Load Shedding in Stream Databases: A Control-Based Approach

Yi-Cheng Tu† Song Liu‡ Sunil Prabhakar† Bin Yao‡

†Department of Computer Sciences
Purdue University
250 N. University St.
West Lafayette, Indiana, USA
{tuyc, sunil}@cs.purdue.edu

‡School of Mechanical Engineering
Purdue University
140 S. Intramural Drive
West Lafayette, Indiana, USA
{liu1, byao}@purdue.edu

ABSTRACT

In Data Stream Management Systems (DSMSs), query processing has to meet various Quality-of-Service (QoS) requirements. In many data stream applications, processing *delay* is the most critical quality requirement since the value of query results decreases dramatically over time. The ability to remain within a desired level of delay is significantly hampered under situations of overloading, which are common in data stream systems. When overloaded, DSMSs employ load shedding in order to meet quality requirements and keep pace with the high rate of data arrivals. Data stream applications are extremely dynamic due to bursty data arrivals and time-varying data processing costs. Current approaches ignore system status information in decision-making and consequently are unable to achieve desired control of quality under dynamic load. In this paper, we present a quality management framework that leverages well studied feedback control techniques. We discuss the design and implementation of such a framework in a real DSMS - the Borealis stream manager. Our data management framework is built on the advantages of system identification and rigorous controller analysis. Experimental results show that our solution achieves significantly fewer QoS (delay) violations with the same or lower level of data loss, as compared to current strategies utilized in DSMSs. It is also robust and bears negligible computational overhead.

1. INTRODUCTION

Applications related to processing of data streams have attracted a great deal of attention from the database community. With great social/economical interests, these applications flourish in a number of fields such as environment monitoring, system diagnosis, financial analysis, and mobile services. Unlike traditional data that are mostly static, stream data are produced continuously (e.g., from a sensor network) and are generally too large to be kept in storage after being processed. Furthermore, most queries against stream data are persistent queries that continuously output results as they are produced. In recent years, a number of Data Stream Management Systems (DSMSs) have been developed [3, 1, 12, 8].

Query processing in DSMSs has to meet various quality[1] requirements [3]. Similar to those in other real-time applications, quality in DSMSs describes the timeliness, reliability, and precision in data processing. Important quality parameters in DSMSs include: processing delay, data loss ratio, sampling rate, etc. A salient feature of DSMSs is the real-time constraints associated with query processing. In many applications of DSMS, query results are required to be delivered before either a firm (e.g., tracking of stock prices) or soft (e.g., network monitoring for intrusion detection) deadline. Therefore, processing delay is the most critical quality parameter in these applications. On the other hand, users may accept query processing at different levels of accuracy as a result of lost or incomplete data [18, 21]. This provides us with optimization opportunities to trade those quality parameters (e.g., loss ratio, sampling rate) that are less important for shorter delays in case of congestion.

It is difficult to provide delay guarantees in a DSMS due to physical resource limitations and the unpredictable pattern of resource usage by streams. In practice, a DSMS could easily accommodate hundreds or even thousands of streams. Delay requirements may be violated even with careful query optimization and admission control, which are the first line of defense against overloading and generally based on static estimations of each stream's resource consumption. The runtime fluctuations of application resource usage (e.g., bursty arrivals) may cause temporary congestion that interferes with real-time data processing. Under this situation, we need to dynamically adjust application behavior by reducing its non-critical quality parameters. For example, we can increase data loss rate by *load shedding* [21] or reduce the window size for windowed operations [4]. We call such adjustment of application parameters *adaptation*. Streaming data are intrinsically dynamic in terms of their bursty arrival patterns and ever-changing tuple processing costs [25, 19]. Thus, an adaptation architecture should promptly detect the change of quality by continuously monitoring the system and determine whether adaptation should be performed.

[1]In this paper , the words 'QoS' and 'quality' are used interchangeably.

VLDB '06, September 12-15, 2006, Seoul, Korea.

While maintaining processing delays under an appropriate level, degradation of other quality should also be controlled. For example, we can always achieve low delays by constantly discarding most of the load. However, query accuracy decreases unnecessarily due to excessive load shedding. It would be desirable to achieve low delays while minimizing data loss. Attempting to solve this problem, current DSMSs employ simple and intuitive strategies to make important adaptation decisions such as the time and magnitude of load shedding. For example, the following load shedding algorithm is used (explicitly) in Aurora [21] and (implicitly) in STREAM [6].

```
for every T time units
    if measured load L is greater than CPU capacity L0
        do shedding load with amount L − L0
    else allow L0 − L more load to be admitted
```

Figure 1: Load shedding algorithm in Aurora

The idea behind this algorithm is: QoS degrades when the load injected into the system is higher than its processing capacity. To maintain QoS under overloading, we only need to make the input load smaller than capacity L_0. However, in a dynamic environment where the input rate keeps changing, this approach may either make the DSMS unstable (i.e., QoS deviates unboundedly from the desirable value) or overreact by discarding too much load. In Section 4.3.1, we elaborate on this issue.

To remedy the above problems in a systematic way, however, is not trivial. Firstly, we need to understand the nature of the DSMS's response to changes of inputs. Specifically, a quantitative model that describes how adaptation of stream behavior affects quality (delay) is needed. Secondly, our adaptation algorithm should be robust, meaning that its performance should not be affected by patterns of load fluctuations and cost variations. Another challenge is the design of the monitoring process: it should be light-weight and still able to effectively capture changes of status.

In this paper, we present our approach to address the above challenges. Our solution takes advantage of proven techniques from the field of control theory. Feedback control is extensively utilized in the fields of mechanical, chemical, and aeronautical engineering to deal with systems that are subject to unpredictable dynamics [11]. In this work, we view quality-driven load shedding in DSMS as a feedback control problem and solve it with a controller designed from a dynamic DSMS model we develop. Specifically, this paper makes the following contributions:

1. We develop a dynamic model to describe the relationship between average tuple delays and input rate of a DSMS. From this model, we propose the idea of controlling the somewhat unmeasurable delay signal by manipulating the number of outstanding data items;

2. We design a controller to make load shedding decisions via rigorous analysis of the system model. By exploiting results from control theory, our design achieves guaranteed system performance;

3. We implement and evaluate our load shedding framework on a real DSMS. By working on a real system, we achieve better understanding of the DSMS model and obtain more convincing results supporting the validity of our approach; and

4. We identify several problems that are unique in the control of DSMS load shedding and propose system-specific strategies to solve these problems.

The rest of this paper is organized as follows: we compare our work with related research efforts in Section 2. Section 3 describes the basic DSMS model and problem formulation. Details of our feedback control framework are presented in Section 4. We show experimental results in 5 and conclude the paper in Section 6.

2. COMPARISON TO RELATED WORK

Current efforts on DSMSs have addressed system architecture [7, 12], query processing [10], query optimization [23], and operator scheduling [5]. Relatively less attention has been paid to the development of a unified framework to support QoS.

Load shedding has been extensively utilized to deal with overloading in DSMSs [21, 6]. Ref. [6] discusses load shedding strategies that minimize the loss of accuracy of aggregation queries. In the LoadStar system [9], statistical models are utilized to maximize the quality of stream mining results when load shedding has to be performed. Earlier work on QoS-driven load shedding in the context of the Aurora [21] DSMS (now evolving to the Borealis project [2]) is closely related to our study in this paper. In [21], three critical questions about load shedding are raised: *when*, *where*, and *how much* to shed. To answer these questions, Aurora checks system load periodically and triggers shedding when excessive load is detected. A precomputed Load Shedding Roadmap (LSRM) that holds possible shedding plans is used to determine where to shed load. Given the amount of total load to shed, the LSRM finds the best plan to accomplish this such that system utility loss (due to lost data) is minimized.

