Efficient Scheduling of Heterogeneous Continuous Queries [*]

Mohamed A. Sharaf   Panos K. Chrysanthis   Alexandros Labrinidis   Kirk Pruhs

Advanced Data Management Technologies Laboratory
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA

{msharaf, panos, labrinid, kirk}@cs.pitt.edu

ABSTRACT

Data Stream Management Systems (DSMS) typically host multiple Continuous Queries (CQ) that process streams of data. In this paper, we examine the problem of how to schedule CQs in a DSMS to optimize for average QoS. We show that unlike standard on-line systems, scheduling policies in DSMSs that optimize for average response time will be different than policies that optimize for average slowdown which is more appropriate metric to use in the presence of a heterogeneous workload. We also propose a hybrid scheduling policy based on slowdown that strikes a fine balance between performance and fairness. We further discuss how our policies can be efficiently implemented and extended to exploit sharing in optimized multi-query plans and multi-stream CQs. Finally, we experimentally show using real data that our policies outperform currently used ones.

1. INTRODUCTION

The growing need for *monitoring applications* has led to a new data processing paradigm and created a new generation of data processing systems, called *Data Stream Management Systems* (DSMSs) that can support *continuous queries* (CQ). In such systems, each monitoring application registers a set of CQs that continuously process continuous data streams looking for data that represent events of interest to the end-user.

Currently, we are developing a DSMS, called *AQSIOS*, that can help support monitoring applications such as the real-time detection of disease outbreaks, tracking the stock market, environmental monitoring via sensor networks, and personalized and customized Web pages. One of the main goals in the design of *AQSIOS* is the development of a scheduling policy that optimizes *Quality of Service* (QoS).

This goal is complicated by the fact that the scheduling policy must take into account that the CQs are heterogeneous, i.e., they may have different time complexities (the amount of processing required to find if input data represents an event), and different productivity or selectivity (the number of events detected by the CQ). For example, consider two CQs, `GOOGLE` and `ANALYSIS` on streams of stock market data. `GOOGLE` is a simple query that asks the DSMS to be notified when there is a stock quote for *GOOGLE*. `ANALYSIS` is a complex query that asks the application to provide some specific technical analysis for any new stock price. Obviously, `GOOGLE` has low cost and it detects less events, whereas `ANALYSIS` has high cost and it detects more events.

The mostly commonly used QoS metric in the literature is *average response time*. In [19], we showed that if the objective is to optimize the response time, then the "right" strategy is to schedule CQs according to their *output rate*. Specifically, in [19] we presented a new scheduling policy called *Highest Rate (HR)*. *HR* generalizes the *Rate-based policy (RB)* [23] for scheduling operators in multiple CQs as opposed to *RB* that has been proposed for scheduling operators within a single query. Under *HR*, the priority of a query is set to its output rate where the output rate of the query is the ratio between its expected selectivity and its expected cost.

However, there are some well known disadvantages to the average response time metric when the workload is heterogeneous. In the above example, the user that issued the `ANALYSIS` query likely knows that it is a complex query, and is expecting a higher response time than the user that issued the `GOOGLE` query. A metric that captures this phenomenon is *average slowdown*. The slowdown of a job is the response time of the job to the ideal processing time of the job [17]. So, for example, if each job had slowdown 1.1, then each user would experience a 10% delay due to queuing (although the responses could be very different).

Interestingly, in most on-line systems (e.g., Web servers), *Shortest-Remaining-Processing-Time (SRPT)* is one policy that is optimal for average response time and near optimal for average slowdown [17]. A surprising discovery of this paper is that this is not the case with the *HR* policy that optimizes average response time of CQs. In general, *HR* will not optimize average slowdown because of the "probabilistic" nature of CQs where the selectivity might not equal to 1. In this paper, we argue that if the objective is to optimize average slowdown then the "right" scheduling strategy is to set the priority of a query to the ratio of its selectivity over the product of its expected cost and its ideal total processing cost.

The average slowdown provided by the DSMS captures the average-case performance of the system. However, improving the average-case performance usually comes at the expense of unfairness toward certain classes of queries that might experience *starvation*. Starvation is typically captured by measuring the *maximum slow-*

[*]This work was partially supported by NSF IIS-0534531. The first author was also supported in part by the Andrew Mellon Predoctoral Fellowship.

down of the system [8]. That is, the perceived worst-case performance.

Starvation is an unacceptable behavior in a DSMS that supports monitoring applications where all kinds of events are equally important. Hence, it is important to balance the trade-off between the average-case and worst-case performances of the DSMS. Toward this, we propose a hybrid scheduling policy that optimizes the ℓ_2 norm of slowdowns [7]. As such, it is able to strike a fine balance between the average- and worst-case performances and hence it avoids starvation and exhibits higher degree of *fairness*.

In addition to new scheduling policies, we consider two special features that are unique to CQs and should be exploited by the query scheduler. First, we address the scheduling of *multi-stream queries with time-based sliding window join operators*. We formulate the definition of slowdown for composite tuples produced by join operators and extend our proposed scheduling policies to handle such multi-stream queries. Second, we consider the scheduling of *multiple queries with shared operators* where we show that a proper setting of the priority of shared operators significantly improves the system performance.

Contributions The contributions of this paper can be summarized as follows:

1. We propose a policy for scheduling multiple CQs that maximizes the average-case performance of a DSMS.
2. We propose a hybrid policy that strikes a fine balance between the average- and worst-case performances.
3. We consider two issues that are very particular to DSMSs. Namely, we propose: (1) extending our proposed policies to handle multi-stream continuous queries; and (2) exploiting sharing in optimizing multi-query plans.
4. To ensure that our proposed hybrid policy can be efficiently realized in AQSIOS, we propose a low-overhead implementation which uses clustering in addition to efficient search pruning techniques from [3, 12].

Our extensive experimental evaluation using real and synthetic data shows the significant gains provided by our proposed policies under different QoS measures compared to existing scheduling policies in DSMSs.

Road Map Section 2 provides the system model. Section 3 and 4 define our QoS metrics and presents our proposed scheduling policies. Section 5 focuses on multi-stream queries. In Section 6 and 7, we discuss implementation details and extend our work to consider queries with shared operators. Sections 8 and 9 discusses our simulation testbed and our experimental results. Section 10 surveys related work.

2. SYSTEM MODEL

In a DSMS, users register continuous queries that are executed as new data arrives. Data arrives in the form of continuous streams from different data sources. where the arrival of new data is similar to an *insertion* operation in traditional database systems. A DSMS is typically connected to different data sources and a single stream might feed more than one query.

A continuous query evaluation plan can be conceptualized as a data flow tree [10, 5], where the nodes are operators that process tuples and edges represent the flow of tuples from one operator to another (Figure 1). An edge from operator O_x to operator O_y means that the output of O_x is an input to O_y. Each operator is associated with a *queue* where input tuples are buffered until they are processed.

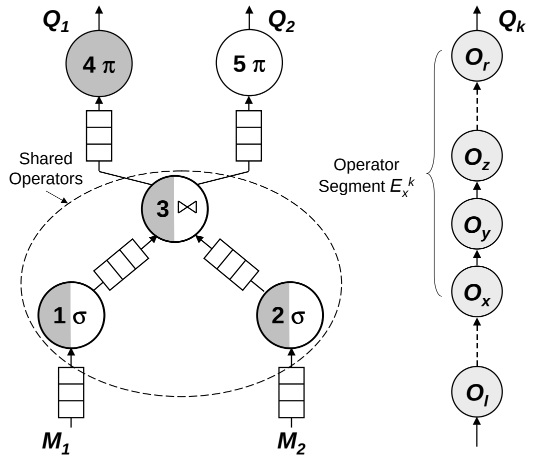

Figure 1: Continuous Queries Plans

Multiple queries with common sub-expressions are usually merged together to eliminate the repetition of similar operations [18]. For example, Figure 1 shows the global plan for two queries Q_1 and Q_2. Both queries operate on data streams M_1 and M_2 and they share the common sub-expression represented by operators O_1, O_2 and O_3.

