Quality of Service Adaptation in Data Stream Management Systems: A Control-Based Approach

Yi-Cheng Tu, Yuni Xia, and Sunil Prabhakar

Department of Computer Sciences, Purdue University
1398 Computer Sciences Building
West Lafayette, IN 47907-1398
U.S.A.
{tuyc, xia, sunil}@cs.purdue.edu

Abstract

Unlike snapshot queries in traditional databases, the processing of continuous queries in Data Stream Management Systems (DSMSs) needs to satisfy user-specified QoS requirements. In this paper, we focus on three major QoS parameters in a DSMS environment: *processing delay*, *querying frequency* and *loss tolerance*. To minimize processing delays, the Earliest Deadline First (EDF) CPU scheduling policy is recommended. QoS control for data streams is a difficult problem because both the processing cost and data rate can be unpredictable. One salient problem is that DSMSs are vulnerable to overloading as the data points tend to come in bursts. QoS degrades significantly when the system is overloaded. We propose a QoS adaptation framework that smartly adjusts application QoS and performs admission control based on the current and historical system status. The design of the QoS adaptation framework takes advantages of techniques from the area of control theory. Algorithms derived from control theory are mathematically analyzable and this makes the parameter tuning an easy job. In addition to load shedding, our QoS adaptor adjusts system load by changing multiple QoS parameters for streams. Extensive simulations using both synthetic and real data sets are carried out to evaluate the performance of the proposed control-based QoS adaptation framework. The results show that: compared to current solutions for QoS control in DSMS, the control-based strategy provides better guarantees of QoS and achieves higher resource utilization. Our solution is also light-weight and easy to implement.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

The recent flourishment of data streaming applications has attracted a great deal of research interest from the database community. A number of Data Stream Management Systems (DSMSs) are being developed for the purpose of information/knowledge extraction (querying) from stream data [1, 2, 3, 4]. The main challenge of handling data streams is imposed by the fact that the volume of data is unbounded as they are generated in a continuous manner. As a result, most queries against stream data are persistent queries that output results continuously whenever they are available. A salient feature of queries injected to DSMSs is the associated real-time constraints. Unlike a traditional DBMS where people are only interested in the *correctness* of query results, queries in a DSMS are generally required to be delivered in a timely fashion. However, we may accept query results of different levels of accuracy in a DSMS due to resource constraints [5, 6]. The set of parameters that describes the temporal/spatial requirements of applications are usually called *Quality of Service* (QoS)[1]. Similar to those in multimedia applications, the QoS parameters on queries in a DSMS are closely related to timeliness, reliability, and precision. We will introduce the QoS parameters found in DSMSs in Section 2.

Generally speaking, the QoS control problem is a re-

[1]Here QoS is different from the concept of *data quality*. In this paper , the words 'QoS' and 'quality' are used interchangeably.

source management problem and has to be considered at both application and system levels [7]. QoS guarantees for multiple applications in a system are difficult to maintain for two major reasons: physical resource limitations and sharing of these resources by statistical multiplexing. These are of no exception for the environment where a DSMS runs. A data stream system could easily accommodate thousands of streams and further service hundreds of continuous queries, all of which share the same resource pool. As more data streams and queries request admission to the system, a decision has to be made so that the system is not overloaded. QoS requirements for individual applications may still be violated even with careful admission control. The fluctuations of resource availability (e.g. bandwidth variations of a wireless link) and application resource usage (e.g. bursty traffic) may cause temporary overloading that interferes with the real-time processing of concurrent jobs. In a DSMS, data generally arrives in a bursty fashion. This is mainly caused by the value-based updating method whose frequency is not known *a priori*. Although strictly time-based data refreshing is also entertained in a DSMS, we focus on QoS control under the value-based scheme in this paper as it is a more common and challenging problem.

Research on QoS control was first motivated by the real-time requirements imposed by multimedia applications. Most of these efforts emphasize system and network level resource management. QoS guarantees are provided as a service of the operating system [8] or middleware [9]. The system maps QoS requirements of applications to resource use (system QoS) and QoS control is accomplished by regulating resource allocation to individual applications. For DSMSs, however, system support is not the ideal solution for supporting quality parameters of applications. First of all, the basic computation unit that can register their QoS needs via a real-time operating system or middleware is a thread [2]. We cannot run each data stream as a thread as their number is in the thousands and no known system can support this many threads [10]. Secondly, a real-time operating system lacks the ability to capture the application-level semantics of QoS. For example, when looking at the same streams of bank transaction data, a security manager requires timely report of unusual events while a marketing analyst is more interested in sampling the data with a stable frequency. For the above reasons, we believe that QoS control and resource management should be an integrated functionality of a DSMS.

As mentioned earlier, streaming data are intrinsically dynamic in their arrival pattern. This makes mapping from application level quality requirements to resource use (i.e. system and network QoS) extremely difficult as the estimation of future resource consumption could easily fail. As a result, traditional reservation-based QoS control methods [11] are not applicable to DSMSs. What we need is an quality-oriented adaptation framework that smartly adjusts QoS levels of applications in response to fluctuations of system and input status. Specifically, the adaptation framework should have the following features:

1. A mechanism that continuously monitors QoS and detects status changes. The measurements are sent to a decision-making entity as a feedback signal. Special care is needed in identifying the appropriate system parameter(s) to be monitored;

2. Based on signals obtained from the monitor, we need a means to determine whether adaptation should be performed and the magnitude of the adaptation action (i.e. set a target on how much to adapt). The decision-making should take internal and environmental uncertainties into account and be optimized towards quick convergence and long-term system stability;

3. QoS adaptation should accurately meet the adjustment target set by the above controlling entity. More important is to maintain fairness among streams while preserving system utility.

In this paper, we present a QoS adaptation framework to meet all three challenges listed above.

The idea of QoS control in DSMSs is not new. Current work in both the Aurora and STREAM projects studied strategies to maximize individual quality parameters such as query precision [12] and loss tolerance [1]. Various strategies of load shedding as a means to handle overloading in DSMSs is discussed in Aurora (see Section 5 for more detail). These researches suffer from two major drawbacks:

1. *Lack of a QoS control model.* DSMS-level QoS support is generally accomplished as a control loop. Consider the load shedding example: the DSMS first detects the load injected into the system and determines the amount of traffic to shed. The problem in current solution is that the choice of input/output signals to the controlled system as well as the relationship between the signals are not justified by an analytical model. Therefore, the direction and magnitude of the adaptation task (e.g. shedding) have to be determined in a rather simple way that leaves much room for improvement.

