to a query from a lower service level". We will now compute Δ for this QoS, assuming $N_1 = N_\Delta$. From Equations 14 and 13(The processing times need not be taken into account for ratio of wait times):*

$$r = \frac{\ln p + \ln p\Delta}{\ln \frac{p}{\Delta} + \ln p} \implies r = \frac{\ln p^2\Delta}{\ln \frac{p^2}{\Delta}}$$

$$\implies \ln p^2\Delta = r \ln \frac{p^2}{\Delta} \implies p^2\Delta = \frac{p^{2r}}{\Delta^r}$$

$$\implies \Delta = p^{\frac{2(r-1)}{r+1}}$$

This would mean for example that for an average size of 10, and $r = 3$, $\Delta = 10$, i.e, the waiting time of a query of size 10 and normalized service level 3 will be $\frac{1}{10}^{th}$ the waiting time of corresponding query of size 10, but with a normalized service level of 1.

5. EXPERIMENTAL RESULTS

In this section we will show that rFEED has the desired FEED properties of fairness, effectiveness and differentiation. The fourth property of efficiency is built into the rank function as discussed previously.

Creating an MWS, testing it and comparing it with other schemes on a live production system can be prohibitively expensive. Hence, our experimental results were obtained from simulations. (All our simulations were done on the Extend Simulation Environment).

We perform three different experiments. For each experiment we compare rFEED with the state of the art, which has been modeled as a resource share system, RS_{opt}. For each experiment we compare the l_2 norms for stretch. We provide a comparison of the average values of stretch as well. The details are presented below.

We begin our discussion by presenting a single metric by which we simultaneously measure fairness and effectiveness.

5.1 The l_2 Norm for Stretch

We have previously discussed that the two requirements of fairness and effectiveness can be at odds with each other, i.e., a fair system could be ineffective or an effective system could be unfair. Effectiveness can be understood as the average case performance and hence is measured by the average value and the fairness can be thought of as the worst-case performance and measured by the maximum value. We need a metric that captures both.

A common trade-off between the average case and the worst-case performance is the l_2 norm [1, 23]. We use the l_2 norm for stretch to demonstrate 'goodness' of our results. The l_2 norm for stretch is defined as:

DEFINITION 4. *The l_2 norm of stretch, for a set of queries $\{q_i : i = 1 \ldots n\}$ with stretches $\{s_i : i = 1 \ldots n\}$ is $\sqrt{(\sum_1^n s_i^2)}$.*

The aim of a good MWS should be to lower the l_2 norm for stretch. We will use this metric to compare rFEED with the resource share system(RS_{opt}) and show that the l_2 norm for stretch for rFEED is consistently less than that for RS_{opt} and sometimes by an order of magnitude.

5.2 Experimental Framework

5.2.1 Experiments

We have done three sets of simulations with different approaches to set the query processing time for scheduling purpose. This processing time is assumed to be the time the query would take to run if run by itself on a system.

Pareto Distribution: We simulate a set of queries whose processing times are drawn from a Pareto distribution with minimum value 1 and the Pareto Index of 1.

Optimizer Cost Estimate: Next we introduce a optimizer cost estimate with error. Our rank function would work with this cost the same way as it would with actual processing times, if the cost were directly proportional to the actual processing time. However, very often even this is not the case. To account for such an error, for each query we multiply the processing time with a function to give us the estimated cost. We draw the error from a normal distribution with $N(1, 0.2)$. In real life we would have the optimizer cost and the actual processing time would be an error of the optimizer cost.

Actual Processing Times: We obtained the execution data from a production system of a customer. We used these execution times as processing times for queries. The execution times however were obtained when the system was loaded, i.e., a number of queries were running concurrently. The system was fully loaded all the way and we assume that all queries were equally affected. Hence, this data is clean enough to be used as an approximation of processing time. (It is very impractical to get the execution time data from queries running by themselves on a production system.)

5.2.2 Simulation Environment

For each of the experiments above, we compare rFEED with a generalization of the state of the art resource share system RS_{opt}. For rFEED we model two different scenarios, rFEED_1 and rFEED_{10} to study differentiation with different normalized service levels. The settings of the three scenarios for corresponding service levels are listed in Table I.

