WPI-CS-TR-04-16 April 2004

Adaptive QoS-Driven Scheduling Framework for a Continuous Query System

by

Timothy M. Sutherland
Bradford Pielech and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Adaptive QoS-Driven Scheduling Framework for a Continuous Query System

Timothy M. Sutherland, Bradford Pielech and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{tims, winners, rundenst}@cs.wpi.edu

Abstract

In a continuous query environment, different applications may have distinct Quality of Service (QoS) requirements. Given the unpredictability of streaming data, utilizing a single scheduling algorithm, as done by current state-of-the-art stream query engines, is no longer sufficient. Current scheduling algorithms used in these systems are typically one-dimensional, limiting the ability to perform well under changing system conditions.

We propose a novel algorithm selection framework used in our CAPE system. This framework leverages the strengths of current scheduling algorithms to meet sets of QoS requirements. In CAPE, each algorithm can be compared in terms of its past ability to improve the QoS, knowing nothing about the characteristics of the algorithm. This knowledge can be used to determine the algorithm that probabilistically has the best chance of improving the QoS. Our framework has the flexibility to add new algorithms, query plans and data sets during runtime, with no need to fine-tune the algorithm to the system. Using standard data sets and query plans from existing literature on scheduling, our experiments show in fact that, this new framework combines the relative strengths of these algorithms while adhering to given QoS requirements.

Keywords: Operator Scheduling, Scoring Function, Adaptive Algorithm Selection, CAPE.

1 Introduction

Many modern applications process queries over unbounded streams of data. These applications include monitoring remote sensors [11], online transaction processing [18], and intrusion detection in networks [17]. An effort is being undertaken by the database community to derive a new class of query systems, called Continuous Query Systems. These systems execute queries on data that is continuously arriving, possibly time-varying and high volume, and returns the result of the query to the user in a streaming real-time fashion.

Unlike a traditional database system, where all data is first stored in persistent storage before being queried, continuous query systems must process the data as it flows by. In a traditional DBMS data access is under strict control, however streaming data comes from sources around the world at varying arrival times. Furthermore, the streams can be extremely bursty with high volumes of data arriving all at once or conversely with no data arriving for quite some time. Also, a continuous query is typically registered with the system, and then is continually processed against the streaming data.

1.1 Motivation

Applications may have different Quality of Service (QoS) requirements because of physical limitations such as network speed or limitations on memory and processing resources. For example, a security system typically registers queries that will recognize intrusion, with high priority to report results in real time. Network traffic monitors may want to sort and group potentially large sets of data before the results are given to the user, but are particularly concerned with keeping memory utilization to a minimum. These varying QoS requirements play a significant role in determining how a query will be processed.

Multiple Dimensions of Variability. In addition to QoS requirements, Continuous Query Systems have many uncontrollable constantly changing external factors to tend with during execution. First, the number of queries may change as queries are registered or removed from the system at any time [4]. Also a query may be optimized during execution, causing a change in the query plan [21]. The stream characteristics may also change significantly in terms of arrival rates and also value distributions. As a consequence, at times a join operator may have a very high selectivity, when other times the data entering the join operator may not join at all with data from the second stream. In order for a Continuous Query system to perform well in environments with such diverse and time-varying characteristics, these uncontrollable factors must be addressed.

1.2 Adapting to Changing Characteristics

Current continuous query systems typically have some measure of adaptability built-in to cope with such unexpected external changes. Aurora [5] allows an administrator to input QoS specifications and the system monitors execution performance based on these QoS metrics. If the QoS drops below an acceptable level, the system will shed load until the performance increases. This may not be acceptable for many applications, especially when quality of the data contents are critical. Consider a government application that scours internet data to try to uncover terrorist activity. If one piece of information is missing it could lead to making an incorrect decision, or worse yet no decision at all.

Another method to cope with these external factors is to use a scheduling algorithm to decide the order in which query operators process data is typically used. Rather than randomly selecting operators to process data, a scheduling algorithm systematically selects operators that can improve performance in an area execution, such as increasing output rate or reducing memory costs. Scheduling algorithms offer Continuous Query Systems fine-grained control over the operators during execution. Current systems either employ traditional scheduling algorithms borrowed from the realm of operating systems [20][8], such as Round-Robin and FIFO or use customized algorithms designed specifically for continuous query systems. These new algorithms include Chain [2], introduced by the STREAM system, and Aurora's Train [6] scheduling.

Chain works well at minimizing memory, but this comes at the expense of decreased throughput and possible operator starvation. Train scheduling is excellent at batching tuples together to be processed in sequence to lower fixed operator costs. However, if query operators do not have a large batch of tuples waiting to be processed, but rather small queues spread throughout the query plan, Train scheduling is not as effective as other algorithms. Other types of algorithms such as Round Robin do not consider the cost of executing an operator and thus may under-utilize inexpensive operators.

We experimentally compared these existing algorithms under a variety of stream workloads in a real Continuous Query System. Our experiments revealed that although these algorithms are good at improving system performance, each typically optimizes for only one parameter in the system, such as reducing memory (Chain) or increasing the query plan output rate (FIFO). Thus, even though there are many scheduling algorithms that work well for a particular query environment, there is no one algorithm that a system can utilize to satisfy all such requirements, especially given the wide variability that a continuous query system encounters.

Our experimental study has confirmed that it is very difficult to design a "one size fits all" scheduling algorithm due to the fact that future arrival rate of incoming streams may be unknown, along with the data contained within the stream. Also, other factors such as changing QoS requirements and the addition or optimization of query plans add yet another level of variability into the Continuous Query System.

1.3 Our Approach

Thus, we propose an **adaptive scheduling framework** that has the ability to select a single algorithm from a pool of algorithms available to the system. By providing several algorithms that optimize different QoS requirements, we aim to heuristically pick an algorithm that has a high probability of performing well during a particular point of query execution, and still adhere to strict (and possibly changing) QoS requirements in the system.

