Load Shedding in a Data Stream Manager

Nesime Tatbul, Ugur Cetintemel, Stan Zdonik,
Mitch Cherniack, Michael Stonebraker

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-03-03
March 2003

Load Shedding in a Data Stream Manager

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

Abstract

A Data Stream Manager accepts push-based input 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 without too much 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 [7, 9, 17, 8, 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 [7, 2, 1] is one such new system that is being developed at Brandeis University, Brown University, and M.I.T. The Aurora data stream manager uses application-level semantics to make intelligent decisions about resource allocation. Typically, the resources in question include processor cycles, memory, and bandwidth [9].

Our designs have been influenced heavily by several specific applications. In particular, we have studied an application that performs remote triage over a battlefield given a large number of soldiers with bio-sensors 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 [1].

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. In these cases, the only 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 *unimportant* data.

The work described in this paper concentrates on the processor as the limited resource and exploits application-specific Quality-of-Service (QoS) information when making load shedding decisions. We assume that QoS is specified separately for each application. It 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.

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

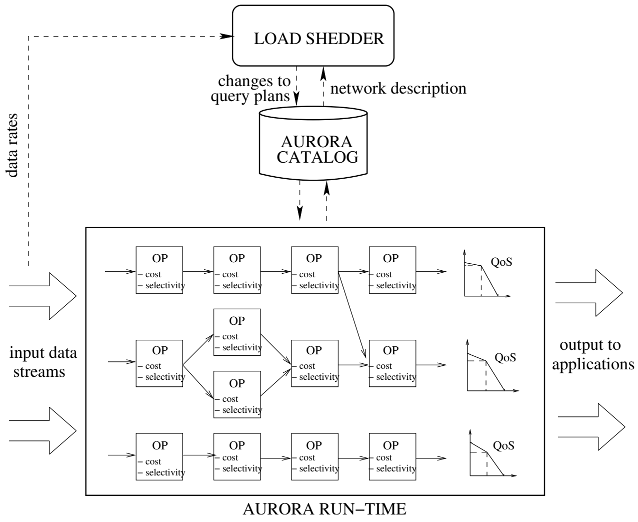

Figure 1: Aurora Architecture and Load Shedder

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 balance these two effects so that we end up gaining more than we lose.

Load shedding is not a new idea. The fields of operating systems, networking [18], and multimedia [10] have 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 (value-based QoS). We also have a much more limited 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., selectivities and operator costs) about the processing elements in a way that is difficult in other contexts.

This paper explores scalable and efficient load shedding techniques for large processing networks. 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 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.

We can not discuss the Aurora data model and the operator set in full due to space limitations. We will however, briefly mention only the operators we consider in the framework of this work. 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 in practice that 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 [7]. The latency graph indicates how utility drops as an answer is delayed. The value-based graph shows which values of the output 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 a statistics about the distribution of values in the output. An important assumption we make about QoS functions of the types (1) and (3) is that they are concave: utility decrease speeds up as the QoS metric increases.

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. The way the scheduler works is a topic for another paper. We will make the assumption here, though, that the scheduler will always use available processor cycles in an optimal way with respect to improving QoS. This assumption is relevant to the load shedding approach taken in this paper.

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 additional load reducing operators into a running network. A load reducing operator must produce fewer output tuples than it gets as input. We call these *drop operators* (or just *drops*). In this paper, we consider two fundamental types of drop operators.

1. **Random Drop.** This operator takes a single parameter p that expresses the percentage 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 also decide which tuples to discard (i.e., the form of the predicate).

The general problem can be stated as follows. We are given a processing 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 $Load(N(I)) > C$. 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:

$$min\{U_{accuracy}(N(I)) - U_{accuracy}(N'(I))\}, \text{ and}$$

$$Load(N'(I)) < H * C$$

$U_{accuracy}$ is the utility that is measured from the loss-tolerance QoS graph, and the constant H is the *headroom factor* which indicates the percentage of processing resources that can be used at steady state [1]. $U_{accuracy}(N(I))$ represents the measured utility when there is no load shedding (i.e., there are no inserted drops). Thus, $U_{accuracy}(N(I)) - U_{accuracy}(N'(I))$ is the loss of utility introduced by load shedding. It is this quantity that we want to minimize.