A *single-stream query* Q_k has a single *leaf* operator O_l^k and a single *root* operator O_r^k, whereas a *multi-stream* query has a single root operator and more than one leaf operators. In a query plan Q_k, an *operator segment* $E_{x,y}^k$ is the sequence of operators that starts at O_x^k and ends at O_y^k. If the last operator on $E_{x,y}^k$ is the root operator, then we simply denote that operator segment as E_x^k. Additionally, E_l^k represents an operator segment that starts at the leaf operator O_l^k and ends at the root operator O_r^k. For example, in Figure 1, $E_1^1 =< O_1, O_3, O_4 >$, whereas $E_1^2 =< O_1, O_3, O_5 >$.

In a query, each operator O_x^k (or simply O_x) is associated with two parameters:

1. *Processing cost* or *Processing time* (c_x): is the amount of time needed to process an input tuple.
2. *Selectivity* or *Productivity* (s_x): is the number of tuples produced after processing one tuple for c_x time units. s_x is less than or equal to 1 for a filter operator and it could be greater than 1 for a join operator.

Given a single-stream query Q_k which consists of operators $< O_l^k, ..., O_x^k, O_y^k, ..., O_r^k >$ (Figure 1), we define the following characterizing parameters for any operator O_x^k (or equivalently, for any operator segment E_x^k that starts at operator O_x^k):

- **Operator Global Selectivity (S_x^k):** is the number of tuples produced at the root O_r^k after processing one tuple along operator segment E_x^k.

$$S_x^k = s_x^k \times s_y^k \times ... \times s_r^k$$

- **Operator Global Average Cost ($\overline{C}_x^k$):** is the expected time required to process a tuple along an operator segment E_x^k.

$$\overline{C}_x^k = (c_x^k) + (c_y^k \times s_x^k) + ... + (c_r^k \times s_{r-1}^k \times ... \times s_x^k)$$

If O_x^k is a leaf operator ($x = l$), when a processed tuple actually satisfies all the filters in E_l^k, then $\overline{C}_l^k$ represents the ideal total processing cost or time incurred by any tuple *produced* or *emitted* by query Q_k. In this case, we denote $\overline{C}_l^k$ as T_k:

- **Tuple Processing Time (T_k):** is the ideal total processing cost required to produce a tuple by query Q_k.

$$T_k = c_l^k + ... + c_x^k + c_y^k + ... + c_r^k$$

We extend the above parameters for multi-stream queries in Section 5.

3. AVERAGE-CASE PERFORMANCE

In this section, we focus on QoS for single-stream queries and present our scheduling policies for optimizing these metrics. Multi-stream queries are discussed in Section 5.

3.1 Response Time Metric

In DSMSs, the arrival of a new tuple triggers the execution of one or more CQs. Processing a tuple by a CQ might lead to discarding it (if it does not satisfy some filter) or it might lead to producing one or more tuples at the output which means that the input tuple represents an event of interest to the user who installed the CQ. Clearly, in DSMS, it is more appropriate to define response time from data/event perspective rather than from query perspective as in traditional DBMSs. Hence, we define the *tuple response time* or *tuple latency* as follows:

DEFINITION 1. *Tuple response time, R_i, for tuple t_i is $R_i = D_i - A_i$, where A_i is t_i's arrival time and D_i is t_i's output time. Accordingly, the average response time for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} R_i$.*

Notice that tuples that are filtered out do not contribute to the metric as they do not represent any event [22].

3.2 Slowdown Metric

Average response time is an expressive metric in a homogeneous setting. That is, when all tuples require the same processing time. In a heterogeneous workload, as in our system, the processing requirements for different tuples may vary significantly and average response time is not an appropriate metric since it cannot relate the time spent by a tuple in the system to its processing requirements. Other on-line systems with heterogeneous workloads such as DBMSs, OS, and Web servers have adopted *average slowdown* or *stretch* [17] as another metric. This motivated us to consider stretch as the metric in our system.

The definition of slowdown was initiated by the database community in [16] for measuring the performance of a DBMS executing multi-class workloads. Formally, the slowdown of a job is the ratio between the time a job spends in the system to its processing demands [17]. In DSMS, we define the slowdown of a tuple as follows:

DEFINITION 2. *The slowdown, H_i, for tuple t_i produced by query Q_k is $H_i = \frac{R_i}{T_k}$, where R_i is t_i's response time and T_k is its ideal processing time. Accordingly, the average slowdown for N tuples is: $\frac{1}{N}\sum_{i=1}^{N} H_i$.*

Intuitively, in a general purpose DSMS where all events are of the same importance, a simple event (i.e., event detected by a low-cost CQ) should be detected faster than a complex event (i.e., event detected by a high-cost CQ) since the latter contributes more to the load on the DSMS.

3.3 Highest Normalized Rate Policy (HNR)

Based on the above definitions, we developed the *Highest Normalized Rate (HNR)* policy for minimizing average slowdown.

To illustrate the intuition underlying *HNR*, consider two operator segments E_x^i and E_y^j starting at operators O_x^i and O_y^j respectively. For each of the two operator segments, we compute its global selectivity and global average cost as described above. Further, assume that the current wait time for the tuple at the head of O_x^i's queue is W_x^i and for the tuple at the head of O_y^j's queue is W_y^j.

In a policy A where E_x^i is executed before E_y^j, the total slowdown of tuples produced under this policy is:

$$H_A = S_x^i \times H_{A,i} + S_y^j \times H_{A,j} \tag{1}$$

where S_x^i and S_y^j is the number of tuples produced by E_x^i and E_y^j respectively, and $H_{A,i}$ and $H_{A,j}$ are the slowdowns of the E_x^i tuples and the E_y^j tuples respectively.

Recall that the slowdown of a tuple is the ratio between the time it spent in the system to its ideal processing time. Hence, $H_{A,i}$ and $H_{A,j}$ are computed as follows:

$$H_{A,i} = \frac{T_i + W_x^i}{T_i} \qquad H_{A,j} = \frac{\overline{C}_x^i + T_j + W_y^j}{T_j}$$

where $\overline{C}_x^i$ is the amount of time E_y^j will spend waiting for E_x^i to finish execution. By substitution in (1),

$$H_A = S_x^i \times \frac{T_i + W_x^i}{T_i} + S_y^j \times \frac{\overline{C}_x^i + T_j + W_y^j}{T_j}$$

Similarly, under an alternative policy B, where E_y^j is executed before E_x^i, the total slowdown H_B is:

$$H_B = S_y^j \times \frac{T_j + W_y^j}{T_j} + S_x^i \times \frac{\overline{C}_y^j + T_i + W_x^i}{T_i}$$

In order for H_A to be less than H_B, then the following inequality must be satisfied:

$$S_y^j \times \frac{\overline{C}_x^i}{T_j} < S_x^i \times \frac{\overline{C}_y^j}{T_i} \tag{2}$$

The left-hand side of Inequality 2 shows the *increase* in total slowdown incurred by the tuples produced by E_y^j when E_x^i is executed first. Similarly, the right-hand side shows the increase in total slowdown incurred by the tuples produced by E_x^i when E_y^j is executed first. The inequality implies that between the two alternative execution orders, we should select the one that minimizes the increase in the total slowdown. That is, we should select the segment with the smallest negative impact on the other one.

Thus, in our *HNR* policy, each operator O_x^k is assigned a priority V_x^k which is the *weighted rate* or *normalized rate* of the operator segment E_x^k that starts at operator O_x^k and it is defined as:

$$V_x^k = \frac{1}{T_k} \times \frac{S_x^k}{\overline{C}_x^k} \tag{3}$$

The term $S_x^k/\overline{C}_x^k$ is basically the *global output rate (GR_x^k)* of the operator segment starting at operator O_x^k as defined in [23]. As such, the priority of each operator O_x^k is its normalized output rate, or equivalently, the normalized output rate of the operator segment E_x^k starting at O_x^k. Hence, executing O_x^k implies the pipelined execution of all the operators on E_x^k unless it is interrupted by a higher priority operator as we will describe in Section 6.

