Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries

Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs
Department of Computer Science, University of Pittsburgh

The emergence of monitoring applications has precipitated the need for Data Stream Management Systems (DSMSs) which constantly monitor incoming data feeds (through registered continuous queries), in order to detect events of interest. In this paper, we examine the problem of how to schedule multiple Continuous Queries (CQs) in a DSMS to optimize different Quality of Service (QoS) metrics. We show that, unlike traditional on-line systems, scheduling policies in DSMSs that optimize for average response time will be different from policies that optimize for average slowdown, which is a more appropriate metric to use in the presence of a heterogeneous workload. Towards this, we propose policies to optimize for the average-case performance for both metrics. Additionally, we propose a hybrid scheduling policy that strikes a fine balance between performance and fairness, by looking at both the average- and worst-case performance, for both metrics. We also show how our policies can be adaptive enough to handle the inherent dynamic nature of monitoring applications. Furthermore, we 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 consistently outperform currently used ones.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Data Stream Management System, Continuous Queries, Operator Scheduling

1. INTRODUCTION

The growing need for *monitoring applications* [Carney et al. 2002] has forced an evolution on data processing paradigms, moving from Database Management Systems (DBMSs) to Data Stream Management Systems (DSMSs). Traditional DBMSs employ a store-and-then-query data processing paradigm, where data is stored in the database and queries are submitted by the users to be answered in full, based on the current snapshot of the database. In contrast, in DSMSs, monitoring applications register Continuous Queries (CQs) which continuously process unbounded

This work is supported in part by the National Science Foundation under grant IIS-0534531.
Authors' addresses: Department of Computer Science, University of Pittsburgh, Pittsburgh PA 15260; emails: {msharaf, panos, labrinid, kirk}@cs.pitt.edu.

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 scheduling policies that optimize *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 fewer events, whereas `ANALYSIS` has high cost and it detects more events.

The most commonly used QoS metric in the literature is *average response time*. In this paper, we show that if the objective is to optimize the response time, then the "right" strategy is to schedule CQs according to their *output rate*. Specifically, we present a new scheduling policy called *Highest Rate (HR)*. *HR* generalizes the *Rate-based policy (RB)* [Urhan and Franklin 2001] 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.

Although scheduling to minimize average response time works well for homogeneous workloads, there are some well known disadvantages to using average response time as the metric to optimize when the workload is heterogeneous. In the above example, the user who 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 ratio of the response time of the job to its ideal processing time [Muthukrishnan et al. 1999]. 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 [Muthukrishnan et al. 1999]. A surprising discovery that motivated this work is that this is not the case with the *HR* policy which optimizes average response time of CQs [Sharaf et al. 2006]. In general, *HR* will not optimize average slowdown because of the "probabilistic" nature of CQs where the selectivity might not equal 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. We call this policy the *Highest Normalized Rate (HNR)* policy.

The average slowdown provided by the DSMS captures the system's average-case performance. 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 slowdown* of the system [Bender et al. 1998], i.e., 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 crucial 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 [Bansal and Pruhs 2003]. 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 problems that are unique to DSMSs and should be exploited by the query scheduler. The first problem is inherent in the dynamic nature of data streams where the distribution of data may vary significantly over time. Towards solving this problem, we propose an *adaptive scheduling* mechanism that allows our proposed policies to react quickly to changes in data distribution. The second problem is inherent in the inter-dependency between operators in CQs due to the presence of join or shared operators. Towards this, we first 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 system performance.

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

(1) We propose policies for scheduling multiple CQs that maximize the average-case performance of a DSMS, for response time and for slowdown.

(2) We propose hybrid policies that strike a fine balance between the average- and worst-case performances, for response time and for slowdown.

(3) We consider three issues that are very particular to DSMSs. Namely, we propose: (a) extending our proposed policies to handle multi-stream continuous queries, (b) exploiting sharing in optimizing multi-query plans, and (c) utilizing an adaptive scheduling mechanism.

(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 adopted from [Aksoy and Franklin 1999; Fagin et al. 2001].

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. Our experiments also highlight the memory requirements of each of our proposed policies and the trade-off between optimizing for QoS vs optimizing for memory usage.

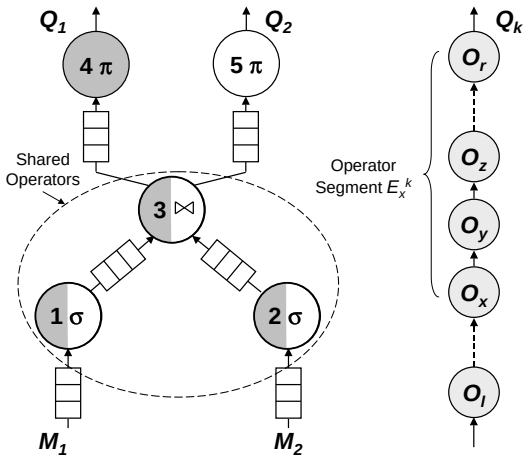

Fig. 1. Continuous Queries Plans

Road Map Section 2 provides the system model. Sections 3 and 4 define our QoS metrics and present our proposed scheduling policies. Section 5 focuses on multi-stream queries. In Sections 6 and 7, we discuss implementation details and extend our work to consider queries with shared operators. Sections 8 and 9 discuss our simulation testbed and our experimental results. Finally, 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 [Carney et al. 2002; Babcock et al. 2003], 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.

Multiple queries with common sub-expressions are usually merged together to eliminate the repetition of similar operations [Sellis 1988]. 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, as illustrated by the half-shaded pattern for these operators.

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 metrics 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 predicate) 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 registered the CQ. Clearly, in DSMSs, it is more appropriate to define response time from a data/event perspective rather than from a 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 [Tian and DeWitt 2003].

3.1.1 *Highest Rate Policy (HR).* The *Rate-based* policy (*RB*) has been shown to improve the average response time of a single query [Urhan and Franklin 2001]. In Aurora [Carney et al. 2003], *RB* was used for scheduling operators within a query, after the query had been selected by *Round Robin (RR)*. Below, we present a policy that extends *RB* for scheduling both queries and operators [Sharaf et al. 2005; Sharaf 2007].

In the basic *RB* policy, each operator path within a query is assigned a priority that is equal to its output rate. The path with the highest priority is the one scheduled for execution. In our proposed *Highest Rate* policy (*HR*), we simply view the network of multiple queries as a set of operators and at each scheduling point we select for execution the operator with the highest priority (i.e., output rate).

Specifically, under *HR*, each operator O_x^k is assigned a value called *global output rate* (GR_x^k). The output rate of an operator is basically the expected number of tuples produced per time unit due to processing one tuple by the operators along the operator segment starting at O_x^k all the way to the root O_r^k. Formally, the output rate of operator O_x^k is defined as follows:

$$GR_x^k = \frac{S_x^k}{\overline{C}_x^k} \tag{1}$$

where S_x^k and $\overline{C}_x^k$ are the operator's global selectivity and global average cost as defined in Section 2. The intuition underlying *HR* is to give higher priority to operator paths that are both productive and inexpensive. In other words, the highest priority is given to the operator paths with the minimum latency for producing one tuple.

The priority of each operator O_x^k is set to its global output rate GR_x^k, or equivalently, the output rate of the operator segment E_x^k starting at O_x^k. Hence, the priority of E_x^k is basically equal to the priority of O_x^k and 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 (or operator segment) as we will describe in Section 6.

3.2 Slowdown Metric

Average response time is an expressive metric in a homogeneous setting, i.e., when all tuples require the same processing time. However, 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. Given this realization, other on-line systems with heterogeneous workloads such as DBMSs, OSs, and Web servers have adopted *average slowdown* or *stretch* [Muthukrishnan et al. 1999] as another metric. This motivated us to consider stretch as the metric

Symbol	Description
O_x^i	Operator x in query i
$E_{x,y}^i$	Segment of operators that starts at O_x^i and ends at O_y^i
E_x^i	Segment of operators that starts at O_x^i and ends at the root O_r^i
c_x^i	Processing time/cost of operator O_x^i
s_x^i	Selectivity of operator O_x^i
$\overline{C}_x^i$	Expected processing time/cost of operator segment E_x^i
S_x^i	Selectivity of operator segment E_x^i
W_x^i	Wait time for tuple at the head of O_x^i's input queue
T_i	Ideal processing time/cost of a tuple produced by query Q_i
V_x	Window interval for join operator O_x
τ_l	Mean inter-arrival time of data stream M_l
SE_x	Set of operator segments starting at shared operator O_x
$\overline{SC}_x$	Expected processing time/cost of set of segments in SE_x

Table I. Table of Symbols

in our system.