2. *Insufficient support of user-level QoS semantics.* In the researches mentioned above, QoS is considered more on the system level and user preference on QoS is ignored in adaptation. We believe user preference is essential as QoS is defined to capture user satisfaction in the first place.

As an improvement to the first problem, we come up with a solution that is derived from the theory of feedback control and well-established results of real-time

[2] Some only provide services on the process level.

systems. We first build a QoS control model centering around a resource scheduler that provably minimizes QoS violations under certain conditions. Based on this model, we formulate QoS adaptation in DSMS as a feedback control problem and develop algorithms to make adaptation decisions (challenges 1 and 2) in a quantitative way. Our model takes unpredictable traffic pattern and processing cost into account thus it deals with these uncertainties better than the simple solution. The use of a model also provides help in runtime system tuning, which could be a painful experience if no model is used. For the second point, we propose application-oriented adjustment of multiple QoS parameters as the major approach for adaptation (challenge 3), in addition to single-QoS solutions such as load shedding. In doing so, user preference on QoS is better maintained as we assign higher utility to QoS combinations that reflect such preferences.

There are two major contributions made in this paper. First of all, we propose a unified framework for QoS provisioning in DSMSs with the focus of QoS adaptation. The adaptation strategy emphasizes satisfaction of user-specific QoS requirements. Secondly, we introduce a new approach for dealing with QoS adaptation: the feedback control approach. It is shown in our experiments that the control-based approach significantly improves QoS compared to simple solutions that play by rules of thumb. To the best of the authors' knowledge, this is the first work that applies control theory to data stream management research. The control-based approach may also be applicable in a broader range of research topics in database systems.

The rest of this paper is organized as follows: Section 2 describes the basic DSMS model on which we build our QoS framework and identifies the major QoS parameters in such systems. Details of our feedback control QoS adaptation scheme can be found in Section 3. Performance evaluation of our proposed framework is presented in Section 4. A literature survey of related research is given in Section 5. Finally, the paper is concluded by Section 6.

2 Model Description

We follow the push-based data stream query processing model proposed by the Stanford STREAM project [12]. In this model, each query plan consists of a small number of *operators* chained as a pipeline. Each stream is associated with such an operator pipeline. Multi-stream joins are performed as hash joins over a *sliding window* whose width is specified by the application either in number of tuples or time. Upon the arrival of a single data point (*tuple*) of a stream, the operators are processed one by one with the tuple as input. The processing of all operators in the pipeline for a tuple is viewed as the basic unit for resource scheduling. In other words, we do not break the pipeline of operators.

Each operator pipeline has a CPU cost model in the form of $(c_1, s_1)(c_2, s_2)\cdots(c_n, s_n)$ where (c_i, s_i) is the cost and selectivity of a single operator in the pipeline. The aggregated cost for the whole pipeline $h = \sum_{i=1}^{n}\prod_{j=1}^{i-1} s_j c_i$ [6] is the cost we are interested in as the pipeline is always scheduled as a whole. We assume h can be estimated through sampling with a variance factor G_h. For example, a G_h of 2 means the real cost measured could go as high as $2h$ at runtime. The cost is applied for every tuple that arrives. This includes the tuples for multiple streams in a window join query. Under the circumstances of stream sharing among multiple queries, we duplicate the tuple for each query. We assume in-memory processing of all operators. The choice of DSMS model is not critical in this work. The control-based framework works for other DSMS models such as that of Aurora [1].

QoS is defined for each stream in our system. Depending on the application we are interested in, the selection of QoS varies. We believe the following QoS parameters are applicable to a majority of DSMS applications:

- **Tuple delays**. This parameter captures the time elapsed between the generation and the end of processing of a tuple. This is a critical QoS in many systems with real-time features such as stock trading and temperature monitoring in chemical reactions. In such systems, longer delay usually means poor data freshness and higher uncertainty therefore the utility drops quickly. Generally, there are *network delays* and *processing delays*. We focus on the latter in this paper. Users can only choose their delay requirements from a set of discrete values.
- **Updating frequency**. Also called *data coherency* in some literature [13]. It shows how often a stream should generate a new value and ship to the database. The frequency can be defined either in time intervals (e.g. *report new readings every 5s*) or value intervals (e.g. *report whenever the stock price changes by 1%*). The second type of updating frequency is a major reason for the bursty arrivals of updates and is the focus of this research.
- **Data loss ratio**. This shows the level of load left unprocessed in QoS adaptation. As tuples are inevitably discarded in case of overloading, each user can specify the maximum percentage of tuple losses for the stream he/she owns.

Users specify QoS in the form of a vector $\vec{q} = (q_1, q_2, \ldots, q_n)$ with q_i being the value on a single QoS dimension. The user QoS preference is captured by a weight w_i of individual QoS parameters. Thus, the utility of any $\vec{q}$ is

$$\mathcal{U}(\vec{q}) = \sum_{i=1}^{n} w_i u_i(q_i) \tag{1}$$

where u_i is the dimensional utility function and we have $\sum_{i=1}^{n} w_i = 1$ for a total of n QoS parameters. We assume all dimensional utility functions are monotonically increasing, convex, and take values in the range $[0, 1]$.

Our research concentrates on minimizing violations of tuple delays, or, deadline misses (DMs) as we believe this is the most important QoS for all users. The limited number of delay classes is just a design decision. It makes no difference if we allow deadline classes with a finer granularity. The ignorance of network delays can be justified by the features of systems where transmission delays and jitters are effectively controlled. We understand network delay dominates in other situations and we leave it to future research.

Various system resources including CPU [6], memory [14], and network bandwidth [15] could be the bottleneck in DSMS query processing. With the previous assumption of sufficient memory and network throughput, we only need to consider the management of CPU cycles.

3 The Control-Based QoS Adaptation Framework

3.1 A brief tutorial on feedback control

The central idea of feedback control is the *feedback control loop*, which consists of the following components (Fig 1): 1. A *plant* to be controlled; 2. A *monitor* (sensor) that periodically measures the relevant status of the plant; 3. The measurements from the monitor are sent to a *controller* as an *output signal*. The controller compares the value of the signal with a *target* value that is set beforehand. The difference between the output signal and the target is called the *error*. The controller then maps the error to a *control action* (or *input signal*); 4. An *actuator* adjusts the behavior of the plant based on the control action.