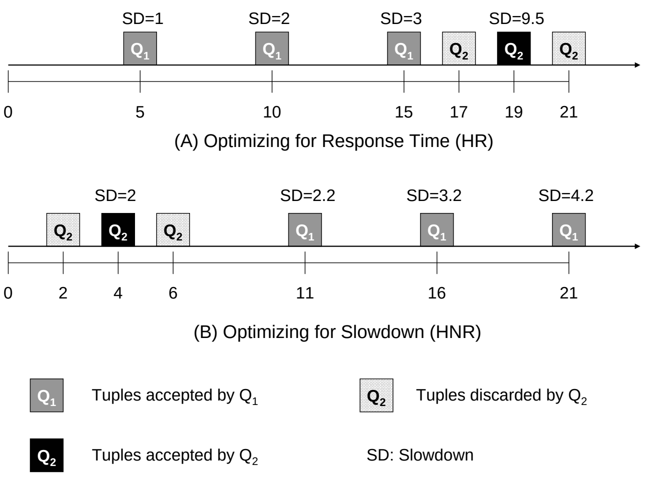

Figure 2: The output of Example 1

3.4 HNR vs. HR

It is interesting to notice that if the objective is optimizing the response time, then the ideal total processing cost T should be eliminated from the denominators of all the above equations resulting in setting the priority V_x^k of operator O_x^k to:

$$V_x^k = \frac{S_x^k}{\overline{C}_x^k} = GR_x^k \quad (4)$$

In fact, this is the prioritizing function we use in our *Highest Rate (HR)* policy for optimizing the response time [19] (as mentioned in the Introduction). As such, *HR* schedules jobs in descending order of output rate which might result in a high average slowdown because a low cost query can be assigned a low priority since it is not productive enough. Those few tuples produced by this query will all experience a high slowdown, with a corresponding increase in the average slowdown of the DSMS.

Our policy *HNR*, like *HR*, is based on output rate, however, it also emphasizes the ideal tuple processing time in assigning priorities. As such, an inexpensive operator segment with low productivity will get a higher priority under *HNR* than under *HR*.

Example 1 To further illustrate the difference between the *HR* and the *HNR* policies, let us consider an example where we have two queries Q_1 and Q_2. Each query consists of a single operator. For Q_1, the cost of the operator is 5 ms and its selectivity is 1.0. For Q_2, the cost of the operator is 2 ms and its selectivity is 0.33. Further, assume that there are 3 pending tuples to be processed by the 2 queries and that all 3 tuples have arrived at time 0.

Under the *HR* policy, Q_1's priority is $\frac{1.0}{5.0} = 0.2$, whereas Q_2's priority is $\frac{0.33}{2.0} = 0.1667$ (which is the output rate of each query). Figure 2(A) shows the queries' output under the *HR* policy where Q_1 is executed first and it accepted/emitted all the pending 3 tuples, then Q_2 is executed and it only accepted one of the 3 pending tuples (since its selectivity is 0.33) and say it was the middle one.

Under the *HNR* policy, Q_1's priority is $\frac{1.0}{5.0\times5.0} = 0.04$, whereas Q_2's priority is $\frac{0.33}{2.0\times2.0} = 0.08$. Hence, under *HNR*, Q_2 is scheduled before Q_1 resulting in the output shown in Figure 2(B).

Table 1 shows that *HNR* provides the lower average slowdown compared to *HR*. The reason is that the one tuple accepted by Q_2 experienced a slowdown of $\frac{4}{2} = 2.0$ under *HNR* while its slowdown under *HR* is $\frac{19}{2} = 9.5$. This unfairness of *HR* toward Q_2 resulted in a higher overall average slowdown compared to *HNR*.

	Response Time	Slowdown
HR	12.25	3.875
HNR	13.0	2.9

Table 1: Results of Example 1

3.5 HNR vs. HR vs. SRPT

It should be clear that under *HR*, if all the operators' selectivities are equal to one, then Equation 4 is simply the inverse of the processing time. Hence, in this case, *HR* is equivalent to *SRPT*. Similarly, if all the operators' selectivities are equal to one, then in Equation 3, $\overline{C}_x^k$ is equal to T_k and O_x^i is executed before O_y^j if $1/(T_i)^2 > 1/(T_j)^2$. By taking the square root of both sides, then *HNR* is also equivalent to *SRPT*.

The above observation shows the effect of the selectivity parameter on this problem. That is, under a probabilistic workload, *HR* reduces the response time, whereas, *HNR* reduces the slowdown. However, as the workload becomes deterministic, both *HR* and *HNR* converge to a single policy which is the *SRPT* policy.

4. AVERAGE-CASE VS. WORST-CASE PERFORMANCE

Here, we first define the worst-case performance and a policy that minimizes it. Then, we introduce our scheduling policy for balancing the trade-off between the average- and worst-case performance.

4.1 Worst-case Performance

It is expected that a scheduling policy that strives for minimizing the average-case performance might lead to a poor worst-case performance under a relatively high load. That is, some queries (or tuples) might starve under such a policy. Such a worst-case performance is typically measured using *maximum slowdown* [8].

DEFINITION 3. *The maximum slowdown for N tuples is $max(H_1, H_2, ..., H_N)$.*

Intuitively, a policy that optimizes for the worst-case performance should be pessimistic. That is, it assumes the worst-case scenario where each processed tuple will satisfy all the filters in the corresponding query. An example of such a policy is the *Longest Stretch First (LSF)* [2]. Under *LSF*, each operator O_x^k is assigned a priority V_x^k which is computed as:

$$V_x^k = \frac{W_x^k}{T_k} \quad (5)$$

where W_x^k is the wait time of the tuple at the head of O_x^k's input queue and T_k is the ideal processing cost for that tuple.

LSF is a greedy policy under which the priority assigned to an operator O_x^k is basically the current slowdown of the tuple at the top of O_x^k's input queue, where the current slowdown of a tuple is the ratio of the time the tuple has been in the system thus far to its processing time.

4.2 Balancing the Trade-off between Average-case and Worst-case Performance

A policy that strikes a fine balance between the average-case and worst-case performance needs a metric that is able to capture this trade-off. In this section, we first present such a metric, then we describe our proposed scheduling policy which optimizes that metric.

4.2.1 The ℓ_2 Norm Metric

On one hand, the average value for a QoS metric provided by the system represents the expected QoS experienced by any tuple in the system (i.e., the average-case performance). On the other hand, the maximum value measures the worst QoS experienced by some tuple in the system (i.e., the worst-case performance). It is known that each of these metrics by itself is not enough to fully characterize the system performance.

To get a better understanding of the system performance, we need to look at both metrics together or, alternatively, we can use a single metric that captures both of these metrics. The most common way to capture the trade-off between the average-case and the worst-case performance is to measure the ℓ_2 *norm* [7]. Specifically, the ℓ_2 norm of slowdowns is defined as:

DEFINITION 4. *The ℓ_2 norm of slowdowns for N tuples is equal to $\sqrt{\sum_1^N H_i^2}$.*

The definition shows how the ℓ_2 norm considers the average in the sense that it takes into account all values, yet, by considering the second norm of each value instead of the first norm, it penalizes more severely outliers compared to the average slowdown metric.

4.2.2 A Scheduling Policy for Balancing the Performance Trade-off

In order to balance the trade-off between the average- and worst-case performance, we are proposing a new scheduling policy that minimizes the ℓ_2 norm of slowdowns. We will call this new policy *Balance Slowdown (BSD)*. To understand the intuition underlying *BSD*, we will use the same technique from the previous section but with the objective of minimizing the ℓ_2 norm of slowdowns.

