Tuning QoD in Stream Processing Engines

Mohamed A. Sharaf[1] **Panos K. Chrysanthis**[2] **Alexandros Labrinidis**[2]

[1] ECE Department, University of Toronto
[2] CS Department, University of Pittsburgh
msharaf@eecg.toronto.edu {panos, labrinid}@cs.pitt.edu

Abstract

Quality of Service (QoS) and Quality of Data (QoD) are the two major dimensions for evaluating any query processing system. In the context of data stream management systems (DSMSs), multi-query scheduling has been exploited to improve QoS. In this paper, we are proposing to exploit query scheduling to improve QoD in DSMSs. Specifically, we are presenting a new policy for scheduling multiple continuous queries with the objective of maximizing the freshness of the output data streams and hence the QoD of such outputs. The proposed Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ) policy decides the execution order of continuous queries based on each query's properties (i.e., cost and selectivity) as well the properties of the input update streams (i.e., variability of updates). Our experimental results have shown that FAS-MCQ can improve QoD by up to 50% compared to existing scheduling policies used in DSMSs. Finally, we propose and evaluate a parametrized version of our FAS-MCQ scheduler that is able to balance the trade-off between freshness and response time according to the application's requirements.

Keywords: Quality of Service (QoS), Quality of Data (QoD), Data Freshness, Data Stream Management Systems, Continuous Queries, Operator Scheduling.

1 Introduction

Data streams processing is an emerging research area that is driven by the growing need for *monitoring applications*. A monitoring application continuously processes streams of data for interesting, significant, or anomalous events, as defined by the users. Monitoring applications have been used in important business and scientific information systems, for example, monitoring network performance, real-time detection of disease outbreaks, tracking the stock market, performing environmental monitoring via sensor networks, providing personalized and customized Web pages.

For example, consider the University of Pittsburgh's Realtime Outbreak of Disease Surveillance System (http://rods.health.pitt.edu). Such a system receives data from different sources (e.g., hospitals, clinics, pharmacies, etc.) and integrates it together in order to detect correlations or abnormal events. In the event of detecting a disease outbreak, CDC and health departments are notified to start mobilizing their resources.

Efficient employment of monitoring applications needs advanced data processing techniques that can support the continuous processing of rapid unbounded data streams. Such techniques go beyond the capabilities of traditional *store-then-query* Data Base Management Systems. This need has led to a new data processing paradigm and created a new generation of data processing systems, called *Data Stream Management Systems (DSMS)* that support the execution of *continuous queries* on data streams (Terry et al. 1992).

Aurora (Carney et al. 2002), STREAM (Motwani et al. 2003), TelegraphCQ (Chandrasekaran et al. 2003), Tribeca (Sullivan 1996), Gigascope (Cranor et al. 2003), Niagara (Chen et al. 2000) and Nile (Hammad et al. 2004) are examples of current prototype DSMSs. In such systems, each monitoring application registers a set of *continuous queries* (CQs), where a CQ is continuously executed with the arrival of new relevant data (Figure 1). In the Realtime Outbreak of Disease System (RODS) example, health officials register queries for tracking specific indicators of disease outbreaks by monitoring multiple input data streams (e.g., prescription data from pharmacies). The arrival of new updates on the input data streams triggers the execution of the registered CQs. The output of such a frequent execution of a continuous query is what we call an *output data stream* (see Figure 1).

As the amount of updates on the input data streams increases and the number of registered queries becomes large, advanced query processing techniques are needed in order to efficiently synchronize the results of the continuous queries with the available updates. Efficient *scheduling* of updates is one such query processing technique which successfully improves the *Quality of Data (QoD)* provided by interactive systems.

QoD can be measured in different dimensions such as accuracy, completeness, freshness etc. (Yeganeh et al. 2009). In this paper, we focus on improving QoD in a DSMS in terms of *freshness* as we assume complete processing of stream data where the DSMS operates under a reasonable load without the need for employing load shedding or approximation techniques. As such, the DSMS provides complete and accurate results that, however, might be stale which makes freshness the main QoD dimension to consider for improvement.

Freshness is especially important, when we are interested in an accurate view of the current physical

The first author is supported in part by the Ontario Ministry of Research and Innovation Postdoctoral Fellowship. This work is also partially supported by NSF under project AQSIOS (IIS-0534531) and Career award (IIS-0746696).

world, be it an outbreak of a disease (as in the RODS system) or the detection of traffic patterns and congestions in an urban setting during a physical disaster. Such accurate views must reflect *all positive event "signals"* (i.e., updates) that satisfy the registered CQs.

Freshness, as well as scheduling policies for improving freshness, has been studied in contexts such as replicated databases (Cho & Garcia-Molina 2000, 2003), Web databases (Labrinidis & Roussopoulos 2001, Qu et al. 2006, Qu & Labrinidis 2007), and distributed caches (Olston & Widom 2002). To the best of our knowledge, our work is the first to study the problem of freshness in the context of data streams. In this respect, our work can be regarded as complementary to the current work on the processing of continuous queries, which considers mainly *Quality of Service (QoS)* metrics like response time and throughput such as (Carney et al. 2003, Chandrasekaran et al. 2003, Babcock et al. 2003, Sutherland et al. 2005, Bai & Zaniolo 2008, Sharaf et al. 2008).

The contributions of this paper are as follows:

1. We propose a policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. The proposed policy, FAS-MCQ, has the following salient features:
   - It exploits the variability of the processing costs of different continuous queries registered at the DSMS.
   - It utilizes the divergence in the arrival patterns and frequencies of updates streamed from different remote data sources.
   - It considers the impact of *selectivity* on the freshness of the output data stream. Reverting back to our RODS/event detection example, our proposed policy will favor queries that lead to positive signals instead of "blindly" processing queries that lead to negative signals.
2. Beyond the basic FAS-MCQ policy, we have also explored a *weighted version* of our FAS-MCQ scheduling policy that supports applications in which queries have different priorities. These priorities could reflect *criticality*, and hence their importance with respect to QoD captured by freshness, or *popularity*, and thus be used to optimize the overall user satisfaction.
3. We study the trade-off between scheduling CQs with the goal of improving QoD (using FAS-MCQ) as opposed to scheduling to improve QoS, which is provided by Rate-based policies such as the ones proposed in (Sharaf et al. 2008, 2006, Urhan & Franklin 2001)
4. Finally, we propose a parametrized version of our FAS-MCQ scheduler that is able to balance the trade-off between QoD and QoS according to the application's requirements.

In order to evaluate our proposed scheduling policies, we have implemented a simulator of such DSMS scheduler and ran extensive experiments. As our experimental results have shown, FAS-MCQ can improve QoD by up to 55% compared to existing scheduling policies used in DSMSs. FAS-MCQ achieves this improvement by deciding the execution order of continuous queries based on individual query properties (i.e., cost and selectivity) as well as properties of the update streams (i.e., variability of updates).

