Scheduling Multiple Continuous Queries to Improve QoD*

Mohamed A. Sharaf, Alexandros Labrinidis, Panos K. Chrysanthis, Kirk Pruhs
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{msharaf, labrinid, panos, kirk}@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 the new data stream management stystems (DSMSs), multi-query scheduling has been exploited to improve QoS. In this paper, we are proposing to exploit scheduling to improve QoD. 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 increase freshness by up to 50% compared to existing scheduling policies used in DSMSs.

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. 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.

*This is an extended version of our paper "Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web", which appears in the Proceedings of the 8th ACM WebDB Workshop (June 2005, held in conjuction with SIGMOD 2005).

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 (CQ)* on data streams [23].

Aurora [4], STREAM [18], TelegraphCQ [5], Tribeca [21], Gigascope [10], Niagara [7] and Nile [11] are examples of current prototype DSMSs. In such systems, each monitoring application registers a set of CQs, where a CQ is continuously executed with the arrival of new relevant data (Figure 1). In the Real-time Outbreak of Disease System (RODS) example, the health officials register queries for tracking specific indicators of disease outbreaks. As another example, a user might register a query to monitor news about *tsunamis*. Thus, as new articles arrive into the system, all the *Tsunami*-related ones have to be propagated to that user. As such, the arrival of new updates triggers the execution of a set of corresponding queries, since portions of the new updates may be relevant to different queries. The output of such a frequent execution of a continuous query is what we call an *output data stream* (see Figure 1). In this particular example, an output data stream can be used to continuously update a user's personalized Web page where a user logs on and monitors updates as they arrive. It can also be used to send email notifications to the user when new results are available.

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 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. In this paper, we are focusing on scheduling continuous queries for improving the QoD of output data streams.

QoD can be measured in different ways, one of which is *freshness*. Freshness, as well as scheduling policies for improving freshness, has been studied in the contexts of replicated databases [8, 9], derived views [13], and distributed caches [19]. 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 only Quality of Service metrics like response time and throughput (e.g., [7, 20, 3, 5, 1]).

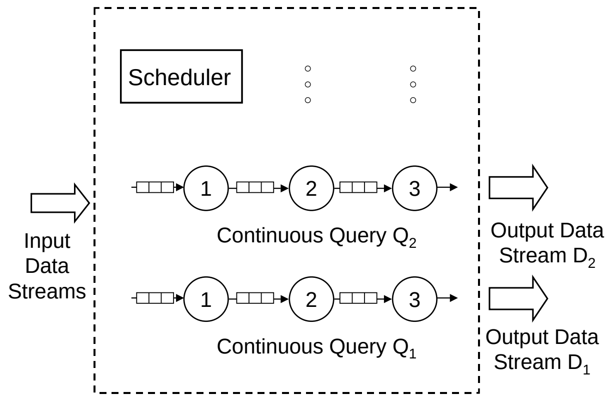

Figure 1: A DSMS hosting multiple continuous queries

Specifically, the contribution of this paper is proposing a policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. The proposed policy, FAS-MCQ, has the following salient features:

1. It exploits the variability of the processing costs of different continuous queries registered at the DSMS.
2. It utilizes the divergence in the arrival patterns and frequencies of updates streamed from different remote data sources.
3. It considers the impact of *selectivity* on the freshness of output data stream.

To illustrate the last point on the impact of selectivity, let us assume a continuous query which is used to project the number of trades on a certain stock, if its price exceeds $60. Further, assume that there is a 50% chance that this stock's price exceeds $60. With the arrival of a new update, if the new price is greater than $60 then a new update is added to the continuous output data stream. Otherwise, the update is discarded and nothing is added to the output data stream. So, in this particular example, the arrival of a new update renders the continuous output data stream stale with probability 50%. FAS-MCQ exploits such probability of staleness in order to maximize the overall QoD.

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.

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 increase freshness by up to 50% 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 is presented in Section 4. Section 5 describes our simulation testbed, whereas Section 6 discusses our experiments and results. Section 7 surveys related work. We conclude in Section 8.