Specifically, consider a policy A where operator segment E_x^i is executed before operator segment E_y^j. The ℓ_2 norm of slowdowns of tuples produced under this policy is:

$$L_A = \sqrt{S_x^i \times (H_{A,i})^2 + S_y^j \times (H_{A,j})^2}$$

where S_x^i, $H_{A,i}$, S_y^j, and $H_{A,j}$ are calculated as in Section 3. Similarly, we can compute L_B which is the ℓ_2 norm of slowdowns of tuples produced under policy B. In order for L_A to be less than L_B, then the following inequality must be satisfied:

$$\frac{S_y^j}{\overline{C}_y^j (T_j)^2}(2W_y^j + 2T_j + \overline{C}_x^i) < \frac{S_x^i}{\overline{C}_x^i (T_i)^2}(2W_x^i + 2T_i + \overline{C}_y^j)$$

As an approximation, we drop $(2T_j + \overline{C}_x^i)$ and $(2T_i + \overline{C}_y^j)$ from the above inequality which yields to:

$$\frac{S_y^j}{\overline{C}_y^j T_j} \times \frac{W_y^j}{T_j} < \frac{S_x^i}{\overline{C}_x^i T_i} \times \frac{W_x^i}{T_i}$$

Hence, under our proposed policy *BSD*, each operator O_x^k is assigned a priority value V_x^k which is the product of the operator's normalized rate and the current highest slowdown of its pending tuples. That is:

$$V_x^k = \left(\frac{S_x^k}{\overline{C}_x^k T_k}\right)\left(\frac{W_x^k}{T_k}\right) \quad (6)$$

Notice that the term $S_x^k / \overline{C}_x^k T_k$ is the normalized output rate of operator O_x^k as defined in (3), whereas the term W_x^k / T_k is the current highest slowdown experienced by a tuple in O_x^k's input queue. As such, under *BSD*, an operator is selected either because it has a high weighted rate or because its pending tuples have acquired a high slowdown. This makes our proposed heuristic a hybrid between our previous policy for reducing the average slowdown (i.e., *HNR*) and the greedy heuristic to optimize maximum slowdown (i.e., *LSF*). Comparing the priority used in *BSD* to that used by *HNR*, we find that *BSD* considers the waiting time of tuples, and gives greater emphasis to the cost.

5. MULTI-STREAM QUERIES

In this section, we extend our work to handle multi-stream queries which contain *Join* operators and specifically, time-based sliding window joins. To simplify the discussion, we assume *Symmetric Hash Join* (SHJ) [25, 14] which is a non-blocking, in-memory join processing algorithm.

To illustrate the semantics of a time-based sliding window join, let us assume a sliding window continuous query Q that performs a join between two streams M_l and M_r with a window interval V. Each tuple that arrives at the system has a *timestamp* which is either assigned by the data source or the DSMS. For such a query Q, when a tuple t arrives at stream M_l, it will be compared against the tuples from M_r that are within V time units from t's timestamp [5, 10]. Out of those tuples, the ones that satisfy the join predicate are streamed up the query plan.

To use SHJ for performing the join operation in the query described above, hash tables HT_l and HT_r are defined over streams M_l and M_r, respectively. As a tuple t with timestamp $t.ts$ arrives at one of the streams (say M_l), it is first hashed and inserted into HT_l, then the hash value is used to probe HT_r for tuples with matching key. Out of those matching tuples, each tuple that satisfies the window predicate is concatenated to the input tuple t and a new *composite* tuple is generated.

5.1 Metrics For Joins

Now, we extend the metrics described in Section 3 for composite tuples generated by multi-stream queries.

5.1.1 Response Time of Joined Tuples

Definition 1 can be used directly to measure the response time of a composite tuple as long as the arrival time is defined. This arrival time is easily defined by considering the dependency between the two joined tuples. That is, the composite tuple cannot be generated until the arrival of the second one (similarly to [5]). Hence,

DEFINITION 5. *The arrival time A_i of a composite tuple t_i that is produced from concatenating two tuples t_l and t_r with arrival times A_l and A_r respectively is equal to $max(A_l, A_r)$.*

Thus, the response time R_i for tuple t_i is $R_i = D_i - A_i$, where D_i is the tuple output time and A_i is the arrival time.

5.1.2 Slowdown of Joined Tuples

In order to measure the slowdown of a composite tuple produced by a multi-stream query Q_k, we first need to identify the ideal processing time T_k incurred by such a tuple. For simplicity, in this section, we drop the query identifier from our notation. To compute T_k, let us consider a query consisting of four components (Figure 3): (1) a join operator O_J; (2) a left operator segment preceding the join operator E_L; (3) a right operator segment preceding the join operator E_R; and (4) a common operator segment following the join operator down to the query root E_C. Each of these segments might compose of one or more operators. In the simplest case when each segment is composed of one operator, the query plan looks like Q_1 or Q_2 in Figure 1.

A tuple that is generated by such a query is the result of concatenating two tuples t_l and t_r received from the left and right inputs,

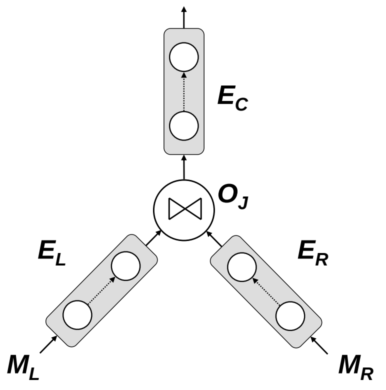

Figure 3: An example of a multi-stream query plan

respectively. The tuple t_l is first processed by E_L, then at O_J, the hash, insert, and probe operations are performed on t_l. Similarly, t_r is processed by E_R and O_J. Ultimately, the concatenated tuple generated by the join is processed by E_C. Hence,

DEFINITION 6. *The ideal processing time T_k of a composite tuple processed by a multi-stream query Q_k composed of join operator O_J, a left segment E_L, a right segment E_R, and a common segment E_C is defined as:*

$$T_k = C_L + C_R + (2 \times C_J) + C_C$$

where C_L, C_R, C_J, and C_C are the ideal total processing costs of the operators in E_L, E_R, O_J, and E_C respectively.

To compute the slowdown of a tuple it is important not to penalize the DSMS for the *dependency delay*. That is, the time that the first tuple has to spend waiting for the arrival of its matching tuple. As such, we define the slowdown incurred by a composite tuple t_i produced by a multi-stream query Q_k as follows:

$$H_i = 1 + \frac{D_i^{actual} - D_i^{ideal}}{T_k}$$

where D_i^{actual} is the actual departure time of the composite tuple which includes: 1) processing time; 2) dependency delay; and 3) queuing delay, whereas D_i^{ideal} is the ideal departure time of the composite tuple if it were the only tuple in the system and it includes all the components in D_i^{actual} except for the queuing delay.

5.2 Scheduling Multi-stream Queries

In order to solve the problem of scheduling multi-stream queries, we follow the same technique in [23, 5] where we reduce the problem to that of scheduling individual segments. Specifically, we view a multi-stream query as a set of disjoint virtual single-stream queries and assign a priority value to each operator in these virtual queries.

However, computing such priorities requires global knowledge about the selectivity of the multi-stream query. Specifically, we need to re-define the prioritizing parameters S_x and $\overline{C}_x$ in the presence of windowed-join operators. As such, let us consider a multi-stream query Q which contains a join operator O_J and operator segments E_L, E_R, and E_C as shown in Figure 3. Further, assume that the selectivities of the operators in Q are known, hence, we can compute the segments' global selectivities S_L, S_R, and S_C. Finally, assume that data arrives at the left and right streams with mean inter-arrival times τ_l and τ_r, respectively and that the query performs a time-based windowed join where the window interval is denoted by V time units.

For scheduling, we view the above query as two operator segments E_{LL} and E_{RR} where $E_{LL} =< E_L, O_J, E_C >$ and $E_{RR} =< E_R, O_J, E_C >$. For simplicity, assume we are implementing a non-preemptive scheduling policy, then it is sufficient to compute the priority values for the leaf operators in E_{LL} and E_{RR}. Let O_x be the leaf operator in E_{LL}, then the parameters S_x and $\overline{C}_x$ are defined as follows:

- **Global Selectivity** S_x is the number of tuples produced due to processing one tuple down segment E_{LL} and is defined as follows:

$$S_x = S_L \times S_J \times (S_R \times \frac{V}{\tau_R}) \times S_C$$

where $(S_R \times \frac{V}{\tau_R})$ estimates the number of tuples present in hash table HT_r at any point of time (as in [14, 5]).

- **Global Average Cost** $\overline{C}_x$ is the expected time required to process an input tuple along segment E_{LL} and is defined as:

$$\overline{C}_x = C_L + (S_L \times C_J) + (S_L \times S_J \times S_R \times \frac{V}{\tau_R} \times C_C)$$

where the first two terms define the cost for processing the input tuple, and the third term is the cost for processing all the tuples generated by concatenating the input tuple with the matching tuples in HT_r.

Using the above parameters as well as the total processing time parameter computed in Definition 6, we set the priority of each operator according to the used scheduling policy as in Sections 3 and 4. For multi-stream queries with multiple join operators, the above parameters are defined recursively.