The Aurora/Borealis work focuses more on the question 'where to shed load' (i.e., construction of LSRM) than the questions of 'when' and 'how much'. As shown in Fig. 1, it uses a heuristic to determine the amount of load shedding and handles processing delays implicitly. The system does not provide information about how the monitoring period T is set. In this paper, we concentrate on the control of delay QoS under heavy fluctuations/bursts and time-varying processing costs of data inputs, which are common in data stream applications. For this purpose, we need to find a solution that is different from the Aurora load shedder shown in Fig. 1. In other words, our work aims to provide better answers to the questions of *when* and *how much* to shed load. Our solution can also be used to guide quality adaptation mechanisms other than load shedding.

Another strategy to deal with system overloading is to downgrade the accuracy in query processing. In [20], data tuples are routed to a *data triage* when long processing delays are observed.

Control-theoretic approaches have been used to solve various problems in the areas of networking [15], real-time systems [17], and multimedia [16]. Our work differs significantly from these efforts: first, we address different problems with a different system to be controlled. For example, [17] focuses on deadline misses in a real-time resource scheduler and [16] discusses precision in a video tracking application. Control theory is basically a collection of many mathemat-

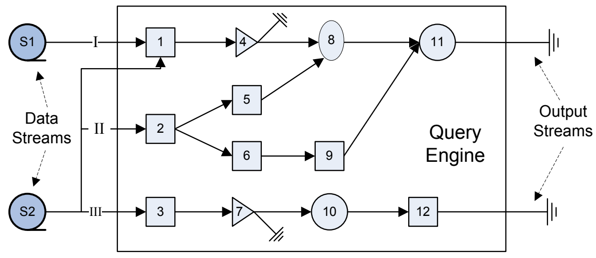

Figure 2: A general system model of DSMS.

ical tools for analyzing system features and designing controllers towards guaranteed performance. Therefore, application of control theory to different problems and systems are not straightforward as the choice of appropriate control techniques are essential. Controller design and performance depend heavily on the dynamic model of the system of interest. Derivation of such models is generally non-trivial and involves various techniques for different systems. In a word, the results in above studies give little insights on how our problem can be solved as distinct systems and control targets are involved. Second, we raise several DSMS-specific issues in this paper (Section 4.5) and provide solutions to these problems. From a control theory viewpoint, these issues bring new challenges that are never before addressed in traditional control applications.

We have presented the idea of control-based load shedding in a short paper [22] where we found that even a crude controller outperforms the *ad hoc* method. However, we only validate our idea with a simulator in [22] therefore the real challenges of QoS control in real-world systems (i.e., contributions listed in Section 1) are not addressed.

3. DSMS MODEL, PROBLEM DESCRIPTION, AND NOTATIONS

In this paper, we study load shedding under a push-based DSMS model, which is a generalization of those of the STREAM [4] and Aurora [3] stream managers. In this model, each query plan consists of a number of *operators* connected to form a branched (e.g., I and III in Fig. 2) or unbranched (e.g., II in Fig. 2) *execution path*. Multiple queries form a network of operators so that they can share computations. Multi-stream joins are performed over a *sliding window* whose size is specified by the application either in number of tuples or time. Data from a stream can enter any number of entry points in the query network. Each operator has its own queue to hold unprocessed inputs and a *scheduler* determines the processing order of operators at runtime.

With respect to a data tuple, *processing delay* is defined as the time elapsed since it arrives at the network buffer of the query engine till it leaves the query network[2]. For example, data from stream source $S1$ in Fig. 2 departs either after being discarded by the filter operator 4 or entering an output stream after operator 11. For data that could enter multiple execution paths, we can choose the longest path to record its departure time (e.g., 2-6-9-11 or 3-7-10-12 for $S2$ data in Fig. 2). Processing delay consists of CPU time spent to execute the operators and time spent in queues[3]. We target a system where data tuples arrive in a dynamic pattern such that future data rates are unpredictable. Furthermore, the expectation of per-tuple CPU cost changes over time. Variations in CPU cost arise from changes in factors such as query network structure (due to addition/deletion of queries), and operator selectivity [21]. In this paper, we assume such variations happen less frequently than the fluctuations of data arrival rates. We believe this is a reasonable assumption as none of the above factors would change abruptly.

[2]Here we ignore network delays. This can be justified by the use of networks where transmission delays are either effectively controlled or significantly smaller than our control target.

Table 1: Notations and symbols.

Symbol	Definition	z-domain
k	discrete time index	-
T	control period	-
y_d	target value for delays	-
H	CPU power for query processing	-
y	processing delay	$Y(z)$
f_{in}	data input rate	$F_{in}(z)$
f_{out}	data output rate	$F_{out}(z)$
u	controller output	$U(z)$
v	desired data input rate	-
c	per-tuple processing cost	-
q	number of outstanding tuples	$Q(z)$
$C(z)$	controller transfer function	$C(z)$
$G(z)$	system (DSMS) transfer function	$G(z)$

Load shedding is performed to reverse the increase of processing delays due to overloading. We allow the system administrator to specify a target delay time y_d. The goal is to maintain the average processing delay of data tuples that arrive within a small time window T (we will discuss more about the choice of T later) to be under y_d. We accomplish this by dynamically dropping load from the system in case of overloading. The problem is how to derive the right time and amount of load shedding such that data loss is as low as possible. The selection of shedding locations is not a focal point of this study. However, our framework is designed to work with current strategies that construct shedding plans such as the current load shedder in Borealis.

Symbols used throughout this paper are listed in Table 1.

4. FEEDBACK CONTROL-BASED LOAD SHEDDING FRAMEWORK

4.1 Overview

The term *control* generally refers to the operations to manipulate particular feature(s) (i.e., output signal) of a *process* by adjusting inputs into the process. The main components of a feedback control system form a *feedback control loop*, as shown in Fig. 3. The runtime operations of this loop are performed as follows: a *monitor* measures the output signal of the *plant*, which is the system to be controlled; The measurements are sent to a *controller*,which compares the output signal with a *target* value and maps the difference between them (i.e., *control error*) to a *control signal*; An *actuator* adjusts the behavior of the plant according to the control signal. The goal of the control operations is to

[3]This implies that CPU power is the bottleneck, which is a reasonable assumption [21]. We understand that limited memory could result in blocking of data processing. However, this should have little effect on our problem because our goal is to control overloading so that the system runs in a zone without such nonlinearities.

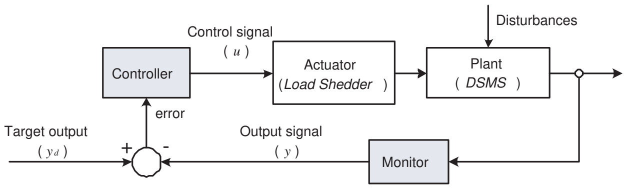

Figure 3: The feedback control loop.

overcome the effects of system and environmental uncertainties named *disturbances*. Readers interested in more details on control theory can refer to [11].

The above general model can be translated into a concrete model that serves as a blueprint for our load shedding framework. We still use Fig. 3 to illustrate this. Note that the shaded boxes represent new components that are not found in existing DSMSs. The plant to be controlled is the query engine of the DSMS and the actuator is the existing load shedding algorithm that adjusts load injected into the plant. In addition, we have a monitor that measures the output signal and a controller to generate the control signal. The unpredictable arrival patterns and processing costs are all treated as disturbances. In this loop, the output signal is the processing delay of tuples, denoted as y and the control signal (i.e., controller output and system input) is the desirable incoming data rate u.

We can easily see that the most critical part of the control loop is the controller, which determines the quantity of the input signal (to DSMS) based on the control error. The beauty of control theory is that it provides a series of mathematical tools to design and tune the controller in order to obtain guaranteed performance under disturbances. In the following, we discuss the design of our feedback control, which consists of two phases: system modeling (Section 4.2) and controller design (Section 4.4). We use the open-source Borealis data stream manager [2], whose query engine is derived from Aurora [3], as our experimental system.

4.2 System Modeling

An accurate mathematical model of the plant is of great importance to control system design. In this study, the model we are interested in is one that describes the relationship between the delay time y and the incoming data flow rate f_{in}. Due to the complexity of the controlled system, we may not be able to derive a model solely based on rigorous analysis. In this case, we can use *system identification* techniques to study system dynamics experimentally.

