Load Shedding in a Data Stream Manager *

Nesime Tatbul[1] Uğur Çetintemel[1] Stan Zdonik[1]
Mitch Cherniack[2] Michael Stonebraker[3]

[1]Brown University
{tatbul, ugur, sbz}@cs.brown.edu

[2]Brandeis University
mfc@cs.brandeis.edu

[3]M.I.T.
stonebraker@lcs.mit.edu

Abstract

A Data Stream Manager accepts push-based inputs from a set of data sources, processes these inputs with respect to a set of standing queries, and produces outputs based on Quality-of-Service (QoS) specifications. When input rates exceed system capacity, the system will become overloaded and latency will deteriorate. Under these conditions, the system will *shed load*, thus degrading the answer, in order to improve the observed latency of the results. This paper examines a technique for dynamically inserting and removing *drop* operators into query plans as required by the current load. We examine two types of drops: the first drops a fraction of the tuples in a randomized fashion, and the second drops tuples based on the importance of their content. We address the problems of determining when load shedding is needed, where in the query plan to insert drops, and how much of the load should be shed at that point in the plan. We describe efficient solutions and present experimental evidence that they can bring the system back into the useful operating range with minimal degradation in answer quality.

1 Introduction

New applications that must deal with vast numbers of input streams are becoming more common. These include applications that process data from small embedded sensors, applications that must correlate financial data feeds, and applications that must manage input from a very large number of geo-positioning devices. A new class of data management systems is emerging in response to these applications [4, 7, 6, 15, 19]. These systems will provide the same kind of infrastructure to stream-based applications that database management systems have provided for data processing applications.

The *Aurora Data Stream Management System* [4, 2, 1] is one such new system that is being developed at Brandeis University, Brown University, and M.I.T. Aurora uses application-level semantics to make intelligent decisions about resource allocation. Typically, the resources in question include processor cycles, memory, and bandwidth [7].

The Aurora design has been influenced heavily by several specific applications [1]. In particular, we have studied an application that performs remote triage over a battlefield given a large number of soldiers with biosensors embedded in their uniforms. Another application involves a next-generation airplane that collects reconnaissance data about the positions of enemy units and that intelligently disseminates this information to a series of ground stations, each with different requirements. We are also beginning an experimental effort to manage fish respiratory data as an indicator of the presence of toxins in a reservoir.

All of these applications are characterized by a large number of push-based data sources in which the data arrival rates can be high and unpredictable. Each of these applications is responsible for monitoring data to detect critical situations, during which the data rates can significantly increase and exceed system capacity. If no corrective action is taken, queues will form and latencies will increase without bound. Since such overload situations are usually unforeseen and immediate attention is vital, adapting the system capacity to the increased load by adding more resources or distributing computation to multiple nodes [7] may not be feasible or economically meaningful. In these cases, the only immediate solution is to shed some of the load. In general terms, *load shedding* is the process of dropping excess load from the system. In order to gracefully degrade the performance of the system, load shedding should identify and discard the relatively less important data [17].

The work described in this paper concentrates on the processor as the limited resource and exploits application-specific Quality-of-Service (QoS) informa-

*This work has been supported by the NSF under the grant IIS-0086057.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

tion when making load shedding decisions. We assume that QoS is specified separately for each application and describes the relationship between various characteristics of an answer and its usefulness (i.e., utility). Thus, we model QoS as a set of functions that relate a parameter of the output to its utility. For example, in many applications, answers are only useful if they are timely. Therefore, the utility of an answer can be a function of the latency involved in its creation. Also, the utility of an answer can be a function of the output value, expressing how important each value is for purposes of the application. For example, in a medical application that monitors patient heartbeats, extreme values are certainly more interesting than normal ones.

It should be noted that while dropping tuples will certainly reduce the processing requirements on the system, and thus, reduce the effective load, it will also have a detrimental effect on the accuracy of the answer. Said another way, load reduction to improve latency and accuracy are fundamentally at odds. When we improve utility by shedding load and reducing latency, we necessarily lose utility by producing an *approximate answer*. The technical challenge in this work is to improve latency with minimal loss in answer accuracy.

Load shedding is not a new idea. The fields of networking [18] and multimedia [8] have long studied load shedding; however, the context is rather different here. First, we consider applications that share processing, and second, we allow for QoS specifications that include semantics (i.e., value-based QoS). We also have a much more restricted view of the processing that is carried out by an Aurora application. We only allow applications to be built out of our seven operators. In this way, we can gather useful statistics (e.g., operator costs and selectivities) about the processing elements in a way that is difficult in other contexts.

This paper explores scalable and efficient load shedding techniques for data stream processing systems. In particular, we address and provide solutions to three key load shedding questions: determining (1) when, (2) where, and (3) how much to shed load. We also present the results of an experimental study that quantitatively demonstrates the effectiveness of our approach by comparing it to alternative, traditional shedding techniques under a variety of workload and processing scenarios.

The next section is a brief overview of the Aurora System. Section 3 presents a detailed discussion of the problem. Section 4 contains the description of our load shedding algorithms. Sections 5 and 6 present the experiments and their results. Section 7 outlines related work, and Section 8 summarizes the work and provides future directions.

2 Aurora Overview

The Aurora System has been designed to deal with very large numbers of data streams. An input stream is a potentially infinite sequence of tuples generated by

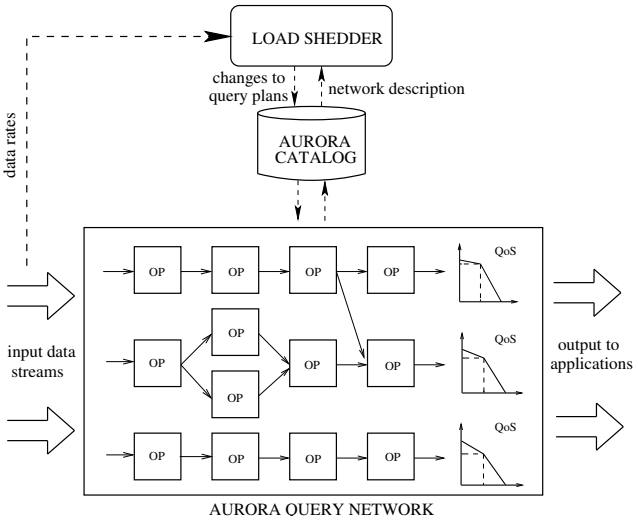

Figure 1: Load Shedder within Aurora

a data source. Possible data sources include computer programs and hardware sensors that continuously push data to Aurora. Aurora is expected to process a large number of queries that are built out of a small set of operators [2]. A query may take an arbitrary number of input streams and always ends at a single output. An operator may be connected to multiple downstream operators. All such splits carry identical tuples and enable sharing of computation among different queries. Multiple streams can also be merged since some operators accept more than one input. An Aurora query network is a collection of such queries.