The rest of this paper is organized as follows. Section 2 provides the system model. In Section 3, we define our freshness-based QoD metrics. Our proposed policies for improving freshness are presented in Section 4. In Section 5, we study the trade-off between QoD and QoS. Section 6 describes our simulation testbed, whereas Section 7 discusses our experiments and results. Section 8 surveys related work. We conclude in Section 9.

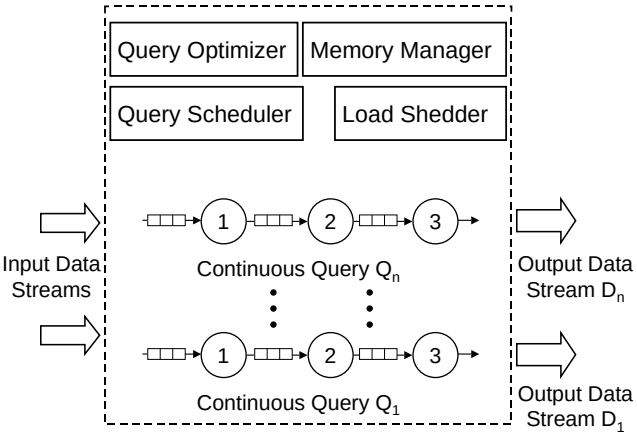

Figure 1: A DSMS hosting multiple continuous queries

2 System Model

We assume a data stream management system (DSMS) where users register multiple continuous queries over multiple input data streams (as shown in Figure 1). In this section, we discuss the details of data stream processing in a DSMS, whereas in Section 3, we define our freshness-based QoD metrics on data streams.

In a DSMS, each input data stream consists of updates at remote data sources that are either continuously pushed to the DSMS or frequently pulled from the data sources. For example, sensor networks readings are continuously pushed to the DSMS, whereas updates to Web databases are frequently pulled using Web crawlers.

Each update u_i is associated with a *timestamp* t_i. This timestamp is either assigned by the data source or by the DSMS. In the former case, the timestamp reflects the time when the update took place, whereas in the latter case, it represents the arrival time of the update at the DSMS.

In this work, we assume single-stream queries where each query is defined over a single data stream. However, data streams can be shared by multiple queries, in which case each query will operate on its own copy of the data stream. Queries can also be shared among multiple users, in which case the results will be shared among them.

A single-stream query plan can be conceptualized as a data flow diagram (Carney et al. 2002, Babcock et al. 2003), i.e., as a sequence of nodes and edges, where the nodes are operators that process data and the edges represent the flow of data from one operator to another (as in Figure 1). A query Q starts at a *leaf* node and ends at a *root* node (O_r). An edge from operator O_1 to operator O_2 means that the output of operator O_1 is an input to operator O_2. Additionally, each operator has its own input queue where data is buffered for processing.

As a new update arrives at a query Q, it passes through the sequence of operators that compose Q. An update is processed until it either produces an output or until it is discarded by some predicate in

the query. An update produces an output only when it satisfies all the predicates in the query.

In a query, each operator O_x is associated with two values:

- *processing time* or *cost* (c_x), and
- *selectivity* or *productivity* (s_x).

As in traditional database systems, an operator with selectivity s_x in a DSMS produces s_x tuples after processing one tuple for c_x time units. s_x is typically less than or equal to 1 for operators like filters. Selectivity expresses the behavior or power of a filter. Additionally, for a query Q_i, we define three parameters

1. *maximum cost* (C_i),
2. *total selectivity* or *total productivity* (S_i), and
3. *average cost* (C_i^{avg}).

For a query Q_i which is composed of a stream of operators $< O_1, O_2, O_3, ..., O_r >$, the maximum cost C_i, the total selectivity S_i and the average cost C_i^{avg} are defined as follows:

$$C_i = c_1 + c_2 + ... + c_r$$

$$S_i = s_1 \times s_2 \times ... \times s_r$$

$$C_i^{avg} = c_1 + c_2 s_1 + c_3 s_2 s_1 + ... + c_r s_{r-1} ... s_1$$

The total selectivity measures the probability that a new update will satisfy all the query predicates, while the average cost measures the expected time for processing a new update until it produces an output or until it is discarded. The average cost is computed as follows. An update starts going through the chain of operators with O_1, which has a cost of c_1. With a "probability" of s_1 (equal to the selectivity of operator O_1) the update will not be filtered out, and, as such, continue on to the next operator, O_2, which has a cost of c_2. Moving along, with a "probability" of s_2 the update will not be filtered out, and, as such, continue on to the next operator, O_3, which has a cost of c_3. Up until now, on average, the cost will be $C^{avg} = c_1 + c_2 s_1 + c_3 s_2 s_1$. This is generalized in the formula for C_i^{avg} above as in (Urhan & Franklin 2001). The maximum cost is a special case of the average cost when the selectivity of each operator in the query is 1.

3 Freshness of Data Streams

In this section, we describe our proposed metric for measuring the quality of output data streams. Our metric is based on the *freshness* of data and is similar to the ones previously used in (Cho & Garcia-Molina 2000, Labrinidis & Roussopoulos 2001, Olston & Widom 2002, Cho & Garcia-Molina 2003, Labrinidis & Roussopoulos 2004). However, it is adapted to consider the nature of continuous queries and input/output data streams.

3.1 Average Freshness for Single Streams

In a DSMS, the output of each continuous query Q is a data stream D. The arrival of new updates at the input queue of Q might lead to appending a new tuple to D. Specifically, let us assume that at time t the length of D is $|D_t|$ and there is a single update at the input queue, also with timestamp t. Further, assume that Q finishes processing that update at time t'. At this time we distinguish two cases:

- If the tuple satisfies all the query's predicates, then $|D_{t'}| = |D| + 1$. In this case, the output data stream D is considered **stale** during the interval $[t, t']$ as the new update occurred at time t and it took until time t' to append the update to the output data stream.
- If the tuple does not satisfy all the predicates, then $|D_{t'}| = |D|$. In this case, the output data stream D is considered **fresh** during the interval $[t, t']$ because the arrival of a new update has been discarded by Q. Obviously, if there is no pending update at the input queue of D, then D would also be considered fresh.

Equivalently, if we view a tuple that matches all the predicates of a query as a *positive "signal"*, then the current definition of freshness measures the amount of time that passes before the signal becomes "visible" to the end users.

Formally, to define freshness, we consider each output data stream D as an object and $F(D,t)$ is the freshness of object D at time t which is defined as follows:

$$F(D,t) = \begin{cases} 1 & \text{if } \forall u \in I_t, \sigma(u) \text{ is false} \\ 0 & \text{if } \exists u \in I_t, \sigma(u) \text{ is true} \end{cases} \quad (1)$$

where I_t is the set of input queues in Q at time t and $\sigma(u)$ is the result of applying Q's predicates on update u.