We assume that the query network can be very large. We further 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. We need not worry about how the cycles might be used. This decoupling of the scheduler and the load shedder is an important simplification. By making this assumption, the load shedder will not need to consider latency QoS directly.

[1] The headroom is reserved to guard the system against thrashing.

4 The Algorithms

We have developed a scalable load shedding algorithm with two variants, one for inserting random drops (namely, *drop-based load shedder*) and one for inserting semantic drops (namely, *filter-based load shedder*). This section first gives the overall structure of the algorithms and then provides some detail on how we build and use the fundamental data structures to make the three important decisions in load shedding.

4.1 Algorithm Structure

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

The first major action in this loop is to inspect the current load of the query network. A monitoring process periodically checks the load level compared to the known system capacity. We will call this the *load evaluation step*.

The load evaluation step first figures out the current load of the network. For this, we do a simple calculation that involves the current input rates, and the operator costs and selectivities. This calculation has two pieces: the load created by the current stream rates, which we call the *stream load (S)*; and the load due to any queues that may have built up since the last load evaluation step, which we call the *queue load (Q)*. The *total load (T)* is the sum of the two pieces.

If $T > H * C$, then the system is overloaded and load must be shed by inserting drops into the query network. On the other hand, if $T < H * C$, 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 static table called the *Load Shedding Road Map (LSRM)* that is constructed off-line once for any given network. The LSRM captures a sequence of plans for achieving progressively higher levels of load reduction. This means that determining the Drops to insert can be reduced to a series of table lookups. The intuition behind the table is whenever we want to reduce the load by a particular amount L, there will be a single *best* way to do that. The best way to reduce the load by L is to use the static plan generated for L. Successive rows of the table increase the amount of load that is shed. Hence, the very last entry in the table provides the maximum level of savings while shedding all data from the network. Figure 2 illustrates the basic structure of the LSRM.

At run-time, when an overload is detected, we use the LSRM to find a plan for recovering $T - H * C$ processor cycles. Each row in the table has an indication of cycle savings. Each table entry also contains a plan for where to place drop operators together with their drop amounts. Whenever load needs to be shed, all we need to do is to search the LSRM for the right amount of cycle savings. Once we locate the entry that saves enough cycles, we adopt the plan stored in that row.

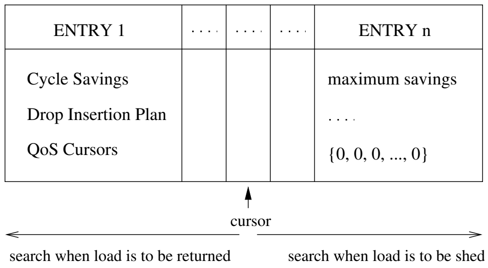

Figure 2: Load Shedding Road Map (LSRM)

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 headroom bound and there are still Drops in the network (cursor is non-zero), then the search can be done in the reverse direction.

4.2 Load Evaluation using Load Coefficients

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

Each input to the network has an associated load coefficient that can be statically computed. The load coefficient represents the number of processor cycles required to push a single input tuple through the network to the outputs. Consider the input stream I given in Figure 3, running through a simple query of n pipelined operators and producing an output stream O.