Aurora has seven primitive operators. Some of these operators generate tuples with new values that are possibly different from their input tuples (*Map*, *Aggregate*, *Resample*) while others process the tuple values as they are (*Filter*, *Union*, *Bsort*, *Join*) [2]. Note that while *Join* does produce new tuples, they are simple concatenations of the inputs. Thus, no new attribute values are created. The work in this paper considers operators of the second type only, namely, *Filter*, *Union*, and *Join*. *Filter* takes a predicate as parameter and filters out the input tuples that do not satisfy this predicate. *Union* merges two input streams into a single stream. *Join* correlates two input streams on equality of a common attribute, taking a time-based window size as parameter.

We also consider the use of system generated *Drop* operators. A *Drop* operator eliminates some fraction of its input. It achieves this by either randomly selecting tuples to drop or by using a predicate to detect and drop tuples containing less important values. In the latter case, *Drop* is effectively the same as *Filter*. Note that if *Drops* are inserted into a network containing only *Filter*, *Union*, and *Join*, the output is guaranteed to be a subset of the actual answer. This observation motivates our restriction on query operators, but it should be noted that in practice there are

actually many applications that can be serviced within this restriction (e.g., many categorization and routing applications like those in financial services and intelligent dissemination). Extending these techniques to work for our other operators is left as future work.

Figure 1 illustrates a small Aurora query network. The output of each query in the network is routed to the relevant monitoring application. Furthermore, each output is supplied with a QoS specification. Currently, QoS in Aurora is captured by three functions (1) a latency graph, (2) a value-based graph, and a (3) loss-tolerance graph [4]. The latency graph indicates how utility drops as an answer tuple is delayed. The value-based graph shows which values of the output tuple space are most important. The loss-tolerance graph is a simple way to describe how averse the application is to approximate answers. The quality metric of the loss-tolerance graph is the percentage of output tuples delivered to an application, 100% indicating zero loss. The loss-tolerance graph can either be explicitly defined for an application or be automatically derived from a value-based graph and statistics about the distribution of values in the output. An important assumption we make about QoS functions of types (1) and (3) is that they have concave shapes, i.e., the negative slope of the function is monotonically increasing.

The Aurora catalog (as shown in Figure 1) contains information regarding the network topology, inputs, outputs, QoS, and relevant statistics (e.g., selectivity and average processing cost for an operator). The Aurora run-time system reads the network description from the catalog and executes the queries. Data tuples are queued for processing by the Aurora run-time. The scheduler selects an operator with waiting tuples and executes that operator on one or more of the input tuples [4, 5].

As shown in Figure 1, the load shedder component of Aurora receives information on data rates and reads the network description from the catalog. If it detects an overload in the system, it sheds load by inserting load reducing *Drop* operators into the running query network. The changes to the query plans are stored in the catalog, thus, updating the network being executed by the Aurora run-time. The load shedder can also determine when *Drop* operators become unnecessary as data rates subside. In this case, the query plans are modified to reduce the amount of data being shed.

3 The Problem

In what follows, we explore techniques for scalable load shedding in Aurora. We model load shedding as the automatic insertion of drop operators into a running network. In this paper, we consider two fundamental types of drop operators.

1. **Random Drop.** This operator takes a single parameter p that expresses the fraction of tuples that should be dropped. This is implemented by tossing a coin that is weighted by p.

2. **Semantic Drop.** This operator is essentially a filter operator with a predicate whose selectivity corresponds to $1-p$ and that discards tuples with the lowest utility.

The load shedding process consists of three fundamental decisions.

1. **Determining when to shed load.** The processing load of the query network needs to be continuously evaluated. If there is overload, it should be detected quickly.

2. **Determining where to shed load.** Tuples can be dropped at any point in the processing network. Obviously, dropping them early avoids wasting work; however, because a stream can fanout to multiple streams, an early drop might adversely effect too many applications.

3. **Determining how much load to shed.** Once we have determined where to insert a drop operator, we must decide the magnitude of that drop. In the case of a random drop, this involves deciding on the percentage of tuples to drop. In the case of a semantic drop, we must decide the form of the predicate.

Load shedding is an optimization problem and can be formally stated as follows. We are given a query network N, a set of input streams I with certain data arrival rates, and a processing capacity C for the system that runs N. Let $N(I)$ indicate the network N operating on inputs I, and $Load(N(I))$ represent the load as a fraction of the total capacity C that network $N(I)$ presents. Load shedding is typically invoked when $Load(N(I)) > H \times C$ [1]. The problem is to find a new network N' that is derived from network N by inserting drops along existing arcs of N such that $Load(N'(I)) < H \times C$ and $U_{accuracy}(N(I)) - U_{accuracy}(N'(I))$ is minimized. $U_{accuracy}$ is the aggregate utility that is measured from the loss-tolerance QoS graphs of the application set. $U_{accuracy}(N(I))$ represents the measured utility when there is no load shedding (i.e., there are no inserted drops).

We assume that any processor cycles that are recovered by the load shedder will be used sensibly by the scheduler to improve the overload situation thereby best improving the latency. We can, therefore, simply figure out how much capacity we need to recover and then produce a plan to do so. The scheduler will do the rest. This decoupling of the scheduler and the load

[1] The constant H is the *headroom factor* that is a conservative estimate of the percentage of processing resources required by the system at steady state. The headroom is reserved to guard the system against thrashing.

shedder is an important simplification. By making this assumption, the load shedder need not consider latency QoS directly.

4 The Algorithms

We have developed a scalable load shedding algorithm with two variants, one for inserting random drops (called *Random Load Shedder*) and one for inserting semantic drops (called *Semantic Load Shedder*). This section first gives the overall structure of the algorithms and then provides some detail on our fundamental data structures and how we use them to make the three load shedding decisions mentioned earlier.

4.1 Algorithm Structure

The load shedder always operates in a loop, detecting important changes to the load status and reacting accordingly.

The first action is to check the current load level against the known system capacity. The current load of the network is evaluated with a simple calculation that involves the current input rates, and the operator costs and selectivities. If $Load(N(I)) > H \times C$, then the system is overloaded and load must be shed by inserting drops into the query network. On the other hand, if $Load(N(I)) < H \times C$, then load can be returned by removing drops that might have been inserted in previous steps.