The definition of slowdown was initiated by the database community in [Mehta and DeWitt 1993] 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 [Muthukrishnan et al. 1999]. In DSMS, we define the slowdown of a tuple as follows [Sharaf et al. 2006; Sharaf 2007]:

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 equal importance, a simple event (i.e., an event detected by a low-cost CQ) should be detected faster than a complex event (i.e., an 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. Table I summarizes the parameters used for describing the *HNR* policy for single-stream queries as well as the other scheduling policies discussed in the next Section. It also includes the parameters used for join operators (Section 5) and shared operators (Section 7).

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.

We then consider two different scheduling policies:

—**Policy (A)**, where E_x^i is executed before E_y^j, and

—**Policy (B)**, where E_y^j is executed before E_x^i.

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

$$H_A = S_x^i \times H_{A,i} + S_y^j \times H_{A,j} \tag{2}$$

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 (2),

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

Similarly, under the 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:

$$S_y^j \times \frac{\overline{C}_x^i}{T_j} < S_x^i \times \frac{\overline{C}_y^j}{T_i} \tag{3}$$

The left-hand side of Inequality 3 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.

In order to select the segment with the smallest negative impact, 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:

$$V_x^k = \frac{1}{T_k} \times \frac{S_x^k}{\overline{C}_x^k} \tag{4}$$

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 [Urhan and Franklin 2001]. 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.

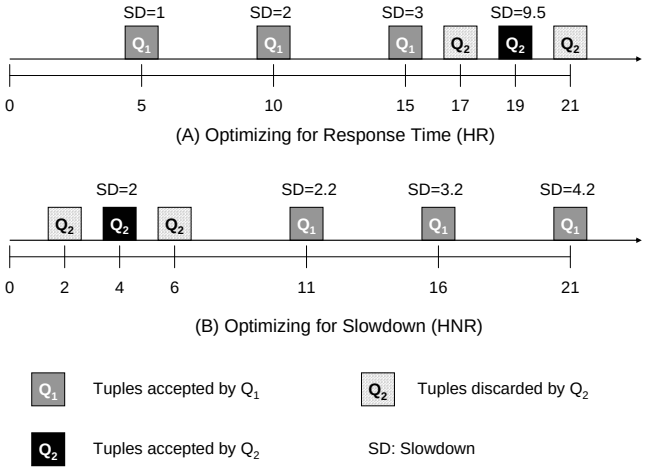

Fig. 2. The output of Example 1

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:

$$V_x^k = \frac{S_x^k}{\overline{C}_x^k} = GR_x^k \tag{5}$$

In fact, this is the prioritizing function we use in our *Highest Rate (HR)* policy for optimizing the response time presented in Section 3.1.1. The *HR* policy, 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); we assume it was the middle one in this example.

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).

	Response Time	Slowdown
HR	12.25	3.875
HNR	13.0	2.9

Table II. Results of Example 1

Table II summarizes the results of the two different policies and 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*.

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 5 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 4, $\overline{C}_x^k$ 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, which has been shown to be optimal for task scheduling when looking at response time and near optimal when looking at slowdown.

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 to minimize 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. The worst-case performance is typically measured using *maximum response time* or *maximum slowdown* [Bender et al. 1998].

DEFINITION 3. *The maximum response for N tuples is* $max(R_1, R_2, ..., R_N)$.

DEFINITION 4. *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 traditional *First-Come-First-Serve (FCFS)* that has been shown to optimize the maximum response time metric in [Bender et al. 1998]. Similarly, the traditional *Longest Stretch First (LSF)* [Acharya and Muthukrishnan 1998] has been shown to optimize the maximum slowdown. Under *LSF*, each operator O_x^k is assigned a priority V_x^k which is computed as:

$$V_x^k = \frac{W_x^k}{T_k} \tag{6}$$

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; 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, and then 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 system performance.

To get a better understanding of 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* [Bansal and Pruhs 2003]. Specifically, the ℓ_2 norm of response times, R_i, is defined as:

DEFINITION 5. *The ℓ_2 norm of response times for N tuples is equal to* $\sqrt{\sum_1^N R_i^2}$.

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.

Similarly, the ℓ_2 norm of slowdowns, H_i, is defined as:

DEFINITION 6. *The ℓ_2 norm of slowdowns for N tuples is equal to* $\sqrt{\sum_1^N H_i^2}$.

In the following sections, we present our policies for balancing the trade-off between the average and worst cases.

4.2.2 *Balancing the Trade-off for Slowdown.* Our proposed *HNR* policy is still biased toward certain classes of queries. These classes are:

(1) Queries with high productivity; and/or
(2) Queries with low processing cost.

For example, under *HNR*, a query with high cost and low productivity comes at the bottom of the priority list. When the system is overloaded, such low priority query will starve waiting for execution. This behavior may be viewed as being unfair as it yields a system with a high value for the maximum slowdown metric. The *LSF* policy, on the other hand, avoids the starvation of tuples yet yields a poor average-case performance.

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:

$$V_x^k = \left(\frac{S_x^k}{\overline{C}_x^k T_k}\right)\left(\frac{W_x^k}{T_k}\right) \tag{7}$$

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 (4), 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.

4.3 Balancing the Trade-off for Response Time

We use the same observations from above to devise a policy that balances the trade-off between average response time and maximum response time. Specifically, our proposed heuristic for balancing the trade-off under the response time metric is a hybrid of our proposed *HR* policy (that optimizes average response time) and the *FCFS* policy (that optimizes maximum response time). As such, under our proposed *Balance Response Time (BRT)* policy, each operator O_x is assigned a priority value V_x which is defined as:

$$V_x^k = (\frac{S_x^k}{\overline{C}_x^k})\,(W_x^k) \tag{8}$$

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) [Wilschut and Apers 1991; Kang et al. 2003] 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 [Babcock et al. 2003; Carney et al. 2002]. 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

Next, 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 [Babcock et al. 2003]). In other words, the composite tuple "inherits" the arrival timestamp of the latest of the tuples used to create it. Hence, the arrival time is defined as follows:

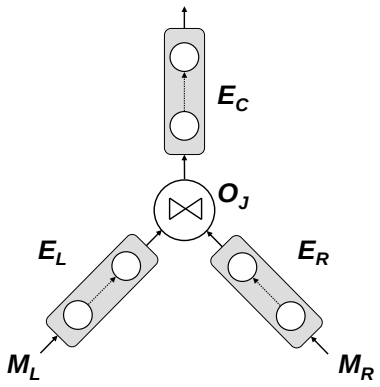

Fig. 3. An example of a multi-stream query plan

DEFINITION 7. *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 those segments might consist 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, 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, the ideal processing time of a composite tuple is defined as follows:

DEFINITION 8. *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 as in [Urhan and Franklin 2001; Babcock et al. 2003], 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, we assume we are implementing a non-preemptive scheduling policy; as such, 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 [Kang et al. 2003; Babcock et al. 2003]).

—**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 8, we set the priority of each operator by substitution in the prioritizing function corresponding to the used scheduling policy (i.e., *HR*, *HNR*, *BSD*, or *BRT*) as defined in Equations 1, 4, 7, and 8 respectively. 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 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 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. We will denote that static component $S_x^k/(\overline{C}_x^k \times T_k^2)$ 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 known 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 $\epsilon^i \leq \Phi_x \leq \epsilon^{i+1}$.

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. Clearly, this "batching" can provide significant savings in computing priorities.