1.4 Contributions

This paper contributes to Continuous Query Systems, particularly query processing, in the following ways:

- Experimentally studied the performance of a wide variety of state-of-the-art scheduling algorithms in our CAPE processing engine to determine the advantages and disadvantages of algorithms under varying QoS requirements, data stream arrival rates, and query plans. The experiments confirm that each algorithm has unique properties that make it more ideal in some circumstances and less ideal in others.
- Designed an adaptive framework that has the ability to learn the behavior of the continuous query system **without significant processing overhead, knowledge of the scheduling algorithm, query plan or data set**. This knowledge is then used to guide the selection of algorithms that probabilistically have the best chance to fulfill a given set of QoS requirements.
- Built CAPE, a Continually Adaptive Processing Engine for streaming data from the ground up, embedding all of these existing scheduling algorithms as well as and our proposed framework.
- Performed an experimental study that supports our claim that we can in fact leverage the strengths of several existing scheduling algorithms to improve the overall performance of a continuous query system given a set of QoS requirements.

1.5 Outline

The remainder of this paper is structured as follows. Section 2 illustrates using an intuitive example that different scheduling algorithms have particular strengths and weaknesses. Section 3 describes design choices for our CAPE system augmented by our proposed adaptive framework. Section 4 experimentally compares the scheduling

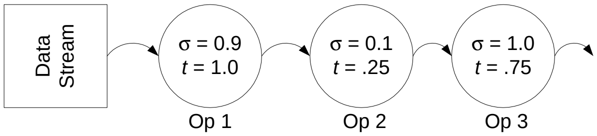


Figure 1. Selectivity σ and Average Tuples Processing Time t for the example query plan.

algorithms used in our work to find the pros and cons of each such algorithm. Section 5 describes key aspects of our adaptive technique, including quality of service requirements, algorithm scoring functions, and algorithm selection heuristics. Section 6 presents an experimental study. Section 7 reviews the related research. Conclusions and future work are discussed in Section 8.

2 Motivating Example

We will illustrate in the example below and then confirm via experimental studies using the system described in Section 3, that scheduling algorithms typically favor a single QoS requirement.

Let us now consider the query plan in Figure 1. The plan contains three filter operators Op_1 through Op_3. We will assume that the input stream will place 1 tuple in the input buffer of Op_1 every time unit, starting at time t_o. When we refer to a tuple, we are really referring to a group of tuples that are organized in some logical batch, such as a disk page or a memory block. For simplicity let us assume that switching between operators takes zero time. When told to run, an operator will consume one tuple from its input queue if available, process the data for a fixed amount of time, t, and then output a percentage of the tuples corresponding to its selectivity, σ. For instance, if an operator consumes 0.5 tuples from its input queue and outputs 0.45 tuples, its σ is 0.9. Figure 1 shows σ and t values for the operators in our example plan.

Now consider two different scheduling algorithms. Algorithm 1 is a FIFO scheduler that will take tuples from the input queue of Op_1 and process them until completion, before processing the next tuples from Op_1's input queue. Algorithm 2 is called Most Tuple In Queue (MTIQ) that runs the operator with the most tuples in its input queue. The difference in scheduling policies becomes apparent when looking at resource allocation, output rate, operator utilization, and freshness of results, as illustrated below.

Table 2 summarizes the number of tuples in all queues and the throughput (number of tuples Op_3 outputs) for each algorithm as execution progresses. The execution would happen as follows: First 1 tuple is removed from the input queue of Op_1 and after processing for 1 time unit, 0.9 tuples are outputted (0.9= 1 x $\sigma(Op_1)$). Then 0.9 tuples are processed by Op_2 and 0.09 are outputted (0.09 = 0.9 x $\sigma(Op_2)$). Finally, 0.09 are consumed by Op_3 and 0.09 are outputted to the end user because $\sigma(Op_3)$ = 1.

While FIFO is propagating tuples through the query plan, more tuples enter the input queue of Op_1. Thus the queue sizes grow. Since it takes 2 time units (1 + 0.25 + 0.75) to operate on a tuple, it would mean that in the time that it takes to process 1 tuple to completion, 2 more tuples would have entered the system. As we can see, FIFO is exceptional at guaranteeing throughput. However it comes at the expense of increased memory consumption.

MTIQ behaves the same as FIFO during t_0, but differs starting with t_1. At t_1, there is 1 tuple queued for Op_1 and 0.9 tuples queued for Op_2. MTIQ chooses to run Op_1 again. At t_2, there is 1 tuple queued for Op_1 and 1.8 tuples queued for Op_2 (0.9 + 0.9). MTIQ then runs Op_2. Op_2 finishes at time $t_{2.25}$ (because it started at t_2 and processed for 0.25 time units) and now the queue sizes are 1, 0.8, 0.1 for Op_1, Op_2, Op_3 respectively. MTIQ runs Op_1 again. At t_3, there is one new tuple for Op_1 and still 0.8 and 0.1 tuples at Op_2 and Op_3. The process continues until time t_{14}, when Op_3 finally has the largest input queue, and is subsequently processed.

This example shows that each algorithm performs differently for different QoS requirements. The MTIQ algorithm keeps its queue sizes smaller than those of FIFO, but it does not output any results for a (relatively) long

Table 1. Total Queue Sizes and Throughput for Example Query Plan from Fig. 1.

Time	FIFO Queue Size	MTIQ Queue Size	FIFO Throughput	MTIQ Throughput
0	1.0	1.0	0.0	0
1	1.9	1.9	0.0	0
2	2.0	2.8	0.09	0
3	2.9	1.9	0.09	0
4	3.0	1.9	0.18	0
5	3.9	1.9	0.18	0

time. MTIQ's throughput is much more bursty than FIFO's. MTIQ will take approximately 14 time units to output its first tuple. The next output will come slightly more quickly, but the output pattern will not be as consistent as FIFO that outputs every 2 time units.

3 CAPE Overview

In this section, we will first present CAPE, our **C**ontinually **A**daptive **P**rocessing **E**ngine. CAPE is similar in system architecture to other systems such as STREAM, Aurora and NiagaraCQ. The main difference is the way that our system uses an adaptive scheduler for determining execution, while other current systems employ a single scheduling algorithm for the duration of execution.

3.1 Architecture

CAPE is composed of five primary components as depicted in Figure 2. The *Stream Receiver* is responsible for receiving the streaming data from *Stream Sources* across the Internet and submitting the data to the *Storage Manager*. The Storage Manager manages the tuple data, deciding if tuples should be stored in memory or persistent storage. It attempts to keep as many tuples in main memory as possible to improve the performance of query plan execution.