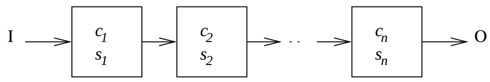

Figure 3: Calculating load coefficients for stream I

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

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

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 $L * r$ (cycles per time unit). If there are m inputs, we can compute the total *stream load* S as

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

Likewise, each operator in the network has a pre-computed load coefficient which represents the number of processor cycles required to push a single tuple from that operator to the outputs [2]. We use this coefficient to calculate the contribution of each operator to the total *queue load* Q. First, we define a global system parameter called $MELT\_RATE$. This parameter is a lower bound on how fast we want to melt (i.e., shrink) the queues. It represents the queue length reduction per time unit as a fraction of the current queue length. An operator i with a load coefficient L_i and queue of length q_i contributes to the total queue load by $MELT\_RATE * L_i * q_i$. The queue load of all operators are summed to calculate Q. Essentially, by having $Q > 0$, we cause an "overshedding" in the network to melt the queues. After all the queues are melted, the load shedder will discover that the current level of dropping can be reduced causing some of the Drops to be removed (or selectivity increased).

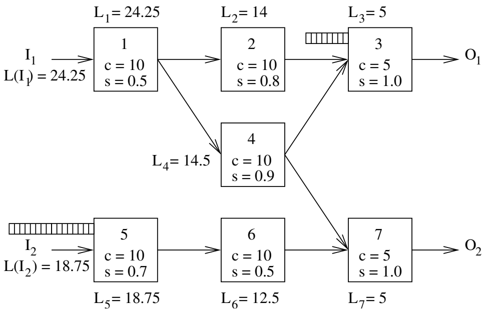

Figure 4: Example query network with load coefficients

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 two input streams as well as load coefficients for each operator in the network. 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 stream load of this network becomes $S = 24.25 * 10 + 18.75 * 20 = 617.5$ cycles per time unit. Additionally, if it turns out that operator 3 has a queue of 50 tuples and operator 5 has a queue of 100 tuples given that $MELT\_RATE = 0.1$, then the queue load of this network becomes $Q = 0.1 * (5 * 50 + 18.75 * 100) = 212.5$. Thus, the total current load of the network adds up to $T = S + Q = 617.5 + 212.5 = 830$ cycles per time unit. If this value is over the system's available capacity, then the load shedder must be invoked.

This section concludes our coverage of the load evaluation step. We now describe how we construct and use the LSRM in detail.

4.3 The Load Shedding Road Map (LSRM)

This subsection describes our techniques to make the last two decisions in load shedding: where in the query network and how much load to shed.

[2]Note that the formula for operator load coefficients is the same as formula (1), except that index i of the outer sum starts from the operator index.

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 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.

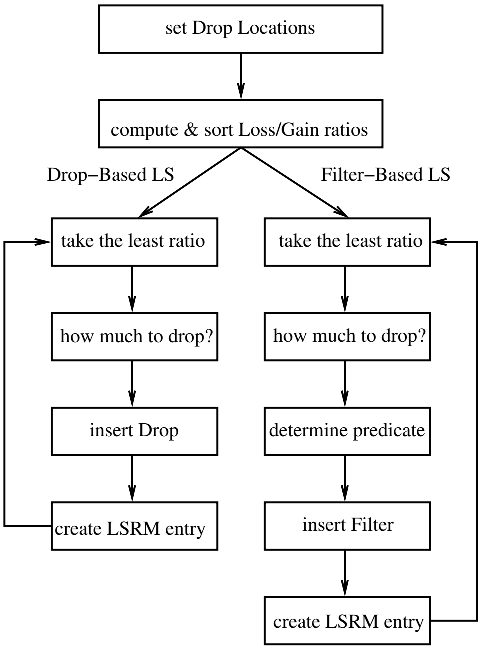

Figure 5: LSRM Construction

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

4.3.1 Drop Locations

The basic principle behind the construction of the LSRM is that each entry in the table must guarantee that the utility sacrificed by adopting its DIP is the minimum that could be achieved 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 must be maximized. Hence, when creating a new LSRM entry, the criteria for choosing the best DIP among many possible ones is to select the one with the minimum *Loss/Gain* ratio. (We further discuss

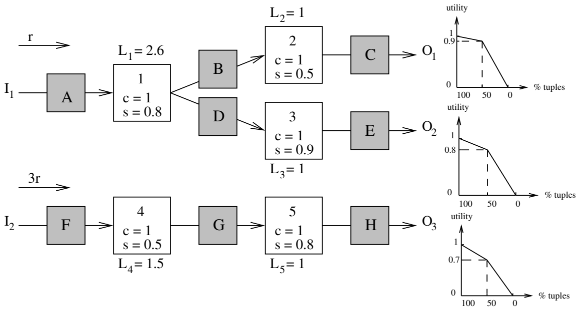

Figure 6: Example query network with candidate drop locations

this ratio in the next subsection.)