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 $R \times W$ policy [Aksoy and Franklin 1999] and later generalized by *Fagin's Algorithm (FA)* which quickly finds the exact answer for *top k* queries [Fagin et al. 2001].

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 belong 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 [Carney et al. 2003] 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.

Example 2 Figure 4 shows an example that illustrates the three implementation techniques described above. The figure shows two query clusters C_x and C_y together with their pending tuples. It also shows the system's input queue where tuples are sorted according to their wait time W and the clusters list where clusters of queries are sorted according to their static priority Φ. A link between a tuple t and a cluster C means that t is the tuple at the head of C's input queue. Notice that t could be at the head of several input queues at the same time, however, at any point of time, it is only associated with the one cluster that has the highest static priority among these clusters. Finally, the priority of a pair $< t, C >$ is computed using t's wait time W and C's priority Φ as described above.

In this example, we assume that the static priority Φ_x of cluster C_x is higher than the static priority Φ_y of cluster C_y. The figure shows the status of the system's queues right after tuple t_1 has been processed by the queries in cluster C_x. At that moment, tuple t_1 is disassociated from cluster C_x and it is instead, associated with cluster C_y which follows C_x in the priority list. Additionally, tuple t_2 is associated with cluster C_x since it is the tuple currently at the head of C_x's input queue.

Using *FA*, the two lists are searched for the pair that has the highest priority to be executed. If the pair $< t_2, C_x >$ is executed first, then at the next scheduling point, tuple t_3 would be the one associated with C_x. However, if the pair $< t_1, C_y >$ is executed first, then at the next scheduling point, t_1 would be associated with the

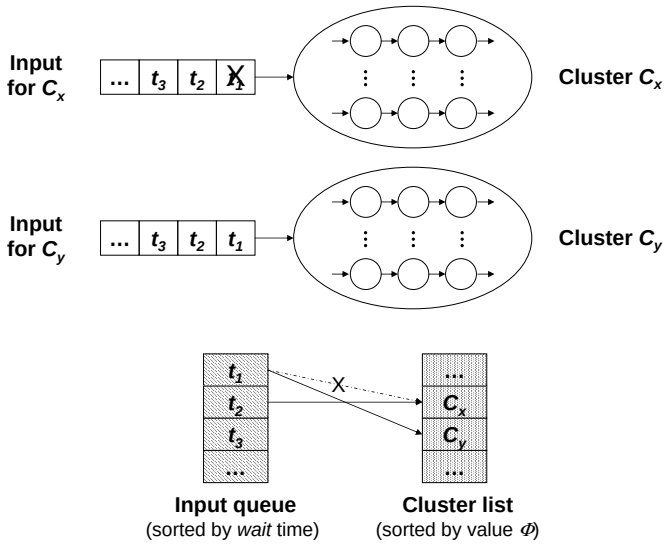

Fig. 4. An example that illustrates the different implementation techniques

next cluster in the cluster list or it would be eliminated from the queue if it has been processed by all clusters.

6.3 Adaptive Scheduling

It should be clear that the success of any of our scheduling policies proposed above relies heavily on the DSMS being able to estimate the processing time and the selectivity parameters for each operator. This would enable the scheduler to compute the right priority for each query which in turn would lead to optimizing the desired QoS metric.

The first of these parameters (i.e., the processing time of an operator) is a fairly static parameter which could be estimated once when a CQ is registered and used throughout the lifetime of the CQ. However, the selectivity parameter of an operator could be very dynamic as it depends on the data distribution in the input data stream which may vary significantly over time. For example, in an environment monitoring application, a filter like `where temp <` $40^\circ F$ will have higher selectivity during the night than during the day. (at least in some parts of the world, including Pittsburgh!).

To circumvent this problem of dynamic selectivity, we propose an *adaptive scheduling* mechanism that enables our proposed policies to deliver the expected performance all the time. Under this mechanism, the DSMS continuously monitors the execution of queries and updates the current priorities of queries based on the new estimations.

Specifically, the DSMS monitors the input and output of query operators over a time window and updates the selectivity of the operator at the end of the window. If the new selectivity is different from the old one, then the operator is assigned a new priority based on the new selectivity. The new selectivity S_{new} assigned to an

operator is basically computed as follows:

$$S_{new} = (1 - \alpha) \times S_{old} + \alpha \times \frac{N_O}{N_I}$$

where S_{old} is the current selectivity of the operator and N_O and N_I are respectively, the number of output and input tuples of an operator during the window interval. Finally, α is an aging parameter that determines how much is the weight assigned to the newly observed selectivity as compared to the selectivity currently assigned to an operator.

For instance, if α is set to 0, then the selectivity would never be updated and the system is static. On the contrary, if α is set to 1, then the system always ignores the past and the new selectivity is basically the one that has been observed during the last window. This might lead to a very unstable system especially with a short monitoring window. Hence, a value of α that is greater than 0 and less than one should allow for a stable and and adaptive system. In fact, we found that setting α to 0.175, the same value used in network congestion control mechanisms [Jacobson 1988], provides the best performance.

Notice that our mechanism for monitoring and adapting is very similar to the *ticket scheme* used in eddies-based query processing [Madden et al. 2002]. However, the ticket scheme is basically used for routing tuples between operators rather than scheduling the execution of multiple continuous queries. Specifically, the ticket scheme provides dynamic query plans that can adapt to changes in workload.

Under the ticket scheme, a lottery scheduling mechanism [Waldspurger and Weihl 1994] is used where the eddy gives a ticket to an operator whenever it consumes a tuple and takes a ticket away whenever it sends a tuple back to the eddy for further processing. To choose an operator to which a new tuple should be routed, a lottery is conducted between operators, with the chances of a particular operator winning is proportional to the number of tickets it has acquired. On the one hand, the ticket scheme gives higher priority to operators with low selectivity, which is beneficial for query plan optimization. On the other hand, our proposed policies, generally give higher priority to operators with high selectivity, which is beneficial for multiple query scheduling for improved online performance.

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 optimization. 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 5) where for each segment E_x^i, we can compute the selectivity S_x^i and the average cost $\overline{C}_x^i$.

Further, assume that the 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

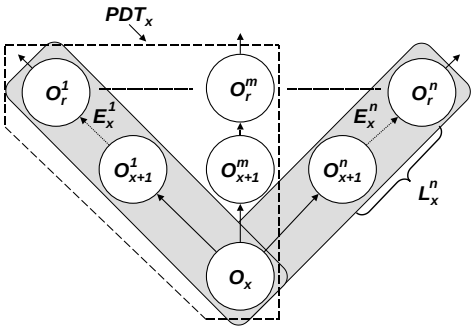

Fig. 5. Multiple CQs plans sharing operator O_x

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.

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* policy 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 the definition of Inequality 3. 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:

$$V_x = \frac{\sum_{i=1}^{N} \frac{S_x^i}{T_{x,i}}}{\overline{SC}_x} \tag{9}$$

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 9. However, that is not always the case because Equation 9 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).

In order 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 9.

For any operator segment E_x^i that does not belong to PDT_x, such segment can be viewed as two components: O_x and L_x^i (as shown in Figure 5). 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 [Chen et al. 2002; Madden et al. 2002] 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 two extra attributes to each packet in the trace and assigned each attribute 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 introduced attributes. 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 13).

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 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 *Round-Robin (RR)* is used to schedule queries and *Rate-based (RB)* is used to schedule operators within a query. Collectively, we refer to the Aurora scheme in our experiments as *RR*.

We also considered the *SRPT* policy where the priority of an operator segment is inversely proportional to its total ideal processing time, as well as the *Chain* scheduling policy [Babcock et al. 2003] which minimizes memory usage.

Here is a list of the rest of the policies considered in our experiments:

—**FCFS:** First Come First Served policy for minimizing maximum response time (Section 4.1).

—**LSF:** Longest Stretch First policy for minimizing maximum slowdown (Section 4.1).

—**HR:** Highest Rate policy for minimizing average response time (Section 3.1.1).

—**HNR:** Highest Normalized Rate policy for minimizing average slowdown (Section 3.3).

—**BRT:** Balance Response Time policy for minimizing ℓ_2 norm of response times (Section 4.3).