The *Statistics Gatherer* stores, calculates, and sorts statistics about any part of a query plan, in particular operators and queues. We use these statistics for many types of calculations, such as deciding how well a particular query plan is running given a cost model, or even simply how many tuples are in main memory at a given time.

The *Execution Engine* is responsible for overseeing the execution of the query plan. The Execution Engine tells the Statistics Gatherer to obtain the latest statistics, asks the *Operator Scheduler* which operator should process data next, and then runs the operator in the order decided by the Operator Scheduler.

3.2 Adaptive Framework Extension

We take a closer look at the architecture of the Operator Scheduler in Figure 3. In most typical continuous query systems [4][7][5], the operator scheduler employs one static scheduling algorithm such as Round Robin or Chain. This scheduler's task is to report to the Execution Engine the operator that will process data next based on that algorithm. Instead, our scheduling component is extended into an adaptive framework. It is equipped with a library of several possible scheduling algorithms and will periodically select one scheduling algorithm from that library that has statistically performed best given the set of QoS requirements.

In order to be able to schedule each operator in a specific order, it is essential that all of the query operators in the system are in the same thread. This will guarantee that the system has fine-grained control over the scheduling

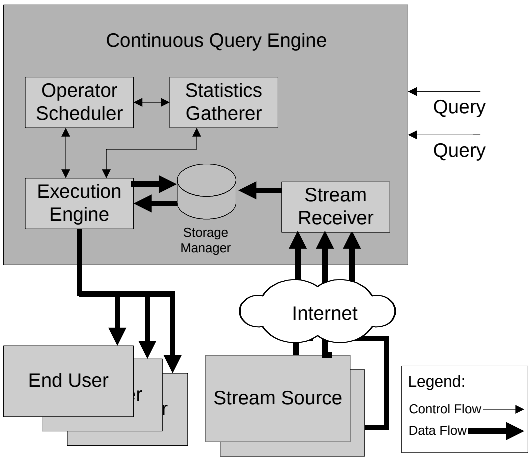


Figure 2. Architecture of Continuous Query System.

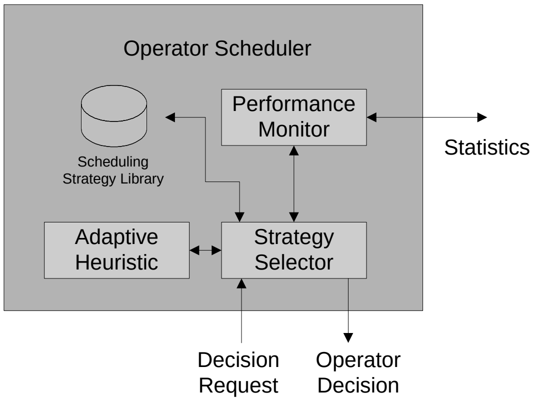


Figure 3. Architecture of Operator Scheduler.

of the operator, rather than the underlying operating system. Research has also shown that as query plans grow larger, current operating systems cannot handle the large number of threads that would need to be scheduled [6].

We keep a *library of scheduling algorithms* available for use to the system at any time. As new scheduling algorithms are developed, they can easily be plugged into the library to be used by the adaptive framework. Our *performance monitor* keeps a score of how well each scheduling algorithm has done thus far during execution. This monitor utilizes statistics already collected in the system for other purposes, thus keeping overhead for the framework to a minimum. Section 5.2 discusses how the algorithms are scored. The *Strategy Selector* then selects which scheduling algorithm to use based on some selection protocol. In CAPE, we use a very lightweight *Adaptive Heuristic* called the Roulette-wheel heuristic [12]. This heuristic takes the calculated scores for each algorithm into account to select the next best candidate. The operator scheduler then simply asks the active scheduling algorithm to pick the next operator to process data, and this decision is then reported to the Execution Engine.

Since statistics are stored about the query plans and operators on a consistent basis for query optimization purposes, we find virtually zero overhead in computing scores for the scheduling algorithms. This is confirmed in our experimental study described in Section 6. By using this architecture, we always select the scheduling algorithm that has statistically performed the best. If the performance degrades, we simply pick another algorithm to aim to boost performance.

4 Operator Scheduling

In order to develop a framework that could leverage the strengths of scheduling algorithms, current algorithms had to be studied. It is important to understand in what query environments a particular scheduling algorithm excels. In this section we will describe the set of most common scheduling algorithms used in Continuous Query Systems and then discuss an experimental study on these scheduling algorithms in an actual continuous query system to confirm that there is indeed no ”one size fits all” algorithm.

4.1 Scheduling Algorithms

Most scheduling algorithms seem to be one-dimensional in terms of meeting some QoS metric. The adaptive technique proposed in this paper focuses on selecting a particular scheduling algorithm when its advantages can be exploited to improve the QoS.

Now we describe several scheduling algorithms employed by our adaptive scheduling framework, and explain their advantages and disadvantages.

Round Robin (RR) is perhaps the most basic scheduling algorithm, and is used as the default scheduler by many Continuous Query systems such as [4]. It works by placing all runnable operators in a circular queue and allocating a fixed time slice to each. Round Robin's most desirable quality is the avoidance of starvation. An operator is guaranteed to be scheduled within a fixed period of time. However, Round Robin does not adapt to changing stream conditions.

FIFO operates on the oldest tuples first to push them through the query plan. FIFO generates a high throughput, because the oldest tuples are given a higher priority over newer ones. But it has the same drawbacks as Round Robin - no adaptability and no consideration of operator properties.

Most Tuples in Queue (MTIQ) scheduler is a greedy algorithm that assigns a priority to each operator equivalent to the number of the tuples in its input queues. MTIQ is a simplified batch scheduler, similar to Train [7]. Batch schedulers work under the assumption that the average tuple processing cost can be reduced if an operator can work on more tuples at a time. Operators typically have a start-up cost associated with their execution and the batch scheduler can amortize this cost over a larger group of tuples.

The most obvious advantage is that MTIQ works well at minimizing memory consumption. By running the operator with the most tuples enqueued, the algorithm will guarantee that no queue will grow unbounded, a claim that cannot be made by any of the previous algorithms.