First of all, the expectation of per-tuple processing cost c can be precisely estimated in the current Borealis system. Readers can refer to Section 4.2 of [21] for details. For now, we assume c is a constant. We will discuss the effects of this assumption in Section 4.4.

The current version of Borealis uses a *round robin* policy to schedule operators and place intermediate results in waiting queues of individual operators. These queues extract input in a first-in-first-out (FIFO) manner therefore we see no priorities assigned to tuples as a result. Let us first consider an ideal situation: all tuples in the network share the same query paths and the system has the same inflow and outflow rates. If there are q outstanding data tuples in the query network when a tuple A enters, the total processing delay of A is

$$y = (q+1)c \tag{1}$$

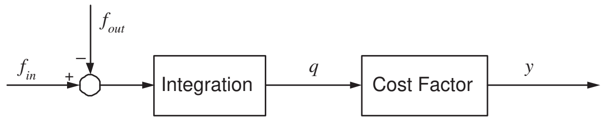

Figure 4: Database model structure.

The reason for this is: when A sits in the queue of any operator, it will not be processed until all of the q tuples are cleared from that queue. If the execution path of A consists of n operators, a total number of $nq+n$ operators would have been executed by the system when A is finished. The cost of the n operators in a path is c therefore the total cost becomes $(q+1)c$. Among the total time of $qc+c$, qc is time spent in waiting queues and c is the processing time of A itself. In other words, it is equivalent to processing tuples as a whole (rather than by operators) in the order they arrive. The outstanding tuples can be regarded as entries in a virtual FIFO queue with length q.

In practice, we cannot use Eq.(1) to model delay time of individual tuples because the execution paths for different tuples are different. For example, if a tuple is discarded by a *selection* operator in the early part of its possible path, it has a shorter delay as compared to one that passes the selection box. Fortunately, instead of delay time y of single tuples, we are interested in the average delay time of a series of tuples arriving in a period of time[4]. Let us denote the length of this period, which is called *control period* or *sampling period*, as T and the average delay of tuples within the kth period as $y(k)$. We propose the following generalization of Eq.(1):

$$y(k) = \frac{c}{H} q(k-1) \tag{2}$$

where $q(k-1)$ is the virtual queue length at period $k-1$ and H (named *headroom factor*) is the fraction of processing power used for query processing. We always have $H < 1$ as resources must be consumed for running the operating system and other maintenance tasks. Note that we ignored the processing time in Eq.(2) because it is a constant and is much smaller than the waiting time (as well as the control target) in practice. The intuition behind Eq.(2) is: we can study data tuples with execution paths of the same length in a group. The same reasoning to generate Eq.(1) holds true for each such group: a tuple A will not leave the network until all other tuples in its group (that entered the network before A) are processed. Taking a weighted average of all such groups, each of which can be described by Eq.(1), we get a form that is close to Eq.(2). This leads to a system model for Borealis as shown in Fig. 4: the incoming data flow f_{in} less the data processing rate f_{out} is accumulated in the virtual queue. Therefore the queue length before period k, $q(k-1)$, is equal to the integration of $f_{in} - f_{out}$ at all times before the k-th period. Eq.(2) becomes:

$$y(k) = \frac{c \cdot T}{H} \sum_{i<k} [f_{in}(i) - f_{out}(i)].$$

Model verification. As Eq.(1) is an intuitive result, we need to verify it by experiments. The verification is done

[4]To guarantee delays for individual tuples, real-time schedulers [14] are generally deployed. Interestingly, in our system, if we can guarantee average delays, those for individual tuples can also be well maintained as the round robin policy is a 'fair' policy.

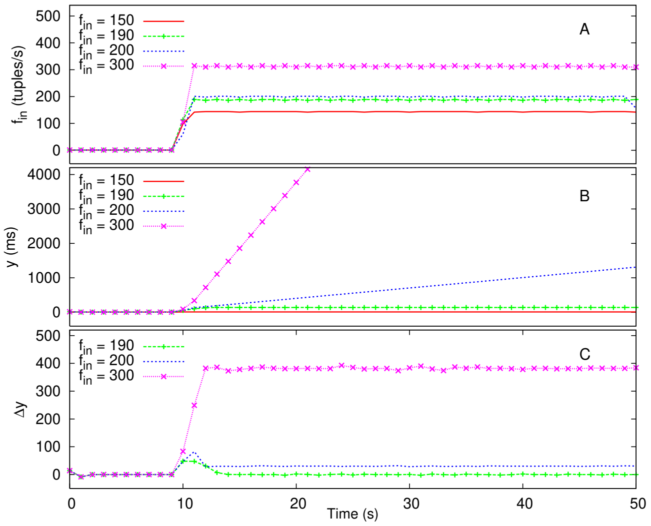

Figure 5: System responses to step inputs.

experimentally in accordance of system identification techniques. We feed the Borealis system with synthetic data streams having various arrival patterns and record responses in terms of delay time y. To set the cost factor c to a constant, we construct a Borealis query network with a number of (14 in this case, details omitted due to space limitations) operators, each of which has a fixed CPU cost. Then we generate stream data whose values follow uniform distributions to fix the selectivity of all filtering operators. By doing these, the average CPU cost of the query network becomes stable. In Fig. 5, we report system responses to a stream whose arrival rates follow a step function of time (i.e., rate starts at very low level and jumps to a high value at the 10-th second, as shown in Fig. 5A). It is shown in Fig. 5B that, when f_{in} is less than 190 tuples per second, all data can be processed immediately and a constant processing delay is observed. This implies that the per-tuple CPU cost is approximately $1000.0/190 = 5.26ms$ as 190 can be viewed as the threshold load that equals the CPU processing capacity (i.e. $f_{in} = f_{out} = 190$/sec assuming $H = 1$). On the other hand, when f_{in} exceeds 190/sec, i.e., more data entering the system than the CPU can handle, data accumulates in the virtual queue and delay y keeps increasing. This confirms the existence of the integration part in the proposed model. Fig. 5C shows the changing rate of y (calculated by $\Delta y = y(k) - y(k-1)$). The fact that Δy converges quickly to a stable value means that there is either no other dynamics or unknown dynamics with insignificant effects in the proposed model.

To further verify the model and determine the model parameter, we compare the real $y(k)$ values and the calculated $y(k)$ values based on our system model in Eq.(2). We collect $q(k)$ values at runtime to calculate $y(k)$. The results of experiments using the same step inputs as before are plotted in Fig. 6. According to Fig. 6A, the values given by our model fits the real data very well for all three choices of H. However, if we magnify the difference between calculated and real values (Fig. 6B), we can see that, with $H = 0.97$, modeling errors are far less than those with the other two values of H.

We also test the system with sinusoidal inputs and similar

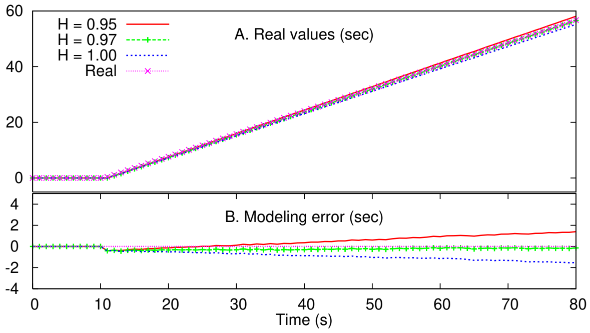

Figure 6: Model verification with step inputs.