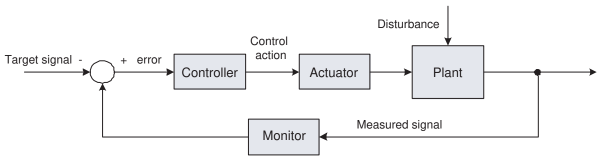

Figure 1: A basic feedback control system.

An essential part of the control loop is the control function in the controller that maps error to input signal to the plant. Numerous theoretical studies have been conducted on the derivation of mapping functions under various situations. Intuitively, this is the main feature that distinguishes the feedback control from a simple solution that uses rules of thumb. The latter is not unusual in practice, especially in complex systems that are not controllable. The TCP rate control in response to congestion is an immediate example. Feedback control is not suitable for all adaptation systems. Several conditions have to be met before we can apply control theory to the scenario of interest. First of all, signal monitoring should reflect the status of the system with certain accuracy (*observability*). Secondly, the system should be controllable, or have the capability to move to the desired state specified by the actuator (*controllability*). Last but not least, we need a model that describes the dynamics of the controlled plant. This model should accommodate input/output signals, internal uncertainties, and environmental disturbances. The more precise the model, the better control we get.

Readers interested in more details on control theory can refer to [16]. In the following section, we discuss how to transform the above generic model into a concrete model that solves our QoS adaptation problem.

3.2 The real-time resource scheduler

Let us start our discussions of the control-based QoS framework by modeling the plant. Here the plant is simply the original DSMS augmented with the resource management component, which is indispensable in any QoS-provisioning system. In our QoS-driven plant with the main focus of timely query processing within bounded delays, a real-time CPU scheduler is deployed.

There are two main categories of real-time CPU scheduling policies: dynamic priority policies and fixed priority policies [17]. The first class is represented by the Earliest Deadline First (EDF) algorithm while the second class by the Relative Deadline Monotonic (RDM) algorithm. Both algorithms are straightforward: EDF schedules jobs strictly in an increasing order of their absolute deadlines while RDM does so by their relative deadline. In our system, let r be the arrival time of a tuple and t be its user-specified processing delay, EDF sorts the jobs by $r + t$ while RDM by t. We choose EDF to implement our scheduler as it is shown to have a *utilization bound* [3] of 1.0 for both periodic [17] and sporadic tasks [18]. We tested the utilization bound of our scheduler by injecting a series of randomly generated loads into a simulated CPU queue (Fig 2, experimental setup in Section 4). As we can see, very few DM events occurred when the request utilization (denoted as U_r) of the injected load is smaller than 1.0. When the requested load with U_r values in the range of $(1.0, 1.3)$, it shows a very clear linear relationship between DM ratio (denoted as M) and U_r. Regression analysis of the data points in this range gives a gradient of 0.64 with a regression coefficient of 0.94. Note that this does not mean the linearity holds for U_r values beyond 1.3, but this is good enough for our controller design.

[3] The threshold utilization value to guarantee zero deadline misses asymptotically.

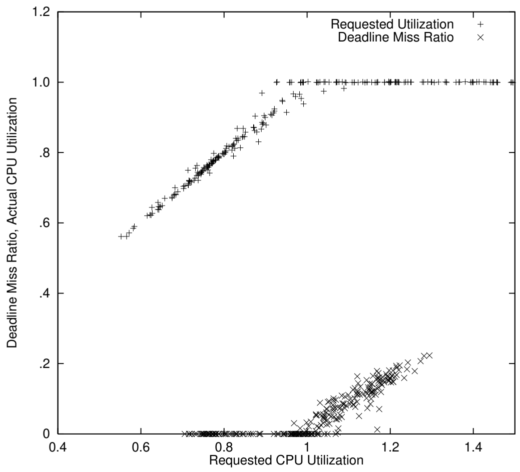

Figure 2: Verification of the linear control model.

Another result we can derive from Figure 2 is the linear relationship between U_r and the actual utilization U. This is easy to understand because the actual CPU utilization equals the requested CPU when the latter is smaller than the CPU capacity. Otherwise the CPU is overloaded and U stays around 100%.

At this point, we can see that the nice linear relationships between U_r and M, U_r and U form a solid foundation for a feedback control system. The variable M represents the QoS achieved by the plant and is a perfect candidate for the output signal. The request load U_r represents a controllable plant status and can be used as a control signal. For example, we can adjust the requested utilization to the system by shedding some of the loads when we detect DMs. The amount to shed can be calculated based on the quantity of DMs measured. As the DM curve in Figure 2 saturates when $U_r < 1.0$, we can use U as output signal because the linearity of the U-U_r model still holds true at this range of U_r.

Soft vs. Hard Real-Time. Our QoS framework may be used in a soft real-time environment: jobs that have already missed their deadlines will not all be discarded. Which jobs to discard is the actuator's decision. Most likely, we can add DM ratio as a new QoS parameter under the soft real-time scenario. The modeling of the plant will also be different. The requested utilization (U_r) is no longer a valid input signal. We introduce a new signal, the *total accumulated utilization* (U_c), which is defined as the sum of the load in the CPU queue leftover by previous monitoring cycle and U_r. Linearity still holds between U_c and M, U (data not shown). For simplicity of reasoning, we still use U_r in the rest of this paper assuming a hard real-time DSMS.

Scheduling and context switch overhead. Theoretically, the time complexity of EDF is logarithmic to the queue size because each job needs to be inserted into a priority queue upon arrival. This is not as good as constant time algorithms such as RDM. But the latter suffers from starvation and is harder to model. With the limited number of processing delay classes, we may find efficient ways to implement EDF but that is not the focal point of this paper. In practice, as tuples are discarded in case of system overloading, the queue size will be kept small. Therefore, insertion into the queue only takes a few memory accesses. As discussed in STREAM [14], context switch in our DSMS model is accomplished by performing procedure calls, which bears extremely low CPU costs. The weight of our scheduler is even lower as we always schedule the operator chain as a whole.