The next major action in the load shedding algorithm is to determine what drops to insert and where to put them. Our approach depends on a pre-computed, tabular data structure called the *Load Shedding Road Map* (*LSRM*). As illustrated in Figure 2, the LSRM is a table in which each subsequent row represents a load shedding plan that sheds more load than its predecessor. A load shedding plan consists of an indication of the expected cycle savings (c_i indicating the cycle savings coefficient for input i), the network arcs to place the drop operators, the corresponding drop amounts, and the effect of the drops on QoS graphs (p_i indicating the QoS cursor for output i). At run-time, when an overload is detected, we use the LSRM to find a plan for recovering $Load(N(I)) - H \times C$ processor cycles by simply performing a table lookup for the first plan that sheds at least this much load.

As shown in Figure 2, we keep a cursor that indicates which row in the LSRM was used last. Later on, if additional load needs to be shed, the search on the LSRM can begin at the cursor. On the other hand, if the load is ever determined to be within the capacity bound and there are still drops in the network (cursor is non-zero), then the search can be done in the reverse direction.

Materializing load shedding plans in advance using the LSRM significantly reduces the run-time overhead of making load shedding decisions.

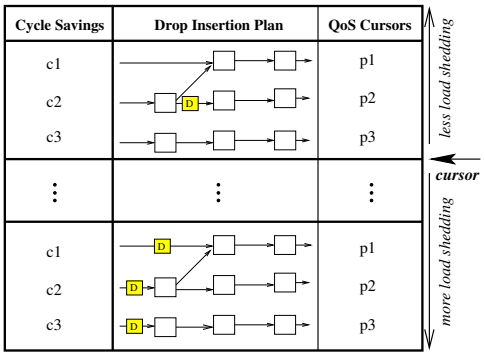

Figure 2: Load Shedding Road Map (LSRM)

4.2 Overload Detection

We now describe our techniques for making the first of the three decisions in load shedding: when to shed load.

Each input to the network has an associated *load coefficient*. The load coefficient represents the number of processor cycles required to push a single input tuple through the network to the outputs. Consider an input stream I running through a query of n operators and producing an output stream O as shown in Figure 3.

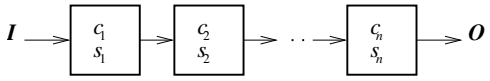

Figure 3: Calculating load coefficients for stream I

Assume that each operator i has an associated cost c_i (cycles per tuple) and selectivity s_i. We assume that such statistics can be gathered over some characteristic run period. The load coefficient for input I is computed as

$$L = \sum_{i=1}^{n} (\prod_{j=1}^{i-1} s_j) \times c_i \qquad (1)$$

for all n operators on a path from input I to the output O. If an input has load coefficient L (in processor cycles per tuple) and input rate r (in tuples per time unit), the actual run-time load for that input is $L \times r$ (cycles per time unit). If there are m inputs, we can compute the total load as

$$\sum_{i=1}^{m} L_i \times r_i$$

The load coefficient formulation given in (1) is based on the flat query shown in Figure 3. We now illustrate how we generalize it to query networks with fan-outs and binary operators. In Figure 4, we provide an example query network with two continuous queries and two input streams. We present load coefficients for each of the input streams as well as load coefficients for each intermediate stream in the network. When there is fan-out, each different path's load is added to

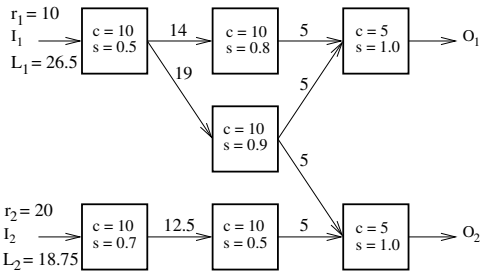

Figure 4: Example query network with load coefficients

an input's load coefficient. When there is a binary operator, input rate flowing into the operator is an aggregation of both inputs. Note that all of these coefficients can be computed statically. Later, the actual run-time load can easily be calculated using these pre-computed coefficients. For example, if it turns out that input streams I_1 and I_2 have rates of 10 and 20 tuples per time unit, respectively, then the load of this network becomes $26.5 \times 10 + 18.75 \times 20 = 640$ cycles per time unit. If this value exceeds the capacity of the system, then the load shedder must be invoked.

Note that the actual load is a combination of current input rates and any queues that may have built up since the last load evaluation step. With the headroom constant $H > 0$, queues will eventually disappear as long as the load shedder maintains the headroom space. Alternatively, load shedder may exploit the queue length information to speed up this process through "overshedding" [16].

4.3 The Load Shedding Road Map (LSRM)

This subsection describes our techniques to make the last two decisions in load shedding: where to shed load and how much load to shed. Making these decisions constitute the major steps in construction of the LSRM.

As shown in Figure 2, the LSRM is an ordered sequence of entries, each of which is a triple of the form:

< Cycle Savings Coefficients (CSC),
Drop Insertion Plan (DIP),
Percent Delivery Cursors (PDC) >

The Drop Insertion Plan (DIP) is a set of drops that will be inserted at specific points in the network. Cycle Savings Coefficients (CSC) is a list of input streams that will be affected by the plan along with their associated savings coefficients. CSC's are used to determine how many cycles will be saved along the path of each input stream if the corresponding DIP is adopted. Percent Delivery Cursors (PDC) is a list of cursors for the loss-tolerance QoS graphs, one for each output. They indicate where the system will be running (in terms of percent of tuple delivery) if the drop insertion plan is adopted.

Figure 5 shows the steps followed in the construction of the LSRM. In this section, we will describe these steps one by one.

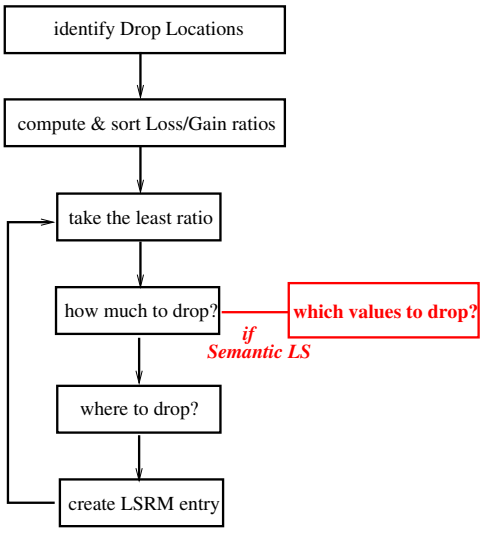

Figure 5: LSRM construction

4.3.1 Choosing Drop Locations

Each entry in the LSRM must guarantee that the utility sacrificed by adopting its DIP is the minimum that is necessary to save the number of cycles promised in its CSC. For this, the amount of utility loss on the loss-tolerance QoS graphs must be minimized. Additionally, the number of cycles gained in return must be maximized. Hence, when creating a new LSRM entry, the best network location (i.e., arc) to insert a drop is the one with the minimum *Loss/Gain* ratio. This guarantees minimal aggregate utility loss if statistical values accurately reflect the system state [16].