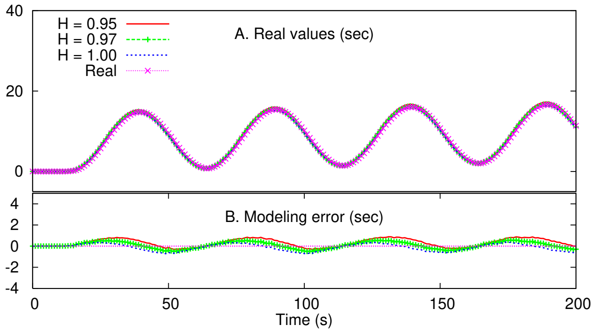

Figure 7: Model verification with sinusoidal inputs.

results are obtained (Fig. 7). In this set of experiments, the incoming data rate f_{in} changes sinusoidally within the range of $[0, 400]$. Small, periodical modeling errors can be seen. This means that there are probably unknown dynamics that our model fails to capture. This is not surprising due to the complexity of the Borealis system. As we shall see later, feedback controllers, if properly designed, have the power to reduce the effects of modeling errors, especially those that impose small errors such as the one we observe here.

Model transform. For the convenience of control analysis, we transform Eq.(2) to a model in the z-domain[5]:

$$Y(z) = \frac{c}{H}Q(z) = \frac{c \cdot T}{H(z-1)}\left[F_{in}(z) - F_{out}(z)\right] \quad (3)$$

where $Y(z)$, $Q(z)$, $F_{in}(z)$ and $F_{out}(z)$ are z-transforms of signals $y(k)$, $q(k)$, $f_{in}(k)$ and $f_{out}(k)$, respectively. Thus, the transfer function of the (Borealis) system in Fig. 4 is:

$$G(z) = \frac{c \cdot T}{H(z-1)}. \quad (4)$$

4.3 Why feedback control?

Before going into the design of controller, we identify some of the problems of non-feedback-control strategies and briefly discuss the basic ideas of feedback control theory.

4.3.1 Problems of current load shedding solution

The existing load shedding strategy (Fig. 1) suffers from poor performance as detailed in the following examples.

[5]The z-transform is a mathematical tool that transforms *difference equations* to *algebraic equations* [11], similar to the Laplace transform used for *differential equations*. From now on, all control-related analysis in this paper will be performed in the z-domain.

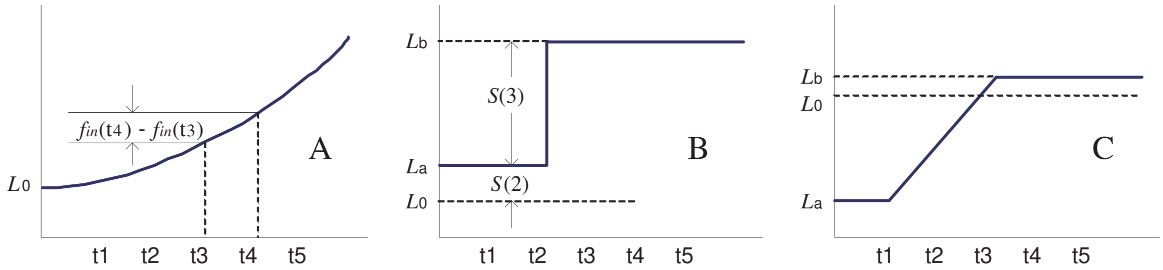

Figure 8: Different cases in which open-loop control has poor performance.

Example 1. *Instability when incoming data rate increases monotonically.* During certain period of time, the incoming data rate may keep increasing as shown in Fig. 8A. This is very typical in data stream systems. In this case, we are not shedding enough load: at any time t_i, we decide to shed load with amount $L_{t_i} - L_0$ in the next period yet the incoming load in the next period $L_{t_{i+1}}$ is greater than L_{t_i}. As a result, more and more tuples are accumulated in the DSMS and system output y may increase unboundedly.

Example 2. *Convergence to wrong value in response to step changes.* As illustrated in Fig. 8B, when the incoming data rate undergoes a step change from L_a to a much larger value L_b, queue length will increase by $L_b - L_a$. If the incoming rate stays on L_b, no further increase of queue length will occur and system output $y(k)$ stabilizes. However, $y(k)$ could converge to a value that is higher than the target value y_d. And the system is unable to self-correct the deviation.

Example 3. *Unnecessary data loss.* When the incoming data rate changes from a stable small value L_a to a higher value L_b that is slightly greater than L_0, the algorithm will discard data with amount of $L_b - L_0$ (Fig. 8C). However, more data should be allowed to enter the DSMS because there are no outstanding tuples before the change. In this case, although the delay time $y(k)$ is smaller or better than the target value y_d, the extra data loss is unnecessary.

The above three cases do not occur just in the Aurora method. Any method that does not use system output in decision-making could face the same or similar problems.

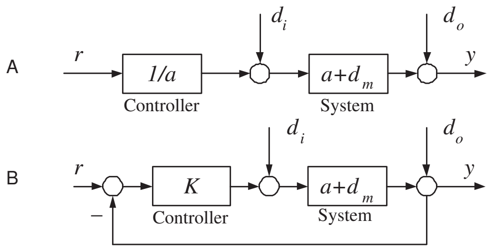

Figure 9: Block diagrams of generic open-loop (A) and closed-loop (B) control systems.

4.3.2 Open-loop vs. closed-loop

The unique feature of feedback control is that the output signal is used (as feedback) in generating the control signal. As there exists a complete loop (Fig. 9B) in the system block diagram, feedback control is also called *closed-loop* control. In contrast, strategies such as the one shown in Fig. 1 are *open-loop* control: system output or state information is not used in the controller, therefore it forms an open loop from the reference value to the system output (Fig. 9A). Here r is the reference input or desired system output, y is the actual system output, a is the system model, d_m, d_i and d_o represent modeling error, input disturbance and output disturbance. Relating this to our problem, the fluctuations of data arrival rates are modeled as input disturbances and the variable processing costs (c) as modeling errors.

In an ideal case, when there are no model uncertainties (i.e., $d_m = 0$), no input or output disturbances (i.e., $d_i = d_o = 0$), the best open-loop controller would be $1/a$ given the nominal system model a. This is because we have $y = r\frac{1}{a}a = r$, hence the output signal is exactly the reference value. However, in the real world, there are always modeling errors and input/output disturbances, therefore the open-loop system output y is:

$$y = \left(\frac{r}{a} + d_i\right)(a + d_m) + d_o = r + r\frac{1}{a}d_m + (a + d_m)d_i + d_o \quad (5)$$

From (5), it is obvious that the open-loop system output is subject to modeling error d_m, input disturbance d_i and output disturbance d_o, and there is no way to reduce their effects. On the other hand, in a closed-loop system where the feedback controller K is also designed based on the nominal system model a, we have

$$[(r - y)K + d_i](a + d_m) + d_0 = y,$$

and the system output y becomes

$$y = \frac{K(a + d_m)}{1 + K(a + d_m)}r + \frac{(a + d_m)}{1 + K(a + d_m)}d_i + \frac{1}{1 + K(a + d_m)}d_o$$

If the controller K is chosen large enough, i.e. $K >> 1$ and $K(a + d_m) >> 1$, the closed-loop system output y is approximately:

$$y \approx r + \frac{1}{K}d_i + \frac{1}{K}d_o \quad (6)$$

Clearly, the effects of modeling error d_m, input and output disturbances d_i and d_o can be reduced by a factor of $1/K$.

The above simple examples show why closed-loop control is better than open-loop control. In summary, the main advantage of closed-loop control over open-loop control is the reduction of the effects of modeling error, input and output disturbances therefore it can be exploited to handle dynamics in our load shedding problem.

4.4 Controller design

In Section 4.3.1, we demonstrated the shortcomings of open-loop solutions. However, even with the system model, it is still not clear how to use system output to make control decisions such that problems like the one in Example 1 do

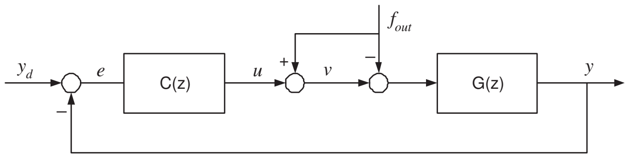

Figure 10: Control system block diagram.