3.3 The feedback control loop

Figure 3 shows the architecture of the control-based QoS adaptation framework. This is basically a map from the architecture shown in Figure 1. Plant status are monitored periodically and output signals sent to the controller. We do not have a separate monitor here since the signals are just some variables kept by the scheduler. The actuator here is broken into two parts: the Admission Controller (Section 3.4) and the QoS adaptor(Section 3.5). The former decides whether new streams can be accommodated by the system while the latter controls the load into the CPU queue. Both components make their decisions according to the input signals got from the controller.

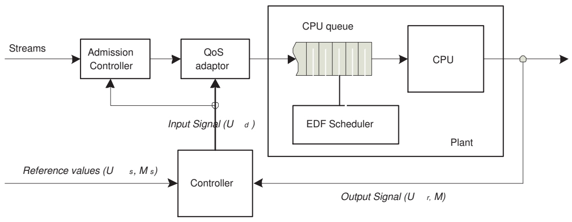

Figure 3: The QoS adaptation framework based on feedback control.

Next, we describe how the input signal is generated. We start by the model of the control loop. Based on observations from Section 3.2, we have the following model for U:

$$U(k+1) = U(k) + G_a(k)U_d(k) + D(k). \qquad (2)$$

The item $G_a(k)$ is added to capture the variance of the load change by QoS adaptation (due to imprecise cost estimation). The irregular traffic load is modeled as disturbance ($D(k)$) to the plant. Similarly, we have the following for M:

$$M(k+1) = M(k) + mG_a(k)U_d(k) + D(k) \qquad (3)$$

where m is the slope of the U_r-M line in Fig 2 and $m = 0.64$ in our case.

On the system level, QoS adaptation is accomplished by controlling the load (U_r) injected into the plant such that the output signals stay around the reference value. The input signal U_d represents the quantity of the load to be changed. For example, a U_d value of -0.15 tells the adaptor to decrease the load by 15%.

With the assumption that the internal dynamics of the controlled plant follow the above equations, we choose the standard Proportional-Integral-Derivative (PID) controller to calculate U_d. Specifically, we use the following equations

$$U_d(k) = \alpha E(k) + \beta \sum_{i=k-W}^{k-1} E(i). \quad (4)$$

$$E(k) = \begin{cases} U_s - U(k) & \text{if } U(k) \geq 1.0 \\ m(M_s - M(k)) & \text{otherwise} \end{cases} \quad (5)$$

where W is the width of a time window used to calculate the integral part, α and β are constants representing the weight of P and I controllers, respectively. The D controller is ignored as it increases overshoots in response to disturbance. According to Eq (5), error correction has two cases using two different output signals M and U. This is because the U-U_r line saturates beyond $U_r = 1.0$ and the model we can use in this area is the M-U_r relationship.

We choose PID controller because it is simple, lightweight, and effective. According to control theory, it performs well in terms of stability, sensitivity, and steady-state error when used for low-order linear systems such as ours [16]. Due to space limitations, mathematical analysis of controller features and plant dynamics are skipped.

3.4 Admission control

We use an admission controller (AC) to help protect the system from unexpected high load. The AC checks if a newly registered stream can be served by the system given the current load. For a new task T_i, the theoretical admission condition is as follows:

$$U_r + h_i r_i \leq U_s$$

where h_i and r_i are the per-tuple cost and time-based updating frequency of T_i, respectively. The practical value of the above admission condition is limited for two reasons: 1. U_r only reflects the current load situation of the system; 2. both h_i and r_i are unknown when the stream first registers. We use some techniques to remedy the above conditions. We first make estimations of the the requested utilization by taking a moving average of current and historical data. For example, an estimation (U_e) of U_r can be written as

$$U_e(k) = \gamma U_r(k) + (1-\gamma) \sum_{i=k-a}^{k-1} U_r(i), \ (0 < \gamma < 1)$$

suppose we are on the k-th control period. Then we just use an optimistic admission controller as follows:

$$U_e(k) < U_s.$$

In utilizing the above optimistic condition, we assume the cost of one single stream is small compared to the total CPU capacity. Even if we make a bad decision at the AC, the QoS adaptor will provide a second chance to correct this decision. By controlling the system workload, AC increases the quality of the PID controller indirectly. The PID controller suffers from control errors when the incoming load fluctuates wildly. AC can smooth the aggregated traffic to make the fluctuations less dramatic.

3.5 QoS Adaptor

Load shedding is frequently used to protect DSMSs from overloading [6, 19]. In order to effectively relieve system load and yet preserve utility, the load shedding algorithm needs to know when, where to shed load and how much load to shed. When we utilize a load shedder within our QoS adaptor, the feedback control loop immediately gives answers to two of the questions: *when* and *how much*. We keep a system variable, the *shed factor*, to guide the shedding action assuming we have a single load shedder deployed as part of the QoS adaptor. With values in the range $[0, 1]$, a non-zero shed factor means load shedding needs to be performed. For any control period k, the shed factor ($S(k)$) is obtained by

$$S(k) = 1 - (1 - S(k-1))(1 + U_d(k-1)). \quad (6)$$

We can see that the shed factor is incrementally changed based on the previous value $S(k-1)$ and the input signal U_d. Equation (6) is derived via the following reasoning: the actual load during period $k-1$ is $U_r(k-1)(1-S(k-1))$ and the desired load for the next period k is $U_r(k)(1-S(k-1))(1+U_d(k-1))$ where the item $1+U_d(k-1)$ shoot in as an adjustment.

The above algorithm does not specify where (which streams, to be specific) to shed load. The simplest load shedding strategy basically chooses victim tuples randomly. Such a statistical shedding method is obviously not optimized towards inter-stream fairness and total system utility. The Aurora system [6] smartly sheds load by choosing the streams with the lowest utility-to-cost ratio as victims. Our QoS adaptor follows the same idea but it has higher flexibility in adjusting system load.

3.5.1 QoS adaptation as an optimization

Our QoS adaptor regulates system load by changing the QoS level of applications. Among the three QoS parameters, degradation of either loss ratio or updating frequency will lead to decreased load request. On the contrary, relaxing processing delay of individual streams has no effect on their CPU use. Therefore, we concentrate on how the first two QoS are assigned to streams. The goal of QoS adaptation is to adjust load to U_s while the system profit is maximized. The profit achieved by an algorithm is generally modeled by utility, as described in Section 2. An advantage of our algorithm over load shedding is that less resource will be wasted when the system is underloaded. Under this situation, a positive control signal is generated and QoS levels of certain applications increase.