—**BSD:** Balance Slowdown policy for minimizing ℓ_2 norm of slowdowns (Section 4.2.2).

Parameter	Value
Base-case policies	RR, SRPT, Chain
Adopted policies	FCFS, LSF, HR, HNR, BRT, BSD
Queries	500 3-operator queries
Operator cost	$K \times 2^0$ – $K \times 2^4$ Secs
Operator selectivity	0.1 – 1.0
Window interval	1 – 10 Secs
System Utilization	0.1 – 0.99

Table III. Simulation Parameters

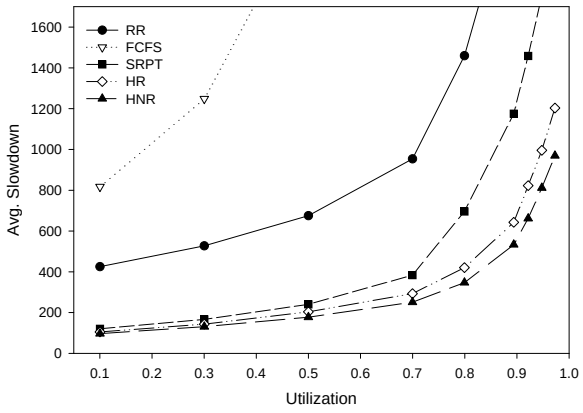

Fig. 6. Avg. slowdown vs. system load

Table III summarized the simulation parameters described above.

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 6 shows how average slowdown increases with utilization. Clearly, *HNR*, our proposed policy, 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*.

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

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

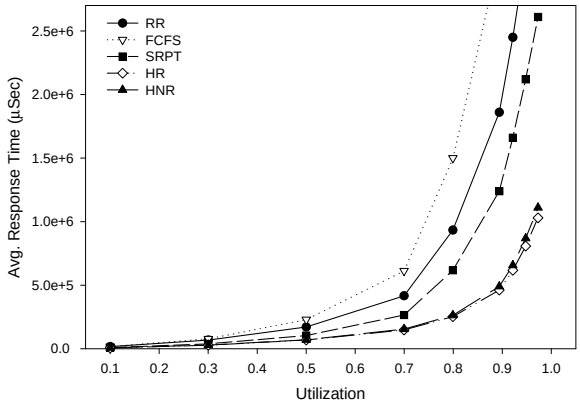

Fig. 7. Avg. response vs. system load

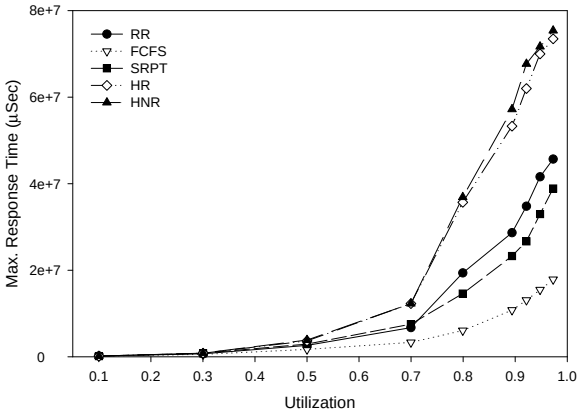

Fig. 8. Max. response time vs. system load

Figure 7. 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 Response Time.* In terms of worst-case performance, Figure 8 shows that FCFS provides the lowest maximum response time which is 75% lower than HR at 0.97 utilization. However, that improvement comes at the expense of poor average-case performance as shown in Figure 7 where the average response time provided by FCFS is 630% that of HR.

9.1.4 *Maximum Slowdown.* Similar to FCFS, Figure 9 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 depicted in Figure 10).

9.1.5 *Trade-off in Slowdown.* Figure 10 shows that *BSD* can strike a fine balance between average slowdown and maximum slowdown. For instance, as shown in Figure 10, at 0.95 utilization, *BSD* decreases the maximum slowdown by 44% compared to *HNR* while decreasing the average slowdown by 80% compared to *LSF*

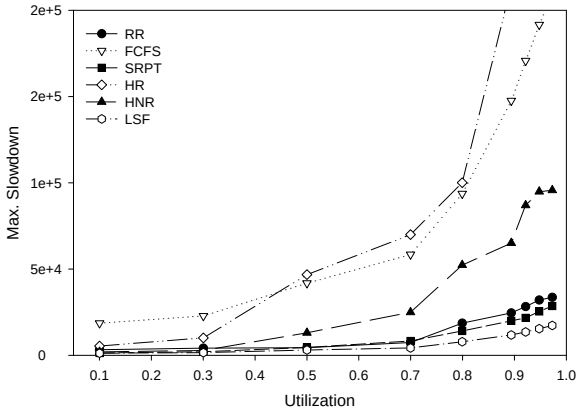

Fig. 9. Max. slowdown vs. system load

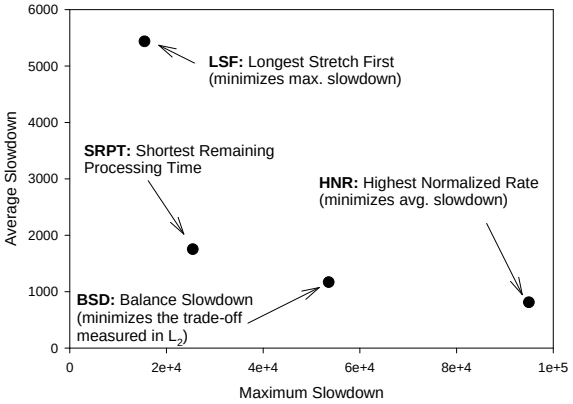

Fig. 10. Max. vs Avg. Slowdown for HNR, LSF, SRPT, and BSD

under the same utilization.

9.1.6 *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 11 shows the ℓ_2 norm of slowdowns as the utilization of the system increases. The figure shows that *BSD* reduces the ℓ_2 of slowdowns by up to 57% compared to *LSF* and by 24% compared to *HNR*.

9.1.7 *ℓ_2 norm of Response Times.* Similar to *BSD*, *BRT* reduces the ℓ_2 norm of response times by up to 51% compared to *FCFS* and 23% compared to *HR* as shown in Figure 12.

9.1.8 *Slowdown per Class.* To gain better insight into the behavior of the different policies toward different classes of queries, we split the workload into distinct classes (as suggested in [Acharya and Muthukrishnan 1998]). Tuples belong to the same class if they were processed by operators with similar costs and selectivities. In Figure 13, we show the slowdown of tuples processed by the class of low-cost

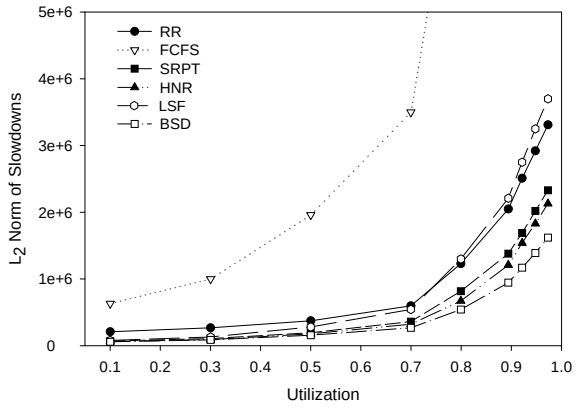

Fig. 11. ℓ_2 of slowdowns vs. system load

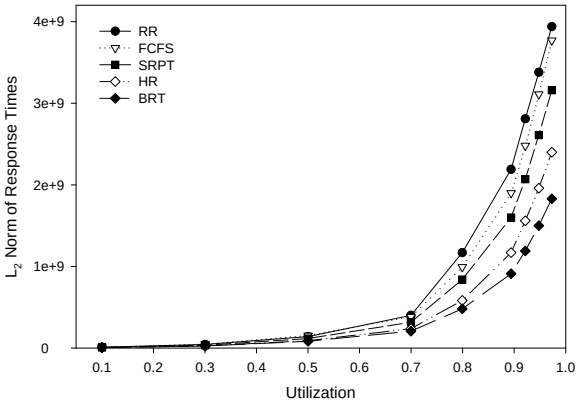

Fig. 12. ℓ_2 of response times vs. system load

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. 11.