To measure the freshness of a data stream D over an entire discrete observation period from time T_x to time T_y, we have that:

$$F(D) = \frac{1}{T_y - T_x} \sum_{t=T_x}^{T_y} F(D,t) \quad (2)$$

Figure 2 shows an example of measuring the freshness of a data stream. Specifically, the figure shows two output data streams; (1) the *ideal* stream, which shows the times instants when updates became available at the DSMS; and (2) the *actual* stream, which shows the time instants when updates became available to the user. The delay between the time an update is available at the system until the time it is propagated to the user is composed of two intervals: (a) the interval where the continuous query is waiting to be scheduled for execution; and (b) the interval where the continuous query is processing the update. The sum of these two intervals represents the overall interval when the output data stream deviates from the ideal one. That is, when the output data stream is stale compared to the physical world.

In the example illustrated in Figure 2, the output data stream is stale for the intervals t_1, t_2 and t_3. Hence, the staleness of the data stream is computed as: $(t_1 + t_2 + t_3)/(T_y - T_x)$, equivalently, the freshness of the data stream is computed as: $((T_y - T_x) - (t_1 + t_2 + t_3))/(T_y - T_x)$.

3.2 Average Freshness for Multiple Streams

Having measured the average freshness for single streams, we proceed to compute the average freshness over all the M data streams maintained by the DSMS. If the freshness for each stream, D_i, is given by $F(D_i)$ using Equation 2, then the average freshness over all data streams will be:

$$F = \frac{1}{M} \sum_{i=1}^{M} F(D_i) \quad (3)$$

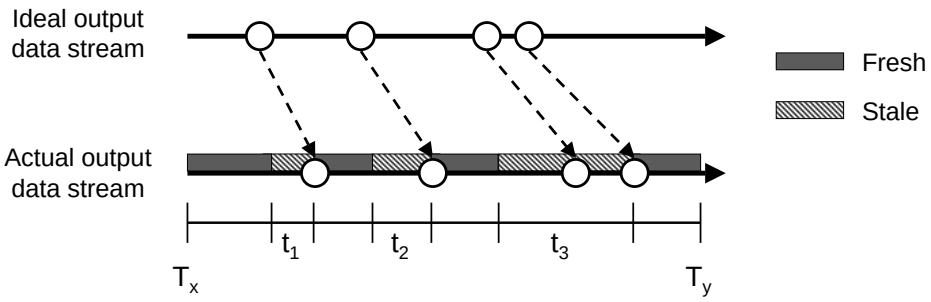

Figure 2: An example on measuring the freshness of a data stream

4 Freshness-Aware Scheduling of Multiple Continuous Queries

In this section we describe our proposed policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*.

Current work on scheduling the execution of multiple continuous queries focuses mainly on QoS metrics such as response time and throughput. Examples of such work appeared in (Chen et al. 2000, Shanmugasundaram et al. 2002, Carney et al. 2003, Chandrasekaran et al. 2003, Babcock et al. 2003, Sutherland et al. 2005, Sharaf et al. 2006, Bai & Zaniolo 2008, Sharaf et al. 2008). The scheduling policies proposed in that body of work mainly exploited the variability in the *output rate* of different CQs to improve the provided QoS. Meanwhile, previous work on synchronizing database updates exploited the variability in the *amount (frequency)* of updates to improve the provided QoD (Cho & Garcia-Molina 2000, Olston & Widom 2002, Cho & Garcia-Molina 2003).

In contrast to the work mentioned above, our proposal, *FAS-MCQ*, exploits both the variability in output rate as well as the amount of updates to improve the QoD, i.e., freshness, of output data streams.

In our previous work (Sharaf et al. 2005), we provide preliminary work on improving the freshness of data streams. In the next sections, we provide detailed analysis of our approach as well as extensions for handling multi-class continuous queries (Section 4.4) and for efficient implementation (Section 4.5). Additionally, in Section 5, we address the problem of balancing the trade-off between scheduling with the goal of improving QoD vs. QoS.

4.1 Scheduling without Selectivity

In order to explain the intuition underlying our FAS-MCQ scheduler, let us assume two queries Q_1 and Q_2, with output data streams D_1 and D_2. Each query is composed of a set of operators, each operator has a certain cost, and the selectivity of each operator is one. Hence, we can calculate for each query Q_i its maximum cost C_i as shown in Section 2. Moreover, assume that there are N_1 and N_2 pending updates for queries Q_1 and Q_2 respectively. Finally, assume that the current wait time for the update at the head of Q_1's queue is W_1, similarly, the current wait time for the update at the head of Q_2's queue is W_2.

In order to determine which of the two queries should be scheduled first for execution, we compare two policies X and Y:

- Under policy X, query Q_1 is executed before query Q_2,
- Under policy Y, query Q_2 is executed before query Q_1.

Under policy X, where query Q_1 is executed before query Q_2, the total loss in freshness, L_X, (i.e., the period of time where Q_1 and Q_2 are *stale*) can be computed as follows:

$$L_X = L_{X,1} + L_{X,2} \quad (4)$$

where $L_{X,1}$ and $L_{X,2}$ are the staleness periods experienced by Q_1 and Q_2 respectively. Since Q_1 will remain stale until all its pending updates are processed, $L_{X,1}$ is computed as follows:

$$L_{X,1} = W_1 + (N_1C_1)$$

where W_1 is the current loss in freshness (i.e., increase in staleness) and (N_1C_1) is the time required to apply all the pending updates. Similarly, $L_{X,2}$ is computed as follows:

$$L_{X,2} = (W_2 + N_1C_1) + (N_2C_2)$$

where W_2 is the current loss in freshness plus the extra amount of time (N_1C_1) where Q_2 will be waiting for Q_1 to finish execution. By substitution in Equation 4, we get

$$L_X = W_1 + (N_1C_1) + (W_2 + N_1C_1) + (N_2C_2) \quad (5)$$

Similarly, under policy Y, where Q_2 is scheduled before Q_1, we have that the total loss in freshness, L_Y will be:

$$L_Y = (W_1 + N_2C_2) + (N_1C_1) + W_2 + (N_2C_2) \quad (6)$$

In order for L_X to be less than L_Y, the following inequality must be satisfied:

$$N_1C_1 < N_2C_2 \quad (7)$$

The left-hand side of Inequality 7 shows the total increase in staleness incurred by Q_2 when Q_1 is executed first. Similarly, the right-hand side shows the total increase in staleness incurred by Q_1 when Q_2 is executed first. Hence, the inequality implies that between the two queries, we start with the one that has the lower N_iC_i value. Similarly, in the general case, where there are more than 2 queries ready for execution, we start with the one that has the lowest N_iC_i value since it will have the minimum negative impact on the freshness of the other queries in the system. Finally, it is worth mentioning that minimizing the negative impact on the overall freshness was the same general criterion that we used in our prior work on scheduling updates over materialized WebViews (Labrinidis & Roussopoulos 2001).

4.2 Scheduling with Selectivity

Assume the same setting as in the previous section, with the only difference being that the total productivity of each query Q_i is $S_i \in [0, 1]$, which is computed as in Section 2. The objective when scheduling with selectivity is the same as before: we want to minimize the total staleness. Recall from Inequality 7 that the objective of minimizing the total loss is equivalent to selecting for execution the query that minimizes the loss in freshness incurred by other queries in the system.