6. IMPLEMENTATION ISSUES

At each *scheduling point*, our scheduler is invoked to decide which operator to execute next. The definition of a scheduling point depends on the scheduling level as follows:

- **Query-level Scheduling:** where the scheduling point is reached when a *query* finishes processing a tuple (i.e., non-preemptive)

- **Operator-level Scheduling:** where the scheduling point is reached when an *operator* finishes processing a tuple (i.e., preemptive).

6.1 Priority Dynamics under HNR

Under *HNR*, the priority given to each operator is static over time. Thus, the scheduler simply keeps a sorted list of pointers to operators. At each scheduling point, the scheduler traverses the list in order and selects for execution the first operator with pending tuples.

In the query-level scheduling, it is sufficient to only keep a list of the priorities of leaf operators where the priority of a leaf operator O_l is basically the normalized output rate of segment E_l.

In the operator-level scheduling, the scheduler might decide to proceed with the next operator O_x on the currently executing query or to execute a leaf operator in another query for which new tuples have arrived. As such, it is required to keep a list of the priorities of all operators, where the priority of operator O_x is computed as the normalized output rate of the segment of operators starting at O_x and ending at the root as shown in Section 3.

6.2 Priority Dynamics under BSD

Recall, the priority of an operator O_x under *BSD* depends on its static normalized output rate and the current slowdown of its pending tuple where the latter increases with time. The increase in the current slowdown for different tuples happens at different rates according to each tuple's current wait time (W) and ideal processing cost (T). As such, the priority of each operator under *BSD* should be re-computed at any instant of time. However, such an implementation renders *BSD* very impractical. An obvious way to reduce such an overhead is to implement *BSD* using a query-level scheduler; this approximation will reduce the frequency of scheduling points, however it is not enough. For instance, if there are q installed CQs, then at each scheduling point the scheduler will have to compute the priorities for q leaf operators. Next, we describe techniques for an efficient implementation of *BSD*.

6.2.1 Search Space Reduction

Notice that the priority of an operator under the non-preemptive implementation of *BSD* can be expressed by the product of two components: W_x^k and $S_x^k/(\overline{C}_x^k \times T_k^2)$ where the former is dynamic, while the latter is static which we will denote as Φ_x.

To reduce the search space, we divide the domain of priorities into *clusters* where each cluster covers a certain range in the priority spectrum. An operator belongs to a cluster if its priority falls within the range covered by the cluster. Then each cluster is assigned a new priority and all operators within a cluster inherit that priority.

Using clustering is a well know technique to reduce the search space for dynamic schedulers. In the particular context of DSMSs, Aurora uses a *uniform* clustering method for its QoS-aware scheduler. However, uniform clustering has the drawback of grouping together operators with large differences in their priorities. For example, if the priority domain is $[1, 100]$ and we want to divide it into 2 clusters, then we will end up with clusters covering the ranges [1,50] and [50,100]. Notice how the ratio between the highest and lowest priority in the second cluster is only 2, whereas that ratio in the first cluster goes up to 50.

In this paper, we propose to *logarithmically* divide the domain of priorities into clusters, where the priorities of the operators that belong to the same cluster are within a maximum value ϵ from each other. Specifically, the first cluster will cover the priority range $[\epsilon^0, \epsilon^1]$, the second covers $[\epsilon^1, \epsilon^2]$ etc.. In general, a cluster i will cover the priority range $[\epsilon^i, \epsilon^{i+1}]$ where a cluster i is assigned a *pseudo priority* equal to ϵ^i and an operator O_x will belong to cluster i if $\epsilon^i \le \Phi_x \le \epsilon^{i+1}$.

The number of resulting clusters depends on ϵ and Δ, where Δ is the ratio between the highest and the lowest priorities in the priority domain. Hence, the number of clusters m is: $m = \frac{log(\Delta)}{log(\epsilon)}$. For example, if the priority domain is $[1, 100]$, then at $\epsilon = 10$, the number of clusters is equal to 2 where the first cluster covers the priorities [1,10] and the second covers [10,100]. As one can see from this example, the ratio between the highest and lowest priority in each cluster is equal to ϵ (i.e., 10) as opposed to 2 and 50 when using uniform clustering.

Given such a clustering method, when a new tuple arrives, instead of routing it to the input queue of a leaf operator O_l^k, it is routed to the input queue of the cluster that contains O_l^k. Then at each scheduling point, the priority of each cluster is computed using the W of the oldest tuple in the cluster's input queue and the cluster's pseudo priority.

6.2.2 Search Space Pruning

The clustering method reduces the complexity of the scheduler from $O(q)$ to $O(m)$, however, we can do even better by pruning the search space. Towards this, we use the same method used in the *RxW* policy [3] and later generalized by *Fagin's Algorithm (FA)* which quickly finds the exact answer for *top k* queries [12].

FA quickly finds the exact answer for *top k* queries in a database where each object has g grades, one for each of its g attributes, and some aggregation function that combines the grades into an overall grade. *FA* requires that for each attribute there is a sorted list which lists each object and its grade under that attribute in descending order. In this paper, we do not present the details of *FA*, but we show how to map our search space to that required by *FA*.

As mentioned above, under *BSD*, our function for computing the priority of an operator cluster is the product of W and its pseudo priority. Hence, the system can keep a list of all clusters sorted in descending order of pseudo priority. Additionally, the system's input queue is already sorted by the tuples' arrival time, which makes it automatically sorted in descending order of wait time with each tuple pointing to its corresponding cluster in the cluster list. At a scheduling point, the two lists are traversed according to *FA* with $k = 1$ (i.e., find the *top 1* answer). The answer returned by *FA* is the cluster with the highest priority which is selected for execution. Note that *FA* will provide the same answer as the one returned by a linear traversal of the list. Hence, the only approximation so far is due to using the clustering method.

6.2.3 Clustered Processing

Once a cluster is selected for execution, then the tuple at the top of the cluster's input queue is processed by its corresponding query until emitted or discarded (i.e., pipelined and non-preemptive). However, it is often the case that the same tuple is to be processed by more than one query in the system. As such, once a cluster is selected by the scheduler, we execute a complete set of queries Q_c which belongs to the selected cluster and they all operate on the head-of-the-queue tuple.

This idea of clustered processing is kind of similar to the *train processing* in Aurora [9] where once a query is selected for execution, it will process a batch of pending tuples. However, each tuple in the same queue will have a different wait time, but in our case, all the queries in the same cluster will have the same pseudo priority which reduces the inaccuracy in the scheduling decision.

7. OPERATOR SHARING

Operator sharing eliminates the repetition of similar operations in different queries. Hence, a multi-query scheduler should exploit those shared operators for further optimizations. In this section we show how to set the priority of a shared operator under our proposed policies.

First, let us consider a set of operator segments SE_x in which operator O_x is shared among multiple operator segments E_x^1, E_x^2, ..., E_x^n (Figure 4) where for each segment E_x^i, we can compute: selectivity S_x^i and average cost $\overline{C}_x^i$.

Further, assume that cost of the shared operator O_x is c_x and $\overline{SC}_x$ is the average cost of executing the set of segments SE_x. Intuitively, $\overline{SC}_x$ is equal to the total average cost of executing the N segments with the cost of the shared operator O_x counted only once. Formally, the average cost $\overline{SC}_x$ of N paths sharing an operator O_x is:

$$\overline{SC}_x = \sum_{i=1}^{N} \overline{C}_x^i - \sum_{i=1}^{N-1} c_x$$

where $\overline{C}_x^i$ is the average cost of segment E_x^i and c_x is the cost of the shared operator O_x.

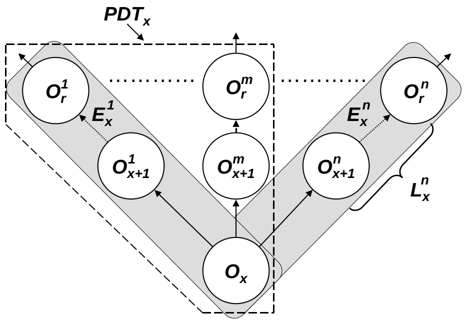

Figure 4: Multiple CQs plans sharing operator O_x

7.1 HNR with Operator Sharing

In this section, we will describe the general method for setting the priority of a shared operator under *HNR*. In the next section, we will describe the particular details of this method. Note that the *BSD* can also be extended in the same way, however the details are eliminated for brevity.