The distribution of queries are independent from the query execution time. The settings of service levels are based on some typical scenarios.

Table 1: Parameter Settings for Service Levels

Service Level	High	Medium	Low
% in Query Workload	30%	50%	20%
Resource % in RS_{opt}	30%	50%	20%
Δs in rFEED_1	1	2	3
Δs in rFEED_{10}	1	10	100

In RS_{opt} the CPU is divided among the three service levels in exact proportion of the number of queries of different service levels. Thus, for example, a query with ideal processing time of 1 second with medium service level will run at least (without waiting for others) 2 seconds in the RS_{opt}. We call it "optimal" in the sense that the proportional resource slicing will keep the CPU always busy, without finishing queries from one service level earlier than others. Within each service level, queries are scheduled using a FIFO queue. Note that in real life this will not be the case. We tried to compare with the best scenario.

We create two scenarios for arrival rates of queries, steady state and peak load. We use Poisson arrival for online queries. We first estimate the mean execution time either through repeated samples (in the case of Pareto Distribution) or through the actual data (in the case of Optimizer Cost Estimate and Actual Processing Times). We then use Little's Law to set the mean arrival intervals of the queries. Little's law states that at steady state the arrival rate is same as the mean service rate. So for the steady state the mean arrival interval is set as the mean execution time. For peak loads we take the square root of the steady state arrival interval.

For rFEED we need the size of the largest query, ψ. For experiments with a Pareto Distribution, this might not be known precisely. Instead we take the average value of the largest value from several samples. Thus the value of ψ used could be away from the real value. This situation is similar with the real life scenarios since the largest query size might not be known in advance. Our experiments show that our rFEED strategy still work well with only approximate ψ.

For each of the six experimental configuration (3 different sets of execution time and two arrival rates) we did 10 different runs. We used 1000 queries per run. The reported results are averages over the 10 different runs. The DBMS is modeled as a concurrent system with the multi-programming level (MPL) fixed to ten.

We now describe our results. For each experiment, we have computed the l_2 norms of stretch under all three scheduling mechanisms for the overall set of queries and also for each of the three different service levels. We plot the l_2 norms as a ratio, where each value is normalized by the value of rFEED_1. As discussed earlier, the lower the l_2 norm of stretch, the better the MWS is. Hence, the ratio of the l_2 norm of stretch of RS_{opt} to that of rFEED, if greater than 1, indicates that rFEED is superior to RS_{opt}. We also plot the average values of stretch for all the experiments in a separate chart.

Since comparing the l_2 norm for different service levels between RS_{opt},rFEED_1 and rFEED_{10} is not fair, we instead focus on studying the l_2 norm between these scheduling strategies for the same service level. Our results clearly differen-

tiates our rFEED from others.

5.3 Exp. 1: Execution Time with Pareto Distribution

This was the experiment as described before where the execution times are drawn from a Pareto distribution with minimum value 1 and the Pareto index 1.

In Figure 4 and Figure 5, we have plotted the results for this experiment. The mean execution time was approximated to 10. Hence the two different arrival rates corresponding to steady state and the peak load were $\lambda_1 = 10$ and $\lambda_2 = 3.16$. We found ψ (as discussed before) to be equal about 4000. K in the Equation 6 was computed using Equation 11 and was found to be 6.3×10^{-8}.