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). Data streams consist 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 frequenly 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. Improving the QoD in the context of multi-stream queries as well as shared queries or operators is part of our future work.

A single-stream query plan can be conceptualized as a data flow diagram [4, 1] (Figure 1): 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. 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 of 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.

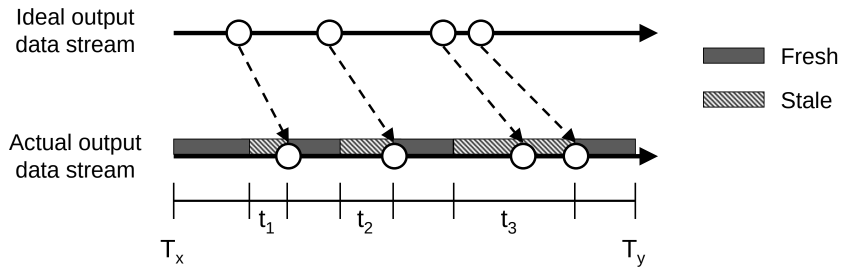

Figure 2: An example on measuring the freshness of a data stream

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

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

Recall that in traditional database systems, an operator with selectivity s_x 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}).

Specifically, for a query Q_i that is composed of a single stream of operators $< O_1, O_2, O_3, ..., O_r >$, C_i, S_i and C_i^{avg} are defined as follows [1]:

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

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

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

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 $C^{avg} = c_1 + c_2 \times s_1 + c_3 \times s_2 \times s_1$. This is generalized in the formula for C_i^{avg} above as in [24]. The maximum cost is a special case of the average cost when the selectivity of each operator in a single-stream query is equal to 1.

[1]In the rest of the paper, we use lower-case symbols to refer to operators' parameters and upper-case ones for queries' parameters.

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 [8, 13, 19, 9, 14]. 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'. If the tuple satisfies all the query's predicates, then $| D_{t'} | = | D | + 1$, otherwise, $| D_{t'} | = | D |$. In the former 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. In the latter case, 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*.

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 on 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: 1) the interval where the continuous query is waiting to be scheduled for execution; and 2) 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 illustared 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:

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

3.3 Fairness in Freshness

Ideally, all data streams in the system should experience perfect freshness. However, this is not always achievable. Especially when the DSMS is loaded, we can have data streams with freshness that is less than perfect, because of a "back-log" of updates that cannot be processed in time [13]. In such a case, it is desirable to maximize the average freshness in addition to minimizing the variance in freshness among different data streams. Minimizing the variance reflects the system's *fairness* in handling different continuous queries.

In this paper, we are measuring fairness as in [17]. Specifically, we compute the average freshness of each output data stream. Then, we measure fairness as *the standard deviation of freshness* measured for each data stream. A high value for the standard deviation indicates that some classes of data streams received unfair service compared to others. That is, they were stale for a longer intervals compared to other data streams. A low value for the standard deviation indicates that the difference in service (freshness) among different data streams is negligible, and, as such, the DSMS handled all streams in a fair manner.

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 on QoS metrics [3, 5, 1] and exploits *selectivity* to improve the provided QoS. Previous work on synchronizing database updates exploited the *amount (frequency)* of updates to improve the provided QoD [8, 19, 9]. In contrast, our proposal, *FAS-MCQ*, exploits both selectivity and amount of updates to improve the QoD, i.e., freshness, of output data streams.

4.1 Scheduling without Selectivity

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.

Next, we compare two policies X and Y. Under policy X, query Q_1 is executed before query Q_2, whereas 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:

$$L_X = L_{X,1} + L_{X,2} \tag{4}$$

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, then $L_{X,1}$ is computed as follows:

$$L_{X,1} = W_1 + (N_1 C_1)$$

where W_1 is the current loss in freshness and $(N_1 \times C_1)$ is the time required until applying all the pending updates.