In the presence of selectivity, we will apply the same principle above. Towards this, we first need to compute for each output data stream D_i its *staleness probability* (P_i) given the current status of the input data stream. This is equivalent to computing the probability that at least one of the pending updates will satisfy all of Q_i's predicates. If S_i is the total selectivity of Q_i, then $(1 - S_i)^{N_i}$ is the probability that all pending updates do not satisfy Q_i's predicates, and hence $P_i = 1 - (1 - S_i)^{N_i}$ is the staleness probability for Q_i.

If out of two queries Q_1 and Q_2, Q_2 is executed before Q_1, then the expected loss in freshness incurred by Q_1 due only to the impact of processing Q_2 first will be:

$$L_{Q_1} = P_1 N_2 C_2^{avg} \tag{8}$$

where $N_2 C_2^{avg}$ is the expected time that Q_1 will be waiting for Q_2 to finish execution and P_1 is the probability that D_1 is stale in the first place. For example, in the extreme case of $S_1 = 0$, if Q_2 is executed before Q_1, it will not increase the staleness of D_1 since all the updates will not satisfy Q_1. However, at $S_1 = 1$, if Q_2 is executed before Q_1, then the staleness of D_1 will increase by $N_2 C_2^{avg}$ with probability one.

Similarly, if Q_1 is executed before Q_2, then the expected loss in freshness incurred by Q_2 only due to processing Q_1 first is computed as:

$$L_{Q_2} = P_2 N_1 C_1^{avg} \tag{9}$$

In order for L_{Q_2} to be less than L_{Q_1}, then the following inequality must be satisfied:

$$\frac{N_1 C_1^{avg}}{P_1} < \frac{N_2 C_2^{avg}}{P_2} \tag{10}$$

Thus, in our proposed policy, each query Q_i is assigned a priority value V_i which is the product of its staleness probability and the inverse of the product of its expected cost and the number of its pending updates. Formally,

$$V_i = \frac{1 - (1 - S_i)^{N_i}}{N_i C_i^{avg}} \tag{11}$$

4.3 The FAS-MCQ Policy

Our proposed policy for *Freshness-Aware Scheduling for Multiple Continuous Queries (FAS-MCQ)* uses the priority function of Equation 11 to determine the scheduling order of different queries. Under this priority function *FAS-MCQ* behaves as follows:

1. If all queries have the same number of pending tuples and the same selectivity, then FAS-MCQ selects for execution the query with the lowest cost.
2. If all queries have the same cost and the same selectivity, then FAS-MCQ selects for execution the query with less pending tuples.
3. If all queries have the same cost and the same number of pending tuples, then FAS-MCQ selects for execution the query with high staleness probability.

In case (1), *FAS-MCQ* behaves like the *Shortest Remaining Processing Time* policy. In case (2), *FAS-MCQ* gives lower priority to the query with high frequency of updates. The intuition is that when the frequency of updates is high, it will take a long time to establish the freshness of the output data stream. This will block other queries from executing and will increase the staleness of their output data streams. In case (3), *FAS-MCQ* gives lower priority to queries with low selectivity as there is a low probability that the pending updates will "survive" the filtering of the query operators and thus be appended to the output data stream.

4.4 Weighted Freshness

In many monitoring applications, some queries are more important than others. That is especially obvious in emergency systems where a few continuous queries can be more critical than others. For example, under the RODS system that monitors for disease outbreaks, it is crucial to monitor for signs of waterborne diseases in areas affected by Hurricane Katrina (and thus consider the corresponding query more crucial than the rest), whereas in other areas of the world it may be more important to monitor for signs of the avian flu. In cases like these, when the system is loaded, it is necessary to maximize the freshness of these critical queries.

Towards handling multi-class CQs, we modify our proposed FAS-MCQ policy to increase the freshness of data streams which have higher levels of importance. Specifically, we assign each continuous query Q_i a *weight* α_i. This assigned weight represents the importance of the query and it takes values in the range $(0.0, 1.0]$ where the weight 1.0 is assigned to the most important query. Hence, the objective of our policy would be to maximize the overall *weighted freshness*. A priority function that allows us to maximize the weighted freshness can be easily deduced from Equations 8 and 9. Recall that Equation 8 measures the expected loss in freshness experienced by Q_1 due to executing Q_2 first, thus, the expected loss in weighted freshness experienced by Q_1 is measured as:

$$WL_{Q_1} = \alpha_1 P_1 N_2 C_2^{avg}$$

Similarly, the expected loss in weighted freshness experienced by Q_2, when Q_1 is executed first, is measured as:

$$WL_{Q_2} = \alpha_2 P_2 N_1 C_1^{avg}$$

In order for WL_{Q_2} to be less than WL_{Q_1}, the following inequality must be satisfied:

$$\frac{N_1 C_1^{avg}}{P_1 \alpha_1} < \frac{N_2 C_2^{avg}}{P_2 \alpha_2}$$

Then, the priority assigned to each query is computed as:

$$V_i = \frac{\alpha_i (1 - (1 - S_i)^{N_i})}{N_i C_i^{avg}} \tag{12}$$

The weights of the queries can be explicitly or implicitly defined, depending on the application. For example, in the case of an application that includes queries that are critical, the critical queries can be explicitly assigned higher weights than the rest of the queries. In applications where explicit criticality/importance information is not given, an implicit

measure of importance can be derived. For example, the popularity of each query (i.e., the number of users that registered that query) can be used as the weight. In such an application, the weighted FAS-MCQ policy will provide high levels of overall user satisfaction in terms of QoD (freshness). Finally, it is worth mentioning that the weight given to a query can be dynamic; for example, it can change depending on the time of day or the day of the week (e.g., for traffic management queries).

4.5 Implementing the FAS-MCQ Scheduler

The FAS-MCQ Scheduler is invoked at every *scheduling point* and uses the current values for N_i and S_i to compute the priority of each query Q_i, according to Equation 11. In our implementation of the FAS-MCQ policy, a *scheduling point* is reached when a query finishes execution. In order to keep the scheduling overhead low when computing priorities, we use a *Calendar Queue* (Brown 1988) for priority management. Calendar queues have been widely used for implementing priority-based scheduling algorithms in high-speed networks as well as in the Aurora DSMS (Carney et al. 2003).

A calendar queue is an $O(1)$ priority queue, based on the idea of *Bucket Sort*. Specifically, the calendar queue is structured as buckets where each bucket corresponds to a class of priorities. To insert an element in the calendar queue, a hash function is used to map its priority to the corresponding bucket. To retrieve elements from the calendar queue, buckets are traversed in order. A calendar queue allows us to avoid re-computing the priorities of queries that received no new updates between consecutive scheduling points. Additionally, for queries with new updates, the amortized cost of updating the priority is of $O(1)$.