The search space for all possible DIPs may seem very large at first. However, it can be dramatically reduced by exploiting several heuristics. We will discuss these using the query network in Figure 6. 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 show candidate locations for drops.

First, consider a query plan which has no sharing with others. In a network that contains such a plan, a drop insertion to *any* location in that query does not affect the other queries. Hence, the utility loss is only observed at the output that belongs to that query plan. For example, in Figure 6, the bottom-most query has no sharing with the top two. Furthermore, it is best to insert drops as early in the query plan as possible. The closer a drop operator to the inputs, the more cycles it saves. Therefore, if we ever want to insert drop to a query plan which has no sharing with other query plans, the location for that drop is always at the beginning of the query plan. Hence, the small box marked with F is the location of the 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, which further lead to different outputs. This is observed at operator 1 in Figure 6. Any tuple coming out of operator 1 would be routed to both outputs O_1 and O_2. Hence, a drop operator inserted upstream from operator 1 would affect both outputs. Therefore, for the top subnetwork in Figure 6, there are three possible ways that the output utilities are affected by a drop: we may lose utility at only O_1 if we insert the drop at B; or at only O_2 if we insert it at D; or at both O_1 and O_2 if we insert it at A. As a result, if we ever want to insert drops in a query network which has sharing, the possible locations where we can insert them are determined by the different output combinations that are affected. Note that our previous rule of thumb still holds making C, E, G, and H not desirable for inserting drops.

Thus, the possible Drop locations are at Aurora inputs or on arcs that follow a split (i.e., A, B, D, F) . For each such location, we need to compute the Loss/Gain ratios as described in the next section.

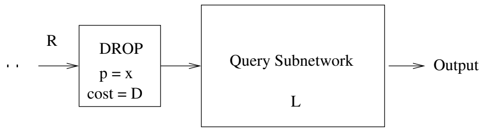

Figure 7: Insertion of a drop

4.3.2 Loss/Gain Ratios

Loss/Gain ratios allow us to sort the possible drop locations by their desirability as candidates. In this ratio, *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 the simple example in 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 percentage, D is the cost of the drop operator itself, and L is the load coefficient of the subnetwork. The gain from inserting this drop is:

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

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

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

Also, to guarantee $G(x) > 0$, i.e., the gain from drop insertion is more than its cost, we must ensure (per above formula for $G(x)$) that $x > \frac{D}{L}$.

Each drop amount must be in increments of a $STEP\_SIZE$ parameter to guarantee that the drop is worth inserting. For this purpose, we define a $STEP\_SIZE$ parameter as $STEP\_SIZE > \frac{D}{min\{L\}}$. This is a conservative bound where $min\{L\}$ is the minimum operator load coefficient in the network. We use this minimum value so that the chosen $STEP\_SIZE$ will work for all locations in the network.

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

Loss is to be determined by using the loss-tolerance QoS graph. In our example, the QoS graph of location B, D, F are the same as that 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 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 guaranteeing that each new entry is built using the smallest possible Loss/Gain ratio. Both the drop-based and the filter-based load shedding algorithm use these ratios in LSRM construction. We next describe in detail how LSRM entries are created for these algorithms.