Similarly, $L_{X,2}$ is computed as follows:

$$L_{X,2} = (W_2 + N_1 C_1) + (N_2 C_2)$$

where W_2 is the current loss in freshness plus the extra amount of time $(N_1 \times C_1)$ where Q_2 will be waiting for Q_1 to finish execution.

By substitution in Equation 4, we get

$$L_X = W_1 + (N_1 C_1) + (W_2 + N_1 C_1) + (N_2 C_2) \tag{5}$$

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

$$L_Y = (W_1 + N_2 C_2) + (N_1 C_1) + W_2 + (N_2 C_2) \tag{6}$$

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

$$N_1 C_1 < N_2 C_2 \tag{7}$$

The left-hand side of Inequality 7 shows the total loss in freshness incurred by Q_2 when Q_1 is executed first. Similarly, the right-hand side shows the total loss in freshness 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 $N_i C_i$ 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 $N_i C_i$ value since it will have the minimum negative impact on the other queries in the system.

4.2 Scheduling with Selectivity

Assume the same setting as in the previous section. However, assume that the productivity of each query Q_i is S_i which is computed as in Section 2. The objective when scheduling with selectivity is the same as before: we want to minimize the total loss in freshness. 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 the other query. In the presence of selectivity, we will apply the same concept.

However, 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 Q_i's predicates. Hence, $P_i = 1 - (1 - S_i)^{N_i}$, where $(1 - S_i)^{N_i}$ is the probability that all pending updates do not satisfy Q_i's predicates.

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

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

where $N_2 \times C_2^{avg}$ 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 $N_2 \times C_2^{avg}$ 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:

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

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 reciprocal 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 Scheduler

The FAS-MCQ scheduler selects for execution the query with the highest priority value at each *scheduling point*. A scheduling point is reached when: (1) a query finishes processing an input update, or (2) when a new update arrives at the system.

In the second case, the scheduler has to decide whether to resume executing the current query or preempt it. A query is preempted if a new update has arrived at a query with priority higher than the one currently executing. Thus, we need to recompute the priority of the currently executing query based on the position of the processed update along the query operators. For example, if the processed update is at the input queue of some operator O_x along the query, then the current priority of the query is computed as:

$$\frac{1 - (1 - S_x)}{C_x^{avg}}$$

where S_x and C_x^{avg} are the expected productivity and expected cost of the segment of operators starting at O_x all the way to the root. If O_x has been processing the tuple for δ_x time units, then the current priority is computed as above by replacing c_x with $c_x - \delta_x$.

4.4 Discussion

It should be noted that under our policy, the priority of a query increases as the processing of an update advances. For instance, let us assume that a query has just been selected for execution. At that moment, the priority of the query is equal to the priority of its leaf node or leaf operator. After the leaf finishes processing the update, the priority of the next operator, say O_x, is computed as shown earlier. Intuitively, S_x and C_x^{avg} are greater than S and C^{avg} of the leaf operator because the remaining processing cost decreases and the expected productivity might increase too. Additionally, N_x is equal to one and our priority function monotonically increases with the decrease in N. Thus, overall, the priority of O_x is higher than that of the leaf node. Similarly, the priority of each operator in the query is higher than the priority of the operator preceding it. As such, a query Q_i is never preempted unless a new update arrives and that new update triggers the execution of a query with a higher priority than Q_i.

Also note that under our priority function (Equation 11), *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.

4.5 Weighted Freshness

In many monitoring applications, some queries are more important than others. That is especially obvious in emergency systems where some continuous queries are more critical than others. For example, a continuous query monitoring the eruption of a volcano is more crtitical than a continuous query monitorig traffic congestions. As such, when the system is loaded, it is necessary to maximize the freshness of these critical queries.

It is fairly easy to modify our proposed FAS-MCQ policy to increase the frehness of data streams which have different 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:

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

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

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

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

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

And the priority assigned to each query is computed as:

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

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).