9.1.9 *Performance over Time.* Figures 14, 15, 16, and 17 show the performance of different scheduling policies over simulation time in the interval from 10^8 to $4 \times 10^8 \mu sec$. The figures show that our proposed policies provide the best performance over time for each of the optimization metrics especially at peak times where traffic is more bursty.

9.1.10 *Impact of Selectivity.* To further study the impact of selectivity, we conducted an experiment where we assigned the same cost to all operators while varying the maximum value of selectivity assigned to an operator. For instance, if the max-

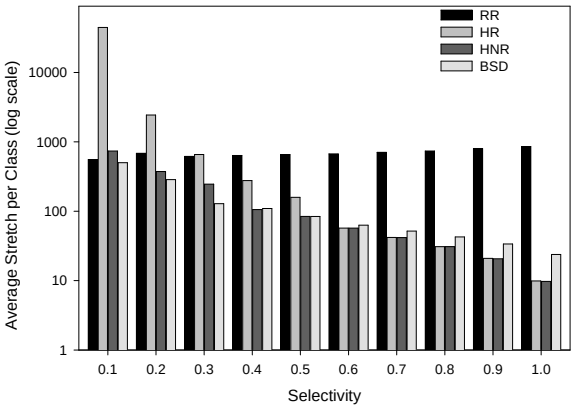

Fig. 13. Slowdown per class for low-cost queries

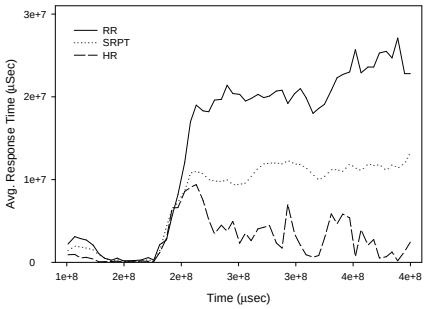

Fig. 14. Response time over time

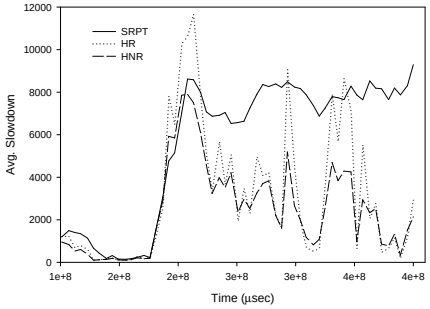

Fig. 15. Slowdown over time

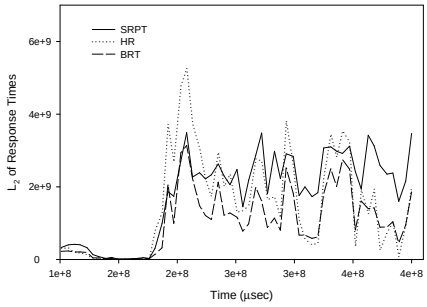

Fig. 16. ℓ_2 of response times over time

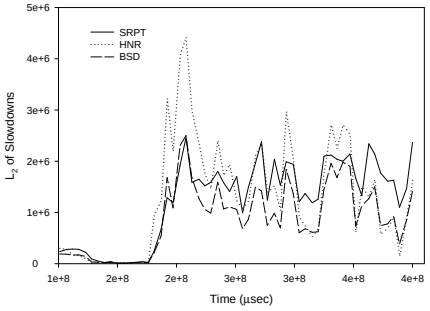

Fig. 17. ℓ_2 of slowdowns over time

imum selectivity is set to 1.0, then the selectivity value assigned to an operator is uniformly distributed in the range [0.1,1.0], whereas if the maximum is 0.5, then the selectivity value assigned to an operator is uniformly distributed in the range [0.1,0.5] etc.

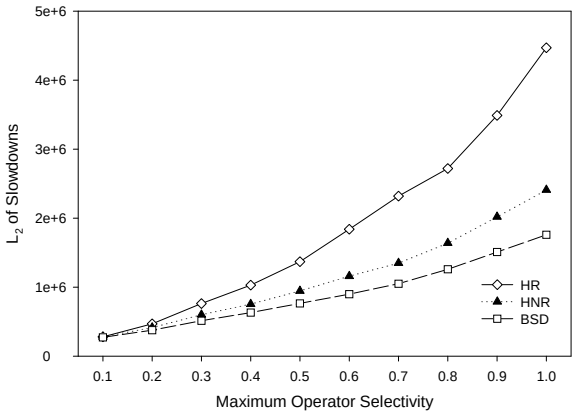

Fig. 18. ℓ_2 of slowdown vs. maximum operator selectivity

Figure 18 shows the ℓ_2 norm of slowdowns for the setting described above. The figure shows that when the maximum selectivity is 0.1, then *HR*, *HNR*, and *BSD* provide almost the same performance since all operators will have the same selectivity of 0.1. As the maximum value of selectivity increases, HR will favor queries with higher selectivity over those with lower selectivity resulting in a high ℓ_2 norm of slowdowns compared to HNR and BSD. The figure shows that BSD always provides the best performance since it considers both the ideal processing time of a query as well as the age of its pending tuples. For instance, when the maximum selectivity is 0.5, BSD reduces the ℓ_2 norm of slowdowns by 44% compared to HR and by 19% compared to HNR; at a maximum of 1.0, the ℓ_2 norm is reduced by 61% compared to HR and by 27% compared to HNR.

9.1.11 *An Oracle Scheduling Policy.* In order to validate our general strategy of using output rate (or normalized output rate) for multiple CQ scheduling, we introduce what we call an *oracle* scheduling strategy. The oracle strategy has the ability to "peek" into an input tuple and determine if it will generate an output event or if it will be discarded.

Clearly, the oracle strategy is not implementable in a real system as it requires processing the input stream (in the same way continuous queries do), before deciding what to schedule. As such, we are introducing this strategy only for the sake of comparison, since it takes the "guess" out of the scheduling decision. Specifically, at each scheduling point, the oracle strategy is able to compute the exact output rate of a query as opposed to its expected output rate as computed by our proposed policies.

As an example, consider a continuous query Q with 5 pending tuples, where only the 5th one is an event. Under regular scheduling, each of the 5 inputs is an event with probability S (which is the selectivity of the query), whereas under the oracle strategy, only the 5th tuple is an event with probability 1.0 and the other tuples are known to be discarded. Given this information, the oracle can compute the instantaneous output rate of Q as 1.0 (the number of tuples produced) divided by the amount of time needed to process the 5 pending tuples.

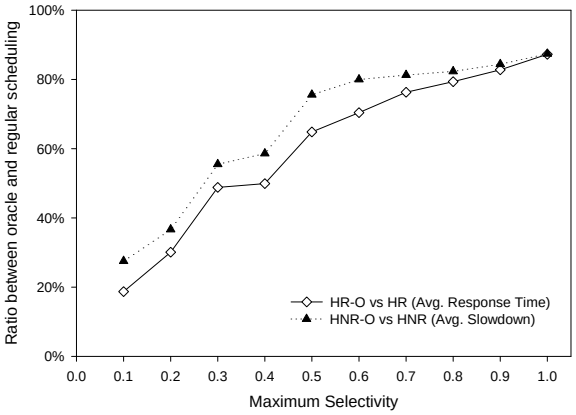

Fig. 19. Performance of an oracle scheduling policy compared to regular scheduling

Clearly, the oracle strategy has the advantage of not relying on selectivity estimation in making the scheduling decision. This is especially beneficial when the expected selectivity deviates significantly from the exhibited one. This is illustrated in Figure 19, where we plot the ratio in performance between regular policies (*HR* and *HNR*) and oracle policies (*HR-O* and *HNR-O*) while increasing the maximum selectivity in the system (as in the previous experimental setup). The figure shows that at low maximum selectivity (i.e., 0.1), the oracle can improve the performance by up to 80%. As the maximum selectivity increases, the gains decrease and drop to 12% when the maximum selectivity is 1.0. The reason is that at low selectivity there is a higher chance that an input tuple is not an event. However, only the oracle knows accurately if it is an event or not, which allows it to make a better decision. As the maximum selectivity increases, there are more queries in the system with high selectivity which means that there is a higher chance that a regular policy's guess about a tuple being an event is correct. This brings the performance of regular policies close to the oracle at higher values of maximum selectivity.