4.3.3 LSRM for Drop-Based Load Shedding

LSRM entries are created incrementally with each new entry choosing the drop location with the next smallest Loss/Gain ratio. This processing involves two main steps:

Determining how much to drop. First, the drop parameter p which denotes the percentage of tuples to be dropped has to be determined. The drop location that we are processing has a certain Loss/Gain ratio, outputs affected by it and the corresponding QoS graph. This Loss/Gain ratio accounts for a particular range of percentages on the loss-tolerance QoS graph where the curve is linear. For example, in Figure 6, F's smaller ratio covers the range from 100% to 50%. Therefore, the drops that we will create have to cover this range. As mentioned in the previous subsection, each drop amount is applied in increments of a $STEP\_SIZE$. The value for this parameter affects the granularity of the entries of the LSRM. It could be chosen as large as the width of the range (in our example, $100-50=50$). However, it is usually desirable to choose this value smaller so that the LSRM provides a larger choice of cycle savings (i.e., more entries). This further provides for a fine-tuned load shedding. If the QoS has a percent delivery cursor at x, we will choose the new drop amount such that the cursor will move to $x - STEP\_SIZE$. The new drop amount will be $p = 1-(x-STEP\_SIZE)/x$.

Placing the drop in the network. After p is determined, a drop operator can be created for the designated drop location. We cannot insert this drop directly because there may be other drops in the same query plan as the new one that were created during an earlier step. In this case, a combination of their drop amounts will determine the number of tuples dropped at the output. This may require that we modify p in some of these drops to achieve the desired tuple percentage at the output. This is because the LSRM is constructed without any knowledge of how the drops will or will not interact.

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. However, if $l > k$, then B's percentage must be reduced to produce a total percentage of l at the output. In other words, drops' percentages must be adjusted such that the output has a percentage drop of $max(k,l)$. This way, the drop insertion decision made in an earlier step is maintained while the new decision is being applied. Our solution to this problem is to pretend that we are inserting the new drop at the output. We then push it towards its desired drop location, fixing the percentages of the drops that are encountered along the way.

All drop operators created since the beginning with their modified percentages form the DIP of our new LSRM entry. Additionally, we compute CSC for each stream that sends tuples to the drops in the DIP. Finally, QoS cursors of the affected outputs are advanced and stored in PDC. The next iteration of the loop takes the current DIP and builds upon it. This way, each LSRM entry has more savings than the previous one.

4.3.4 LSRM for Filter-Based Load Shedding

LSRM entries for the filter-based load shedder are created in almost the same way as for the drop-based 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. Furthermore, the predicate must be filtering out the least valuable data values.

Determining how much to drop. We developed a technique which enables us to follow the same method to determine the drop amount, as the one described for the drop-based 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 and hence how much and at which locations to drop data. We now explain this technique with an example.

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 interval have a constant utility, i.e., the assigned linear function for interval i is in the form of $y = u_i$. Assume also that we have at hand the output data value histograms for each of the intervals specified in the QoS graph. Such histograms are commonly created in conventional DBMS's to assist in query processing. Each such histogram shows the relative frequency of each 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, $w_i = u_i * f_i$
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_i value in ascending order and store them in a table together

interval	u_i	f_i	w_i	n_i
0-50	0.2	0.4	0.08	0.08/0.68 = 0.12
51-100	1	0.6	0.6	0.6/0.68 = 0.88

Table 1: Example value intervals

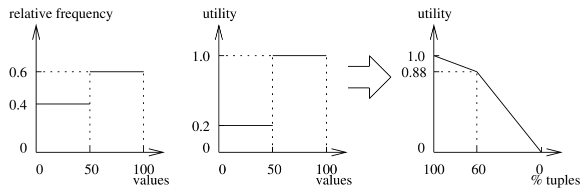

Figure 8: Derivation of the loss-tolerance QoS

with their u_i, f_i, and n_i values. Table 1 illustrates such a table with two value intervals.

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_i). When we drop all values in an interval with normalized utility n_i, then our utility 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 $f_i * 100$ percent of the tuples. Therefore, while tuple utility of 100% is 1, the tuple utility of $(100 - f_i * 100)$% drops to $1 - n_i$. The utility values for the percentages in the range $(100, f_i * 100)$ decrease linearly.