The search space for all possible drop locations may seem very large at first. However, it can be dramatically reduced by exploiting several heuristics. We use the network shown in Figure 6 to discuss how reduction is performed. This network consists of two input streams and three queries. Each query has a loss-tolerance QoS attached to its output. Smaller boxes marked with letters indicate candidate locations for drops.

First, consider a query plan that has no sharing with others. In a network that contains such a plan, a drop insertion at *any* location in that query does not affect the other queries. Hence, the utility loss is only observed at the output of that query plan. For example, in Figure 6, the bottom-most query has no sharing with the top two. In general, it is best to insert drops as early in the query plan as possible since it minimizes wasted work. Therefore, for query plans with no sharing, the best location for a drop is always at its input. Hence, the small box marked with F is the best location for a drop in the bottom query.

Now consider a query network with sharing. Sharing occurs when the output of an operator fans out to more than one downstream operator leading to different outputs. This is observed at operator 1 in Figure

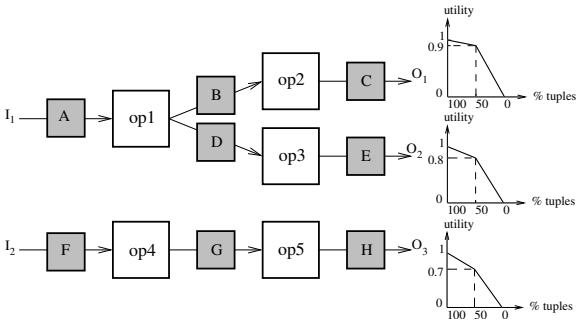

Figure 6: Example query network with candidate drop locations

6. Any tuple coming out of operator 1 would be routed to both outputs O_1 and O_2. Hence, a drop operator inserted at A affects both outputs. Inserting a drop at B or D affects only one output and is thus a way to isolate the effect to a single output. If both B and D end up requiring similar drops, then the drop could be placed at A thereby saving the work of operator 1. However, if A is preceded by a linear stretch of boxes, then by our previous reasoning, the drops should be pushed upstream to the next split point. As a result, output of operators with split points (B, D) and Aurora inputs (A, F) are the only locations that need to be considered for drops. For each such candidate location, we compute the Loss/Gain ratios, as described in the next section.

4.3.2 Loss/Gain Ratios

Loss/Gain ratios allow us to sort the candidate drop locations by their desirability as candidates. Here, *Loss* refers to utility loss from loss-tolerance QoS for each percentage of tuples dropped. *Gain* refers to processor cycles gained for each percentage of tuples dropped. Consider Figure 7, where a drop operator has been inserted upstream from a query subnetwork. R is the input rate flowing into the drop, x is the drop amount, D is the cost of the drop operator itself, and L is the load coefficient of stream flowing into the subnetwork. The gain from inserting this drop is:

$$G(x) = \begin{cases} R \times (x \times L - D) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Assume the loss-tolerance utility function of the affected output is given as $U(x)$. The Loss/Gain ratio is then:

$$\frac{-dU(x)/dx}{dG(x)/dx} = \frac{\text{negative slope of } U(x)}{R \times L}$$

One important point is that since we are sorting Loss/Gain ratios, absolute values of data rates are not needed. Rather, it is enough to know their relative proportions. Hence, if statistics on rate proportions are available, the Gain values can be computed statically.

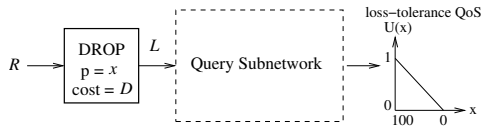

Figure 7: Insertion of a drop

Loss is determined by using the loss-tolerance QoS graph. In our example in Figure 6, the QoS graphs of locations B, D, F are the same as those of O_1, O_2, and O_3, respectively. The QoS of location A is the cumulative QoS from O_1 and O_2 and is obtained by summing the QoS functions of both of these outputs. Each drop location may have as many different Loss/Gain ratios as there are different function pieces in its (cumulative) QoS graph. For example, the percent loss for location F is $(1-0.7)/50$ for the first piece of the QoS function, whereas it is $(0.7-0)/50$ for the second piece. Furthermore, it is always guaranteed that Loss/Gain ratio increases as we move from 100% to 0%. This is a result of our concaveness assumption mentioned earlier.

Once the drop locations and their Loss/Gain ratios are fixed, they are then sorted in ascending order by their Loss/Gain ratios (Figure 5). The entries of the LSRM are created by processing the drop locations in this order. Each new entry builds upon the previous entry, by applying the drop location with the next smallest Loss/Gain ratio.

4.3.3 LSRM for Random Load Shedding

Given the drop location with the smallest Loss/Gain ratio and the previous LSRM entry, the random load shedding algorithm follows two main steps to create a new LSRM entry:

Determining how much to drop. First, the drop parameter p, which denotes the fraction of tuples to be dropped, has to be determined. Each drop amount is applied in increments of a $STEP\_SIZE$. To explain how we set this parameter, we will turn back to the formula for $G(x)$. To guarantee $G(x) > 0$, i.e., the gain from drop insertion is more than its cost, we must ensure that $x > \frac{D}{L}$. For this purpose, we use a $STEP\_SIZE$ parameter such that $STEP\_SIZE > \frac{D}{min\{L\}}$, where $min\{L\}$ is the minimum load coefficient in the network (over all arcs). We use this minimum value so that the chosen $STEP\_SIZE$ will work for all locations in the network. The value for this parameter also affects the granularity of the entries of the LSRM.

Assume that the loss-tolerance QoS for the output that correspond to the given drop location has a percent delivery cursor (PDC) value of x, as stored in the previous LSRM entry. Then we choose the new drop amount such that the cursor moves to $x - 100 \times STEP\_SIZE$. For this, the new drop to be inserted must have the drop parameter $p = 1 - (x - 100 \times STEP\_SIZE)/x$.