Given the above comparisons, it is clear that, in general, using variants of output rate is the right strategy to schedule CQs. However, the exhibited gains in performance depend on the accuracy in computing the rate as illustrated in Figure 19.

9.1.12 *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 20. In this experimental setting, we generated a workload where queries receive input tuples from 2 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 20 shows that *BSD* improves the ℓ_2 norm 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

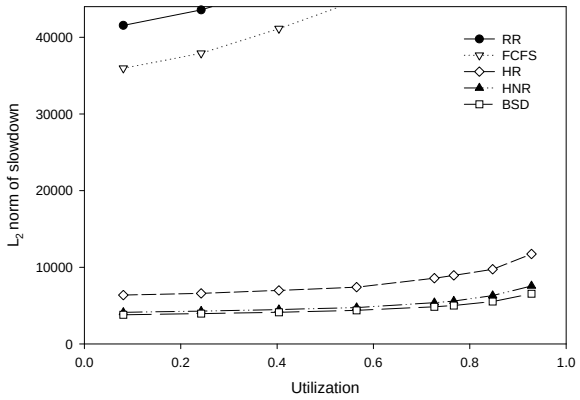

Fig. 20. ℓ_2 of slowdown for multi-stream queries

1.0.

9.2 Memory Usage

Besides CPU, memory is another resource that needs to be considered in a DSMS. For this reason, we also studied the memory requirements of each of the proposed scheduling policies. Figure 21 shows the average memory usage of our proposed policies, along with that of the *Chain* policy [Babcock et al. 2003] that was designed to minimize memory usage; we are including Chain as a yard-stick for comparison.

Figure 21 shows how policies that optimize for slowdown (i.e., the *HNR and BSD*) reduce the memory usage compared to those that optimize for response time (i.e., the *HR and BRT*). For instance, *HNR* reduces the memory usage by up to 22% compared to *HR*. Both *HNR* and *HR* give higher priorities to queries with low processing cost. Similarly, those queries are given higher priorities under policies that optimize for memory usage like *Chain*, since tuples that belong to such queries will spend a short time in memory. However, when it comes to selectivity, *Chain* gives higher priorities to queries with low selectivities, or in other words, to queries whose input tuples have a higher chance to be discarded, since they will save memory space. On the contrary, *HR* gives those queries low priority since they will not generate results. Meanwhile, since *HNR* emphasizes processing cost, it will boost the priority value of a low selectivity query with low processing cost. Hence, it allows *HNR* to schedule such queries earlier and save memory space.

Figure 21 also shows how the *BRT* and *BSD* policies provide more savings in memory usage. The reason is that under such policies, if a low selectivity query has been waiting for a long time, its priority increases until it is eventually executed. For instance, *BSD* decreases the memory usage by up to 13% compared to *HNR*.

In order to put these results into the proper perspective, we also compared the performance of *Chain* to our proposed policies under the different QoS metrics that we have studied in previous experiments. Figure 22 shows that *Chain* consistently suffers under all of the QoS metrics studied in this paper. For example at utilization 0.97, *Chain* provides 3 times the average slowdown of *HNR* which needs only 2 times the memory of *Chain*. Similarly, at utilization 0.97, *Chain* increases the ℓ_2 norm

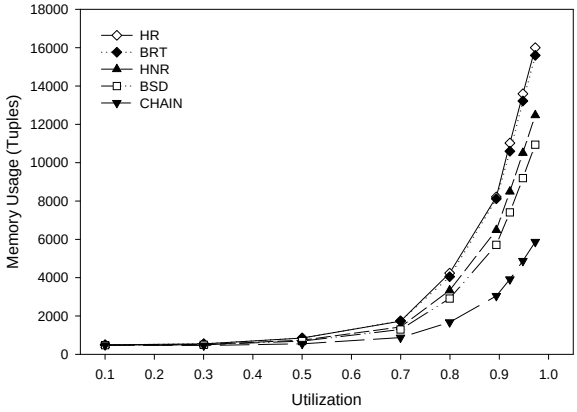

Fig. 21. Memory usage vs. system load

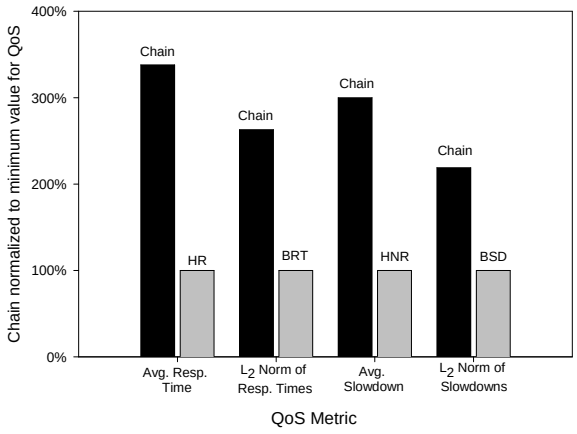

Fig. 22. Performance of Chain under QoS metrics

of slowdowns by 2.2 times compared to *BSD* although *BSD* requires memory space that is only 1.85 times more than that of *Chain*. Thus, *BSD* is able to also strike a fine balance between improving the interactive performance within acceptable memory requirements.

9.3 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* which are defined as follows:

—*BSD-Hypothetical* is a version of *BSD* where we ignore the scheduling overheads.
—*BSD-Uniform* uses uniform clustering as in [Carney et al. 2003].
—*BSD-Logarithmic* uses our proposed logarithmic clustering (described in Section 6).

In both *BSD-Uniform* and *BSD-Logarithmic*, we set the cost of each of the pri-

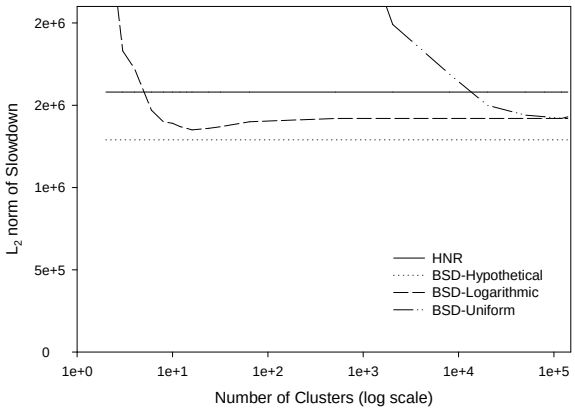

Fig. 23. ℓ_2 of slowdown vs. number of clusters

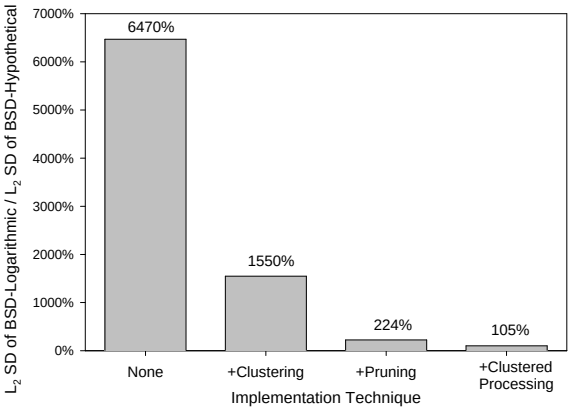

Fig. 24. Efficient implementation of BSD

ority computation and comparison operations to the cost of the cheapest operator in the query plans.

Figure 23 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*, 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 ℓ_2 norm is only 5% higher than *BSD-Hypothetical*. By increasing m beyond 12, its ℓ_2 norm starts increasing again due to increasing the search space. On the other hand, *BSD-Uniform* 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 number of clusters is small (i.e., each cluster range 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 $\Delta \approx 1.2e + 05$).