Consider the simple example in Figure 8. An histogram for two value intervals and a value-based QoS are provided. Our goal is to generate a loss-tolerance QoS from these two. Notice that the first value interval makes up the 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 the Figure 8.

Note that this method is straightforward for constant interval utilities. For non-constant linear utility functions, we simply define a *chunk_size* and assume that each load shedding step will drop data amounts 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 functions.

Determining the predicate for filter. There is a pre-determined order for dropping value intervals imposed by their utilities. We guarantee 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 already 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 20% from interval [0, 50], i.e., half the data in this interval [3]. The interval to be dropped will be $0 \leq value < 25$ and the predicate for the filter 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 filter predicate would be $value \geq 75$.

Placing the filter in the network. This step is almost the same as the one for drop-based load shedding. One additional issue here is that, when a filter is inserted to a query network, selectivities of the operators downstream from this filter may change. To handle this, we sweep the network once starting from the new filter to update the selectivities.

4.3.5 Query Networks with Joins

Due to space limitations, we can only briefly touch on the issues that join operators introduce for load shedding. Sampling techniques for approximate join processing has been explored in previous work [3] and is known to be a hard problem. Our drop-based load shedding technique is also a sampling-based technique and does not work well with joins. However, our filter-based load shedding technique can be applied to query networks with equi-joins. The join predicate must be on a common attribute of two streams and value-based QoS graphs must be defined on the value range of this attribute. If the join is a windowed one [4], then additional issues may arise.

When a filter is pushed across a windowed join, the selectivity of operators downstream from the join (including the join itself) may change. This change may be arbitrary if the window size is small. We might know that input streams to the join have a uniform value distribution. However, join correlates tuples that appear at most a window apart in its inputs (to avoid blocking and infinite buffering). If this window is not large enough, then the distribution of values in a window can not 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 do have a design to handle short windows which relies on keeping histograms to maintain selectivities. We hope to present this approach and its results in a future paper.

5 Experimental Setup

5.1 The Simulator

To experiment on our algorithms, we implemented a query network simulator using the CSIM18 Simulation Engine [16]. We defined our main resource, the CPU, as a CSIM facility with a round robin service policy. With this policy, each process that requests to use the CPU is given a constant time slice in turn.

We modeled each query operator in the network, which has one or more input tuples in its input queue, using a CSIM process. Processing of a tuple by an

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

[4] We consider time-based windows here.

operator is simulated by allowing the CSIM process to use the CPU facility for an amount of time equal to the cost for that operator. Operator semantics are applied to the input tuples, and the output tuples are queued at the inputs of their successor operators.

We also have a monitoring process, which wakes up periodically to evaluate the load. Additionally, we created one process to simulate each input data stream arriving at the query network. The input rates are simulated by controlling the tuple inter-arrival time.

5.2 Workload Generation

Query Networks. To generate our test networks, we implemented a random query network generator. First, we decide on how many applications we want to deliver data to (i.e., the number of queries in the network), say N. Then, we decide on the number of operator levels each query will have (i.e., the depth of the tree representing the query), say L. We then create a two-dimensional grid of operators of size Lx$(2^L * N)$ [5]. To create a query, we first create its very last operator. This operator is placed in the last column of the grid. Each column of the grid forms a level of operators from which we can choose a random operator to connect to the previous level. Starting from the output end, when we process all the columns backwards until we reach the input end. At this point a query has been generated. Lastly, we attach the query to one or more input streams.

We can adjust the relative proportions of operator types in the network by changing their probabilities of being chosen at each level. 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 cost. We assume that drop operators are relatively cheaper than others.

Streams. 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.

Quality of Service. 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. Interval [0, 50] has a lower utility than interval [51, 100], which always has a utility of 1. The utility of the first interval is assigned randomly using a Zipf distribution. Using this distribution, we can control the skewedness of utility values relative to the second (higher) utility interval.