We now fill the QoS model with more details before going into further discussions. Assume there are N streams at the moment of decision-making. Streams are divided into different priority classes, each with an importance factor δ. A QoS vector now contains two elements: q_1 (frequency) and q_2 (loss ratio). For easy analysis, we use *non-loss ratio* as q_2 since we want utility functions to monotonically increase. In addition to the CPU cost h, each stream is also equipped with another parameter r: the estimated data rate. We assume r is mapped from q_1 and q_2 via a function f that monotonically increases on both dimensions:

$$r = f(q_1, q_2) = g(q_1)q_2 \tag{7}$$

where g is the function that maps value-based frequency to time-based frequency. The function f is called *rate mapping function*. The monotonicity of f is easy to understand because data rate increases as either q_1 or q_2 increases. Then the adaptation of QoS in various streams can be formulated into the following optimization problem:

$$\begin{array}{ll} \text{Maximize} & \sum_{k=1}^{N} \mathcal{U}_k(\vec{q}_k)\delta_k, \\ \text{subject to} & \sum_{k=1}^{N} h_k f_k(\vec{q}_k) \leq U_s. \end{array}$$

A typical non-linear optimization with non-linear constraints, the above problem is very difficult to solve online. Our best bet is to find a fast heuristic algorithm that gives satisfactory results. We present one such algorithm in the following section.

3.5.2 User-guided QoS adaptation

For each stream, the QoS adaptor needs to make two decisions: how much resource (CPU utilization) should be saved by the adaptation and how should the stream backup in its QoS to achieve the goal of resource saving. This problem is more complicated when we have multiple QoS dimensions. Consider the following motivating example:
Let us first assume a simple rate mapping function $f(q_1, q_2) = q_1 q_2$ for a specific stream. For an adaptation goal of decreasing the CPU utilization request of the stream by half, we have ${q'}_1{q'}_2 = \frac{1}{2}q_1q_2$ where $\{{q'}_1, {q'}_2)\}$ is the new QoS vector. There are infinite number of adaptation solutions as any point on the $xy = \frac{1}{2}q_1q_2$ surface is feasible. A desirable solution is one that also yields the highest utility $w_1u_1(q_1) + w_2u_2(q_2)$ according to Eq (1): yet another non-linear optimization problem. Needless to mention is that we do not even know how much resource to save.

Given the fact that it is hard for the adaptor to make the above decisions, we let the users decide! In particular, each stream is associated with a *STEPSIZE* value for each QoS dimension. For example, a *STEPSIZE* of 0.1 for q_2 means the user wishes to backup on its non-loss ratio by 0.1 in case of negative control ($U_d < 0$). The same factor can be used for advancing the QoS when a positive control signal arrives. The adaptation for a single stream works as follows: the stream registers with an initial QoS vector $\{q_{1,0}, q_{2,0}\}$ and *STEPSIZE*s S_1 and S_2. When adaptation is needed, it follows a fixed roadmap

$$\{q_{1,0}, q_{2,0}\} \rightarrow \{q_{1,0} - S_1, q_{2,0} - S_2\} \rightarrow \\ \{q_{1,0} - 2S_1, q_{2,0} - 2S_2\} \rightarrow \cdots$$

for negative control. We may also adjust QoS values geometrically using S_1 or S_2 to indicate the proportions to change. The intuition behind this strategy is that users always choose the QoS that is most valuable. When adaptation happens, he/she knows better than anybody about what to sacrifice and what to keep. Note the dimensional weight (w_i) for two QoS can be captured by the relative value of their *STEPSIZEs* such as

$$\frac{w_1}{w_2} = \left(\frac{S_2}{S_1}\right)^a, \quad (a > 0),$$

which means a more important QoS is to be degraded less than an unimportant one. When $a = 1$, we could use the following roadmap for positive controls:

$$\{q_{1,0}, q_{2,0}\} \rightarrow \{S_2 + q_{1,0}, S_1 + q_{2,0}\} \rightarrow \\ \{2S_2 + q_{1,0}, 2S_1 + q_{2,0}\} \rightarrow \cdots$$

To complete our heuristic algorithm, we borrow the idea of benefit-to-cost analysis from [20] and Aurora [6]. The central part of this strategy is to list the candidate streams in order of their current *cost efficiency* given as

$$E_i = \frac{h_i(f_i(\vec{q}_1) - f_i(\vec{q}_0))}{\mathcal{U}(\vec{q}_0) - \mathcal{U}(\vec{q}_1)} \tag{8}$$

where $\vec{q}_0$ is the current QoS vector and $\vec{q}_1$ the vector of the next stop in the adaptation roadmap. As we can see, E_i is basically the resource gain divided by the utility loss if we adapt stream i from $\vec{q}_0$ to $\vec{q}_1$. The algorithm works as follows: a handle to each stream is stored in a list sorted by E, the adaptor iteratively picks the stream with the largest E until the total resource gain satisfies the control signal U_d. For positive control, we pick up streams with the smallest E value. The time complexity of this algorithm is $O(N)$ where N is the number of streams, which is tolerable considering this is done once every control period.

4 Experiments

We evaluate the performance of our control-based QoS adaptation scheme by simulations. We implement the proposed control framework for DSMSs using the TCL scripting language. The architecture of the simulator follows our design shown in Section 3.3 with the addition of a load generator that feeds stream data into the plant.

4.1 Experimental setup

In our simulator, we define four classes of processing delays: 250ms, 500ms, 1s, and 2s. We set the target utilization U_s to 0.9 and M_s to 0. The monitoring cycle is 5 seconds if not otherwise specified. For the PID controller, the α and β values are set to 0.5 ans 0.15, respectively. We use a time window of width 4 for the integral part of the controller. The PID controllers are chosen based on stability conditions derived in [21]. We do not consider the overhead for context switch and operator scheduling as they are relatively small and we assume they are uniformly absorbed by the 0.1 free CPU utilization.