To set the priority of a shared operator under the *HNR* policy, consider two sets of operator segments SE_p and SE_q, where $SE_p = \{E_p^1, ..., E_p^N\}$ sharing operator O_p and $SE_q = \{E_q^1, ..., E_q^M\}$ sharing operator O_q. For now, assume that if a set of segments is scheduled, then all the segments within that set are executed.

To measure the impact of executing one set on the other, we will use the same concept from Inequality 2. Basically, we will measure the increase in slowdown incurred by the tuples produced from one set if the other set is scheduled for execution first. Hence, if the set of segments SE_p is executed first, then the increase in slowdown incurred by tuples from SE_q is computed as follows:

$$H_q = S_q^1 \frac{\overline{SC}_p}{T_{q,1}} + S_q^2 \frac{\overline{SC}_p}{T_{q,2}} + ... + S_q^M \frac{\overline{SC}_p}{T_{q,M}}$$

where $\overline{SC}_p$ is the amount of time that set SE_q will spend waiting for set SE_p to finish execution and $T_{q,i}$ is the ideal total processing time for the tuples processed by E_q^i.

Similarly, we can compute H_p which is the increase in slowdown incurred by tuples from SE_p. In order for H_q to be less than H_p, then the following inequality must be satisfied:

$$\overline{SC}_p \sum_{i=1}^{M} \frac{S_q^i}{T_{q,i}} < \overline{SC}_q \sum_{i=1}^{N} \frac{S_p^i}{T_{p,i}}$$

Hence, the priority of a set of operator segments SE_x that consists of N segments sharing a common operator O_x is:

$$V_x = \frac{\sum_{i=1}^{N} \frac{S_x^i}{T_{x,i}}}{\overline{SC}_x} \quad (7)$$

7.2 Priority-Defining Tree (PDT)

Setting the priority of a shared operator using all the N segments in a set is only beneficial if it maximizes the value of Equation 7. However, that is not always the case because Equation 7 is non-monotonically increasing. That is, adding a new segment to the equation might increase or decrease its value.

We definitely need to boost the priority of a shared operator, however, we do not want segments with low normalized rate to hurt those with high normalized rate by bringing down the overall priority of the shared operator. As such, we need to select from each set what we call a *Priority-Defining Tree (PDT)* which is the subset of segments that maximizes the aggregated value of the priority function. Hence, the priority of a shared operator is basically the priority of that PDT and once a shared operator is scheduled, the segments in the PDT are executed as one unit (unless it is preempted).

Accordingly, to compute the priority value V_x for operator O_x, we sort the segments according to their priority. Then, we visit the segments in descending order of priority, and only add a segment to the priority defining tree of O_x (PDT_x) if it increases the aggregate priority value, otherwise we stop and the shared operator O_x is assigned that aggregate priority value. Hence, for an operator O_x shared between N segments, with a PDT_x that is composed of m segments where $m \leq N$, the priority of O_x under the *HNR* policy is defined as:

$$V_x = \frac{\sum_{i=1}^{m} \frac{S_x^i}{T_{x,i}}}{\sum_{i=1}^{m} \overline{C}_x^i - \sum_{i=1}^{m-1} c_x}$$

If $m = N$, that is, if the PDT consists of all the segments sharing O_x, then V_x is equal to the global normalized rate as defined in Equation 7.

For any operator segment E_x^i that does not belong to PDT_x, such segment can be viewed as two component: O_x and L_x^i (as shown in Figure 4). Executing PDT_x will naturally lead to executing the O_x component of E_x^i. Scheduling L_x^i for execution depends on its priority which is computed in the normal way using its normalized rate as in Section 3. Hence, for example, in a query-level implementation of the *HNR* scheduler, the priority list will contain all the leaf operators in addition to the first operator in each segment that does not belong to any PDT.

8. EVALUATION TESTBED

To evaluate the performance of the algorithms proposed in this paper, we created a DSMS simulator with the following properties.

Queries: We simulated a DSMS with 500 registered continuous queries. The structure of the query is the same as in [11, 15] where each query consists of three operators: select, join and project. For the experiments on single-stream queries, we assume a join with a stored relation; for multi-stream queries we use window join between data streams.

Streams: We used the *LBL-PKT-4* trace from the *Internet Traffic Archive* [1] as our input stream. The trace contains an hour's worth of wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. This trace gives us a realistic data arrival pattern with On/Off traffic which is typical of many applications.

Selectivities: In order to control the selectivity, we added an extra attribute to each packet in the trace and assigned it a uniform value in the range [1,100]. Then the selectivity of the select and join operators is uniformly assigned in the range [0.1,1.0] by using predicates defined on the new attribute. Since the performance of a policy depends on its behavior toward different classes of queries, where a query class is defined by its global selectivity and cost, we chose to use the same selectivity for operators that belong to the same query. This enables us to control the creation of classes in a uniform distribution to better understand the behavior of each policy (e.g., Figure 11).

Costs: Similar to selectivity, operators that belong to the same query have the same cost, which is uniformly selected from five possible classes of costs. The cost of an operator in class i is equal to: $K \times 2^i$ time units, where $i \in$ [0,4] and K is a *scaling factor* that is used to scale the costs of operators to meet the simulated

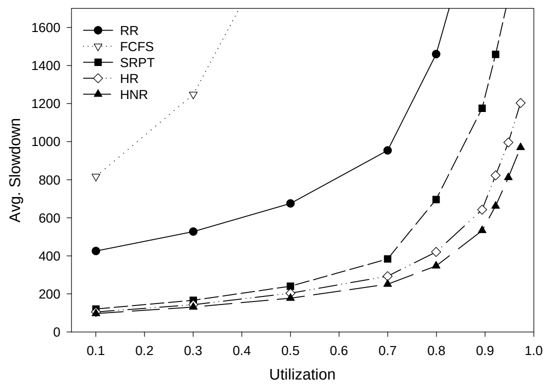

Figure 5: Avg. slowdown vs. system load

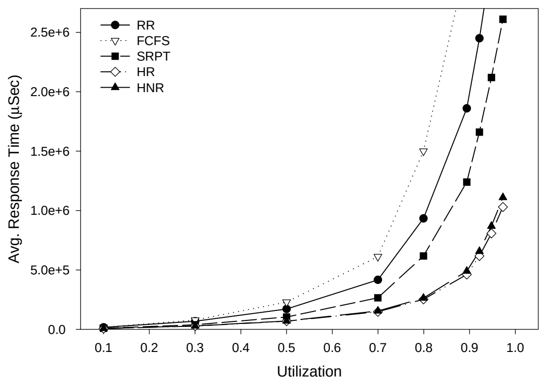

Figure 6: Avg. response vs. system load

utilization (or load). Specifically, we measure the average inter-arrival time of the data trace, then we set K so that the ratio between the total expected costs of queries and the inter-arrival time is equal to the simulated utilization.

Policies: We compared the performance of our proposed policies to the two-level scheduling scheme from Aurora where *RR* is used to schedule queries and *RB* is used to schedule operators within the query. Collectively, we refer to the Aurora scheme in our experiments as *RR*. In addition, we considered *FCFS*, *SRPT*, and our *HR* policy.

9. EXPERIMENTS

In this section, we present the performance of our proposed policies under the different QoS metrics. We also present results on the implementation of the *BSD* policy as well as the performance of the *PDT* strategy for scheduling shared operators.

9.1 Performance under Different Metrics

In this section, we present the performance of our proposed policies under the different QoS metrics.

9.1.1 Average Slowdown

Figure 5 shows how average slowdown increases with utilization. Clearly, *HNR* provides the lowest slowdown followed by *HR*. For instance at 0.7 utilization, the slowdown provided by *HNR* is 74% lower than that of *RR*, 51% lower than *SRPT*, and 18% lower than *HR*. At 0.97 utilization, *HNR* is 75% lower than *RR*, 53% lower than *SRPT*, and 20% lower than *HR*.