Chain [2] is a recently proposed variation of Greedy Scheduling. Conceptually, each operator is assigned a priority that is based on selectivity, tuple processing cost, and the priorities of neighboring operators. By analyzing the priorities of neighboring operators, “Chains” of operators can be scheduled to run together. This method will remove the largest number of tuples from the system in the shortest amount of time.

Chain was shown, using experimental studies, to be an ideal algorithm for keeping queue sizes to a minimum. It may however suffer from starvation and poor response time during times of burst [2].

4.2 Evaluation of Scheduling

Figure 4 shows the performance of several algorithms including Round Robin, Most Tuples In Queue, Chain and First In First Out in our CAPE system. Here we monitor two different quality of service requirements, the number of tuples in memory, and the average tuple delay. For this study, we used a two-stream query plan, depicted in Figure 5. Further details about the experimental testbed are found in Section 6.

As anticipated, Chain and MTIQ performed best when it comes to minimizing memory use. As discussed in Section 4.1, Chain processes operators that remove the largest number of tuples the most quickly. MTIQ processes

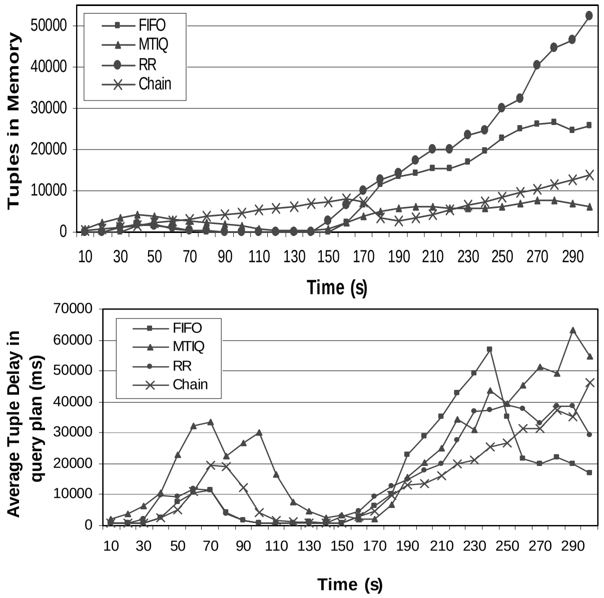


Figure 4. Performance of scheduling algorithms with a two-stream query plan.

operators that have the largest queue in the query plan. Thus it is no surprise that these two algorithms are excellent at reducing memory usage.

However we see very different results when observing how well the algorithms perform when it comes to the average tuple delay. MTIQ and Chain end up being the two worst performers by the end of execution. FIFO, which was only mediocre when considering the memory requirement, actually does quite well keeping the average tuple delay to a minimum. Overall we observe from Figure 4 that no one algorithm has a clear advantage. MTIQ and Chain compete for the best results in memory consumption, while RR, Chain and FIFO compete for the best results for average tuple delay. The key idea that can be deduced from this is that we should be using the best algorithm available at a given time to aim to optimize the QoS requirements. This is a key principle exploited by our solution outlined below.

5 Adaptive Scheduling Framework Details

The adaptive scheduler selector will periodically evaluate the current scheduler's performance for the specified QoS requirements and compare this with the other schedulers' performance. This qualitative comparison is based upon assigning a fitness score [12] to each scheduler that captures how well it performed in several metrics including throughput, memory size, output rate, et al. relative to the other algorithms.

We assume that we have no a priori knowledge about the relative strengths and weaknesses of the set of scheduling algorithms made available to us. Instead, our system aims to empirically learn the behavior of the algorithms over time.

We note that each scheduling algorithm's performance can fluctuate wildly as the data arrival characteristics change as seen in Figure 4. For example, one algorithm may perform very well in a stable system, but break down precipitously during periods of bursty arrival.

There were several challenges associated with creating an adaptive scheduling framework that solves the aforementioned problems. First, a function needs to be developed such that it can quantify how well a algorithm is performing for a particular QoS metric. The scoring function needs to allow the individual QoS metrics to be weighed for relative importance and normalize the collected statistics for those metrics such that one algorithm can be ranked against another.

Second, the adaptive scheduling framework needs to be able to intelligently choose the next scheduling algorithm. It must be able to weigh the benefits of choosing an alternate algorithm vs. staying with the current one. Because of this the Adaptive Heuristic needs to be carefully chosen such that it favors the well-performing (relative to QoS requirements) algorithms, but still allows the other algorithms to be periodically explored.

5.1 Quality of Service Requirements

Our system allows for the system administrator to specify the desired execution behavior as a composition of several possible goals. A QoS requirement consists of three components: the statistic, quantifier, and weight. The statistic corresponds to the requirement that is to be controlled. The quantifier, either maximize or minimize, specifies what the administrator wants to do with this preference. The weight is the relative importance of each requirement, with the sum of all weights equal to 1. We combine all of the QoS requirements into a single set called a QoS specification. This specification is made so we can generalize how we want the system to perform overall. Table 2 shows an example QoS specification. Here, the administrator has specified that the system should give highest priority to minimizing the queue size and next highest to maximizing the throughput.

Table 2. An example QoS specification

Statistic	Quantifier	Weight
Input Queue Size	minimize	0.75
Output Rate	maximize	0.25

QoS requirements are a key concept in our system. They guide the adaptive execution by encoding a goal that the system should pursue. Without these preferences, the system will not have any benchmark to determine how well or poorly a scheduler is performing. It is important to note that the requirements specify the desired behavior in relative terms. That is, the QoS is not specified for an absolute performance goal (i.e., achieve an output rate of X tuples per sec or have no more than Y tuples in the query plan at once), but rather specifies that the system should aim to maximize the output rate or minimize queue size. Absolute requirements are too dependent on data arrival patterns and thus may not be achievable.