We test our QoS framework with both synthetic and real-world stream data. For the synthetic data, we generate streams with update arrival times following the exponential distribution and the *b*-model [22]. The latter produces traffic with very high levels of burstiness. The streams with the Poisson arrival pattern are divided into four classes with average interarrival time from one to four seconds. The skewness of *b*-model traffic is determined by a *bias factor*. We use a bias factor of 0.7, which means, within any time interval, 70% of the data points come in half of the time interval and 30% in the other half. The Internet Traffic Archive [4] provides Internet traffic trace extracted from real servers and we use their *LBL-PKT-4* trace for our experiments. The *LBL-PKT-4* data set includes 863,000 records representing one hour of TCP traffic. Each record contains a timestamp as well as the IPs and port numbers for both source and destination. We treat packets with the same source IP as a stream. The queries to the stream data are simulated in a similar way as in Babcock *et al* [14]. We consider three different types of queries as shown in Table 1. The *cost profile* shows the estimations of the per-tuple processing costs and selectivity of all operators in a query. For each stream, s_1 and s_2 are randomly (uniformly) chosen from the ranges of $[0, 1]$ and $[1.5, 2.2]$, respectively.

Table 1: Experimental queries.

Query Type	*Cost Profile*
Single-stream, no joins	$(0.5, s_1)(4, 0)$ or $(0.4, s_1)(2, s_1)(2.2, s_1)(4, 0)$
Single-stream, join with stored relation	$(0.4, s_1)(1.3, s_2)(1.5, s_1)(4, 0)$
Multi-stream slide-window join	$(4, s_1)(1.5, s_2)(1.3, 0)$

[4]http://ita.ee.lbl.gov/index.html

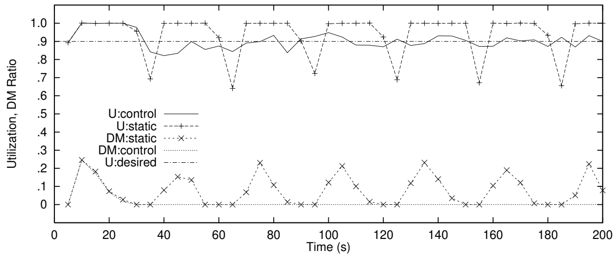

Figure 4: Utilization and deadline misses by different load shedding strategies

4.2 Traffic with Poisson arrival pattern

We compare the performance of the control-based QoS scheme with that of a *static load shedding* strategy. The latter also dynamically configures its shed factor according to a signal representing the load status in the plant. However, the change of its shed factor follows a predefined roadmap. We choose a method that updates the shed factor by increments of a *base factor* (0.1 in this experiment), similar to the approach in the Aurora project [6]. The rate control mechanism in TCP [23] is of the same flavor: a connection exponentially decreases its sending rate when congestion is detected. Figure 4 shows the results of both shedding methods using the same set of data streams. The streams are generated with a Poisson arrival pattern. The total load of the streams is about 1.4, simulating a constant overloading situation. The control-based strategy converges to a stable state at about 35 seconds and the errors afterwards are kept within $\pm 10\%$ of the desired value. On the contrary, the static shedding strategy shows a zigzag pattern in the achieved utilization with an error range significantly larger than that of the control-based method. As a result, its deadline miss ratio reaches about 20% periodically while there are almost no DMs in the control-based experiment. On the other hand, the deep valleys in its utilization curve also translate into unnecessarily discarded loads (data not shown).

One thing to point out is that we did not arbitrarily choose some parameters to make the static shedding case inferior in performance. Instead, we tested a number of base factors and presented the one with the best output here. We could imagine things can be even worse if we use the *multiplicative decrease-slow start* algorithm of TCP rate control. The control-based strategy requires no such tuning process: only 2 sets of parameters are tested and they give similar results.

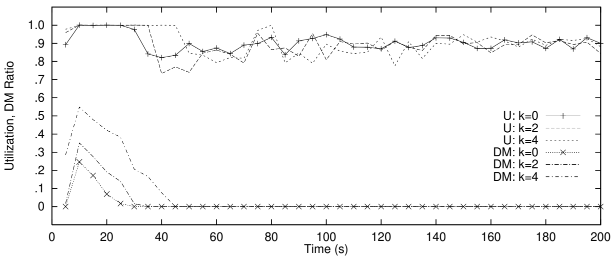

Figure 5: Performance of control-based approach under different precision of per-tuple cost estimation.

The same parameters are used for most of the experiments to be described in the following paragraphs. This shows the most important advantage of adapting control theory in practice: once the proper range of system parameters are found by mathematical analysis, very little effort is needed to tune the system at runtime.

4.3 Response to uncertainties

In the previous experiment, we assume the per-tuple cost estimation of the streams is 100% accurate. However, this may not be true in reality. Although the costs of individual operators are easy to estimate, their selectivity is not. We investigated the response of our adaptation scheme to such imprecise cost estimations (Figure 5). We introduce uncertainties to the costs by selecting the real costs of each query iteration randomly from a range around the profiled value. For example, the $k = 2$ treatment in Figure 5 chooses a cost value for each tuple from the range $[0.5h, 1.5h]$ while h is the profiled CPU cost. This range for the case of $k = 4$ is even wider ($[0.1h, 4.1h]$) and the profiled cost h is not even the mathematical expectation of all the possible values.

Our QoS adaptation strategy seems to handle the incorrect cost estimations well. Comparing to the original experiment where all costs are precisely profiled ($k = 0$), the system achieves (almost) the same level of stability. The only difference is that the time used to converge to the stable status is longer as the uncertainty about costs grows. In other words, the system 'learns' the real value of costs gradually. For all three cases, DM rates are zero for most of the time. In times before convergence, systems with higher uncertainty show more deadline misses.

It is believed that stream data flows cannot be correctly modeled by Poisson distributions [24, 22]. In order to see whether our system converges to stability under the disturbance of bursty data arrivals, we run experiments with the *b*-model data (Fig 6a) and real TCP trace (Fig 6b). The utilization levels recorded are obviously less smooth than those in Fig 4 as more undershoots can be observed. The undershoots become more serious as the traffic gets more bursty. The TCP trace shows higher fluctuations than the *b*-model data (Fig 6c). The interesting thing is that single *b*-model streams are more bursty than single TCP streams (data not plotted) but the combined *b*-model data has more regular arrival pattern. The control-based strategy performs well in controlling the overshoots in comparison with static shedding. This is further supported by the DM ratio measured: DM events are abundant in static shedding while very few are observed in control-based shedding. For both traffic, there are slightly more undershoots in the control-based experiments. In the TCP experiment, the base factor for the static shedding case is 0.15 because the number 0.1 used in Fig 4 gives incredibly high DM readings. On the other hand, the undershoots increase significantly if we choose a value larger than 0.15. Again, we see that the configuration of the static shedding system has to be done in an *ad hoc* way. Note the deep valleys (e.g. 320-350th second in *LBL-PKT-4*) in the utilization curve are partly caused by the sudden drop of requested load (Fig 6c).