Placing the drop in the network. After p is deter-

mined, a drop operator can be created for the designated drop location. However, the new drop operator's placement in the network may affect the drops that already exist in the DIP inherited from the previous LSRM entry. For example, if we were to insert a drop with $p = k$ at A in Figure 6, and B already had a drop with $p = l$ where $l \leq k$, then the drop at B becomes redundant because the drop at A covers the amount B is supposed to drop. However, if $l > k$, then B's percentage must be reduced to produce a total percentage of l at the output. In other words, the drop insertion decision made in an earlier step must be maintained while the new decision is being applied. Our algorithm achieves this by inserting the new drops at the outputs first, and then pushing them towards the desired drop locations. Necessary adjustments are made on drop parameters of the existing drops that are encountered along the way.

New drops combined with the drops of the previous LSRM entry form the DIP of our new LSRM entry. We update the CSCs for each stream that sends tuples to the drops in the DIP. Finally, QoS cursors of the affected outputs are advanced and stored in PDCs. The next iteration of the loop takes the current entry and builds upon it. This way, each LSRM entry has more savings than the previous one.

4.3.4 LSRM for Semantic Load Shedding

LSRM entries for the semantic load shedder are created in almost the same way as for the random one. The major difference is that we need to create semantic drops with proper predicates which will provide us the desired level of load shedding, while filtering out the data values with the lowest utility. Therefore, in this section, we mainly discuss additional techniques that the semantic load shedding algorithm uses to construct the LSRM entries.

We developed a technique that enables us to follow the same method to determine drop amount and location, as the one described for the random load shedder. Our approach derives the loss-tolerance QoS graph from a given value-based QoS graph. In this way, the derived loss-tolerance QoS graph captures the value utility information and can be safely used for deciding Loss/Gain ratios.

Deriving the loss-tolerance QoS. For the purposes of this study, we restrict value-based QoS graphs to be piece-wise linear functions. For simplicity, assume that values in each value interval have a constant utility [2]. Assume further that the output data-value histograms are available for each of the intervals specified in the value-based QoS graph. Such histograms are commonly created in conventional DBMS's to assist in query processing. A histogram shows the relative frequency of each value interval. Using a histogram and a value-based QoS graph, we can produce a loss-tolerance QoS graph as described below. We use the following notation:

u_i : utility of values in interval i
f_i : relative frequency of values in interval i, $\sum f_i = 1$
w_i : weighted utility of values in interval i, $w_i = u_i \times f_i$
n_i : normalized utility of values in interval i, $(n_i = \frac{w_i}{\sum w_i})$

We order the value intervals based on their u values in ascending order and store them in a table together with their u, f, w, and n values. Table 1 illustrates such a table with two value intervals.

interval	u	f	w	n
0-50	0.2	0.4	0.08	$0.08/0.68 = 0.12$
51-100	1.0	0.6	0.6	$0.6/0.68 = 0.88$

Table 1: Example value intervals

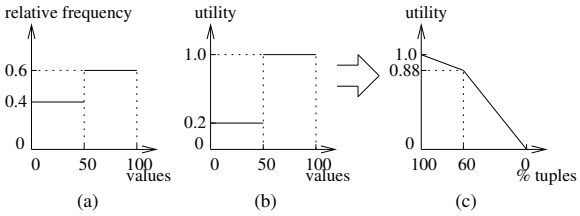

Figure 8: Derivation of the loss-tolerance QoS

The derivation relies on the fact that given a value-based QoS, if we needed to drop some tuples, we would always start dropping from the lowest utility interval (hence the table is ordered on u). When we drop all values in an interval with normalized utility n_i, then the utility for that output drops to $1 - n_i$. Using the relative frequency f_i, we can infer that dropping values of i will lead us to drop about $f_i \times 100$ percent of the tuples. Therefore, while utility of 100% is 1, the utility of $(100 - f_i \times 100)\%$ drops to $1 - n_i$. The utility values for the percentages in the range $(100, 100 - f_i \times 100)$ decrease linearly.

Consider the simple example in Figure 8. A histogram for two value intervals (Figure 8a) and a value-based QoS (Figure 8b) are provided. Our goal is to generate a loss-tolerance QoS from these two. Notice that the first value interval makes up 40% of the values and has a normalized utility of 0.12 (see Table 1). This means that when we drop 40% of the tuples, our utility drops from 1 to $1 - 0.12 = 0.88$. Therefore, the point (60, 0.88) is the inflection point at which the utility function and hence the slope changes. This leads us to the loss-tolerance graph in Figure 8c.

Determining the predicate for semantic drop. There is a pre-determined order for dropping value intervals imposed by their utilities. We capture this by keeping a sorted list of intervals in ascending order of their utilities. The cursor on the loss-tolerance QoS graph, say x, indicates how much of the data we al-

[2] For non-constant linear utility functions, we simply define a *chunk_size* and assume that each load shedding step will drop value ranges in multiples of a chunk. Hence, the values in the same chunk can be assumed to have the average utility of that piece of the utility function.

ready dropped. Each time we need to drop an additional k percent of the tuples, we locate the right entry in the interval table based on relative frequency of the intervals.

Consider the example we presented in Figure 8. Assume that x is 100, i.e., we have not dropped any tuples yet. Assume also that the algorithm has decided to drop 20% of the tuples. Interval [0, 50] has a percentage of 40% and it is the smallest utility interval that we have not dropped from before. We should drop the 20% from interval [0, 50], i.e., half the data in this interval [3]. The interval to be dropped will be $[0, 25)$ and the predicate for the semantic drop operator to be inserted will be $value \geq 25$. If we needed to drop 70%, then interval [0, 50] would not be sufficient. We would have to drop all of interval [0, 50] plus half of the interval [51, 100]. Hence, the predicate would be $value \geq 75$.

4.3.5 A Note on Join Processing

A join operator correlates tuples that appear at most a window apart in its inputs (to avoid blocking and infinite buffering). If the window size is not large enough, then the distribution of values in a given window cannot be assumed to be uniform. Hence, selectivity estimation becomes a problem. Our current implementation works for long window sizes, for which we can easily estimate changes to operator selectivities. We are currently investigating histogram-based techniques for handling short window sizes.

5 Experimental Environment

5.1 The Simulator

To experiment on our algorithms, we implemented a query network simulator using the CSIM18 Simulation Engine [14]. In the simulation, a round-robin CPU scheduling policy is used for operator scheduling. The model also includes a monitoring process, which wakes up periodically to evaluate the load to detect overloads. Additionally, a workload generator simulates each input data stream arriving at the query network based on the specified tuple inter-arrival times.