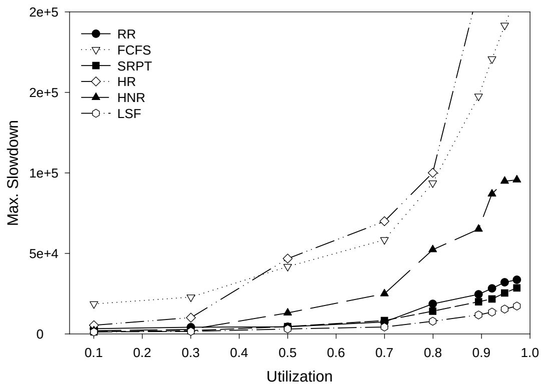

Figure 7: Max. slowdown vs. system load

9.1.2 Average Response Time

As expected, this improvement in slowdown by *HNR* would lead to an increase in response time compared to *HR* as shown in Figure 6. For instance, at 0.7 utilization, *HNR*'s response time is 4% higher than *HR* and it is 7% higher at 0.97 utilization.

9.1.3 Maximum Slowdown

In terms of worst-case performance (i.e., maximum slowdown), Figure 7 shows that *LSF* reduces the maximum slowdown by 80% compared to *HNR*. However, that improvement comes at the expense of poor average-case performance as shown in (Figure 9).

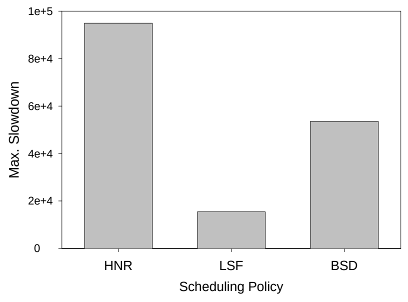

Figure 8: Max. slowdown (HNR, LSF, BSD)

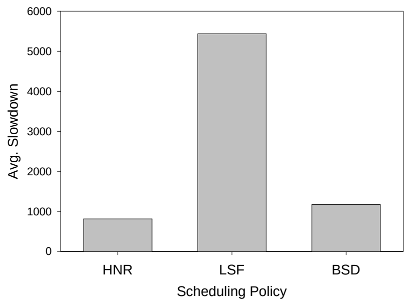

Figure 9: Avg. slowdown (HNR, LSF, BSD)

9.1.4 Trade-off in Slowdown

Figures 8 and 9 show that *BSD* can strike the fine balance between average slowdown and maximum slowdown. For instance, as shown in Figure 8, at 0.95 utilization, *BSD* decreases the maximum slowdown by 44% compared to *HNR* while Figure 9 shows

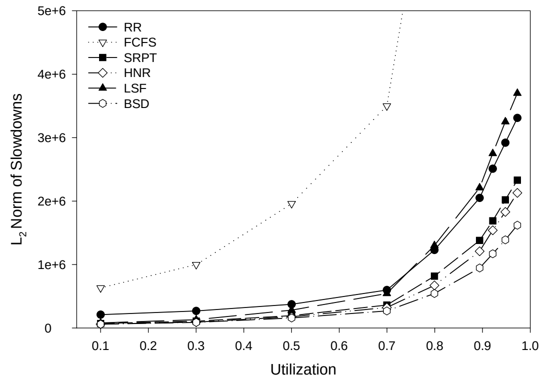

Figure 10: ℓ_2 of slowdowns vs. system load

that *BSD* decreases the average slowdown by 80% compared to *LSF* under the same utilization.

9.1.5 ℓ_2 *norm of Slowdowns*

As mentioned above, the trade-off between average and maximum slowdowns is easily captured using the ℓ_2 metric. Figure 10 shows the ℓ_2 norm of slowdowns as the utilization of the system increases. The figure shows that *BSD* reduces the ℓ_2 by up to 57% compared to *LSF* and by 24% compared to *HNR*.

9.1.6 *Slowdown per Class*

To get better insight into the behavior of the different policies toward different classes of queries, we split the workload into distinct classes (as suggested in [2]). Tuples belong to the same class if they were processed by operators with similar costs and selectivities. In Figure 11, we show the slowdown of tuples processed by the class of low-cost queries (i.e., queries where an operator cost is $K \times 2^0$) and different selectivities. The figure shows how *HR* is unfair toward the low-selectivity queries which leads to significant increase in the slowdown of the tuples processed by those queries. *HNR* is still biased toward high-selectivity queries, yet less than *HR*. Similarly, *BSD* is less biased than *HNR*. That balance allowed *BSD* to provide the best ℓ_2 norm of slowdowns as shown in Fig. 10.

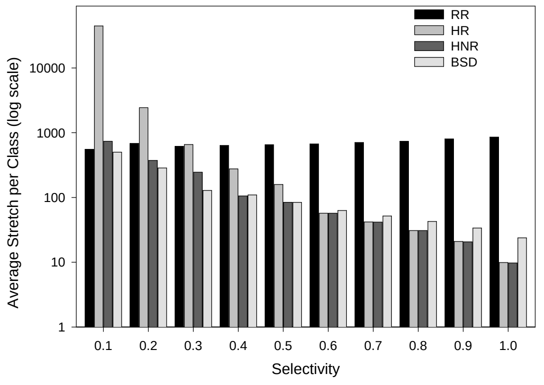

Figure 11: Slowdown per class for low-cost queries

9.1.7 ℓ_2 *norm for Multi-stream Queries*

BSD also provides the lowest ℓ_2 norm of slowdowns for multi-stream queries as shown in Figure 12. In this experimental setting, we generated a workload where queries receive input tuples from 2

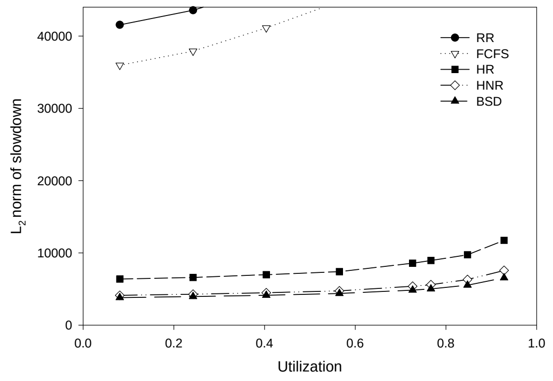

Figure 12: ℓ_2 of slowdown for multi-stream queries

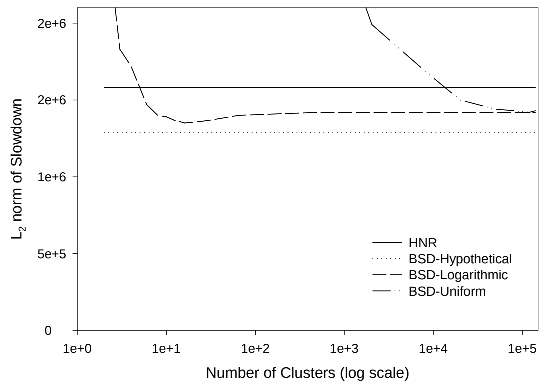

Figure 13: ℓ_2 of slowdown vs. number of clusters

data streams, generated following Poisson arrival. In this workload, the costs and selectivities of the operators are assigned uniformly as before and the windows are in the range of 1 to 10 secs. Figure 12 shows that *BSD* improves the ℓ_2 by up to 14% compared to *HNR*. It is also interesting to notice the large improvement offered by *BSD* over policies like *RR* and *FCFS*. For instance, at 0.9 utilization, *BSD* improves the performance 17 times compared to *RR*, and by 15 times compared to *FCFS*. The reason is that *RR* and *FCFS* do not exploit selectivity which plays a more significant role in the case of multi-stream queries where the selectivity of the join operator often exceeds 1.0.

9.2 Comparison of Implementation Techniques

To evaluate the impact of the implementation techniques proposed in Section 6, we compared the performance of four policies: *HNR*, *BSD-Hypothetical*, *BSD-Uniform*, and *BSD-Logarithmic*. *BSD-Hypothetical* is a version of *BSD* where we ignore the scheduling overheads. In *BSD-Uniform*, we use uniform clustering as in [9], whereas in *BSD-Logarithmic* we use our proposed logarithmic clustering. In both policies, we set the cost of each of the priority computing and comparison operations to the cost of the cheapest operator in the query plans.