5.2 Scoring the Statistics

During execution, the Execution Controller will update the statistics that are related to the QoS requirements. Once these statistics have been updated, the system needs to decide how well the previous scheduler, S_{old}, has performed, and compare this performance to the other scheduling algorithms. Thereafter a decision is made to determine how to continue execution. To accomplish this, the system calculates the mean (μ_H) and the spread of the values ($max_H - min_H$) of each of the statistics specified in the service preferences for the historical category, H. Next, using the statistics from S_{old} the mean μ_S of each of the statistics is calculated. Finally, each μ_S is normalized according to the formula in Equation 1. This normalizes each value in the $[-0.5, 0.5]$ range.

$$z_i = \frac{(\mu_S - \mu_H)}{max_H - min_H} decay^{time} + 0.5 \tag{1}$$

A $decay$ parameter is used to exponentially decay old and out-of-date data to give a higher priority to those algorithms which were run most recently. This data is the most relevant to the current state of the system. The decay is calculated by raising the $decay$ parameter $(0\ <$decay$<\ 1)$ to a time relative to the start of the query execution.

5.3 Scoring Scheduling Algorithms

Next we compute a scheduler's overall score, $scheduler\_score$, by combining the relative performance for all of the QoS metrics in the QoS specification.

In Equation 2, each of the normalized values computed by Equation 1 is multiplied by its corresponding weight w_i. The quantifier from the requirement is used to determine if we wish to maximize or minimize the QoS metric. If the quantifier is to maximize, we will use z_i. If the quantifier is to minimize, we use $-z_i$.

$$scheduler\_score = \sum_{i=0}^{I} (z_i)(w_i) \tag{2}$$

Finally by comparing S_{old}'s $scheduler\_score$ with the scores for all of the other algorithms (that have run so far), the adapter is in a position to select the next scheduling candidate. Notice that this calculation is very cheap. We are able to use a simple Radix sort to rate each algorithm in linear time. Section 5.5 describes the heuristic in which an algorithm is selected.

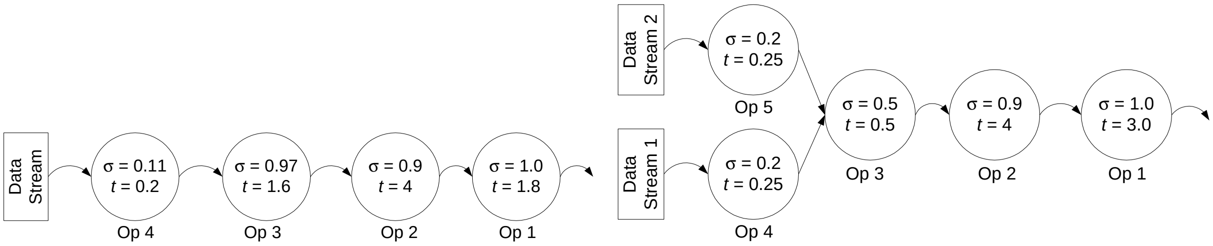
Analysis. Equation 2 gives a better score to QoS requirements with a high weight and a high z value from Equation 1. It maps each scheduler's score for each statistic to a value between 0 and 1 to allow for a fair comparison among different statistics. The weighed sum will also yield a value between 0 and 1 for each scheduler. In our case, we want a complete set of statistic values for one scheduler to map a complete set of statistics for a scheduler into a single value that could be compared against another set.

The score assigned to an algorithm is not based solely on the previous time that it was used, but rather it is an aggregate over time. While the performance of an algorithm is largely coupled to the characteristics of the data, over time the score of the algorithm should reflect its true potential. Therefore, the system is capable of handling reasonable fluctuations in the characteristics of the arriving data.

5.4 Scheduling Observations

Several observations must be considered when using the scores to determine the next scheduling algorithm:

1. Initially, all scheduling algorithms should be given a chance to "prove" themselves, otherwise the decision would be biased against the algorithms that did not yet run. Therefore, at the beginning of execution, we want to allow some degree of exploration on the part of the adapter. However, if the query is relatively short running, allowing too much exploration will prevent the adapter from doing its job.
2. Not switching algorithms periodically during execution (i.e., greedily choosing the next algorithm to run) could result in a poor performing algorithm being run more often than a potentially better performing one. Hence, we have to periodically explore alternate algorithms.
3. Switching algorithms too frequently could cause one algorithm to impact the next and skew the latter's results. For example, using Chain as described in Section 4.1 could cause a glut of tuples in the input queues of the lower priority operators. If MTIQ (Section 4.1) were to be run, its throughput would initially be artificially inflated because of the way Chain operated on the tuples. If we switched to another algorithm soon after, the z-score from Equation 1 for the throughput would be skewed. More generally, when a new algorithm is chosen, it should be used for enough time such that its behavior is not significantly overshadowed by the previous algorithm.



(a) Query Plan 1

(b) Query Plan 2

Figure 5. Query Plans used in Experimentation.

5.5 Adapting Scheduling Selection

After each algorithm has been given a score based on its performance, the system needs to decide if the current scheduling algorithm performed well enough that it should be used again or if better performance could be achieved by changing algorithms. Considering Observation 1 above, initially running each algorithm in a round robin fashion is the fairest way to start the adaptive scheduling.

Once each algorithm has had a chance to run, there are various heuristics that could be applied here to determine if it would be beneficial to change the scheduling algorithm. The heuristics are divided into two groups. The first group will simply choose the best suited algorithm using the equations described above, while the second group will eliminate poor performing algorithms. The idea behind eliminating poor performing algorithms is that if an algorithm has performed worse than all other algorithms for quite some time, there is no need to consider it any longer as its chance to be scheduled in the near future is low. This heuristic cannot be too hasty in removing any algorithm from consideration because stream characteristics could be the cause of poor performance for any one algorithm.

In an effort to consider all scheduling algorithms while still probabilistically choosing the best fit we adopted the Roulette Wheel strategy [12]. This strategy assigns each algorithm a slice of a circular “roulette wheel” with the size of the slice being proportional to the individual’s score. This strategy is also referred to as “fitness proportion selection”[12]. Then the wheel will be spun once and the algorithm under the wheel’s marker is selected to run next. This strategy was chosen for this framework because it is very lightweight, and does not cause significant overhead. In spite of its simplicity, this strategy is sufficient to significantly outperform single scheduling strategies, as we will see in Section 6.

This strategy may initially choose poor scheduling algorithms, but over time should fairly choose a more fit algorithm. The strategy also allows for a fair amount of exploration and thus it prevents one algorithm from dominating.

6 Experimental Evaluation

6.1 Experimental Setup

In this section we briefly discuss our experimental test bed and the results of our adaptive framework. Our goal was to compare our adaptive framework against other scheduling algorithms, including Round Robin, Chain, FIFO, and MTIQ.