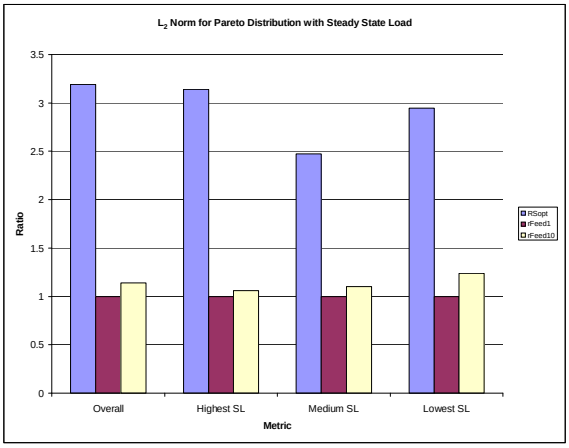

Figure 4: L_2 Norm for Pareto Distribution with Steady State Load

In Figure 4 we have plotted the results for the steady state system. As described before, we have plotted the ratios which are obtained by dividing the values with corresponding values from rFEED$_1$. From the l_2 norm for all the queries, we can observe that the l_2 norm of the RS$_{opt}$ is 3.19 times rFEED$_1$ and 2.79 times rFEED$_{10}$. In other words, rFEED$_1$ provides 319% improvement and rFEED$_{10}$ provide 279% improvement over the state of the art.

rFEED has better l_2 norm for all the service levels for both rFEED$_1$ and rFEED$_{10}$.

In Figure 5 we have plotted the results for peak load. Here the percentage improvement for rFEED$_1$ is 2734% over RS$_{opt}$ and for rFEED$_{10}$ it is a 433% improvement. This shows that rFEED provides significant improvement with peak loads.

There is differentiation that can be seen by comparing different service levels between RS$_{opt}$, rFEED$_1$ and rFEED$_{10}$. The ratio of l_2 norm for the highest service level between rFEED$_{10}$ and rFEED$_1$ is 0.77, i.e., for rFEED$_{10}$, the l_2 norm was 77% of that of rFEED$_1$. This indicated that rFEED$_{10}$ has higher differentiation than rFEED$_1$.

If we compare rFEED$_{10}$ and RS$_{opt}$ we see for the lowest service level, the l_2 norm for RS$_{opt}$ is 1.42 times that for rFEED$_{10}$, for the medium service level it is 31.05 times that of RS$_{opt}$ and for the highest service level RS$_{opt}$'s l_2 norm is 40.32 times that of rFEED$_{10}$. It can be seen that as we go to higher service levels the ratio of the l_2 norms steadily increases indicating that rFEED$_{10}$ differentiates very well.

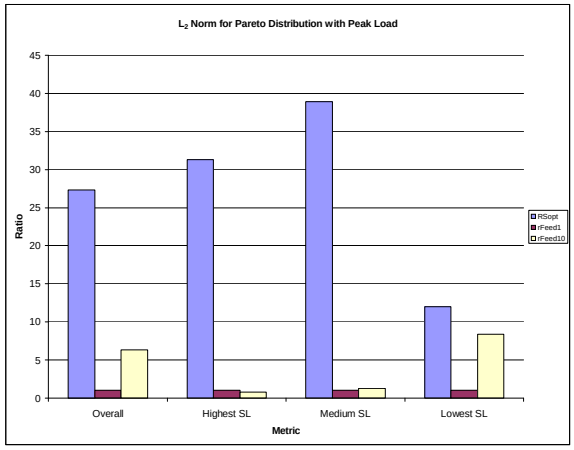

Figure 5: L_2 Norm for Pareto Distribution with Peak Load

Even with such high differentiation, overall rFEED$_{10}$ is 433% better than RS$_{opt}$.

5.4 Exp. 2: Execution Time with Optimizer Estimate

This was the experiment as described before where the execution times are drawn from a Pareto distribution with minimum value 1 and the Pareto index 1 and cost was computed by multiplying the processing time by an error derived from $N(1, 0.2)$. The scheduling was done using the error and the results were computed with the processing time.

In Figure 6 and Figure 7, we have plotted the results for this experiment. Like the previous experiment the mean execution time was approximated to 10, arrival rates corresponding to steady state and the peak load were $\lambda_1 = 10$ and $\lambda_2 = 3.16$ and K was 6.3×10^{-8}.