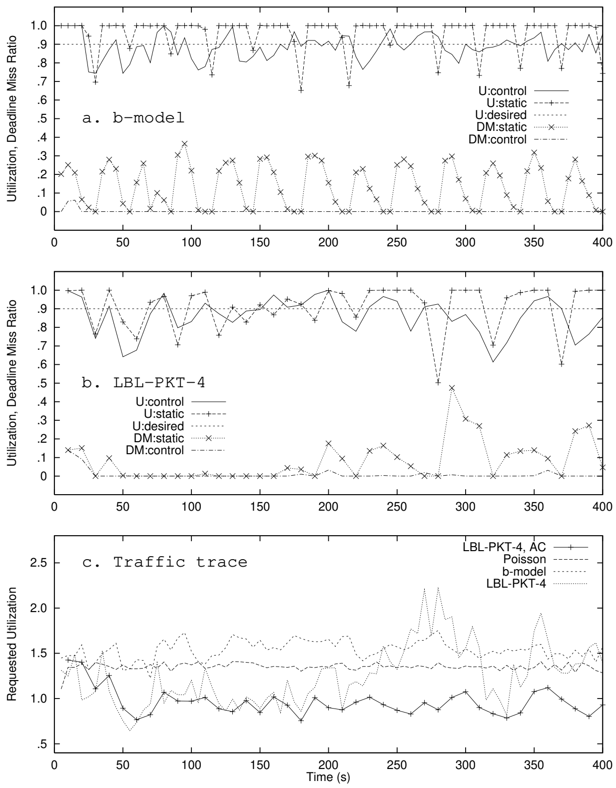

Figure 6: Performance under bursty traffic.

The overall result of this experiment is not surprising: errors are expected to be large when the system contention frequently changes because the control needs some time to converge to stable status. However, the control-based strategy still outperforms a static algorithm as the former uses more information to make decisions. In some sense, the static shedding DSMS

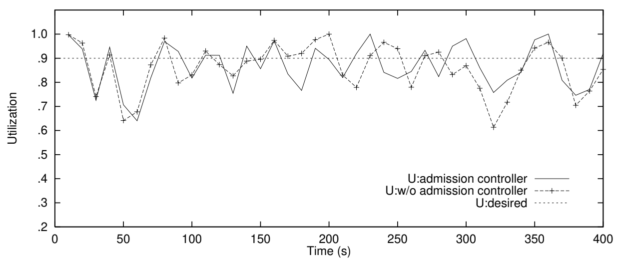

Figure 7: Performance improvement by admission control.

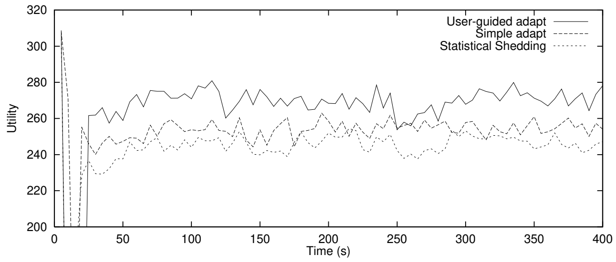

Figure 8: Utility gain by smart QoS adaptation vs. statistical shedding.

can also be viewed as a feedback control system: it calculates the output signal exclusively based on the sign of the current input signal. For the control-based DSMS, the output signal is calculated using the numerical value of a number of recent input signals. The admission controller helps in decreasing undershoots (Fig 7) suffered by the previous experiments (Fig 6). As expected, it does so by smoothing the aggregated traffic (the '*LBL-PKT-4, AC*' line in Fig 6c). The DM ratio is not affected by the use of an admission controller.

4.4 QoS adaptation beyond load shedding

In this section we introduce experiments performed to study the effects of our QoS adaptor. We compare the utility of the proposed adaptor with one that only uses statistical shedding to adapt. To verify the effects of the user-guided adaptation that follows user QoS preference, we also performed a third experiment ('simple adapt' in Fig 8) where streams adapt according to an arbitrary roadmap that is not related to the dimensional weights (w_i). The load we use in this experiment is the one with Poisson interarrival patterns. The experiment simulates the situation of constant overloading (requested utilization at about 1.4). The utility is calculated as the sum of utilities of all streams. For an individual stream, utility is calculated from the maintained QoS within the latest monitoring period using Eq (1). The single-dimension utility function, weight, and resource mapping functions are all generated randomly (parameters not shown due to limited space). The way to interpret the results is as follows: all three experiments start with a similar utility value at the 5th second (around 310), then the total value decreases rapidly as the control is in transition. When the control converges, so do the utility levels of all three experiments. The absolute utility value is less meaningful than the utility loss from the initial value. We may easily conclude that the statistical shedding lost much more utility, compared to our QoS adaptation strategy. The use of our user-guided adaptation roadmap is also essential: the utility loss of user-guided adaptation is significantly less than those of the simple adaptation strategy that does not consider user QoS preference.

5 Comparison to Related Work

Current work on DSMSs has concentrated on system architecture [1, 3], query processing [25, 26], query optimization [27], and stream monitoring [28]. Relatively less attention has been paid to the development of a unified framework to support application-specific QoS requirements.

Research that is most closely related to our work is the QoS-aware load shedding introduced in the context of the Aurora project [6]. In this paper, the authors raised three critical questions about load shedding: when, where, and how much to shed. The Aurora system model is different from ours: it involves a network of operators and each operator is a unit for resource scheduling. Aurora uses semantic shedding to choose victim tuples based on a cost/utility analysis of each operator. Loss ratio is the only QoS parameter considered. It uses a set of intuitive and simple rules to solve the other two problems. In comparison to Aurora, our work focuses on timely processing of tuples and also considers other QoS such as updating frequency. Our control-based adaptation significantly improves the answers to the first and the third questions about load shedding mentioned above. Shedding is not the only means to adjust the load in our framework. Instead, our system adapts to the desired state by changing the QoS levels of streams. The QoS change follows a user-friendly roadmap. In another paper [10] from the Aurora group, various operator scheduling algorithms with different optimization goals are presented. A scheduling algorithm that minimizes runtime memory consumption is given in [14] as part of the STREAM project. In [19], load shedding strategy that minimizes the loss of accuracy of aggregation queries in DSMS is discussed.