We used data from the Internet Traffic Archive [10] as our data set. This data was used to represent the contents of real streaming data. The arrival rates of the streams were set to have a random pattern using poisson distribution. The streams were steady at times, and rather bursty at other times. These streams are sent across a 10

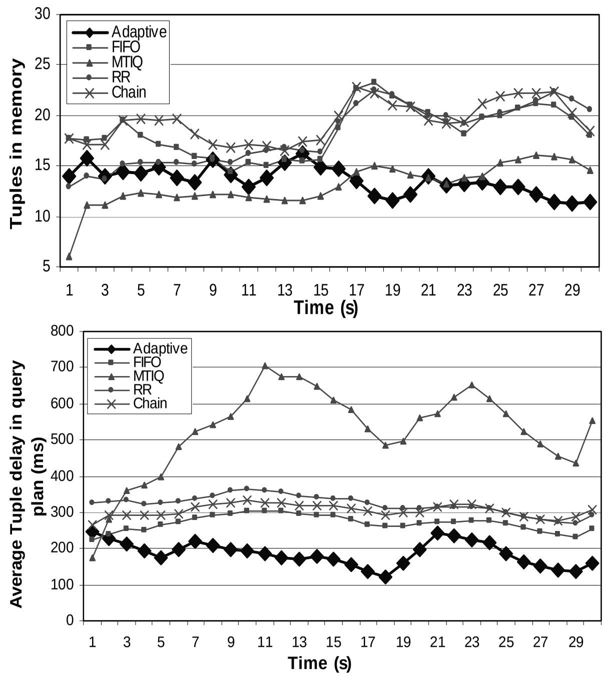


Figure 6. Optimizing query execution with one QoS requirement.

BaseT LAN to show how CAPE will respond to receiving data from a remote source rather than locally generated data. The stream rates were adjusted using custom built Stream Sources that would generate data with different poisson means every 5 seconds. This was done to show that under both steady and bursty conditions, the adaptive framework could respond with good experimental results.

Two query plans were used in the experimental results in this paper. These plans were selected as a basis to compare published algorithms such as Chain[2]. The query plans are made up of “mega-operators” which run for longer periods of time than a typical query operator, but allow us to more closely observe how the scheduling algorithms process data. The first query plan is a simple query plan with four mega-operators. The second query plan utilizes a window join operator [9] with a window of 200ms. That is, any tuples that are received within 200ms of each other are evaluated in the join predicate of the operator. The query plans are shown in Figure 5 with selectivity denoted by (σ) and average tuple processing time by (t).

Our system aimed to optimize up to three QoS requirements: average output rate, average tuple delay, and average memory size (Section 5.1). These requirements were selected for experimentation because each requirement is vastly different and no one scheduling algorithm can optimize for such different requirements.

6.2 Direct Competition with Published Scheduling Algorithms

The first experiment used a QoS specification with only one requirement. This was done to demonstrate that the adaptive framework can pick an optimal scheduling algorithm even for only one requirement. Figure 6(a) shows that the adaptive framework does exceptionally well at selecting algorithms to keep tuples in memory down. In fact, at many times the framework outperforms every single scheduling algorithm in terms of memory.

In Figure 6(b) we can see that the adaptive framework outperforms all individual scheduling algorithms. It can outperform the other algorithms by leveraging their relative strengths. It was observed that MTIQ can exploit queue buildups caused by FIFO. As FIFO begins execution, a buildup of tuples is created at the leaf operator. Since there is a buildup in tuples at the leaf operator MTIQ is selected (at time t=7) and progresses the tuples through the query plan. FIFO is then selected again (at time t=21) as older tuples were still in the query plan that

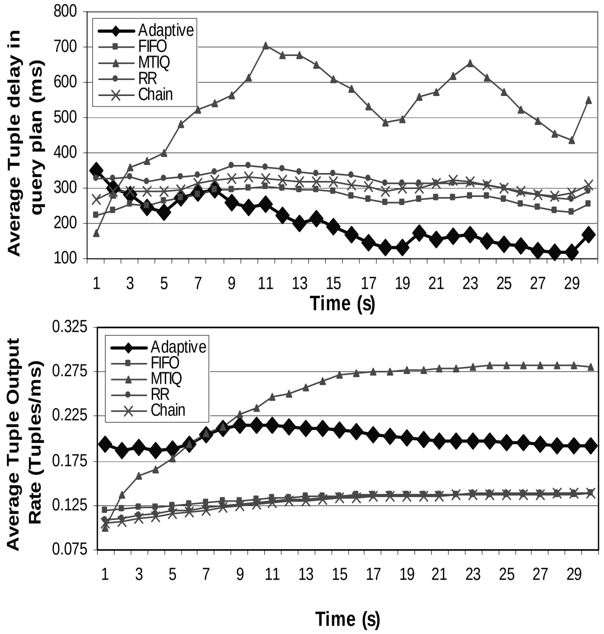


Figure 7. Optimizing query execution with two QoS requirements. 70% focus on minimizing tuple delay, and 30% focus on maximizing output rate (Query Plan 1)

needed to be processed.

6.3 Reaction to Changing QoS Specifications

For our second set of experiments, we show how the adaptive framework reacts to a QoS specification with two requirements. We have two goals in this set of experiments. First we would like to show that if we change the importance of a QoS requirement, the framework will acknowledge this and adapt accordingly. Secondly it is important that the framework performs well in both QoS requirements.

In Figure 7 we depict the results for an experiment for which we place 70% importance on tuple delay and a 30% importance on output rate. Here we observe that the adaptive framework outperforms single algorithms with respect to average tuple delay, and performs about average with respect to the average output rate.

Figure 8 shows our performance when we adjust the percentage of the weights to 70% focus on maximizing output rate, and 30% focus on minimizing tuple delay. We observe that with the change in service requirement, the adaptive framework still does exceptionally well at minimizing tuple delay, and improves significantly at raising the average tuple output rate. This shows that the adaptive framework can adapt accordingly to varying QoS requirements, and also provide significant improvements of single scheduling algorithms.