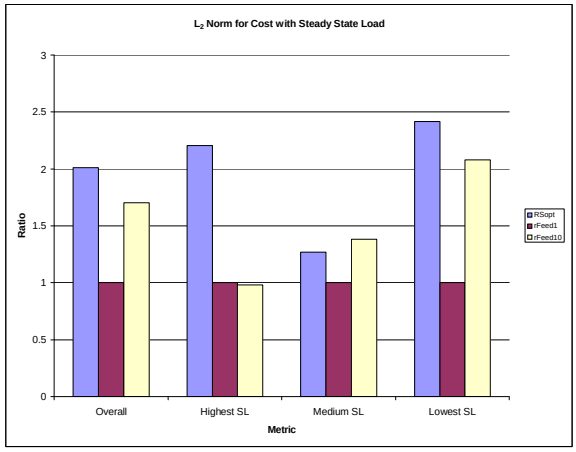

Figure 6: L_2 Norm for Cost with Steady State Load

In Figure 6 we have plotted the results for the steady state system. As before the ratios have been plotted. For the overall l_2 norm, RS$_{opt}$'s l_2 norm is 2.01 times that of rFEED$_1$ and 1.18 times that of rFEED$_{10}$. Both rFEED$_1$ and rFEED$_{10}$ have better l_2 norm than RS$_{opt}$ for all service levels.

There is some differentiation as the ratio of l_2 norm of

rFEED$_{10}$ and rFEED$_1$ increases from highest service level to lowest service level.

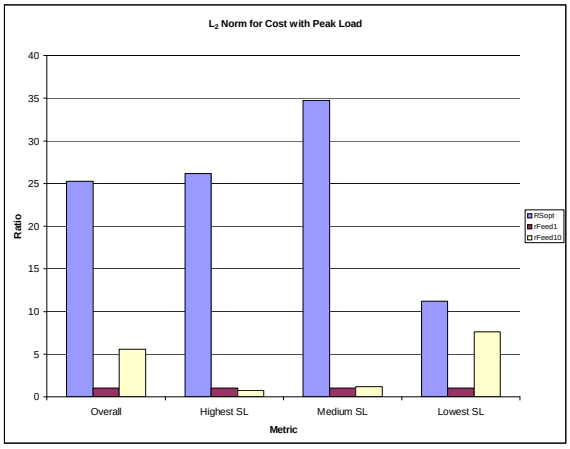

Figure 7: L_2 Norm for Cost with Peak Load

In Figure 7 we have plotted the ratios for peak load. Here the percentage improvement for rFEED$_1$ is 2522% over RS$_{opt}$ and for rFEED$_{10}$ it is a 557% improvement.

Again, like Experiment 1, by comparing rFEED$_{10}$ and RS$_{opt}$ we can see that rFEED$_{10}$ provides significant differentiation without paying a very high cost in the overall results. For the lowest service level, the l_2 norm for RS$_{opt}$ is 1.14 times that for rFEED$_{10}$, for the medium service level it is 28.95 times and for highest service level it is 33.69 times that of rFEED$_{10}$.

When comparing with the results of Experiment 1, it can be seen that our performance does not suffer significantly because of the error in cost estimation. It seems that rFEED is robust to modest mis-estimates in computation of processing times.

5.5 Exp. 3: Execution Time with Actual Processing Time

As mentioned before, we obtained actual execution times from a customer and used them as an approximation of processing times.

We had more than 65000 actual processing times. For each run, we sampled a 1000 of these processing times. The average execution time was 24.5 seconds, so the steady state arrival rate, λ_1 was 24.5 and the peak arrival rate, λ_2 was 4.7. The largest query was close to 10000 seconds, so K was 10^{-8}.

In Figure 8 and Figure 9, we have plotted the results for this experiment.

In Figure 8 we have plotted the ratios for the steady state system. The results are interesting here, in that we see differentiation even for a steady state scenario. For the overall l_2 norm, RS$_{opt}$'s l_2 norm is 2.47 times that of rFEED$_1$ and 1.58 times that of rFEED$_{10}$. In terms of service levels however, rFEED performs better than RS$_{opt}$ only for the highest service level. This behavior is an example of differentiation and is not undesirable since we want the highest service levels to perform better than the other two service levels but not at a significant cost to the overall quality of results.

In Figure 9 we have plotted the results for the peak load scenario. The results this time are really impressive.