5.3 Evaluating Utilities

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 $= \frac{\sum_{i=1}^{k} u_i * n_i}{\sum_{i=1}^{k} n_i}$, where

k: number of epochs [6]

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 $= \frac{\sum_{i=1}^{n} f_i' * u_i}{\sum_{i=1}^{n} f_i * u_i}$, where

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

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

u_i: average value utility for value interval i

Overall utility of a query network in the presence of multiple queries is computed by taking a sum of individual utilities for each query.

6 Experiments

6.1 Admission Control Algorithms

In addition to our drop-based (**Drop-LS**), and the filter-based (**Filter-LS**) load shedding algorithms, we have 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.

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 remainder 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.

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.

6.2 Experimental Data

For the experiments presented in the following subsections, we used a network with 5 queries. The 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 122% and 407% of system capacity, 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] This is the maximum size of the grid when it only consists of binary operators.

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

6.3 Tuple Utility Loss

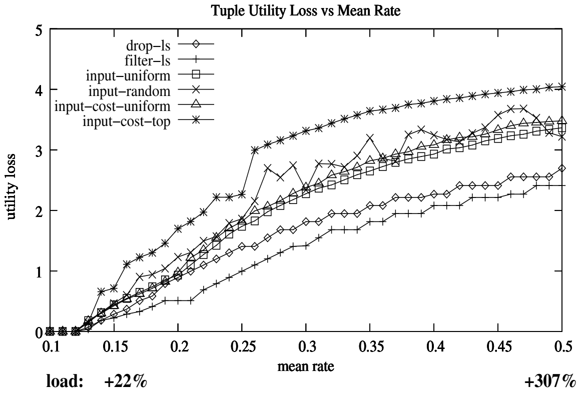

Figure 9: Tuple Utility Loss vs. Mean Rate

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 rates for the input streams. Mean rate is defined as the mean number of tuple arrivals per time unit at each input stream in the network.

As shown in Figure 9, 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 tuple-QoS graph (generated from the same value-QoS graph). Utility loss for filter-ls is observed to be less than drop-ls by a constant amount. This is a result of information in the output value histograms that filter-ls can exploit whereas drop-ls can not. 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 save cycles without causing any utility loss at the output. Filter-ls can capture this with a predicate, but drop-ls is only allowed to drop random tuples. The constant utility difference between the two curves amounts to this free cycle gain in filter-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. An interesting result is that, for smaller input rates, the input-random algorithm performs comparably to the input-uniform algorithms.

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 tuple-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 did not help much and almost performs exactly the same as input-uniform for low excess loads.

6.4 Value Utility Loss

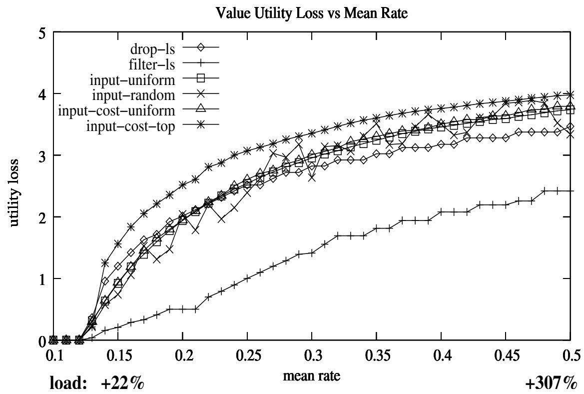

Figure 10: Value Utility Loss vs. Mean Rate

We now investigate the loss in value utility for different algorithms and different levels of input rates. Our primary goal is to quantify the semantic utility gains we can achieve by exploiting information present in the value-QoS graph. The previous set of experiments did not consider the value-QoS graph. We, therefore, 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. Note that drop-ls does not seem to provide any benefit over the admission control algorithms in this case. In fact, it seems to perform somewhat worse than the others in lower excess loads. This interesting behavior could be a result of using the value-based QoS to derive the loss-tolerance QoS which is then used by drop-ls. This derivation assumes that the least utility value intervals will be dropped the first. However, drop-ls drops random tuples. It would thus be more fair to use the tuple utility loss (as opposed to value utility) to compare the algorithms that drop randomly rather than semantically (i.e., admission control algorithms and drop-ls) as we did in the previous experiment.