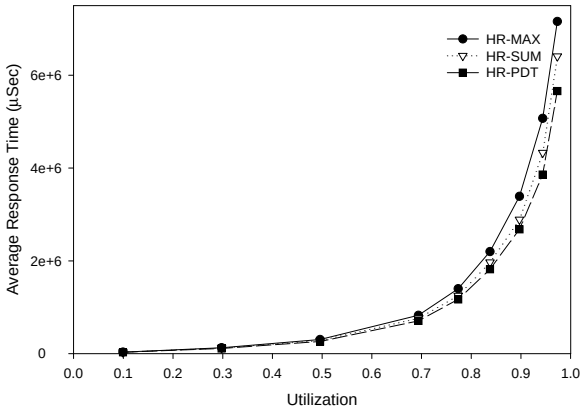

Fig. 25. Response for grouped queries

Figure 24 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*, i.e., the implementation overhead of the *BSD* policy is only 5%.

9.4 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.

Figures 25, 26, and 27 show the performance of different scheduling policies under the response time, slowdown, and ℓ_2 norm of slowdown metrics respectively. In each figure, we compare the performance of three variants for implementing the same policy that optimizes the metric under investigation. These three variants are: *Max*, *Sum*, and *PDT*, defined as follows:

—*Max:* where the shared operator priority is equal to the priority of that *one* segment within the group that has the maximum priority.

—*Sum:* where the shared operator priority is equal to the aggregation of the priorities of *all* the segments in a group.

—*PDT:* where the shared operator priority is equal to the aggregation of the priorities of the segments in its *priority-defining tree* (as described in Section 7).

The figures show that the *PDT* strategy significantly improves the performance of each scheduling policy. For example, Figure 25 shows that, compared to *Max* and *Sum*, *PDT* reduces the response time by 21% and 12% respectively, whereas the reductions in slowdown are 24% and 18% (Figure 26) and finally, the reductions in the ℓ_2 norm of slowdowns are 10% and 8% (Figure 27).

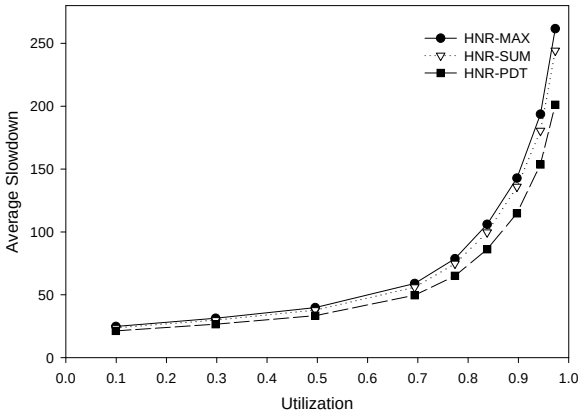

Fig. 26. Slowdown for grouped queries

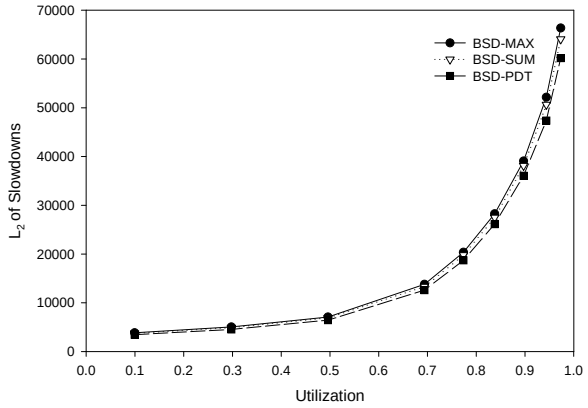

Fig. 27. ℓ_2 of slowdowns for grouped queries

9.5 Adaptive Scheduling

In all the previous experiments, queries operated on data that was generated according to a uniform distribution in the range of [1, 100]. In this experiment, we use a more dynamic setting to study the performance of our adaptive scheduling mechanism. Specifically, we divide the simulation time into 100 intervals, where the data in each interval is generated according to a Gaussian distribution that is specified by a mean and a standard deviation. The mean starts at 50.0 and it is incremented by one with every new interval.

The goal of this set of experiments is to study the behavior of the adaptive variants of our proposed policies; basically, this means that for the adaptive policies, selectivity will be estimated dynamically, as described in Section 6.3.

Figure 28 shows the ratio between the performance of the adaptive and the non-adaptive versions of each policy under the metric optimized by that policy. For instance, it compares the performance of the adaptive *HR* (i.e., *HR-A*) to the non-

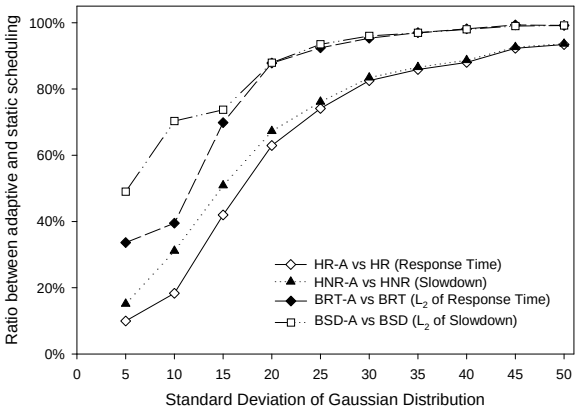

Fig. 28. Ratio of the adaptive scheduler performance vs. the static one for different metrics

adaptive *HR* under the response time metric. For example, a value of 20% for *HR-A* vs. *HR* means that *HR-A*'s response time is 20% of that of *HR*. The non-adaptive *HR* assumes that data is uniformly distributed, whereas *HR-A* monitors the data distribution and adjusts the operators selectivities and priorities accordingly.

Figure 28 shows that the adaptive versions of all policies always outperform the non-adaptive ones especially at low values of standard deviation where the distribution is highly skewed within each interval. For instance, at a standard deviation of 25, HR-A's response time is 74% of HR and HNR-A's slowdown is 76% of HNR, whereas at a standard deviation of 5, these values are 10% and 15% respectively.

Figure 28 also shows that the relative gain provided by HNR-A is lower than that provided by HR-A. This is because HNR uses the ideal processing time in its prioritizing function; this makes its non-adaptive version less sensitive to the fluctuations in selectivity. Similarly, the relative gains provided by BRT-A and BSD-A are lower than HR-A, since both BRT and BSD use the wait time in their prioritizing functions.

Obviously, the improvement in performance provided by adaptive scheduling depends on the choice of values for the monitoring window length and the aging parameter α. In the results shown in Figure 28, we selected a window of length 100 input tuples and a value of α equal to 0.175. In order to choose these specific values, we explored the combinatorial search space of the two parameters. We observed that, in general, very low values of α yield a very unstable system as it gives very low weight to the old observations, while high values of α result in an almost static system that cannot adapt fast enough to changes. Similarly, for the window length, a short window does not have enough data to provide good estimates of selectivity, while long windows provide outdated statistics.

Samples of the search space are provided in Figures 29 and 30 (at standard deviation 5 as in Figure 28). In particular, in Figure 29, we plot the performance of the adaptive scheduler compared to the static one when α is equal to 0.175 and variable window length. Similarly, in Figure 30, we plot the performance when the window length is 100 and α is variable. The figures show that, in general,

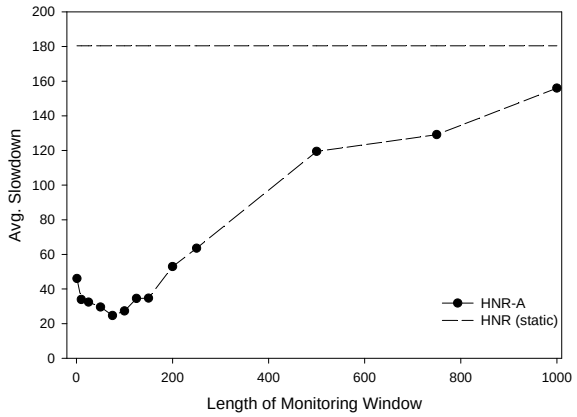

Fig. 29. Impact of monitoring window length on adaptive scheduling

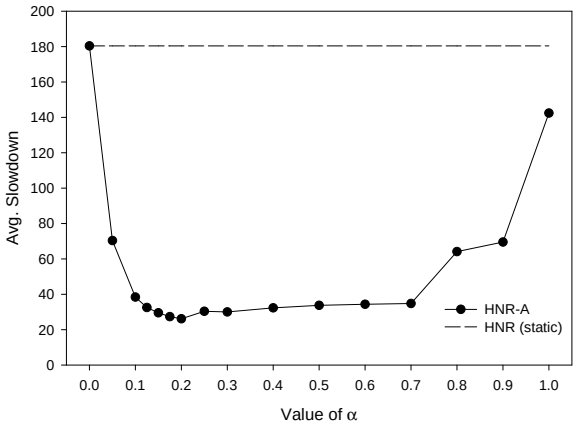

Fig. 30. Impact of α value on adaptive scheduling

windows between 50 and 150 tuples and αs between 0.1 and 0.25 provide the best performance.