5. 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*. In addition, we considered a FCFS policy where updates are processed according to their arrival times. Finally, we adapted the Shortest Remaining Processing Time (*SRPT*) policy, where the priority of a query is the reciprocal of its maximum cost. The SRPT policy has been shown to work very well for scheduling requests at a Web server when the performance metric is response time [12].

Queries: We simulated a DSMS that hosts 250 registered continuous queries. The structure of the query is adapted from [6, 16] where each query consists of three operators: two predicates and one projection. 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: 2^i time units, where i is 0, 1, or 2.

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 uniformly selected from the range [0.1, 1.0].

Streams: The number of input data streams is set to 10 and the length of each stream is set to 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 exact time required for executing the queries in the system, whereas for lower utilizations, the mean inter-arrival time is increased proportionally. To generate a back-log of updates [13], we have a parameter B which controls the number of *bursty* streams. A bursty stream is created by adapting the initially generated Poisson stream using two parameters: *burst probability (p)* and *burst length (l)*. Specifically, we traverse the Poisson stream and at each entry/update we toss a coin, if the tossing result is less than the p, then the arrival time A_b of that update is the beginning of a new burst. Then, the arrival times of each of the next l updates are adjusted so that the new arrival time, A'_i, of an update u_i is set to $(A_i - A_b) * p$, where A_i is the arrival time computed originally under the Poisson distribution. We have conducted several experiments with different settings of the p, l and B parameters. In this paper, we will present the simulation results where p is equal to 0.5, l is equal to 50 updates and B is in the range [0, 10] with the default being 5. Table 1 summarizes our simulation parameters and settings.

Parameter	Value
Policies	FAS-MCQ, RR, FCFS, SRPT
Number of Queries	250
Number of Operators per Query	3
Operators' Costs	1, 2, 4
Operators' Selectivities	0.1–1.0
Utilization	0.1–0.95
Number of Data Streams	10
Stream Length	10K
Number of Bursty Streams	0–10

Table 1: Simulation Parameters

6. EXPERIMENTS

In this section we are going to present our experimental evaluation results on the impact of utilization, impact of burstiness in data streams, impact of selectivities and on fairness. In this paper, we are only reporting the results of our experiments with the *FAS-MCQ* policy (i.e., without weights).

6.1 Impact of Utilization

In this experiment, the selectivity for all operators is set to 1, whereas the processing costs are variable and are generated as described earlier. Figure 3 depicts the average total freshness over all output data streams as the load at the DSMS increases. In this experiment 5 out of the 10 input data streams are bursty.

The figure shows that, in general, the freshness of the output data streams decreases with increasing load. It also shows that the FAS-MCQ policy provides the highest freshness all the time. The freshness provided by SRPT is equal to that of FAS-MCQ for utilizations up to 0.5. After that point, with increasing utilization, queues start building up. That is when FAS-MCQ gives higher priority to queries with shorter queues and low processing cost in order to maximize the overall freshness of data, thus outperforming SRPT. At 95% utilization, FAS-MCQ has 22% higher freshness than SRPT. If we report QoD as staleness (i.e., the opposite of freshness [19]), then FAS-MCQ is 41% better than SRPT, with just a 20% overall average staleness.

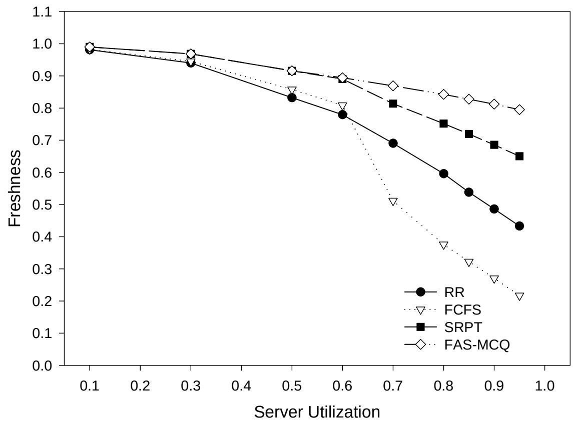

Figure 3: freshness vs. load (selectivity=1.0)

6.2 Impact of Bursts

The setting for this experiment is the same as the previous one. However, the utilization at all points is set to the default value of 90%. In Figure 4, we plot the average total freshness as the number of input data streams that are bursty increases. At a value of 0, all the arrivals follow a Poisson distribution with no bursts, whereas at 10, all input data streams are bursty as described in Section 5.