5.2 The Workload

To generate our test networks, we implemented a random query network generator [16]. Using this program, we first determine the number of queries and the number of operators for each query. We can adjust the relative proportions of operator types in the network by changing their probabilities of being chosen. We can also control the number of splits, i.e., points where an operator's output is connected to more than one operators' inputs. For each filter operator, we also generate a filter predicate and assign its selectivity. We assume uniform input data distributions to simplify the estimation of selectivities. The predicates are simple comparison predicates of the form $value > constant$. Each operator type has a fixed average cost.

We use streams of integers whose values are chosen randomly from the range $[0, 100]$. We generate streams with a constant mean inter-arrival time.

Our main QoS graph is the value-based QoS graph. Given that and the output value histograms, we generate the loss-tolerance QoS graph. We use two different value intervals in our experiments. The utility of the first interval is assigned randomly using a Zipf distribution while the second interval has a utility of 1.0. Using this distribution, we can control the skewedness of utility values on QoS graphs among multiple output applications.

For the experiments in the following section, unless otherwise stated, we use a network with five queries. This query network consists of 47 operators, 34 of which are filters, and the rest are unions. There is no sharing among the queries. The number of input streams is 18. The mean input rate for these streams vary between 10 and 2 units of inter-arrival time (i.e., 1/10 and 1/2 tuples per unit time), representing an excess load between +22% to +307% of the system capacity. The results reported for the Input-Random algorithm are averages of four different runs.

5.3 Algorithms

In addition to our random (**Random-LS**) and the semantic (**Semantic-LS**) load shedding algorithms, we also developed two other algorithms that effectively simulate classical admission control. Neither algorithm takes QoS information into account; rather, they shed load at the network inputs as necessary.

1. Input-Random. When an excess of load, ΔL, is detected, this algorithm randomly selects one input stream and sheds sufficient load on that stream to compensate for ΔL. If shedding all the data from the chosen input does not suffice, we select another input stream and repeat the same step for the remaining excess load. We also implemented a variant of this algorithm, namely **Input-Cost-Top**. It selects the most costly input stream (i.e., the one with the highest load share) instead of a random one.

2. Input-Uniform. Rather than choosing streams one at a time, this algorithm distributes the excess load evenly across all input streams, attempting to shed the same amount of load from each. If an input stream cannot provide its share of cycle gains, then the extra amount is distributed evenly to the other inputs. A variant of this algorithm, **Input-Cost-Uniform** distributes excess load across all input streams weighted by their costs.

[3] Any 50% of this interval could be dropped. However, we restrict our drop predicates to be range predicates. Therefore, we drop contiguous data values from beginning of the interval towards its end.

5.4 Utility Metrics

We use two metrics to evaluate the utility of a query output. *Tuple Utility* refers to the utility based on the loss-tolerance QoS graph. *Value Utility* refers to the utility based on the value-based QoS graph. The following formulas are used to compute these utilities:

Tuple Utility $= \frac{\sum_{i=1}^{k} u_i \times n_i}{\sum_{i=1}^{k} n_i}$, where

k: number of epochs [4]

n_i: number of tuples seen in epoch i ($\sum_{i=1}^{k} n_i = n$)

u_i: loss-tolerance utility of each tuple during epoch i

Value Utility $= \frac{\sum_{i=1}^{n} f'_i \times u_i}{\sum_{i=1}^{n} f_i \times u_i}$, where

f_i: relative frequency of tuples in interval i w/o drops

f'_i: frequency of tuples for interval i w/ drops relative to the total number of tuples

u_i: average value utility for interval i

The overall tuple (value) utility of a query network in the presence of multiple queries is computed by taking a sum of individual tuple (value) utilities for each query.

6 Experimental Results

6.1 Tuple Utility Loss

Our first experiment quantifies the loss in tuple utility for different load shedding schemes and for varying overload levels. Different load levels are characterized by different mean (arrival) rates for the input streams. The mean arrival rate is defined as the mean number of tuple arrivals per time unit at each input stream in the network.

Figure 9 shows that all algorithms are clearly negatively affected by increasing input rates. Because the system has fixed capacity, the percentage of tuples that need to be dropped increases with increasing input rates, thereby, decreasing the loss-tolerance utility of the system.

As expected, we observe that the two QoS-driven algorithms perform much better than the admission control algorithms. They follow a similar pattern, which is not surprising as they make their decisions based on the same loss-tolerance QoS graph (generated from the same value-QoS graph). Utility loss for Semantic-LS is observed to be less than that of Random-LS by a constant amount. This is a result of information in the output value histograms that Semantic-LS can exploit whereas Random-LS cannot. Since some of the input tuples are filtered out by the filter operators before they reach the outputs, they show up in the input streams but not in the output. Those are the tuples that should be dropped from the input in the first place. They provide cycle gain without causing any utility loss at the output. Semantic-LS can capture this with a predicate, but Random-LS is only allowed to drop random tuples. The constant utility difference between the two curves

[4] An *epoch* is the time period during which same percentage of tuples are being received.

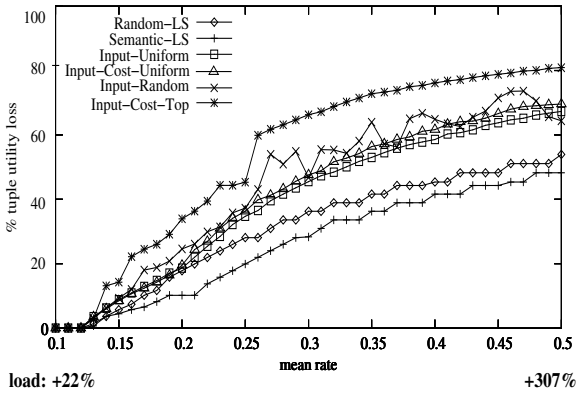

Figure 9: % Tuple Utility Loss vs. Mean Rate

amounts to this free "cycle gain" in Semantic-LS at no utility loss.

The Input-Random and Input-Cost-Top algorithms perform poorly compared to others, incurring relatively higher utility losses for all input rates.

Because Input-Uniform spreads tuple drops uniformly across the applications, for low excess loads, all applications can manage to remain at the top, relatively flat portions of their loss-tolerance QoS graphs. With increased load, as shown in Figure 9, this situation changes and we start observing the benefits of the QoS-driven algorithms over the Input-Uniform algorithm. Weighting drops from input streams based on their costs does not help much and almost performs exactly the same as Input-Uniform for low excess loads.