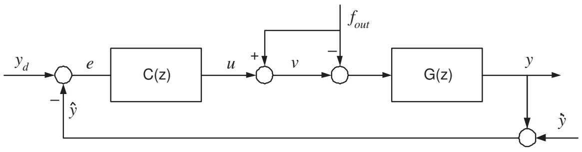

Figure 11: Control system with estimated feedback.

not occur. In this section, we discuss how to design controllers with guaranteed performance. We start this section by introducing our basic design of controller and continue with Section 4.5 to address some DSMS-specific challenges. The basic control scheme is illustrated in Fig. 10, where y_d is the preset reference value for delay time, $e = y_d - y$ is the error signal, and u represents the controller output (with the same unit as inflow rate f_{in}). The meaning of u is: the *increase* of the number of outstanding tuples (i.e., size of the virtual queue) allowed in the next control period. Therefore, we denote $v = u + f_{out}$ as the desired data flow rate to the database as f_{out} tuples will leave the queue.

4.4.1 Design based on pole placement

For a dynamic system, continuous or discrete, one can use system poles to determine its dynamic characteristics. System poles are the roots of the denominator polynomial of the transfer function and zeros are the roots of the numerator polynomial. The location of the system poles can tell how fast the system responds to an input and how well the response would be [11].

Pole placement design, one of the most important controller design techniques, is to add additional poles and/or zeros into the closed-loop system so that the closed-loop system may have desired performance. If a raw system $G(z) = \frac{B(z)}{A(z)}$ has poles as the roots of $A(z) = 0$, the closed-loop system, after adding a feedback controller $C(z) = \frac{N(z)}{D(z)}$, has a closed-loop transfer function $\frac{C(z)G(z)}{1+C(z)G(z)} = \frac{N(z)B(z)}{D(z)A(z)+N(z)B(z)}$. Hence the closed-loop system has poles as the roots of $D(z)A(z)+N(z)B(z) = 0$. Clearly, the system poles have been moved from $A(z) = 0$ in the raw system to $D(z)A(z)+N(z)B(z) = 0$ in the closed-loop system. System performance can be significantly improved by correct selection of $C(z) = \frac{N(z)}{D(z)}$.

The closed-loop performance is evaluated by the speed and smoothness, or *convergence rate* and *damping*, of system's response to disturbances. The closer the system poles are to 0, the faster the system response. Although it is theoretically possible to set the closed-loop poles at 0 and make the system respond very fast, it is practically not a good idea due to the large control authority needed for fast response. In our case, it means that if we want the system respond too fast, we may sometimes have to shed too much data. System damping is another important metric to evaluate closed-loop performance. Smaller damping means more severe oscillation, which is not desirable. When damping is less than 0.7, there exist visible oscillations in the system step response; when damping is bigger than 1, there is no oscillation but the system response becomes slow. Usually we choose the damping between 0.7 and 1.

With the above considerations, we develop the following feedback controller. The detailed design procedure can be found in Appendix A.

$$u(k) = \frac{H}{cT}[b_0 e(k) + b_1 e(k-1)] - a u(k-1) \qquad (7)$$

where a, b_0, and b_1 are controller parameters that can be obtained by solving Eq.(15) and Eq.(16) in Appendix A.

Handling time-varying characteristics of the plant. As time goes by, the average processing cost also changes. Let us denote the per-tuple cost at period k as $c(k)$. As our current controller is designed assuming constant c, we introduce modeling errors to the closed-loop by allowing c to change over time. As mentioned in Section 3, we assume the value of $c(k)$ changes slowly over time, at least compared to moving of data arrival rates. Under this situation, we normally believe the system is still stable with the existing basic controller. Due to its closed-loop nature, the controller should be able to compensate for the effects of such dynamics. Our experimental results (Section 5.1) provides strong evidence favoring this claim. We leave a systematic solution to handle fast-changing c as future work.

4.5 DSMS-specific issues

Whereas the Borealis system model seems to have a simple dynamic structure (Eq.(3)), the control of the system is far from trivial. In addition to the basic controller design, we also have to address the following practical issues.

4.5.1 Unavailability of real-time output measurement.

Accurate measurements of system output in real-time is essential in control system design. Unfortunately, this requirement is not met in our system because the output signal is the delay time. The output measurement is not only delayed, but also delayed by an unknown amount (the amount is the output itself!). To be more specific, the output signal of our controller should be the delay of tuples that have just entered the system when we calculate $u(k)$. However, at time k, we can only measure the delay of those that entered the system some time ago. This is a very interesting challenge to control theory as it does not exist in conventional control systems where the controlled signal can always be measured when we need it.

Given the output signal is not measurable when it is needed, can we derive it from the current system status? The answer is 'yes' and it comes right from the system model. We can easily modify the Borealis system to accurately record the number of outstanding data tuples (virtual queue length) $q(k-1)$. This can be done by just counting all the inflow/outflow tuples. We already know that at any time, $c(k)$ values can be accurately estimated. Therefore, instead of using a measurement of delay y as the feedback signal, we use an estimation of y that is derived from Eq.(2):

$$\hat{y}(k) = q(k-1)\frac{c(k)}{H}. \qquad (8)$$

Naturally, Eq.(8) adds estimation errors to the closed-loop.

We denote the estimation error as $\tilde{y} = y - \hat{y}$. Fortunately, our controller is still found to be robust by the following argument. When estimated output $\hat{y}$ is used as feedback signal, the original control system becomes the one shown in Fig. 11. The output of the closed loop system is hence described by:

$$Y(z) = \frac{C(z)G(z)}{1 + C(z)G(z)} Y_d(z) - \frac{C(z)G(z)}{1 + C(z)G(z)} \tilde{Y}(z) \quad (9)$$

The closed-loop system is still stable as long as $\tilde{y}$ is bounded, which is always true. The Y_d term in Eq.(9) shows that the output of the closed-loop system still tracks the target signal with desired damping and convergence rate. However, the accuracy is compromised due to the introduction of estimation errors (the $\tilde{Y}$ term in Eq.(9)).

4.5.2 Load shedder (actuator) design

Given the desired data flow rate $v(k)$ obtained from the controller, the task of the load adaptor is to cut the incoming data stream (with rate f_{in}) such that the the actual number of tuples accepted into the system is close to $v(k)$. In this paper, we investigate two different ways to accomplish this.

A straightforward way to implement the load shedder is to manipulate the number of data tuples entering the DSMS query network. In other words, we treat the Borealis system as a blackbox by not shedding load within the network. For this purpose, we set a shedding/filtering factor α ($0 \leq \alpha \leq 1$) to all the data streams. When Borealis receives a tuple, it flips an unfair coin with head probability $1 - \alpha$. A tuple is accepted only when the coin shows head. At the end of period k, α should be determined as follows:

$$\alpha = 1 - [v(k)/f_{in}(k+1)]. \quad (10)$$

However, $f_{in}(k+1)$ is unknown when we calculate α. We use its value in the current period $f_{in}(k)$ as an estimation.

Our control-based solution should also work for the existing load shedders such as the one in Borealis. In Borealis, load can be shed from any queues in the query network. Using the network in Fig. 2 as an example, we can drop tuples in front of any combination of operators from 1 to 12 while the simple load shedder mentioned above only allows shedding before operators 1, 2, and 3. This difference, however, does not conflict with our system model (therefore controller design). Our model says $y(k)$ depends on $q(k-1)c$, which is basically the outstanding 'load' in the queue. Shedding only intact tuples (outside the network) or partially processed tuples (in the network) makes no difference: the same 'load' is being discarded and $y(k)$ depends on how much load is left in the queue. Given the $v(k)$ generated by our controller, we know that new load with amount $L_a = v(k)c(k+1)$ can enter the DSMS during the next period $k+1$. However, the outstanding tuples carry a load of $L_q = q(k)c(k)$ and incoming streams carry a load of $L_i = f_{in}(k+1)c(k+1)$, which is approximated by $f_{in}(k)c(k)$. Therefore, load with amount of $L_s = L_q + L_i - L_a$ is to be shed. Pass the L_s value to the Borealis load shedder, it will find the best plan to bring down the total load by L_s.