Figure 4 shows how the total average freshness decreases when the number of bursty data streams increases. It also shows that FAS-MCQ provides the highest freshness compared to the other policies. Notice the relation between FAS-MCQ and SRPT: as the number of bursty streams increases, the difference in freshness provided by FAS-MCQ compared to SRPT increases up until there are 5 bursty streams. At that point, FAS-MCQ has 20% higher freshness than SRPT. At the same time, FAS-MCQ has 1.8 the freshness of the RR policy and 3.6 the freshness of the FCFS policy.

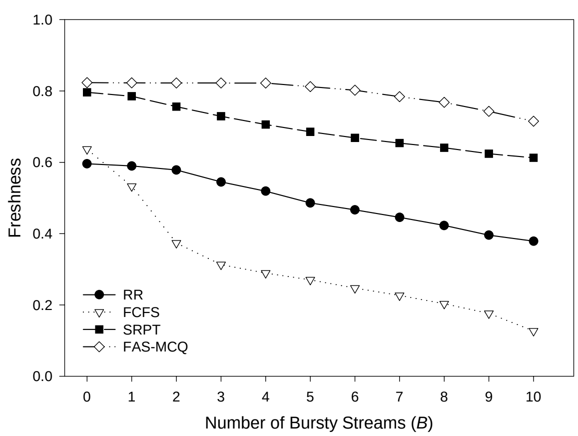

Figure 4: freshness vs. number of bursty streams

After there are 7 bursty input streams, the performance of the FAS-MCQ and SRPT policies get closer. The explanation is that at a lower number of bursty streams, 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, SPRT is performing reasonably well. At 10 bursty streams, FAS-MCQ has only 16% higher freshness than SRPT.

6.3 Impact of Selectivity

In this experiment, the cost for all operators is set to 1 time unit. However, the selectivity is chosen uniformly from the range [0.0, 1.0]. Figure 5 depicts how the freshness decreases with increasing load at the Web server. The figure also shows that FAS-MCQ still provides the highest freshness, as it considers the probability that an update will affect the freshness of the corresponding data stream. That is different from SRPT which will give a higher priority to a query with low cost. Hence, SRPT might spend time executing queries that will only append fewer updates to their corresponding output data streams.

In this setting, RR behaves better than SRPT at high utilizations. At a 95% utilization, FAS-MCQ gives 50% higher freshness than RR and 63% higher than SRPT.

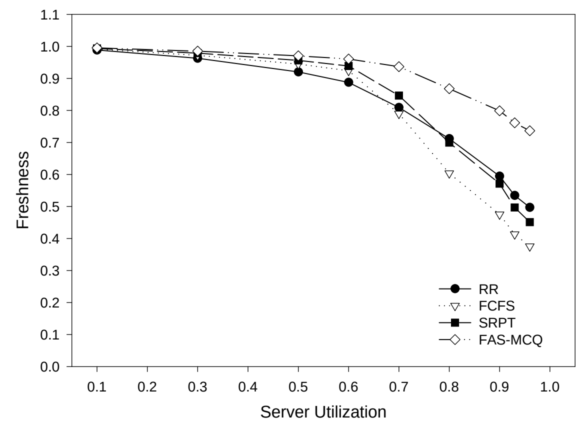

Figure 5: freshness vs. load (variable selectivity)

6.4 Fairness

Figure 6 shows the standard deviation of freshness for the same experiment setting. The figure shows that for all policies, the deviation increases with increasing load where some output data streams are stale for longer times compared to other data streams. However, FAS-MCQ provides the lowest standard deviation for most values of utilization. As the utilization approaches 1 (i.e., when the DSMS is about to reach its capacity), the fairness provided by FAS-MCQ gets closer to that of FCFS. Thus, FAS-MCQ is at least as fair as FCFS, even at very high utilizations.