6.2 Value Utility Loss

We now investigate the loss in the value utility for different algorithms and input rates. Our goal is to quantify the semantic utility gains we can achieve by exploiting information present in the value-QoS graph. We compare our value-based algorithm against others that do not utilize such semantic information.

Figure 10 clearly demonstrates that the semantic drop algorithm significantly outperforms the other approaches in terms of the value utility metric. Note

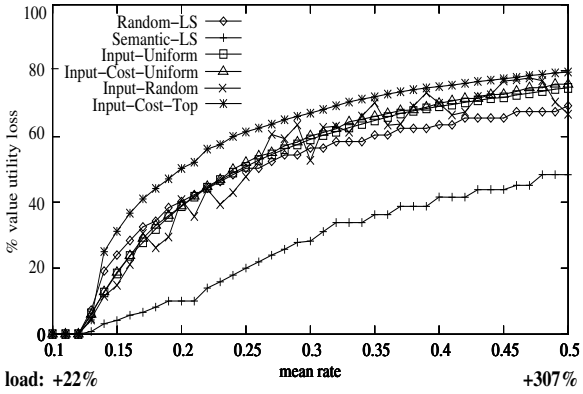

Figure 10: % Value Utility Loss vs. Mean Rate

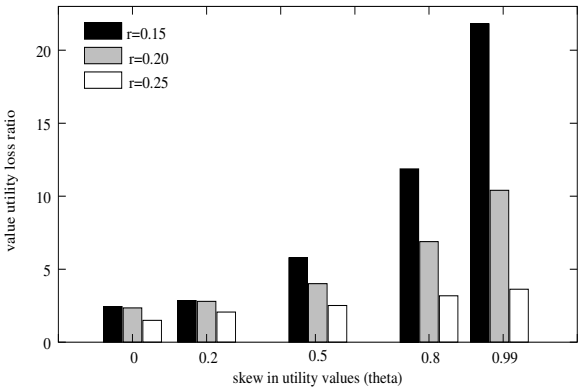

Figure 11: Value Utility Loss Ratio for Random-LS/Semantic-LS vs. Skew in Utility

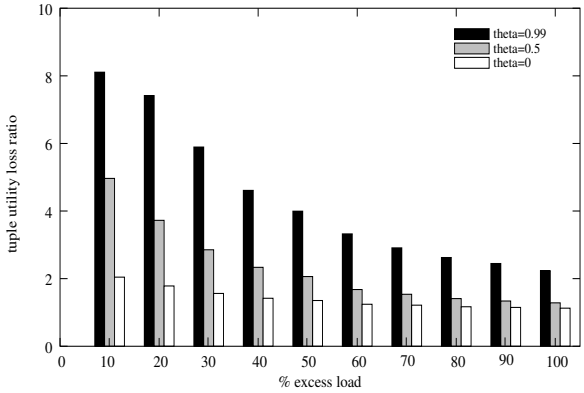

Figure 12: Tuple Utility Loss Ratio for Input-Uniform/Random-LS vs. % Excess Load

that comparing the other approaches among each other based on the outcome of this experiment would not be fair. Since those approaches drop in a randomized fashion, they must be compared on the basis of the tuple utility metric, as we presented in the previous experiment.

6.3 Effect of Utility Skew

In the scenarios we considered so far, the utility values for the first data intervals of different outputs were uniformly distributed. (Remember that the utility value of the second interval is taken as 1.0 for all the outputs.)

In order to characterize the impact of skewed utility distributions for different outputs, we devise a scenario where we inject skew to the utility values for the first data interval. We use a Zipf distribution to generate the utility values and use the Zipf parameter θ to control the skew. For low skew values, the utility values are more evenly distributed. For higher values, low utilities have higher probability of occurrence. Hence, we expect that with high skew, value-based dropping will perform much better than the randomized dropping approach. The rationale is that the latter will tend to drop high utility tuples, whereas the former will be able to fine-select the lower utility tuples for dropping.

We now demonstrate the effect of skew on utility loss for drops and filters for different values of mean input rates. Figure 11 illustrates our results. On the y-axis, we show the ratio of the value utility loss coming from the random load shedding algorithm to that coming from the semantic one. As we hypothesized, as the skew gets larger, the Semantic-LS algorithm gets increasingly more effective compared to the Random-LS algorithm. Interestingly, as the input rates increase, this effect tends to diminish. The reason is that when the rates are sufficiently high, the Semantic-LS algorithm also starts to drop tuples from the higher utility value intervals.

6.4 Effect of Sharing

For our last experiment, we used a network with 20 identical queries. Queries receive input from a single input stream and have one common operator. This operator's output is split into 20 arcs and routed to the query-specific operators on separate arcs to create a scenario of full sharing among the queries.

This experiment investigates the behavior of our algorithms against the admission control algorithms in the presence of shared operators (i.e., splits) in the query network. In this experiment, we compare the tuple utility loss of two algorithms: Input-Uniform, the best of the admission control algorithms as observed in the earlier experiments; and our Random-LS algorithm. Our comparison here is based on the loss-tolerance QoS rather than the value-QoS, to factor out the advantage of our algorithms due to using semantic information.

The bar chart in Figure 12 shows how tuple utility loss ratio between the two algorithms change as the amount of excess load in the system is increased. At each point of excess load, we present three different results. Each result is obtained using a different set of QoS graphs for the 20 queries. The loss-tolerance QoS graphs are generated from value-QoS graphs for which two data intervals are used: [0, 75] with a utility chosen from a Zipf distribution with skew parameter θ, and [76, 100] with utility 1.0. Hence, as the skew parameter theta increases, the uniformity of the QoS graphs decreases.

We observe that, as QoS graphs get more skewed, Random-LS performs better than the Input-Uniform algorithm. The reason is that our algorithm takes slopes of the QoS graphs into account while deciding where to shed load; whereas Input-Uniform always uniformly drops from the inputs. We further observe that the success of Random-LS against Input-Uniform starts to diminish as the amount of excess load gets to extremely high levels. This is because of the fact that, as the load increases to extreme levels, dropping from the inner arcs of the network does not suffice to

recover all extra the cycles. Our algorithm is forced to adopt the plans down in the LSRM, which eventually correspond to dropping at input points of the query network.

7 Related Work

The congestion control problem in data networks [18] is relevant to Aurora and its load shedding mechanism. Load shedding in networks typically involves dropping individual packets randomly, based on timestamps, or using (application-specified) priority bits. Despite conceptual similarities, there are also some fundamental differences between network load shedding and Aurora load shedding. First, unlike network load shedding which is inherently distributed, Aurora is aware of the entire system state and can potentially make more intelligent shedding decisions. Second, Aurora uses QoS information provided by the external applications to trigger and guide load shedding. Third, Aurora's semantic load shedding approach not only attempts to minimize the degradation in overall (accuracy-based) system utility, but also quantifies the imprecision due to dropped tuples.