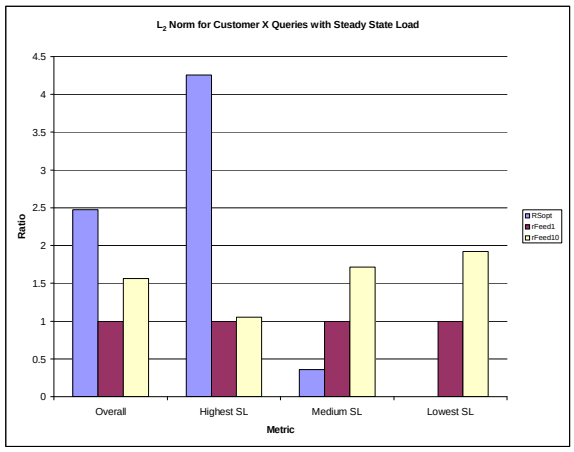

Figure 8: L_2 Norm for Customer X Data with Steady State Load

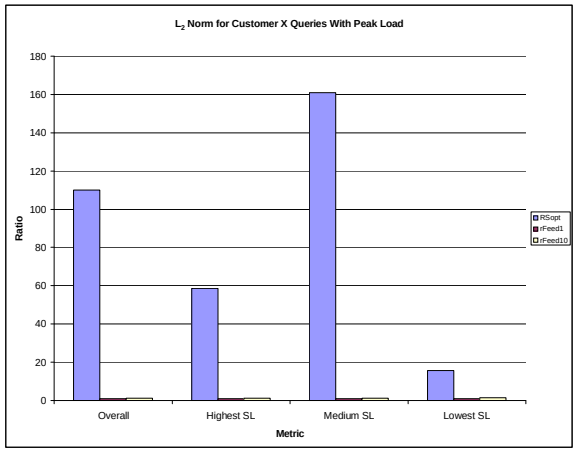

Figure 9: L_2 Norm for Customer X Data with Peak Load

The overall l_2 norm, for RS$_{opt}$ is 110 times that for rFEED$_1$ and 89.7 times for that of rFEED$_{10}$. The ratios are in a similar large for all three service levels.

5.6 Average Stretch

Customers are also often interested in the average value of stretch. In Figure 10, for each experiment we have plotted two bars. The first bar indicates the ratio of the average stretch value for RS$_{opt}$ to the average stretch value for rFEED$_1$ ($\frac{RS_{opt}AverageStretch}{rFEED_1AverageStretch}$) and the second bar indicates the ratio of the average stretch value for RS$_{opt}$ to the average stretch value for rFEED$_{10}$ ($\frac{RS_{opt}AverageStretch}{rFEED_{10}AverageStretch}$).

For all the three experiments rFEED$_1$ and for both steady state and peak load scenarios, the ratios are greater than 1 indicating that rFEED$_1$ has lower average stretch than RS$_{opt}$ for all cases.

For rFEED$_{10}$, the ratio is more than 1 for five out of the six scenarios, where it is slightly higher (20%) for the steady state case of Experiment 3.

Again, with peak loads the ratios are much larger as compared to steady state, with the highest value being 123. This means that the average stretch for RS$_{opt}$ was 123 times that

of rFEED$_1$ for the experiment with actual processing times.

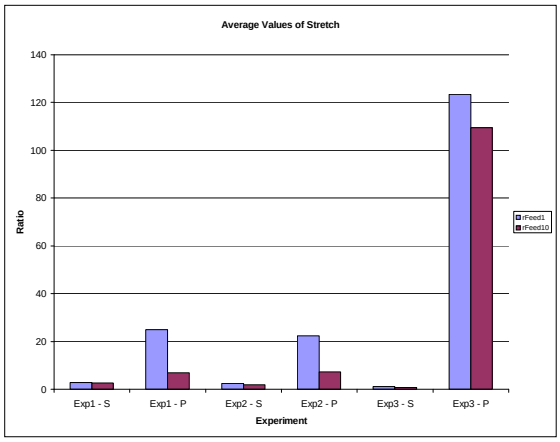

Figure 10: Average value for Overall Stretch for Various Experiments

5.7 Response Time

Although, in this work our primary metric of interest is stretch, response time is also an important consideration. In general, rFEED does very well with response time but it requires a different computation for the value of K. Due, to lack of space we present two experimental results to compare rFEED with other techniques. We compare with FIFO, SJF and a technique called Greedy With Rounding(GWR). GWR is like our technique suitable for online scheduling and also requires an estimate of the largest query size.