Figure 13 shows the ℓ_2 norm of slowdowns provided by the four policies vs. the number of clusters (i.e., m) at 0.95 utilization. The figure shows that for *BSD-Logarithmic*, when m is small (≤ 6), its ℓ_2 might exceed that for *HNR* that is because the priority range covered by each cluster is large which decreases the accuracy of the scheduling. However, as we increase m, its performance gets closer to that of *BSD-Hypothetical* such that at 12 clusters, its provided

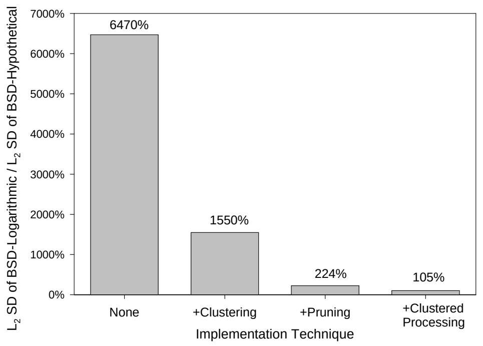

Figure 14: Efficient implementation of BSD

Metric	Policy	Strategy for Computing Priority		
		Max	Sum	PDT
Avg. Slowdown	*HNR*	261.6	244.2	201.1
ℓ_2 norm	*BSD*	66359	64066	60184

Table 2: Performance of Optimized Queries

ℓ_2 is only 5% higher than *BSD-Hypothetical*. By increasing m beyond 12, its ℓ_2 starts increasing again due to increasing the search space. For *BSD-Uniform*, it starts at a very high ℓ_2 and it decreases slowly with increasing m. That is, the accuracy of the solution is very poor when the cluster size is large. As such, *BSD-Uniform* starts to provide acceptable performance (10% higher than *BSD-Hypothetical*) when the cluster range is very small (notice that in this setting $\Delta \approx 1.2e + 05$).

Figure 14 shows the incremental gains provided by each of the proposed implementation techniques when using 12 logarithmic clusters. The figure shows that a naive implementation of *BSD* will increase the ℓ_2 norm by 6470% compared to *BSD-Hypothetical*. By incrementally adding each of the implementation techniques, we achieve a performance that is only 5% higher than *BSD-Hypothetical*.

9.3 Operator Sharing

To measure the performance of the sharing-aware versions of *HNR* and *BSD*, we created a workload in which queries are grouped randomly in sets of 10 queries each where all queries within a set share the same select operator.

Table 2 shows the two measured QoS metrics. Next to each metric is the policy that optimizes it and the performance of this policy using three variants for setting the priority. In *Max*, the overall priority is equal to the priority of that *one* segment within the group that has the maximum priority, whereas, in *Sum*, the priority is the aggregation of the priorities of *all* the segments in a group.

The table shows that the *PDT* strategy significantly improves the performance of each scheduling policy. For example, compared to the *Max* strategy, it provides 23% reduction in slowdown and 10% reduction in ℓ_2. These improvements are due to the fact that both *Sum* and *Max* could underestimate the priority of a shared operator.

10. RELATED WORK

Improving the response time of queries over data streams has been the focus of many research efforts. The work in [24] proposes rate-based query optimization as a replacement of the traditional cost-based approach. For multiple queries, multi-query optimization has been exploited by [11] to improve system throughput in the Internet and by [15] for improving throughput in TelegraphCQ.

TelegraphCQ uses a query execution model that is based on *eddies* [4]. In that model, the execution order of operators is determined at run-time. This is particularly important when the operators' costs and selectivities change over time. Similar to TelegraphCQ, our policies can work in a dynamic environment with support for monitoring the queries' costs and selectivities, and updating the priorities whenever it is necessary.

Operator scheduling has been addressed in several research efforts (e.g., [23, 9, 5, 13, 21]). The work in [23] proposes the rate-based *(RB)* scheduling policy for scheduling operators within a single query to improve response time. Aurora [9] uses a policy called Min-Latency *(ML)* which is similar to the rate-based one; ML minimizes the average tuple latency in a single query. For multiple queries, Aurora uses a two-level scheduling scheme where Round Robin (*RR*) is used to schedule queries and *ML* (or *RB*) is used to schedule operators within the query.

Aurora also proposes a QoS-aware scheduler which attempts to satisfy application-specified QoS requirements. Specifically, each query is associated with a QoS graph which defines the utility of stale output; the scheduler then tries to maximize the average QoS. In this paper, we focus on system QoS metrics that do not require the user to have any prior knowledge about the query processing requirements or to predict the appropriate QoS graph. We also considered balancing the worst- and average-case performance, which results in a more fair system.

Multi-query scheduling has also been exploited to optimize metrics other than QoS. For example, *Chain* is a multi-query scheduling policy that optimizes memory usage [5]. The work on Chain has also been extended to balance the trade-off between memory usage and response time [6]. Another metric to optimize is Quality of Data (*QoD*). In our work in [20], we propose the freshness-aware scheduling policy for improving the QoD of data streams, when QoD is defined as freshness.

Table 3 lists the scheduling policies discussed above. For each policy, it states the optimization metric targeted by the policy. It also states if the policy is used in the context of a single query or multiple queries and whether or not the policy handles multi-stream queries that contain join operators.

11. CONCLUSIONS

In the paper, we considered scheduling multiple heterogeneous CQs in a DSMS for improved QoS. To quantify such QoS, we adopted slowdown-based metrics which are better suited for heterogeneous applications. This led us to the development of a new scheduling policy that optimizes the average-case performance of a DSMS. Additionally, we proposed a hybrid policy that strikes a fine balance between the average-case performance and the worst-case performance. Further, we have extended the proposed policies to exploit operator sharing in optimized multi-query plans and to handle multi-stream queries. Finally, we have evaluated our proposed policies and their implementation experimentally and showed that our scheduling policies outperform previously proposed policies. Our next step is to incorporate our policies in our *AQSIOS* DSMS prototype.

12. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thoughtful and constructive comments. We would also like to thank Shenoda Guirguis for his help in editing the final version of this paper.

13. REFERENCES

[1] http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html.

Policy	Ref.	Objective	Metric	Single CQ	Multiple CQs	Join CQ
Rate-based (RB)	[23]	Average	Response Time	$\surd$	$\times$	$\surd$
Min-Latency (ML)	[9]	Average	Response Time	$\surd$	$\times$	$\times$
Round Robin (RR)	[9]	Average	Response Time	$\surd$	$\surd$	$\times$
Highest Rate (HR)	[19]	Average	Response Time	$\surd$	$\surd$	$\surd$
Highest Normalized Rate (HNR)	§3.3	Average	Slowdown	$\surd$	$\surd$	$\surd$
Longest Stretch First (LSF)	§4.1	Maximum	Slowdown	$\surd$	$\surd$	$\surd$
Balance Slowdown (BSD)	§4.2.2	ℓ_2	Slowdown	$\surd$	$\surd$	$\surd$
Chain	[5]	Maximum	Memory usage	$\surd$	$\surd$	$\surd$
Freshness-Aware Scheduling (FAS)	[20]	Average	Freshness	$\surd$	$\surd$	$\times$

Table 3: Classification of priority-based scheduling policies for CQs

[2] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proc. of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[3] D. Aksoy and M. Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Tran. on Networking*, 7(6):846–860, 1999.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[5] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases (VLDB J.)*, 13(4), 2004.

[7] N. Bansal and K. Pruhs. Server scheduling in the l_p norm: A rising tide lifts all boats. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 2003.

[8] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proc. of the ACM Symposium on Discrete Algorithms (SODA)*, 1998.

[9] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[10] D. Carney, U. Getintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2002.

[11] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2002.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[13] M. Hammad, M. Franklin, W. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[14] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2003.

[15] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[16] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 1993.

[17] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[18] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1), 1988.

[19] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. QoS metrics and algorithms for processing multiple continuous queries. Technical Report 06-134, Computer Science Dept., Univ. of Pittsburgh, 2006.

[20] M. A. Sharaf, A. Labrinidis, P. K. Chrysanthis, and K. Pruhs. Freshness-aware scheduling of continuous queries in the dynamic web. In *Proc. of the International Workshop on Web and Databases (WebDB)*, 2005.

[21] T. Sutherland, B. Pielech, Y. Zhu, L. Ding, and E. A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *Proc. of the International Database Engineering and Applications Symposium (IDEAS)*, 2005.

[22] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2003.

[23] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of the Very Large Data Bases (VLDB) Conference*, 2001.

[24] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[25] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.