4.5.3 Determination of the control period T

The sampling period is an important parameter in digital control systems. An improperly selected sampling period can deteriorate the performance of the closed-loop. In our setup, we consider the following two issues in selecting T:

1. *Nature of disturbances.* In order to deal with disturbances, our control loop should be able to capture the moving trends of these disturbances. The basic guiding rule for this is the Nyquist-Shannon sampling theorem [24]. A fundamental principle in the field of information theory, the theorem states that: when sampling a signal, the sampling frequency must be greater than twice the signal frequency in order to reconstruct the original signal perfectly from the sampled version. In our setup, this means the control period should be at most half of the width of the spikes in input rate (as average processing costs changes more slowly). In practice, a sampling frequency that is one order of magnitude larger than the input signal frequency is often used. Therefore, a high sampling frequency is preferred to capture the time-varying properties of the system and input data.

2. *Uncertainties in system signals.* In our problem, the output signal $y(k)$ and processing cost $c(k)$ are defined as the statistical expectations of a series of tuples. Taking such expectations can eliminate uncertainties brought by the heterogeneity of individual tuples. A larger sampling period (low sampling frequency) is preferred as more smoothing effects can be expected. For example, when tuple processing cost is in the order of milliseconds, setting T to a fraction of one second level would give us tens to a few hundreds of samples to approximate the real values of $y(k)$ and $c(k)$. For higher sampling frequencies, we get fewer samples to estimate $y(k)$ and may encounter estimation errors.

We need to make a tradeoff between the above two factors in choosing the right sampling period.

5. PERFORMANCE EVALUATION

We implemented a controller and a monitoring module in the Borealis data manager[6] based on our design. As the current release of Borealis does not include the load shedder presented in [21], we also built our own load shedder, which allows shedding from the queue and randomly selects shedding locations. In other words, it is more general than the first load shedder we discuss in Section 4.5.2 but lacks the optimization towards non-delay parameters found in the Borealis load shedder.

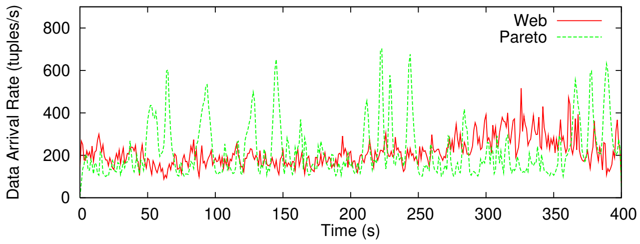

Figure 13: Traces of synthetic and real stream data.

We test our control-based framework with both synthetic and real-world stream data. The real data are traces of requests to a cluster of web servers[7]. The synthetic data are generated in such a way that the number of data tuples per control period follows a long-tailed (Pareto, to be specific) distribution [13]. The skewness of the arrival rates is regulated by a *bias factor* β. The traces of a Pareto stream with $\beta = 1$ as well as the web access data are plotted in Fig.

[6]http://nms.lcs.mit.edu/projects/borealis/

[7]dataset LBL-PKT-4, available at http://ita.ee.lbl.gov

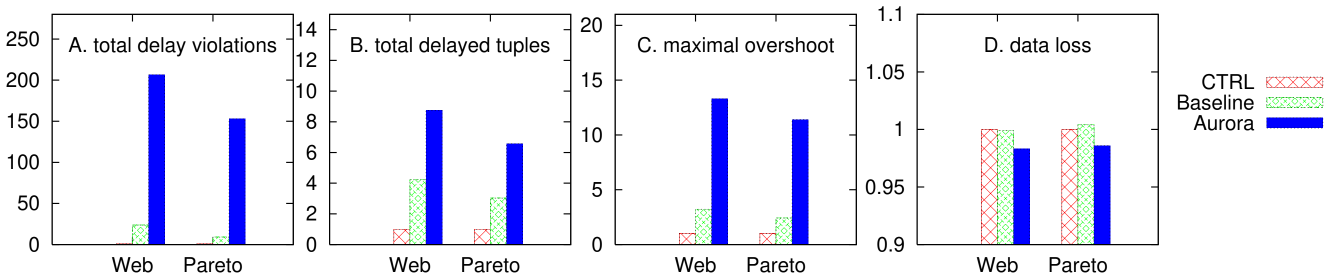

Figure 12: Relative performance of different load shedding strategies.

13. We can see that the fluctuations in the 'Pareto' data are more dramatic than those in the 'Web' data.

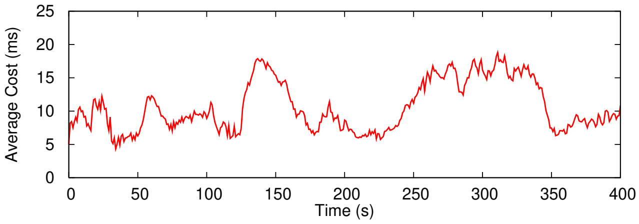

Figure 14: Variable unit processing costs (ms).

We also use synthetic traces to simulate the variations of per-tuple cost c. We first generate the cost variations following a Pareto distribution and then modify the trace by adding 'events' to it. For example, in the trace plotted in Fig. 14, we have a small peak at the 50th second, a large peak with a sudden jump (starting from the 125th second), and a high terrace with a sudden drop (250th to 350th second). The same queries for system identification (Section 4.2) are used for experimental studies.

We compare our control-based framework (referred to as *CTRL* hereafter) with the following two approaches:

- *AURORA*: the algorithm utilized in the current Aurora/Borealis system (Fig. 1). At the k-th control period, the measured load L is $f_{in}(k-1)$. To deal with variable per-tuple cost, we define $L_0 = H/c(k-1)$;

- *BASELINE*: a simple feedback control-based method: it uses system status (i.e., $q(k-1), c(k)$ in our case) in making decisions. Specifically, $v(k)$ is obtained from the system model (Eq.(8)): the target value of y_d would allow $y_d H/c(k)$ outstanding tuples, therefore $u(k) = y_d H/c(k) - q(k-1)$ more tuples can be added to the queue. Consequently, we get $v(k) = u(k) + f_{out}(k) = \frac{y_d H}{c(k)} + \frac{TH}{c(k)} - q(k-1)$. The unknown quantity $c(k)$ is estimated by $c(k-1)$. This method is used to test the importance of controller design.

In all the experiments we report in this section, we set target delay value y_d to 2000 milliseconds unless specified otherwise. We run all tests for 400 seconds. For *CTRL*, the controller parameters[8] identified by our analysis are: $b_0 = 0.4$, $b_1 = -0.31$, and $a = -0.8$. Following the experiments shown in Fig. 6, we set H to 0.97. The control period is set to 1000 milliseconds. Going back to Fig. 13, we see that most of the bursts in both traces last longer than a few (4 to 5) seconds therefore a sampling period smaller than two seconds is preferred according to the sampling theorem. The change of costs c in Fig. 14 has peaks with widths on the order of tens of seconds (with some exceptions) thus one-second period is definitely sufficient. We also test the systems with different choices of T and y_d.

We consider the following metrics in evaluating the adaptation strategy: 1) *Delay Violations*: the primary control goal. Specifically, we record both the *accumulated delay violations* (i.e., $\sum y - y_d$ for all data tuples whose processing delay $y > y_d$), and *total delayed tuples*, which is the total number of tuples whose delays are longer than y_d; 2) *Maximal Overshoot*: the longest delay violation recorded. This metric captures transient state performance; and 3) *Data Loss Ratio*: the percentage of data tuples discarded. This can be viewed as the *cost* of performing load shedding.