We did two sets of experiments both with Pareto distribution with minimum value 1 and the Pareto index 1. For each experiment we present the mean of 10 runs, and there were 10000 queries in each run. The mean arrival rate λ was 10 in Experiment 1 and was 3.16 in Experiment 2. We assumed $\psi = 10000$. We used a single server model with no service levels. The results are tabulated in Table II, where it can be seen that rFEED does much better than FIFO and GWR. SJF is optimal for response time and rFEED performs very closely to SJF. SJF is unsuitable for online scheduling since it causes starvation. Furthermore, if we were to compute $K = \frac{1}{\psi^3}$, our performance would is same as SJF.

Table 2: Results for Response Time

	FIFO	GWR	SJF	rFEED
$\lambda = 3.16$	17589.66	9314.27	1150.312	1165.91
$\lambda = 10$	22024.52	13852.97	2816.87	4920.169

5.8 Discussion

From the experiments the following things can be observed:

1. rFEED performs better than a resource sharing system for the l_2 norm of stretch under various experimental conditions.
2. For a peak load scenario, where the waiting queues could be large in length rFEED does significantly better than a resource share system.
3. Both rFEED$_1$ and rFEED$_{10}$ provide differentiation, where rFEED$_{10}$ provides more differentiation.
4. Differentiation does not come at a large cost to the overall results.
5. rFEED is robust to moderate mis-estimates in processing times.
6. rFEED does significantly better than RS$_{opt}$ even when the average value of stretch is used as a metric.
7. Although, rFEED works well in practice it has a few limitations:
   (a) It depends on knowing the query execution time which is not always known. However, in this case the optimizer cost can be chosen.
   (b) If the value of ψ is very large, it can lead to small values for K, which could result in ignoring of the wait time.

6. CONCLUSIONS

There are four major challenges with managing mixed workloads. First, a typical BI workload is heavy-tailed, with many small queries and a few large queries. In fact, just 10% of the queries often account for up to 90% of the load on a typical EDW. Second, the user perspective is critical in measuring the performance of an EDW, thus making the stretch metric extremely important. Third, a good MWS needs to be fair, effective, efficient and differentiated, all at the same time. Fourth, the BI workloads change dynamically so manual tuning is not practical.

We have designed an MWS that tackles all these challenges. Our design, called the rFEED scheduler, is simple in design, powerful in its performance, and is practical for implementation. We have presented a general scheduling function and then shown how to compute the various parameters to make it implementable. Furthermore, we have modeled the best-of-breed commercial schedulers, and have run our new rFEED scheduler head-to-head against this model. rFEED does significantly better.

The rFEED scheduler has been slated for incorporation into HP's commercial, enterprise class, DBMS offering.

7. ACKNOWLEDGMENTS

We thank Michael Fisher, Rao Kakarlamudi, Vijay Bellam, Zbigniew Omanski, Hans Zeller, Seetha Laksmi, Ahmed Ezzat, Gary Melnik and the many other engineers that have helped with several practical aspects of this work.

8. REFERENCES

[1] N. Bansal and K. Pruhs. Server scheduling in the l$_p$ norm: a rising tide lifts all boat. In *STOC*, pages 242–250, 2003.

[2] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Online weighted flow time and deadline scheduling. *J. Discrete Algorithms*, 4(3):339–352, 2006.

[3] A. Bedekar, S. Borst, K. Ramanan, P. Whiting, and E. Yeh. Downlink scheduling in CDMA data networks. *GLOBECOM*, 5:2653–2657, 1999.

[4] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *SODA*, pages 270–279, 1998.

[5] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *SODA*, pages 762–771, 2002.

[6] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[7] Carrie Ballinger. The Wild World of Mixed Workload: Priorities and resources learn to get along. Teradata Magazine Online. http://www.teradata.com/t/go.aspx/?id=114533.

[8] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 557–568, New York, NY, USA, 2007. ACM.

[9] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *STOC*, pages 297–305, 2002.