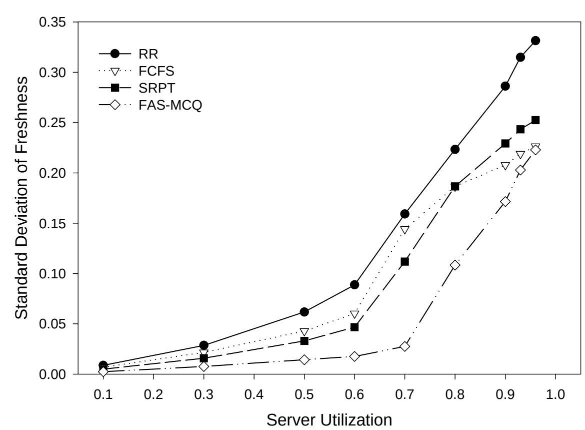

Figure 6: standard deviation of freshness

However, the FCFS policy behaves poorly if we look beyond fairness and into the average total freshness: as shown in Figure 5, FAS-MCQ provides 96% higher average freshness compared to FCFS, despite having the same fairness.

7. 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 [7] to improve the system throughput in an Internet environment and in [16] 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 [3]. The work in [1] employs a scheduler for minimizing the memory utilization.

To the best of our knowledge, no previous work has proposed multi-query scheduling policies for improving the QoD provided by continuous queries. However, *load shedding* has been devised as a technique to control the degree of degradation in the provided QoD under overloaded conditions. The work in [22] describes a load shedding technique that decides which tuples to drop according to the importance of their content. The work in [2] formalizes the load shedding problem for aggregate queries.

Scheduling policies for improving the QoD has been studied in the context of replicated databases and in Web databases. For example, the work in [8, 9] provides policies for crawling the Web in order to refresh a local database. The authors make the observation that a data item that is updated more often should be synchronized less often. In this paper, we utilize the same observation, however, [8, 9] assumes that updates follow a mathematical model, whereas we make our decision based on the current status of the Web server queues (i.e., the number of pending updates). The same observation has been exploited in [19] for refreshing distributed caches and in [15] for multi-casting updates.

The work in [13] 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 [13] by considering updates that are streamed from multiple data sources with different traffic patterns as opposed to a single data source.

8. CONCLUSIONS

Motivated by the need to support monitoring applications, as well as active Web services, both of 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, as well as 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 both properties of queries and input data streams in order to maximize the freshness of output data 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 in which queries have different priorities. 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 increase freshness by up to 50% compared to the existing policies.

Our next step is to study the problem of maximizing the freshness when MCQ plans include shared operators as well as join operators. Additionally, we are planning to investigate the relation between QoD and QoS provided by DSMSs.

Acknowledgments: The authors would like to thank the reviewers for both WebDB 2005 and HDMS 2005 for their helpful comments and suggestions. This work is supported in part by NSF ITR Medium Award (ANI-0325353) and by the Pennsylvania Department of Health Award (ME-01-737). The first author is also supported in part by the Andrew Mellon Predoctoral Fellowship.

9. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[2] B. Babcock, M. Datar, and R. Motwani:. Load shedding for aggregation queries over data streams. In *ICDE*, 2004.

[3] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[4] D. Carney, U. Getintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, V. R. S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. .Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[8] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, 2000.

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

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

[11] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[12] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agarwal. Size based scheduling to improve web performance. *Transactions on Computer Systems*, 21(2):207–233, 2003.

[13] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB*, 2001.

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

[15] W. Lam and H. Garcia-Molina. Multicasting a changing repository. In *ICDE*, 2003.

[16] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[17] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *VLDB*, 1993.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[19] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.

[20] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB*, 2002.

[21] M. Sullivan. A stream database manager for network traffic analysis. In *VLDB*, 1996.

[22] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[23] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD*, 1992.

[24] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, 2001.