5.1 Experimental results

We first compare the long-term performance of *CTRL* with that of the two other algorithms. In Fig.12, we plot the ratios of all four metrics measured (i.e, totals in the 400-second period) from the *AURORA* and *BASELINE* experiments to that of *CTRL*. For example, when injected with the same 'Web' data stream, Fig. 12A shows that *AURORA* rendered 205 times more total delay violations than *CTRL* and *BASELINE* had 23 times. Similar results were obtained in total delayed tuples (Fig. 12B) and maximal overshoot (Fig. 12C). Note that all data points for *CTRL* are 1.0 in Fig. 12. The data loss ratio for all methods are almost the same with *AURORA* losing slightly fewer tuples (0.986 for 'Web' and 0.987 for 'Pareto'). It is easy to see that, for both real ('Web') and synthetic ('Pareto') data inputs, *CTRL* is the easy winner in the three delay-related metrics with almost the same amount of data loss. The *BASELINE* method, as a feedback solution, has worse performance than *CTRL* but it also beats *AURORA*.

To better understand the above long-term results, we show the transient performance of all three methods by plotting $y(k)$ values measured at all control periods in Fig. 15. We can see that, as expected, almost all output in *CTRL* is very close to the target value of two seconds. For *BASELINE* and *AURORA*, we can observe peaks that are large in both height and width. Such peaks are the results of either fluctuations of arrival rate or changes of c (e.g., those at about 50th second and 125th second, and the high terrace starting from the 230th second). Note the first two peaks of c also have impact on the *CTRL* system: average delay increases beyond two seconds. However, with the design goal

[8] Any set of parameters that are solutions to Equations (15) and (16) are supposed to have the same performance. This is verified by our tests with other set of parameters (details skipped).

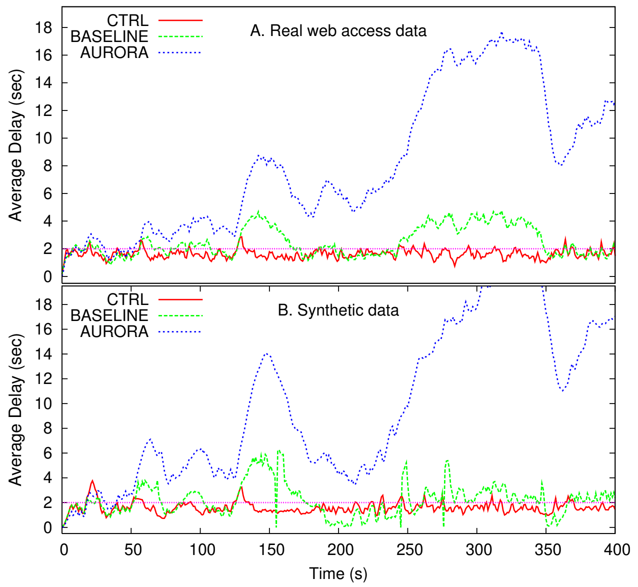

Figure 15: Performance of different load shedding methods.

of fast convergence and high damping, the controller in the *CTRL* system can quickly bring the system back to a stable state thus large peaks of y are avoided. The high terrace has almost no effect on *CTRL*. This is because the value of c increases gradually before the terrace. Our controller can capture and compensate for this kind of gradual change while the open-loop system cannot (i.e., Example 2 in Section 4.3.1). From Fig. 15, we can fairly conclude that the design goal of our controller (Section 4.4.1) is achieved.

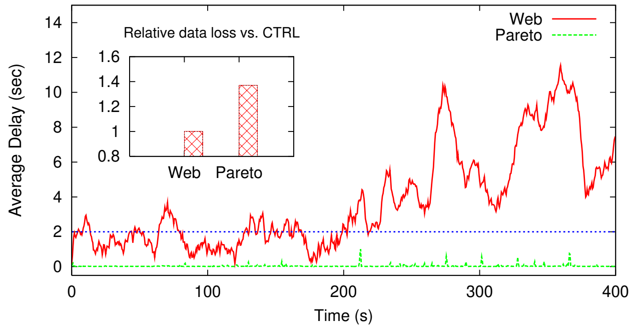

Figure 16: Performance of 'Aurora' with $H = 0.96$.

System robustness. In the above experiments, the *AURORA* method has poor stability: tuple delays increase all the time. A question one might ask is: can we remedy the problem by using a smaller L_0 value (recall the algorithm in Fig. 1) such that more data can be discarded? In our setup, this means the same as changing the H value (even though $H = 0.97$ is proved to be correct in Section 4.2) as we define L_0 to be H/c. Fig. 16 shows the results of the *AURORA* method under both real and synthetic data inputs using a smaller H value of 0.96. For the 'Web' data inputs, the system is still unstable. Surprisingly, no delay violations can be observed for the 'Pareto' inputs. However, the price for this is huge: it costs 37% more data loss than *CTRL* (small graph in Fig. 16). This result shows the poor robustness of open-loop solutions: it is hard to tune the system as performance depends heavily on the pattern of inputs.

To further study the robustness of the three methods, we test them using data streams with different levels of burstiness. Specifically, we feed the systems with synthetic data streams with bias factors β of 0.1, 0.25, 0.5, 1, 1.25, and 1.5. The smaller the bias factor, the more bursty the input. In Fig. 17, we show the change of all four metrics with respect to the bias factor. All numbers plotted are relative to the corresponding value measured in the case of $\beta = 1.5$. As the input stream becomes more bursty, very little difference can be observed in *CTRL* (Fig. 17A) while the changes in *AURORA* (Fig. 17B) are much more dramatic. The performance of *BASELINE* is not significantly affected by the bias factor as well (data not shown).

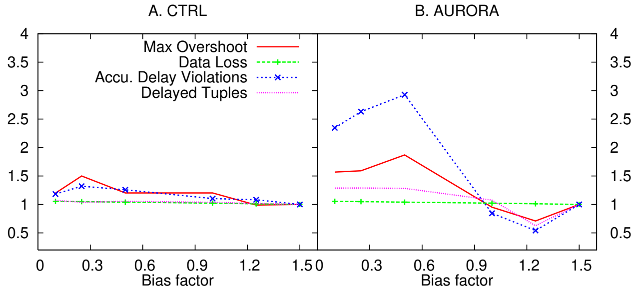

Figure 17: Effects of input burstiness.

In Fig. 18 we show how different systems respond to changes of target value y_d at runtime. In these tests, we set y_d to be 1000 milliseconds initially and change it to 3000 milliseconds at the 150th second and then to 5000 milliseconds at the 300th second. We can see that *CTRL* converges to the new target values very quickly. The *AURORA* method does not respond to the changes of y_d at all as it is open-loop. When y_d changes, it takes the *BASELINE* method very long time to converge to the new target value.

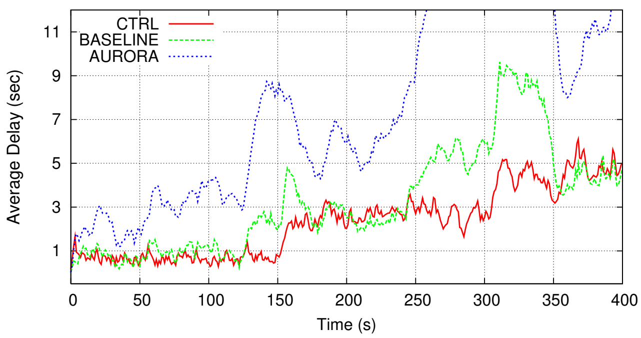

Figure 18: Responses to changes of target value.