5 Scheduling for QoD vs. Scheduling for QoS

In this section we discuss the difference in behavior between scheduling with the goal of improving QoD as opposed to scheduling with the goal of improving QoS (i.e., when the objective is to minimize the *average response time*). We also present a parametrized version of our FAS-MCQ scheduler that balances the trade-off between both the QoD and QoS metrics.

5.1 Scheduling for QoS

In traditional DBMSs, the response time of a query is defined as the amount of time from the time the query arrives at the system until the time when the last tuple of the result is produced.

The definition of response time above captures the behavior of DBMSs which respond to the arrival of queries. In contrast, DSMSs respond to the arrival of new data. Hence, in a DSMS, it is more appropriate to define response time from the perspective of data (instead of queries). Therefore, we define *tuple response time* as follows:

Definition 1 *Tuple response time, T_i, for tuple i is $T_i = D_i - A_i$, where A_i is the tuple arrival time and D_i is the tuple departure time. Accordingly, the average response time for N tuples is: $\frac{1}{N}\sum_i^N T_i$.*

Under this definition, tuples that are filtered out during query processing do not contribute to the overall response time metric (Tian & DeWitt 2003).

The *Rate-based (RB)* policy has been shown to improve the average response time of a single multi-stream query with join operators (Urhan & Franklin 2001). In the basic RB policy, each operator path within a query is assigned a priority that is equal to its production rate. The path with the highest priority is the one scheduled for execution.

In our previous work (Sharaf et al. 2006, 2008), we generalize the basic Rate-based strategy for scheduling multiple continuous queries with the objective of minimizing the average response time. That is, multiple continuous queries are scheduled for execution based on their output rates. In this paper, we call that extended version of RB as *Rate-based for Multiple Continuous Queries (RB-MCQ)*.

Under RB-MCQ, at each scheduling point we select for execution the CQ with the highest priority (i.e., output rate). Specifically, under *RB-MCQ*, each query Q_i has a value called the *global output rate* (GR_i) which is defined in terms of the parameters of the CQ operators. The output rate of a query Q_i, composed of the operators $< O_1, O_2, O_3, ..., O_r >$, is basically the expected number of tuples produced per time unit due to processing one tuple by the operators along the query all the way to the root O_r. Formally,

$$GR_i = \frac{S_i}{C_i^{avg}} \tag{13}$$

or, equivalently,

$$GR_i = \frac{1-(1-S_i)}{C_i^{avg}} \tag{14}$$

where S_i and C_i^{avg} are the CQ's expected selectivity and expected cost as defined in Section 2.

5.2 Balancing the Trade-off between QoD and QoS

In this section, we present our approach for balancing the trade-off between QoS and QoD. In particular, we propose a tunable version of our FAS-MCQ scheduler that balances the trade-off between the provided response time and data freshness in a data stream management system.

Towards tuning the trade-off in the perceived performance, we argue that the distinction between scheduling for QoD and QoS is easily identified by comparing the priority functions used by FAS-MCQ (Equation 11) versus the one used by RB-MCQ (Equation 14). Specifically, the scheduling decision made by FAS-MCQ considers three factors: (1) cost (2), selectivity, and (3) number of pending tuples, whereas RB-MCQ considers only the first two factors.

As a result, FAS-MCQ might favor a query with a relatively expensive cost and very few pending tuples as opposed to RB-MCQ which might favor an inexpensive query with a large number of pending tuples. In such case, RB-MCQ may be appending tuples faster to the output data streams (i.e., providing low response time), however, the appended tuples would be stale most of the time. On the other hand, FAS-MCQ might be relatively slower in appending tuples to the output data streams (i.e., providing high response time) yet would maintain most of those output data streams as fresh as possible.

Given the above observations, we propose a parametrized version of FAS-MCQ that balances the trade-off between the achieved QoD and QoS. We will refer to this policy as FAS-MCQ(β), where β is a parameter that specifies the weight given to the number of pending tuples (i.e., N) when computing the priority of a query.

Formally, under FAS-MCQ(β) each query Q_i is assigned a priority value V_i which is computed as follows:

$$V_i = \frac{1-(1-S_i)^{N_i^\beta}}{N_i^\beta C_i^{avg}} \tag{15}$$

The parameter β takes values in the range [0.0,1.0] and it acts as a *knob* for shaping the system's behavior. For instance, for $\beta = 0.0$, FAS-MCQ(0.0) behaves like the RB-MCQ policy described above, whereas for $\beta = 1.0$, FAS-MCQ(1.0) reverts to the original FAS-MCQ described in Section 4. For settings where $0.0 < \beta < 1.0$, the system achieves the desired balance between QoD and QoS.

6 Evaluation Testbed

We have conducted several experiments to compare the performance of our proposed scheduling policy and its sensitivity to different parameters. Specifically, we compared the performance of our proposed *FAS-MCQ* policy to a two-level scheduling scheme from Aurora where Round Robin is used to schedule queries and pipelining is used to process updates within the query. Collectively, we refer to the Aurora scheme in our experiments as *RR*. We also included the *RB-MCQ* policy (Sharaf et al. 2006) described in Section 5 as well as a FCFS policy where updates are processed according to their arrival times.

Queries: We simulated a DSMS that hosts 250 registered continuous queries. The structure of the query is adapted from (Chen et al. 2002, Madden et al. 2002) where each query consists of three operators: two predicates and one projection.

Real Data Streams: We use the *LBL-PKT-4* traces from the *Internet Traffic Archive* [1]. The traces contain an hour's worth of all wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. In our experiments, we use the *TCP* and *UDP* packet traces as 2 input data streams to the system where the registered queries are uniformly assigned to any of the 2 data streams.

Synthetic Data Streams: In this setting, we generate 10 input data streams each of length 10K tuples. Initially, we generate the updates for each stream according to a Poisson distribution, with its mean inter-arrival time set according to the simulated system utilization (or load). For a utilization of 1.0, the inter-arrival time is equal to the expected time required for executing the queries in the system, whereas for lower utilizations, the mean inter-arrival time is increased proportionally. Moreover, to stress test the system, we generate a back-log of updates where we traverse the Poisson stream and group together every 10 consecutive tuples in a burst setting the arrival time of all tuples that belong to the same burst to be equal to that of the first tuple in the burst. In the default setting, 5 out of the 10 data streams are bursty.

Selectivities: In any query, the selectivity of the projection is set to 1, while the two predicates have the same value for selectivity, which is selected using a Zipf distribution from the range [0.1, 1.0]. The Zipf distribution is defined using a Zipf parameter which determines the degree of skewness. In our setting, the skewness is toward queries with selectivity equal to 1.0 and in the default setting the Zipf parameter is set to 0.0 (i.e., uniform distribution).

Costs: All operators that belong to the same query have the same cost, which is uniformly selected from three possible classes of costs. The cost of an operator in class i is equal to: $K \times 2^i$ time units, where $i \in$ [0–2] and K is the *scaling factor* which is used to scale the costs of operators to meet the desired utilization. For synthesized data, K is equal to 1. For the network traces, we measure the 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. Finally, the cost of each of the calendar queue operations is equal to the cost of the cheapest operator in the system.