Real-time databases, which support the enforcement of timing constraints on transactions, also require effective overload management. The typical solution is to abort transactions that are not expected to meet their deadlines, or perform feasibility analysis and reject any new transactions that are not deemed feasible [12]. This can be regarded as a simple form of load shedding where shedding units are individual transactions and shedding happens only at the system input boundaries. Aurora's load shedding approach is finer-grained since we are dealing with individual tuples and dropping can be performed at any place inside the operator network. Furthermore, Aurora's load shedding is driven by the ultimate goal of maximizing the aggregate accuracy-based QoS perceived by the applications, requiring reasoning about the data values and processing that takes place inside the Aurora network.

Load shedding is essentially an approximate query answering technique. Various techniques for producing approximate answers in exchange for faster execution have been studied in the database literature [3]. In the context of data streams, approximation has to be applied as data continues to arrive. The process is also more dynamic in that the degree of approximation has to be adjusted as the difference between supply and demand on resources changes. More recent work have explored approximate query processing techniques on data streams both for aggregation queries [10, 11] and sliding window joins [9, 13]. Unlike these work, we not only consider individual operations, but also complete query networks. These networks may be composed of a variety of operators and may serve multiple applications with shared operations. Furthermore, we address the load shedding process from end to end; i.e., from overload detection to overload resolution. We originally proposed to do semantic load shedding by filtering data that has lower utility to the applications [4]. Das *et al.* have a different view of semantic load shedding, concentrating on join processing and the semantic distance of the approximate answer [9]. Dropping tuples when input rate exceeds the service rate has also been discussed in rate-based evaluation of window joins [13]. In this work, the focus has been on random drops rather than semantic ones.

The STREAM system uses several approximation techniques on stream queries [15]. Synopses are used to reduce memory requirements of operators in a query plan; random sampling is used as a means of load shedding. We not only provide techniques for sampling using random drops but also provide semantic load shedding based on tuple values.

8 Summary and Conclusions

In this paper, we have described the general problem of shedding load in a data stream management system by discarding tuples that have the least impact on QoS. We discussed the way in which we detect an overload, our mechanism for discarding tuples (i.e., inserting drops), and a technique for determining the proper location and the right magnitude of the drops. The key feature of our solution is that most of the analysis concerning dropping strategies can be done statically and captured in a simple data structure. The dynamic load shedding process involves a very cheap use of the static information. This technique makes our solution practical and scalable.

Also, our solution does not depend on the details of the scheduling algorithm. Instead it assumes that any cycles that are recovered as a result of load shedding are used sensibly by the scheduler to relieve the congestion. This makes our solution much more general in that it works equally well with any good scheduler.

We have shown some experimental evidence that our load shedding techniques outperform basic admission control and its variants. We have also shown that while our probabilistic dropping technique can do fairly well, the method that takes tuple semantics into account can do even better. Our experiments also clearly show that as we increase the difference in importance between the most valuable tuples and the least valuable tuples, semantic load shedding produces more striking benefits. All of these results verify our intuitions. The most crucial observation of this paper is that it is possible to design a low-overhead mechanism for putting these concepts into practice in the context of a stream data manager.

In the future, we plan to study ways to generalize these methods to include more complex operators supported by Aurora. These include stateful operators like aggregation. The effect of pushing drops past these operators has to be considered. Furthermore,

any operator that produces fewer output tuples than it receives as input could potentially be used for load shedding, such as an aggregate. We will also explore such alternative techniques.

In the current study, we have focused on load shedding to reclaim processor cycles. In many stream-oriented applications, cycles are not the limited resource. Often things like bandwidth or battery power will be the resources that must be conserved. Load shedding via dropping of tuples has an important role to play in these environments as well. We intend to investigate ways in which we can design techniques similar to the ones discussed here that can work for other kinds of resource management.

Finally, in stream processing applications, operators must avoid blocking because blocking can obviously destroy latency. Thus, it is reasonable to have an *operator timeout* [2] before it is certain that a semantic condition is reached (e.g., emit an average before all the stock prices for the 3pm report have been received). Once the timeout has happened, further tuples in the previously terminated category (e.g., 3pm reports) have no benefit and can therefore be discarded. This is a type of pre-planned load shedding that is needed when the data rates are too slow and tuples are delayed. We are interested in exploring the relationship between load shedding for high loads and load shedding (timeouts) for low loads. It would be interesting to see if there is some commonality that could be exploited.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R.Yan, and S. Zdonik. Aurora: A Data Stream Management System. In *ACM SIGMOD Conference*, page 666, San Diego, CA, June 2003.

[2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, to appear.

[3] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[4] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB Conference*, pages 215–226, Hong Kong, China, August 2002.

[5] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, pages 269–280, Asilomar, CA, January 2003.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR Conference*, pages 257–268, Asilomar, CA, January 2003.

[8] C. L. Compton and D. L. Tennenhouse. Collaborative Load Shedding for Media-Based Applications. In *Intl. Conf. on Multimedia Computing and Systems*, pages 496–501, Boston, MA, May 1994.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *ACM SIGMOD Conference*, pages 40–51, San Diego, CA, June 2003.

[10] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams Databases. In *ACM SIGMOD Conference*, pages 13–24, Santa Barbara, CA, May 2001.

[11] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *VLDB Conference*, pages 79–88, Roma, Italy, September 2001.

[12] J. Hansson and S. H. Son. Overload Management in Real-Time Databases. In *Real-Time Database Systems: Architecture and Techniques*, pages 125–140. Kluwer Academic Publishers, 2001.

[13] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *IEEE ICDE Conference*, Bangalore, India, March 2003.

[14] Mesquite Software, Inc. CSIM18 Simulation Engine. http://www.mesquite.com/.

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, pages 245–256, Asilomar, CA, January 2003.

[16] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. Technical Report CS-03-03, Brown University, Computer Science, February 2003.

[17] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding on Data Streams. In *ACM Workshop on Management and Processing of Data Streams (MPDS'03)*, San Diego, CA, June 2003.

[18] C. Yang and A. V. S. Reddy. A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. *IEEE Network*, 9(5):34–44, 1995.

[19] Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *CIDR Conference*, pages 233–244, Asilomar, CA, January 2003.