We will now consider the case of having two equally important QoS requirements. Figure 9 shows the performance of the adaptive framework with an equal focus on average output rate and average tuple delay. We make two observations from these charts. First, clearly there is no single optimal scheduling algorithm, as each algorithm exhibits varying performance throughout execution. Second, our adaptive framework is able to outperform all single scheduling algorithms for the duration of execution, on average.

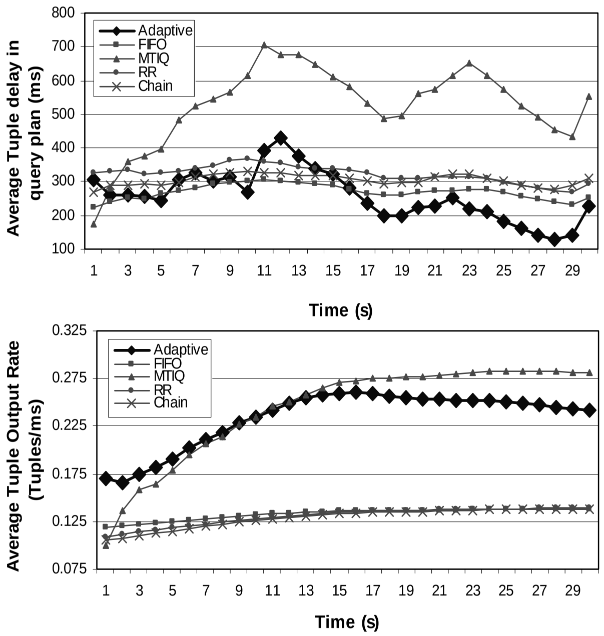


Figure 8. Optimizing query execution with two QoS requirements. 30% focus on minimizing tuple delay, and 70% focus on maximizing output rate (Query Plan 1)

6.4 Adaptive Framework with Multi-Faceted QoS Specifications

In our final set of experiments we compared the performance of the adaptive framework against the single scheduling algorithms with a QoS specification of three requirements. In this example each requirement (average tuple delay, average output rate, and average tuples in memory) was each given equal weight.

In Figure 10 we can see that the adaptive framework again performs well under all three QoS requirements. The biggest improvements are average tuple delay and the number of tuples in memory, where the adaptive framework significantly improves upon practically all single scheduling algorithms.

6.5 Experimental Recap

In the above section we have presented how the adaptive framework boosts overall performance of query plan execution in the continuous query system. In most cases, the adaptive framework performs exceptionally better than any single algorithm alone. In all other cases, the adaptive framework performs at least above average with respect to the single algorithms. The framework also adapts well to changing service requirements. When the focus shifts to a particular requirement the framework is able to adjust to this and act accordingly. Overall the adaptive framework greatly improves both the performance and flexibility of the continuous query system.

7 Related Work

There is a recent surge of ongoing research in the field of executing queries over streaming data. [3][16][14] provide a comprehensive overview of the challenges of executing queries in a stream environment. Most closely related to ours is that of STREAM [3] and Aurora [7].

The STREAM [3] project's goal is to "manage resources carefully, and to perform approximation in the face of resource limitations in a flexible, usable, and principled manner." STREAM focuses on efficiently allocating

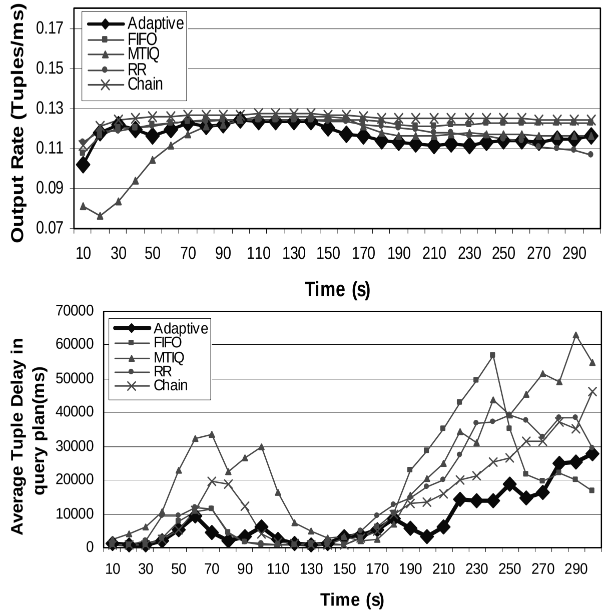


Figure 9. Optimizing query execution with two QoS requirements. 50% focus on minimizing tuple delay, and 50% focus on maximizing output rate (Query Plan 2)

memory to queues, synopses, and operators by making use of stream constraints and the Chain [2] scheduling algorithm. STREAM also provides techniques to best approximate the query result using various static and dynamic techniques such as reducing window sizes.

STREAM differs from our work in the following ways. First, STREAM only supports one scheduling algorithm, namely Chain. The Chain scheduler does not consider QoS specifications such as maximizing tuple throughput or minimizing overall response time. While Chain is an ideal strategy for minimizing intermediate queue sizes, it is not as effective in other QoS requirements. In our work, if a scheduling algorithm starves or is ill-performing, we are able to choose an alternative algorithm that will be able to more closely meet the desired QoS specification. Chain is an ideal algorithm for our framework because we can exploit its advantages whenever the QoS requirements allow us to do so.

Aurora [7] aims to reduce tuple execution costs while maximizing overall QoS goals. They accomplish this by first having queues collect as many tuples as possible without processing and then the operator processes all tuples at once generating a train of data. The benefit is that tuples passed to subsequent operators do not have to go to disk. Aurora allows the administrator to input a graph to define an acceptable QoS. Aurora takes into account several different QoS metrics such as response times, tuple drops, and importance of values. It also allows for arbitrary compositions to be created, similar to our QoS specifications.