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

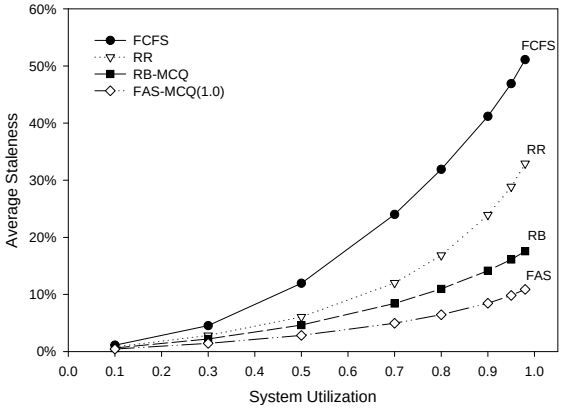

Figure 3: Average staleness vs. system utilization

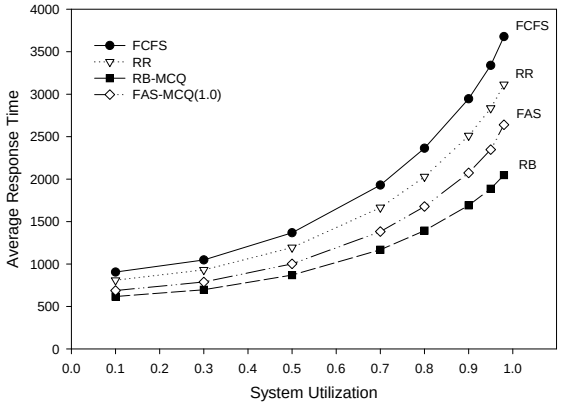

Figure 4: Average response time vs. system utilization

Table 1 summarizes our simulation parameters and settings.

7 Experiments

In this section, we present a representative sample of the experimental results obtained under the settings described in Section 6.

7.1 Impact of Utilization

Figure 3 depicts the average total staleness over all output data streams as the utilization of the DSMS increases. In this setting, we use the basic version of FAS-MCQ which is equivalent to setting β to 1.0 in FAS-MCQ(β). The figure shows that, in general, the staleness of the output data streams increases with increasing load. It also shows that the FAS-MCQ policy provides the lowest staleness for all values of utilization with RB-MCQ being the closest contender. Additionally, the relative improvement provided by FAS-MCQ compared to RB-MCQ increases with increasing utilization. For instance, at 0.1 utilization, FAS-MCQ achieves 30% reduction in staleness compared to RB-MCQ, whereas at 0.95 utilization, RB-MCQ provides a 16% staleness while FAS-MCQ reduces the staleness to 10% (i.e., a 40% improvement).

As expected, the reduction in staleness provided by FAS-MCQ comes at the expense of an increase in response time which is illustrated in Figure 4. The figure shows that, like staleness, the response time increases with increasing load. Moreover, it shows

Parameter	Value
Policies	FAS-MCQ, RB-MCQ, RR, FCFS
Number of Queries	250
Number of Operators per Query	3
Operators' Costs	$1K$, $2K$, $4K$
Operators' Selectivities	0.1–1.0
Utilization	0.1–0.99
Data Streams	real and synthetic
Number of Data Streams	2 (real) and 10 (synthetic)
Number of Bursty Streams	0–10

Table 1: Simulation Parameters

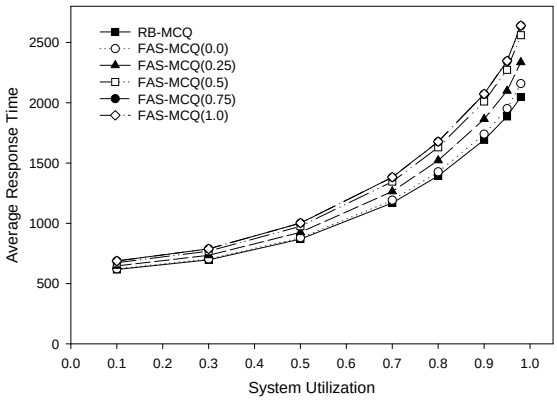

Figure 5: Response time for different βs

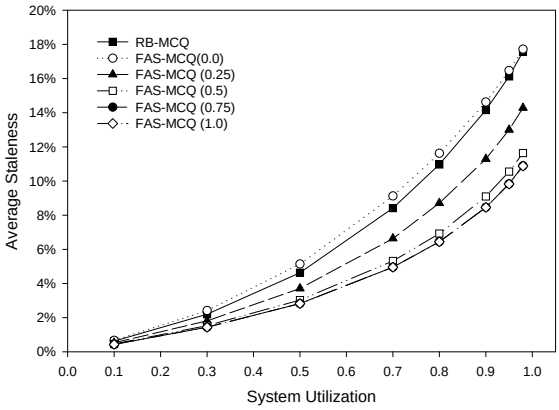

Figure 6: Staleness for different βs

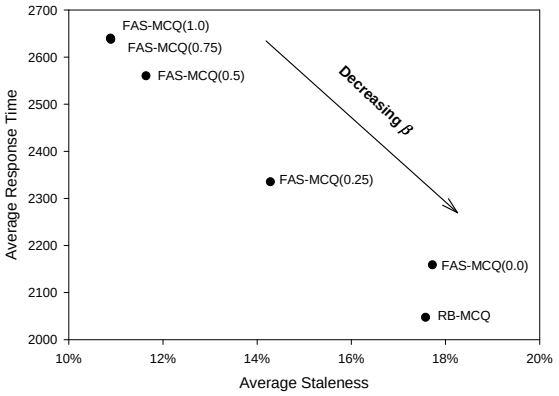

Figure 7: Trade-off between staleness and response time at utilization 0.95

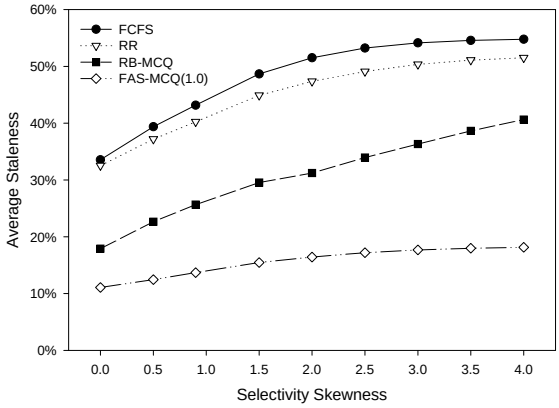

Figure 8: Staleness vs. skewness in selectivity

that RB-MCQ reduces the response time compared to FAS-MCQ. For example, at 0.95 utilization, the response time provided by FAS-MCQ is 23% higher than that of RB-MCQ (at a 40% improvement in staleness compared to RB-MCQ as previously shown in Figure 3). The trade-off between QoD (i.e., freshness) and QoS (i.e., response time) is further illustrated using Figures 5 and 6 as explained next.