The use of feedback control is inspired by the work of Lu *et al* [29]. The major difference between our work and [29] is that we use feedback control to solve different problems (general real-time scheduling vs. QoS adaptation in DSMSs). As a result, there are also differences in the techniques we exploited. For example, they assume incoming loads change in a step-climbing way while we aim at more dramatic fluctuations of job arrivals. Therefore, a P controller is good for their purposes while we utilize the PID controller to massage the effects of the bursty traffic. In [30], feedback control is used to assist QoS adaptation of concurrent video tracking tasks.

Research on CPU scheduling in real-time databases is closely related to our efforts in guaranteeing QoS. Abott and Garcia-Molina [31] compared the performance of major scheduling algorithms under different load situations. Haritsa *et al.* [32] presented a modification to EDF that randomly decreases the priority of some of the jobs under high system contentions.

6 Conclusions and Future Research

We describe a QoS adaptation framework for DSMSs in this paper. Some of the features of stream data applications are analogous to those of multimedia. We argue that continuous queries in DSMSs are quality critical in terms of timeliness, reliability, and precision. Therefore, how to maintain quality in a multi-query environment is a vital problem in the design of a DSMS. In this paper, we regard processing delay as the most critical QoS in such systems. We propose a QoS adaptation framework that emphasizes maintaining low levels of deadline misses. The central part of our framework is a real-time CPU scheduler, a nice feature of which is that no deadline misses will occur if the load is kept under a certain level. One common practice for DSMSs to overcome excessive incoming requests is load shedding. We design a feedback control loop to dynamically adjust load according to status (signals). The beauty of a feedback control loop is that it automatically converges to stable status, even after a major disturbance is encountered. Comparing to previous work, the control-based approach solves the adaptation problem in a more systematic and precise way. Another contribution of this paper is our adaptation method that performs more than load shedding. We view the problem as an optimization of QoS reassignment to all streams. As the first paper that deals with multiple QoS parameters in adaptation in DSMS, our adaptation scheme concentrates on user preference of these parameters.

DSMS is an area with many open problems. The problem brought up in this paper itself needs more work. In the feedback control part, we used a set of parameters for the PID controller derived from known results of relevant work. The fact that these parameters work well in our experiments does not mean there is no room for improvement. We believe a more precise model of the system, especially the bursty traffic pattern, would definitely help our design of more effective PID controller. Our DSMS model comes with assumptions that could be relaxed in order to study a more general environment. One of these is the consideration of network delays. Another extension we can see is the QoS control under the consideration of more than one resources. The validity of a control-based approach under such complex systems would be interesting to investigate.

References

[1] D. Carney, U. Çetintemel, 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 *Procs. of the 28th VLDB Conf.*, pages 84–89, August 2002.

[2] Niagara Project, http://www.cs.wisc.edu/niagara/.

[3] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, March 2003.

[4] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, Wei Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of 1st CIDR Conference*, January 2003.

[5] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Proceedings of ACM SIGMOD '03*, pages 563–574, June 2003.

[6] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th VLDB Conference*, pages 309–320, August 2003.

[7] Klara Nahrstedt and Ralf Steinmetz. Resource Management in Networked Multimedia Systems. *IEEE Computer*, 28(5):52–63, 1995.

[8] D. Yau and S. Lam. Operating System Techniques for Distributed Multimedia. *International Journal of Intelligent Systems*, 13(12):1175–1200, December 1998.

[9] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 2001.

[10] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *Procs. of the 29th VLDB Conf.*, pages 838–849, August 2003.

[11] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resources Reservation and Application Adaptation. In *Proceedings of IWQOS*, pages 181–188, June 2000.

[12] A. Arasu, B. Babcock, S. Babu, M. Datar, J. Rosenstein, K. Ito, I. Nishizawa, and J. Widom. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Procs. of 1st CIDR Conf.*, January 2003.

[13] S. Shah, S. Dharmarajan, and K. Ramamritham. An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data. In *Procs. of 29th VLDB Conf.*, pages 57–68, August 2003.

[14] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems . In *Proceedings of ACM SIGMOD '03*, pages 253–264, June 2003.

[15] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating Probabilistic Queries over Imprecise Data. In *Proceedings of ACM SIGMOD '03*, pages 551–562, June 2003.

[16] G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital Control of Dynamic Systems*. Edison-Wesley, Massachusetts, 1990.

[17] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Communications of ACM*, 20(1):46–61, January 1973.

[18] Tarek Abdelzaher, Vivek Sharma, and Chenyang Lu. A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling. *IEEE Trans. on Computers*, 53(3):334–350, March 2004.

[19] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Procs. of ICDE Conf.*, 2004.

[20] Diane. L. Davison and Goetz Graefe. Dynamic Resource Brokering for Multi-User Query Execution. In *Proceedings of ACM SIGMOD '95*, pages 281–292, June 1995.

[21] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *IEEE Real-Time Systems Symposium*, December 2000.

[22] M. Zhang, T. Madhyastha, N.H. Chan, S. papadimitriou, and C. Faloutsos. Data Mining Meets Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic. In *Proceedings of the 18th ICDE Conference*, pages 507–516, Feburary 2002.

[23] V. Jacobson. Congestion Avoidance and Control. In *Procs. of ACM SIGCOMM*, pages 314–329, August 1988.

[24] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[25] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Procs. of SIGMOD Conf.*, pages 61–72, June 2002.

[26] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for Shared Window Joins Over Data Streams. In *Proceedings of 29th VLDB Conf.*, pages 297–308, August 2003.

[27] S. Viglas and J. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Procs. of SIGMOD Conf.*, pages 37–48, June 2002.

[28] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Procs. of the 28th VLDB Conf.*, pages 358–369, August 2002.

[29] C. Lu, J. Stankovic, G. Tao, and S. Han. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Journal of Real-Time Systems*, 23(1/2):85–126, September 2002.

[30] B. Li and K. Nahrstedt. A Control-Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, September 1999.

[31] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Trans. Database Systems*, 17(3):513–560, September 1992.

[32] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Procs. IEEE Real-Time Systems Symposium*, pages 232–242, 1991.