6.5 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 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 data interval 1. 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

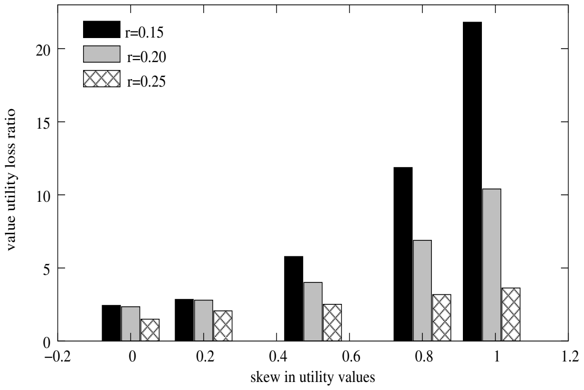

Figure 11: Value Utility Loss for Drop-LS/Value Utility Loss for Filter-LS vs. Skew in Utility

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 drop-based 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 drop-based algorithm to that coming from the filter-based algorithm. As we hypothesized, as the skew gets larger, the filter algorithm gets more effective compared to the drop algorithm. Interestingly, as the input rates increase, this effect tends to diminish. The reason is that when the rates are sufficiently high, the filter algorithm will too have to start dropping tuples from the higher utility value interval.

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 [13, 15]. 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 happens inside the Aurora network.

Aurora load shedding is also related to approximate query answering, data reduction, and summary techniques (e.g., [14, 6, 4]). These techniques trade result accuracy for efficiency of query execution. Whereas most work addressed traditional one-time database queries over stored data, more recent work also investigated techniques for providing approximate answers to queries over data streams. [11] describes histogram-based techniques to compute approximate answers correlated aggregation queries. [12] describes wavelet-based techniques to compute data summaries, which are then used to provide approximate answers to ad-hoc queries. [5] talks about the problem of sampling data from a moving window of recent items over data streams.

By throwing away data probabilistically, Aurora bases its computations on sampled data, effectively producing approximate answers in a way that is functionally equivalent to data sampling. A similar sampling idea for a data stream management system is sketched in the context of the STREAM project [17]. The unique aspect of our approach is that our sampling is driven by QoS specifications that relate result tuple values to utility.

8 Summary and Conclusions

In this paper, we have described the general problem of shedding load in a stream data management system by discarding tuples that have a the smallest 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 highly 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 will study ways to generalize these methods to include more complex operators as is supported by the Aurora system. This includes stateful operators like aggregation. The effect of pushing drops past these operators has to be considered.

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 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 to 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. Çetintemel, 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 *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, CA. to appear.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. Technical Report CS-02-10, Brown University, Computer Science, August 2002.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, Philadephia, PA, June 1999.

[4] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 574–576, Philadephia, PA, June 1999.

[5] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Symp. on Discrete Algorithms (SODA 2002)*, 2002.

[6] 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.

[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 *VLDB*, pages 215–226, Hong Kong, China, August 2002.

[8] 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: Conf. on Innovative Data Systems Research*, pages 269–280, Asilomar, CA, January 2003.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR: Conf. on Innovative Data Systems Research*, pages 257–268, Asilomar, CA, January 2003.

[10] 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, 1994.

[11] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams Databases. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, Santa Barbara, CA, May 2001.

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

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

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[15] B. Kao and H. Garcia-Molina. An Overview of Realtime Database Systems. In W. A. Halang and A. D. Stoyenko, editors, *Real Time Computing*. Springer-Verlag, 1994.

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

[17] 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: Conf. on Innovative Data Systems Research*, pages 245–256, Asilomar, CA, January 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: Conf. on Innovative Data Systems Research*, pages 233–244, Asilomar, CA, January 2003.