Effects of control period. In Section 4.5.3, we discussed the general rules on choosing the right sampling period. We test these rules by running experiments with nine different sampling periods ranging from 31.25 to 8000 milliseconds with the *CTRL* system and 'Web' data stream. In Fig. 19, every data point is the ratio to the lowest corresponding metric in all nine tests and the x-axis has a logarithmic scale. For example, the smallest accumulated delay violations were recorded under $T = 500ms$ and this value is about 40 times as high when $T = 4000ms$. Obviously, the magnitude and frequency of delay violations increase dramatically

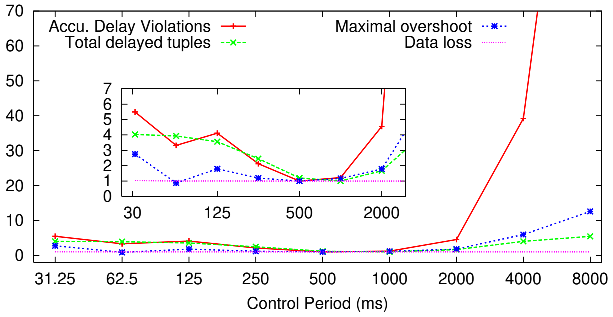

Figure 19: Performance under different values of T.

when T is beyond four seconds. As expected, a shorter control period is preferred. This confirms our discussion about the sampling theorem in Section 4.5.3. When T becomes too small, performance degrades. The best region seems to be $[250, 1000]$ in these experiments (see small graph in Fig. 19). Similar results are obtained for the 'Pareto' data inputs. One thing to point out is: in all experiments with T smaller than 4000ms, *CTRL* outperforms *BASELINE* and *AURORA* with similar difference shown in Fig. 12.

Computational overhead. The operation of our controller only involves several floating point calculations at each control period. In our experimental platform with a Pentium 4 2.4GHz CPU, this time is only about 20 microseconds. This is trivial because the control period is set to be (at least) on the order of hundreds of milliseconds.

Summary of experimental results. From the above experiments, we see that *CTRL* is the winner in all delay-related metrics.The use of feedback signal provides big advantages over the open-loop *AURORA* method. A thorough understanding of system dynamics is also extremely useful. By applying simple rules derived from the system model, the *BASELINE* method achieves far better performance than *AURORA*. Decisions based on rigorous controller design is another plus for our method. With guaranteed convergence, our controller avoids large and long-time deviations from the desired output while the *BASELINE* method suffers from such deviations. An important feature of the control-based solution is its robustness. Note that we only use standard inputs to validate the system model and controller tuning (i.e., setting controller parameters) is accomplished by mathematical reasoning exclusively. On the contrary, tuning of other methods can be *ad hoc*, as evidenced by the dependence of open-loop solutions on the pattern of data inputs. We have reasons to believe that even the current system model (Fig.4) can be used for DSMSs other than Borealis: we noticed (via experiments) that modifying the query network only changes a parameter (c) but not necessarily the structure of the model.

6. CONCLUSIONS AND FUTURE WORK

This paper argues for the importance of managing data processing delays in data stream management systems. Violations of delay are generally caused by system overloading. Load shedding and other adaptation strategies have been exploited to compensate for degraded delays under overloading. We noticed that the strong dynamics such as bursty arrival pattern of data stream applications require an adaptation strategy with excellent transient-state performance (e.g., fast convergence to steady state), which most of the current works in this area fail to provide. We proposed a load shedding framework that leverages various techniques from the field of control theory. We started by developing a dynamic model of a steam management system. We then construct a feedback control loop to guide load shedding through system analysis and rigorous controller design.

We have implemented our design and performed extensive experiments on a real-world system - the Borealis stream manager. It is evident that our approach achieves better performance in terms of reduced delay violations over current strategies that do not consider system status in decision-making. The control-based strategy is also robust and light-weight. Finally, we believe our explorations can give rise to many opportunities to conduct synergistic research between the database and control engineering communities to extend our knowledge in both fields.

Immediate follow-up work includes more experiments on the Aurora load shedder and more dramatic changes of per-tuple costs. The idea is to use adaptive control techniques to capture the internal variations of the system model and provide better control over the whole system. Our control-based framework can also be extended in a few directions. For example, we can improve the quality model by providing heterogeneous quality guarantees for streams with different priorities; and multiple quality dimensions can be supported at the same time by introducing a multi-in-multi-out control model. We are currently investigating the potential of control theory in a number of other topics in DBMS research such as query reoptimization and dynamic resource allocation in traditional DBMSs.

7. REFERENCES

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

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Procs. of CIDR*, January 2005.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal.*, 12(2):120–139, 2003.

[4] 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.*, 2003.

[5] 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.

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

[7] 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.

[8] 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.

[9] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Loadstar: A load shedding scheme for classifying data streams. In *Procs. of SIAM Conf. on Data Mining*, 2005.

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

[11] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall, Massachusetts, 2002.

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

[13] M. Harchol-Balter, M. Crovella, and C. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *Journal of Parallel and Distributed Computing*, 59(2):204–228, November 1999.

[14] K.-D. Kang, S. H. Son, and J. Stankovic. Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases. *IEEE Trans. on Knowledge and Data Engineering*, 16(10):1200–1216, October 2004.

[15] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Procs. of SIGCOMM*, September 1991.

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

[17] 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.

[18] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Procs. of SIGMOD*, pages 563–574, 2003.

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

[20] F. Reiss and J. M. Hellerstein. Declarative Network Monitoring with an Underprovisioned Query Processor. In *Procs. of ICDE*, April 2006.

[21] 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.

[22] Yi-Cheng Tu, Mohamed Hefeeda, Yuni Xia, and Sunil Prabhakar. Control-based Quality Adaptation in Data Stream Management Systems. In *Proceedings of DEXA*, pages 746–755, August 2005.

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

[24] W.D.Stanley. *Digital Signal Processing*. Reston Publishing Co., 1975.

[25] 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.

Acknowledgments

The authors would like to thank Ms. Nesime Tatbul, Prof. Uğur Çetintemel, and Prof. Stan Zdonik from Brown University for their help on the Aurora/Borealis system.

APPENDIX

A. CONTROLLER DESIGN BASED ON POLE PLACEMENT

In this study, we set the desired convergence rate to three sampling periods. This means the system, in response to dynamics, would converge to $1 - \frac{1}{e} \approx 63\%$ of the desired value in 3 control periods and to 98% in 12 periods. We set the system damping to 1 and set the desired closed-loop poles to be on the real axis, at 0.7. Thus, the desired closed-loop characteristic equation (CLCE) is:

$$(z - 0.7)^2 = z^2 - 1.4z + 0.49 = 0 \tag{11}$$

As Eq.(4) shows a first-order system, the controller $C(z)$ will have one pole and its generic format [11] is

$$C(z) = \frac{H(b_0 z + b_1)}{cT(z + a)} \tag{12}$$

where b_0, b_1, and a are controller parameters. Therefore, the closed-loop transfer function (CLTF) becomes

$$\frac{C(z)G(z)}{1 + C(z)G(z)} = \frac{b_0 z + b_1}{z^2 + (a - 1 + b_0)z + (-a + b_1)} \tag{13}$$

and the actual closed-loop characteristic equation is

$$z^2 + (a - 1 + b_0)z + (-a + b_1) = 0. \tag{14}$$

Matching the above CLCE to its desired form shown in Eq.(11), we get the following (Diophantine) equation:

$$z^2 + (a - 1 + b_0)z + (-a + b_1) = z^2 - 1.4z + 0.49 \tag{15}$$

At the steady state, the CLTF should have a static gain that equals one, meaning we want the output y to be exactly the same as y_d. This results in the following equality:

$$\frac{b_0 z + b_1}{z^2 + (a - 1 + b_0)z + (-a + b_1)}\Big|_{z=1} = 1 \tag{16}$$

Solving Equations (15) and (16), one can obtain the controller parameters a, b_0, and b_1.

In summary, the above design results in a closed loop system having two poles, both of which are on the positive real axis at 0.7. Now we can generate the control signal u. Let $U(z)$ and $E(z)$ be the z-transforms of u and error e, respectively. According to Fig. 10, e is the input and u is the output respect to the controller, we have

$$U(z) = C(z)E(z) = \frac{H(b_0 z + b_1)}{cT(z + a)} E(z).$$

Multiplying both sides by $\frac{(z+a)cT}{z \cdot H}$, we get

$$U(z)\frac{cT}{H} + \frac{acT}{H}U(z)z^{-1} = b_0 E(z) + b_1 E(z) z^{-1}.$$

By inverse z-transformation, the above leads to the solution for u as follows:

$$u(k) = \frac{H}{cT}[b_0 e(k) + b_1 e(k-1)] - au(k-1).$$