10. RELATED WORK

The growing need for monitoring applications has led to the development of several prototype DSMSs (e.g., [Carney et al. 2002; Motwani et al. 2003; Carney et al. 2002; Cranor et al. 2003; Chandrasekaran et al. 2003; Hammad et al. 2004]). These prototypes utilize new techniques for the efficient processing of continuous queries over unbounded data streams. For example, [Viglas and Naughton 2002] proposed rate-based query optimization as a replacement to the traditional cost-based approach. Also, new techniques for processing aggregate CQs appeared in [Li et al. 2005], while techniques for processing join CQs appeared in [Golab and Ozsu 2003].

For multiple queries, multi-query optimization has been exploited by [Chen et al. 2002] to improve system throughput in the Internet and by [Madden et al. 2002] for

Policy and Reference		Objective	Supported CQs		
			Single	Multiple	Join
RB	*Rate-based* [Urhan and Franklin 2001]	Average Response Time	√	×	√
ML	*Min-Latency* [Carney et al. 2003]	Average Response Time	√	×	×
RR	*Round Robin* [Carney et al. 2003]	Average Response Time	√	√	×
HR	***Highest Rate*** §3.1.1	Average Response Time	√	√	√
HNR	***Highest Normalized Rate*** §3.3	Average Slowdown	√	√	√
FCFS	***First Come First Served*** §4.1	Maximum Response Time	√	√	√
LSF	***Longest Stretch First*** §4.1	Maximum Slowdown	√	√	√
BRT	***Balance Response Time*** §4.3	ℓ_2 norm of Response Time	√	√	√
BSD	***Balance Slowdown*** §4.2.2	ℓ_2 norm of Slowdown	√	√	√
Chain	Chain [Babcock et al. 2003]	Maximum Memory usage	√	√	√
FAS	*Freshness-Aware Scheduling* [Sharaf et al. 2005]	Average Freshness	√	√	×

Table IV. Classification of priority-based scheduling policies for CQs (policies in bold are presented in this paper)

improving throughput in TelegraphCQ. TelegraphCQ uses a query execution model that is based on *eddies* [Avnur and Hellerstein 2000]. 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., [Urhan and Franklin 2001; Carney et al. 2003; Babcock et al. 2003; Hammad et al. 2003; Sutherland et al. 2005]). The work in [Urhan and Franklin 2001] proposes the rate-based *(RB)* scheduling policy for scheduling operators within a single query to improve response time. Aurora [Carney et al. 2003] 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 focused 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. Specifically, we developed policies that minimize the average response time as well as the average slowdown for multiple CQs that include join and shared operators. We also considered balancing the worst- and average-case performance, and presented policies to do so for response time and for slowdown.

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 [Babcock et al. 2003]. The work on Chain has also been extended to balance the trade-off between memory usage and response time [Babcock et al. 2004]. Another metric to optimize is Quality of Data (*QoD*). In our work in [Sharaf et al. 2005], we propose the freshness-aware scheduling policy for improving the QoD of data streams, when QoD is defined as freshness.

Table IV 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 AND FUTURE WORK

In this paper, we considered the problem of scheduling multiple heterogeneous CQs in a DSMS with the goal of optimizing QoS for end users and applications. To quantify such QoS we first used the traditional metric of response time, which we defined over multiple CQs, including CQs that contain joins of multiple data streams. We also considered slowdown as another QoS metric, since we believe it to be a more fair metric for heterogeneous workloads, and, as such, more suitable for a wide range of monitoring applications.

Having defined the QoS metrics to optimize, we developed new scheduling policies that optimize the average-case performance of a DSMS for response time and for slowdown. Additionally, we proposed hybrid policies that strike a fine balance between the average-case performance and the worst-case performance, thus avoiding starvation (which is crucial for event detection CQs).

Further, we have extended the proposed policies to exploit operator sharing in optimized multi-query plans and to handle multi-stream queries. We have also augmented the proposed policies with mechanisms that ensure their adaptivity to changes in workload. Finally, we have evaluated our proposed policies and their implementation experimentally and showed that our scheduling policies consistently outperform previously proposed policies.

As the experimental results show, the magnitude of improvement in performance provided by our proposed policies depends on the considered metric as well as the query workload and the characteristics of the input data streams. For instance, we showed that the improvement is more significant when queries cover a large selectivity range. We also showed the need for good estimations of selectivity and processing costs, as well as efficient implementation and approximation algorithms. Our experimental results also show the trade-off in optimizing for QoS and optimizing for memory usage. As part of our future work, we are planning to study policies that balance the trade-off between the two objectives.

We are also planning to study the problem of scheduling multiple window-based continuous aggregate queries, which is currently an open research problem. Having that new policy in place, we believe it would be very rewarding to study the performance of all our proposed policies under more complicated workloads. That is, workloads in which different kinds of queries exist at the same time (i.e., filters, joins, or aggregates) as well as workloads with more complicated query structures (e.g., multi-way joins, or queries with shared join and aggregate operators).

We are also considering incorporating the proposed policies as part of the scheduling component in an exiting DSMS prototype, through our *AQSIOS* project. This will provide us with the ability to assess the actual gains provided by these policies in a real-system implementation. It will also provide us with insights of the scheduling overheads involved and the appropriate approximation methods to use in order to balance the trade-off between scheduling benefits and overheads.

Additionally, we are interested in developing policies that are able to balance the trade-off between different QoS metrics as well as QoD metrics. Finally, we would like to investigate policies that consider query importance and/or popularity in the scheduling decision.

REFERENCES

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

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

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

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

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

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

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

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

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

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

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

Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. 2003. Gigascope: A stream database for network applications. In *SIGMOD*.

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

Golab, L. and Ozsu, M. T. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of the Very Large Data Bases (VLDB) Conference*.

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

Hammad, M. A., Mokbel, M. F., Ali, M. H., Aref, W. G., Catlin, A. C., Elmagarmid, A. K., Eltabakh, M., Elfeky, M. G., Ghanem, T. M., Gwadera, R., Ilyas, I. F., Marzouk, M., and Xiong, X. 2004. Nile: A query processing engine for data streams. In *Proc. of the International Conference on Data Engineering (ICDE)*.

Jacobson, V. 1988. Congestion avoidance and control. In *Proc. of the ACM SIGCOMM International Conference on Communications Architectures and Protocols (SIGCOMM)*.

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

Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record 34,* 1, 39–44.

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

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

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. 2003. Query processing, resource management, and approximation in a data stream management system. In *Proc. of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

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

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

Sharaf, M. A. 2007. Metrics and algorithms for processing multiple continuous queries. Ph.D. thesis, University of Pittsburgh.

Sharaf, M. A., Chrysanthis, P. K., and Labrinidis, A. 2005. Preemptive rate-based operator scheduling in a data stream management system. In *Proc. of the Third ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05)*. Cairo, Egypt.

Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A., and Pruhs, K. 2006. Efficient scheduling of heterogeneous continuous queries. In *Proc. of the Very Large Data Bases (VLDB) Conference*.

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

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

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

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

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

Waldspurger, C. A. and Weihl, W. E. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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