[10] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *STOC*, pages 84–93, 2001.

[11] L. Cherkasova and T. Rokicki. Alpha Message Scheduling for Packet-Switched Interconnects. Technical Report HPL-94-71, HP Labs, August 1994.

[12] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[13] E. J. Friedman and S. G. Henderson. Fairness and efficiency in web server protocols. In *SIGMETRICS*, pages 229–237, 2003.

[14] J. R. M. Hosking and J. F. Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29(3):339–349, 1987.

[15] IBM. DB2 Query Patroller. http://www-306.ibm.com/software/data/db2/querypatroller/.

[16] B. Kalyanasundaram, K. Pruhs, and M. Velauthapillai. Scheduling Broadcasts in Wireless Networks. In *Proceedings of the 8th Annual European Symposium on Algorithms (ESA)*, pages 290–301, 2000.

[17] A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of biological requests. In *SPAA*, pages 103–112, 2006.

[18] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[19] A. Mehta, C. Gupta, S. Wang, and U. Dayal. rfeed: A mixed workload scheduler for enterprise data warehouses. In *ICDE '09, Accepted*, 2009.

[20] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, page 433, 1999.

[21] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Trans. Netw.*, 7(3):277–292, 1999.

[22] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. M. Nahum, and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *ICDE*, page 60, 2006.

[23] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, pages 511–522, 2006.

APPENDIX

A. COMPUTING Δ FOR A PARETO DISTRIBUTION

We now demonstrate the computation Δ given a statistical distribution. For this demonstration we assume a Pareto distribution. However, the methodology for computing Δ is general and can be used with any distribution.

We consider a Pareto distribution where the minimum value is 1 and the Pareto Index is also 1. Then, for such a Pareto distribution we get the frequency distribution as: $f(x) = \frac{1}{x^2}$.

Assume that a number of queries $|S|$ arrive at time, $t = 0$ and they are from two different SLs, one with service level modifier $\Delta_q = 1$ and the other with service level modifier $\Delta_q = \Delta$.

Now we compute the total execution time of queries with execution time less than or equal to say t. Let this quantity be T_t. To compute T_t, consider a sample of size S that follows our Pareto distribution. Let $|S_t|$ be the number of queries with execution time less than t and let there expected value be $E(S_t)$. Then: $T_t = (|S_t|)E_t$. We first compute $|S_t|$:

$$|S_t| = |S| \int_1^t \frac{1}{x^2} dx \implies |S_t| = |S|(1 - \frac{1}{t}) \tag{15}$$

To compute $E(S_t)$, we need to consider a truncated Pareto distribution, with maximum value of the random variable being t. A truncated distribution has the form: $\frac{f(x)}{F(x)}$, where F is the cumulative distribution function. Now:

$$E(S_t) = \frac{\int_1^t x \frac{1}{x^2} dx}{\int_1^t x} \implies E(S_t) = \frac{\ln t}{1 - \frac{1}{t}} \tag{16}$$

Then from Equation 15 and 16, we get:

$$T_t = |S_t|E(S_t) \implies T_t = |S| \ln t \tag{17}$$

We are now ready to compute the ratio of interest in Equation 13. Let the ratio r, be the ratio of stretches for a query from $\Delta = 1$, and execution time p to $\Delta = \Delta$, and execution time p.

Let N_1 be the number of queries with $\Delta = 1$ and N_Δ be the number of queries with $\Delta = \Delta$. Then from Equation 17 and Equation 13 we get:

$$r = \frac{N_1 \ln p + N_\Delta \ln p\Delta + p}{N_1 \ln \frac{p}{\Delta} + N_\Delta \ln p + p} \tag{18}$$

Note that the first term in the denominator is zero if $p < \Delta$. Now we compute Δ from this. From Equation 18, after some basic manipulation we get:

$$\ln \Delta(N_\Delta + rN_1) = \ln p((r-1)(N_\Delta + N_1)) + (r-1)p$$

$$\implies \Delta = e^{\frac{(r-1)(N_1 \ln p + N_\Delta \ln p + p)}{N_\Delta + rN_1}} \tag{19}$$