7.2 Staleness vs. Response Time

Figures 5 and 6 show the average staleness and average response time for the same simulation settings used in the previous experiment. In addition to illustrating the difference in behavior between RB-MCQ and FAS-MCQ, the figures also show the performance of the parametrized FAS-MCQ(β) policy. Figure 5 shows how the response time of FAS-MCQ decreases by decreasing the value of β down to $\beta = 0.0$. Notice that at $\beta = 0.0$, the response time of FAS-MCQ(0.0) is just slightly higher than RB-MCQ which is due to the scheduling overheads. On the other hand, Figure 5 shows the reduction in staleness with increasing values of β.

To better assess the magnitude of the trade-off, we plot the performance of the different policies at utilization 0.95 in Figure 7. For instance, the figure shows that FAS-MCQ(1.0) reduces the staleness by 40% while increasing the response time by 23%, whereas FAS-MCQ(0.25) reduces the staleness by 20% and increases the response time by 14%.

7.3 Impact of Selectivity

Figure 8 shows the average staleness for an experiment where all operators have the same cost, utilization is set to 95%, and the skewness of selectivity is variable. Recall that we control the degree of skewness using a Zipf parameter. Specifically, setting the Zipf parameter to 0.0 results in a uniform distribution of selectivity, whereas by the increasing its value the distribution is skewed towards high values. That is, most of the registered CQs are productive.

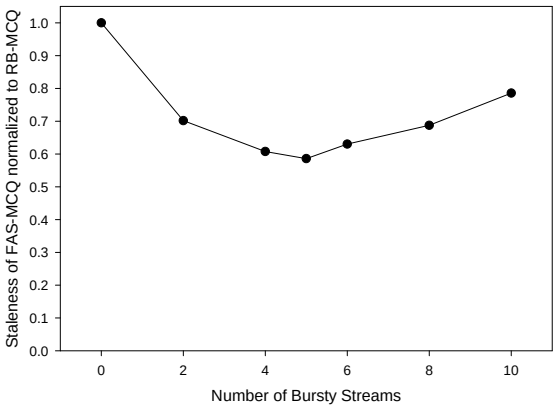

Figure 9: Staleness vs. number of bursty streams (out of 10)

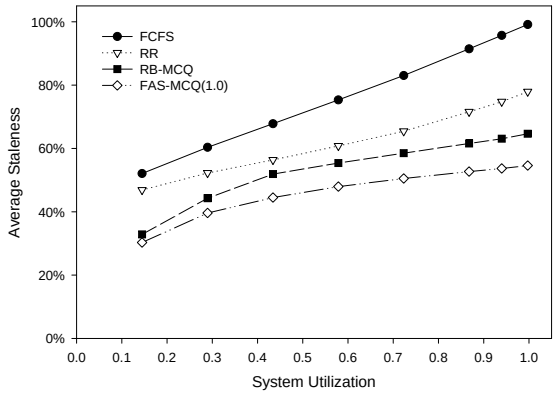

Figure 10: Staleness vs. system utilization (real data traces)

Figure 8 shows that by increasing the skewness, the staleness provided by all policies increases. This is because when most of the queries have high selectivity, then the arrival of new updates will render the output data streams stale most of the time. The figure also shows that the gains provided by FAS-MCQ compared to RB-MCQ increase with increasing the skewness. For instance, at 0.0, FAS-MCQ reduces the staleness by 40% compared to RB-MCQ. This reduction goes up to 55% when the distribution is highly skewed. The reason is that at a highly skewed distribution, all queries will have the same cost and most of them will have the same selectivity, hence, for RB-MCQ most queries will have the same priority. That is in contrast with FAS-MCQ which will utilize the extra information provided by the number of pending tuples to differentiate between queries and to assign higher priorities to queries that feed a stale stream or a stream that could be quickly brought to freshness.

7.4 Impact of Bursts

The setting for this experiment is the same as the default one. The DSMS utilization, however, is set to 95% at all points. Additionally, we plot the average staleness as the number of bursty input streams increases as shown in Figure 9. For instance, at a value of 0, all the arrivals follow a Poisson distribution with no bursts, whereas at 10, all input streams are bursty.

Figure 9 shows the staleness of FAS-MCQ normalized to that of RB-MCQ. Hence, the smaller the value the bigger the reduction. The figure shows that as the number of bursty streams increases, the reduction in staleness provided by FAS-MCQ compared to RB-MCQ increases up until there are 5 bursty streams. At that point, FAS-MCQ reduces the staleness by 40%. After that point, the performance of the two policies gets closer. The explanation is that at a moderate number of bursty streams (up to 5 streams for this setting), FAS-MCQ has a better chance to find a query with a short queue of pending updates to schedule for execution. As the number of bursty streams increases, the chance of finding such a query decreases, and as such, RB-MCQ is performing reasonably well. For instance, at 10 bursty streams, FAS-MCQ reduces the staleness by only 22% compared to RB-MCQ.

7.5 Real Data

Figure 10 shows the results for our final experiment where we use real network traces. The selectivities and costs of operators are the same as in the first experiment.

In Figure 10, the behavior of the different scheduling algorithms is consistent with the previous experiments, where FAS-MCQ provides the lowest staleness followed by RB-MCQ, then RR and FCFS. Additionally, it shows the relatively high values of staleness exhibited by all policies, which is explained by the fact that the two traces are highly bursty, reflecting an ON/OFF traffic pattern.

8 Related Work

Improving the QoS of multiple continuous queries has been the focus of many research efforts. For example, multi-query optimization has been exploited in (Chen et al. 2000) to improve the system throughput in an Internet environment and in (Madden et al. 2002) for improving the throughput of a data stream management system. Multi-query scheduling has been exploited by Aurora to achieve better response time or to satisfy application-specified QoS requirements (Carney et al. 2003). The work in (Babcock et al. 2003) employs a scheduler for minimizing the memory utilization. Moreover, balancing the trade-off between multiple QoS metrics has been studied in (Sutherland et al. 2005, Bai & Zaniolo 2008)

To the best of our knowledge, no previous work has proposed multi-query scheduling policies for improving the QoD provided by continuous queries. *Load shedding*, however, has been devised as a technique to control the degree of degradation in the provided QoD under overloaded conditions. Several load shedding techniques have been proposed in (Tatbul et al. 2003, Babcock et al. 2004, Tatbul & Zdonik 2006, Tatbul et al. 2007)

Scheduling policies for improving the QoD have been studied in the context of replicated databases and in web databases. For example, the work in (Cho & Garcia-Molina 2000, 2003) provides policies for crawling the Web in order to refresh a local database, where it made the observation that a data item that is updated more often should be synchronized less often. The same observation has been exploited in (Olston & Widom 2002) for refreshing distributed caches and in (Lam & Garcia-Molina 2003) for multi-casting updates. In this paper, we utilize the same observation for improving data stream freshness. Our work, however, makes the scheduling decision based on the current status of the DSMS queues (i.e., the number of pending updates) as opposed to assuming a mathematical model for updates as in (Cho & Garcia-Molina 2000, 2003).