This differs from our work in that Aurora makes use of one dynamic scheduling algorithm as opposed to using different algorithms to try to improve system performance. While Aurora also focuses on maintaining administrator-specified QoS requirements, the key difference is the way that QoS is inputted into the system. Aurora requires a convex graph to allow the administrator to specify the specify the “quality” given an absolute performance metric, such as tuple delay, or tuple values. This has proven to be very effective, since Aurora uses tuple shedding when the performance degrades significantly. In CAPE we specify a QoS set as a linear function of each QoS and its relative weight in the system. Rather than trying to obtain an absolute performance, we aim to perform as well as we can given the current system state. This is also easier for the administrator creating the QoS set because they do not need to know absolute values to create graphs. They can input what requirements

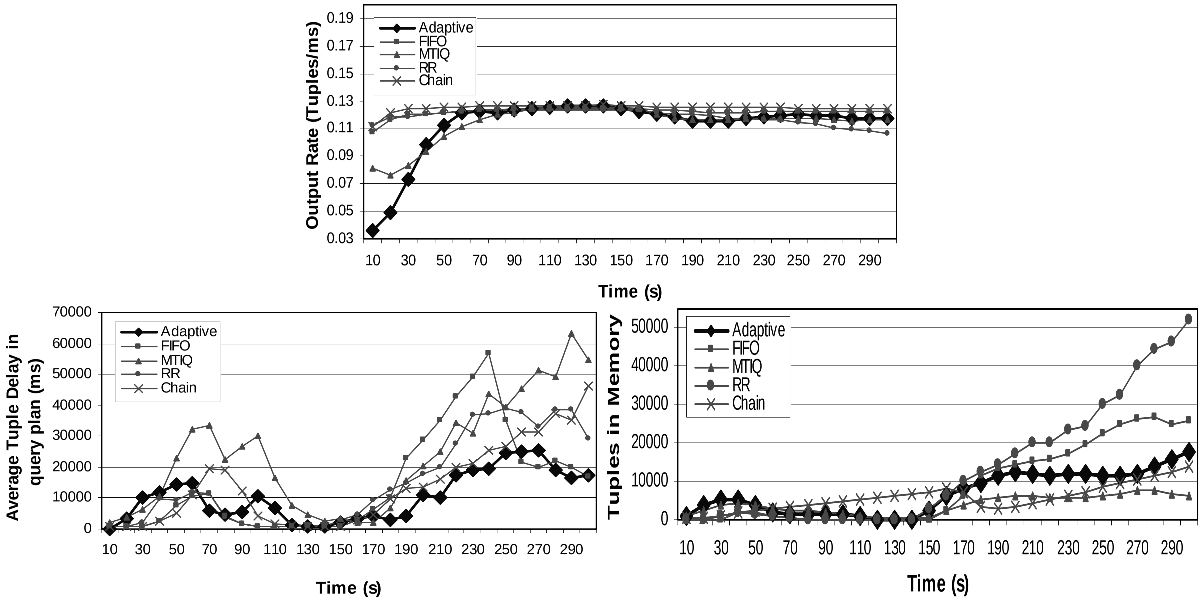


Figure 10. Optimizing query execution with three equal QoS requirements (Query Plan 2)

they want to perform well, relative to other requirements in the system.

Telegraph [15][11] is another adaptive query system that makes use of Eddies [1] to adapt execution for each tuple. Eddies uses a lottery-type scheduler to route tuples to an operator that will need to eventually process the tuple. The goal is to prevent tuples from waiting in input queues for a slow operator to be ready to process them. [15] extended the previous Eddies work by providing support for queries over streaming data. This level of adaption is much finer compared to what is used in our work. Our work instead is more comparable with the ”traditional” notion of a query plan (and thus maintains intermediate data in queues) as adapted by all stream systems besides Telegraph, including STREAM, Aurora and NiagaraCQ. Eddies does not allow for QoS metrics to be specified, rather it follows its hard coded optimization goal.

Niagara [13] is a continuous query system that uses XML as data format. Niagara focuses on efficiently sharing processing between large amounts of continuous queries. No work on scheduling has been reported to date.

Rate-based stream scheduling in [19] deals with ordering the execution of input streams so that the stream with the highest output rate will have a higher priority, and thus will be executed more often. The goal is to produce tuples as quickly as possible (maximize throughput). They consider cases where the tuples have equal and nonequal importance.

8 Conclusion

In this paper, we proposed a novel scheduling-strategy selection framework that leverages the strengths of individual scheduling strategies to meet the arbitrary compositions of QoS requirements. Our framework uses the recent performance of each algorithm in determining how to best adapt given the system’s current state. Using a lottery based heuristic, an algorithm is selected based on its likeliness to succeed based this recorded performance.

We have found that not only was our adaptive framework able to significantly improve performance for a combination of QoS requirements, but it was also able to react to requirements as they changed over time. The framework was also able to succeed where existing scheduling algorithms would fail because of their one-dimensional nature. As a result, our technique was able to aid several highly tuned algorithms, such as Chain [2], in areas where the algorithm would normally not produce satisfactory results.

In addition, our extension is general and complimentary to typical systems such as Aurora or STREAM. This framework could easily be plugged into these existing systems. Thus the benefit of these ideas could be far reaching for Continuous Query Systems.

As ongoing work, we will study further ways to make the framework more flexible. We are currently working on creating a distributed continuous query engine that will be able to handle even larger volumes of data and query plans. Finally, another area of future work will be to study how a QoS specification can be defined for individual queries or sub-queries to improve performance at a more fine-grained level.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272. ACM Press, 2000.

[2] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264. ACM Press, 2003.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, et al. Models and issues in data stream systems. In *ACM SIGMOD-SIGACT-SIGART*, pages 1–16. ACM Press, 2002.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, May 2000.

[5] D. Abbadi, D. Carney, U. Cetintemel, et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, pages 120–139, 2003.

[6] D. Carney and U. Cetintemel and A. Rasin et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[7] D. Carney, U. Cetintemel, M. Cherniack, et al. Monitoring streams: A new class of data management applications. In *VLDB*, pages 215–226, 2002.

[8] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152. ACM Press, 1990.

[9] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, September 2003.

[10] Internet Traffic Archive. http://www.acm.org/sigcomm/ITA/, 2003.

[11] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[12] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[13] J. F. Naughton, D. J. DeWitt, D. Maier, et al. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[14] R. Motwani and J. Widom and A. Arasu, et al. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, pages 245–256, 2003.

[15] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, et al. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, New York, NY 10036, USA, 2002. ACM Press.

[16] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[17] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX*, pages 13–24, Berkeley, USA, June 15–19 1998. USENIX Association.

[18] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3):555–568, 2003.

[19] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, Los Altos, CA 94022, USA, 2001.

[20] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS*, pages 214–225. ACM Press, 1990.

[21] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, June 2004, to appear.