The work in (Labrinidis & Roussopoulos 2001) studies the problem of propagating the updates to derived views. It proposes a scheduling policy for applying the updates that considers the divergence

in the computation costs of different views. Similarly, our proposed *FAS-MCQ* considers the different processing costs of the registered multiple continuous queries. Moreover, *FAS-MCQ* generalizes the work in (Labrinidis & Roussopoulos 2001) by considering updates that are streamed from multiple data sources with different traffic patterns as opposed to a single data source.

Finally, the problem of considering user preferences to manage the trade-off between QoS and QoD has been addressed in the context of web databases. The work in (Qu et al. 2006) provided an admission control framework, whereas the work in (Qu & Labrinidis 2007) introduced a two-level scheduling algorithm to address the problem.

9 Conclusions

Motivated by the need to support monitoring applications which involve the processing of update streams by continuous queries, in this paper we studied the different aspects that affect the QoD of these applications. In particular, we focused on the freshness of the output data stream and identified that both the properties of queries, i.e., cost and selectivity, and the properties of the input update streams, i.e., variability of updates, have a significant impact on freshness.

Our major contribution is a new approach to scheduling multiple queries in Data Stream Management Systems. Our approach exploits the properties of the continuous queries as well as the input data streams in order to maximize the freshness of output streams. We proposed a new scheduling policy called *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)* and a weighted version of it that supports applications with multi-class queries. We also introduced a generalized variant of FAS-MCQ that balances the trade-off between QoD and QoS, according to application requirements. We have experimentally evaluated our proposed FAS-MCQ policy against scheduling policies used in current DSMS prototypes as well as Web servers. Our experiments show that FAS-MCQ can reduce staleness by up to 55% compared to the other policies.

References

Babcock, B., Babu, S., Datar, M. & Motwani, R. (2003), Chain: Operator scheduling for memory minimization in data stream systems, *in* 'SIGMOD'.

Babcock, B., Datar, M. & Motwani:, R. (2004), Load shedding for aggregation queries over data streams, *in* 'ICDE'.

Bai, Y. & Zaniolo, C. (2008), Minimizing latency and memory in dsms: a unified approach to quasi-optimal scheduling, *in* 'SSPS'.

Brown, R. (1988), 'Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem', *Communications of the ACM* **31**(10), 1220–1227.

Carney, D., Cetintemel, U., Rasin, A., Zdonik, S., Cherniack, M. & Stonebraker, M. (2003), Operator scheduling in a data stream manager, *in* 'VLDB'.

Carney, D. et al. (2002), Monitoring streams: A new class of data management applications, *in* 'VLDB'.

Chandrasekaran, S. et al. (2003), TelegraphCQ: Continuous Dataflow Processing for an Uncertain World, *in* 'CIDR'.

Chen, J., DeWitt, D. J. & Naughton, J. F. (2002), Design and evaluation of alternative selection placement strategies in optimizing continuous queries, *in* 'ICDE'.

Chen, J., DeWitt, D. J., Tian, F. & .Wang, Y. (2000), NiagaraCQ: A scalable continuous query system for internet databases, *in* 'SIGMOD'.

Cho, J. & Garcia-Molina, H. (2000), Synchronizing a database to improve freshness, *in* 'SIGMOD'.

Cho, J. & Garcia-Molina, H. (2003), 'Effective page refresh policies for web crawlers', *ACM Transactions on Database Systems* **28**(4), 390–426.

Cranor, C., Johnson, T., Spataschek, O. & Shkapenyuk, V. (2003), Gigascope: A stream database for network applications, *in* 'SIGMOD'.

Hammad, M. et al. (2004), Nile: A query processing engine for data streams, *in* 'ICDE'.

Labrinidis, A. & Roussopoulos, N. (2001), Update propagation strategies for improving the quality of data on the web, *in* 'VLDB'.

Labrinidis, A. & Roussopoulos, N. (2004), 'Exploring the tradeoff between performance and data freshness in database-driven web servers', *VLDB J.* **13**(3), 240–255.

Lam, W. & Garcia-Molina, H. (2003), Multicasting a changing repository, *in* 'ICDE'.

Madden, S., Shah, M. A., Hellerstein, J. M. & Raman, V. (2002), Continuously adaptive continuous queries over streams, *in* 'SIGMOD'.

Motwani, R. et al. (2003), Query processing, resource management, and approximation in a data stream management system, *in* 'CIDR'.

Olston, C. & Widom, J. (2002), Best-effort cache synchronization with source cooperation, *in* 'SIGMOD'.

Qu, H. & Labrinidis, A. (2007), Preference-aware query and update scheduling in web-databases, *in* 'ICDE'.

Qu, H., Labrinidis, A. & Mossé, D. (2006), Unit: User-centric transaction management in web-database systems, *in* 'ICDE'.

Shanmugasundaram, J., Tufte, K., DeWitt, D. J., Naughton, J. F. & Maier, D. (2002), Architecting a network query engine for producing partial results, *in* 'WebDB'.

Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A. & Pruhs, K. (2006), Efficient scheduling of heterogeneous continuous queries, *in* 'VLDB'.

Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A. & Pruhs, K. (2008), 'Algorithms and metrics for processing multiple heterogeneous continuous queries', *ACM TODS* **33**(1).

Sharaf, M. A., Labrinidis, A., Chrysanthis, P. K. & Pruhs, K. (2005), Freshness-aware scheduling of continuous queries in the dynamic web, *in* 'WebDB'.

Sullivan, M. (1996), A stream database manager for network traffic analysis, *in* 'VLDB'.

Sutherland, T. M., Zhu, Y., Ding, L. & Rundensteiner, E. A. (2005), An adaptive multi-objective scheduling selection framework for continuous query processing, *in* 'IDEAS'.

Tatbul, N., Çetintemel, U. & Zdonik, S. B. (2007), Staying fit: Efficient load shedding techniques for distributed stream processing, *in* 'VLDB'.

Tatbul, N., Cetintemel, U., Zdonik, S. B., Cherniack, M. & Stonebraker, M. (2003), Load shedding in a data stream manager, *in* 'VLDB'.

Tatbul, N. & Zdonik, S. B. (2006), Window-aware load shedding for aggregation queries over data streams, *in* 'VLDB'.

Terry, D. B., Goldberg, D., Nichols, D. & Oki, B. M. (1992), Continuous queries over append-only databases, *in* 'SIGMOD'.

Tian, F. & DeWitt, D. J. (2003), Tuple routing strategies for distributed eddies, *in* 'VLDB'.

Urhan, T. & Franklin, M. J. (2001), Dynamic pipeline scheduling for improving interactive query performance, *in* 'VLDB'.

Yeganeh, N. K., Sadiq, S. W., Deng, K. & Zhou, X. (2009), Data quality aware queries in collaborative information systems, *in* 'APWeb/WAIM'.