Load Shedding in a Data Stream Manager

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

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-02-12
December 2002

Load Shedding in a Data Stream Manager

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

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

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

A Data Stream Manager (DSM) 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, 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 the query plan as required by the current load. We examine two types of Drops: the first drops a fraction of the tuples in a random 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 present an algorithm for accomplishing this and experimental evidence that it can react quickly and can bring the system back into the useful operating range.

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. These systems provide the same kind of infrastructure to stream-based applications that database management systems have provided for data processing applications. A stream data manager uses application-level semantics to make intelligent decisions about resource allocation. Typically, the resources in question include storage, processor cycles, and bandwidth.

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 ten to twenty bio-sensors embedded in each soldier's uniform. We have also worked with an application that collects reconnaissance data from an advanced airplane about the positions of enemy units and that is concerned with the intelligent dissemination of 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 this data to detect critical situations. During these events, the data rates can increase, and it is much more important that relevant data get delivered in a timely fashion. In the Aurora system, an expression of the relative importance of output events is captured as a QoS specification. The system must always try to maximize the total delivered QoS. During times of stress, the input rates can create a load that exceeds the system capacity. In these cases, the only way to operate within the QoS bounds is to shed some of the load. In general terms, load shedding is the process of dropping out excess load from the system. In a query network of operators, one obvious way of load shedding is to drop out data values from the data processing network. Dynamically choosing where in the network to shed load and how much load to shed is a challenging problem. This paper explores scalable load shedding techniques for large processing networks.

The work described in this paper concentrates on the processor as the main resource with the overload problem and exploits application-specific QoS information for shedding excess load. We believe that QoS is specified separately for each application. It describes the relationship between various characteristics of an answer and the usefulness (i.e., utility) of that answer. 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. Some values are more interesting than others.

It should be noted that while dropping tuples will certainly

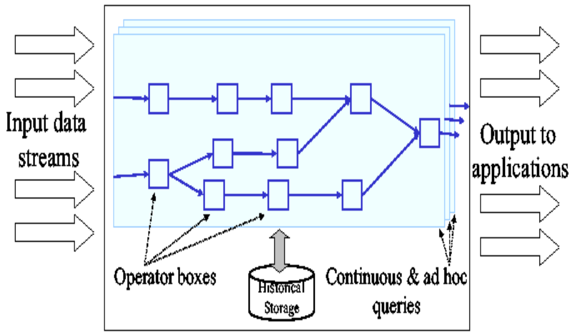

Figure 1: Aurora Model

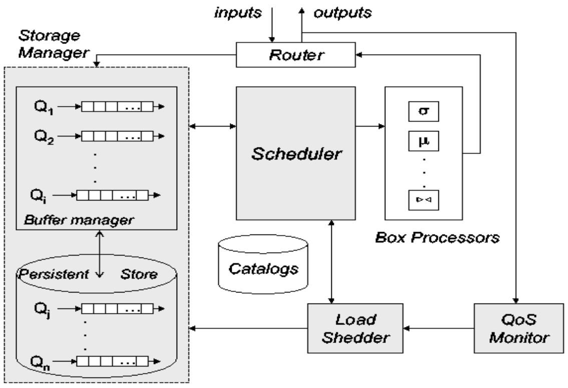

Figure 2: Aurora Architecture

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

The next section is a very brief overview of the Aurora system. Section 3 presents a detailed discussion of the problem. Section 4 contains the description of our load shedding algorithm. 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. Users build queries out of a small set of operators (a.k.a. boxes). The current implementation provides a user interface for tapping into pre-existing inputs and network flows and for wiring boxes together to produce answers at the outputs. While it is certainly possible to accept input as declarative queries, we feel that for a very large number of such queries, the process of common sub-expression elimination is too difficult. Figure 1 illustrates the basic model of Aurora.

A simple stream is a potentially infinite sequence of tuples that all have the same stream ID. An arc carries multiple simple streams. This is important so that simple streams can be added and deleted from the system without having to modify the basic network. A query, then, is a sub-network that ends at a single output and includes an arbitrary number of inputs. Boxes can connect to multiple downstream boxes. All such path splits carry identical tuples. Multiple streams can be merged since some box types accept more than one input (e.g., Join, Union). We do not allow any cycles in an operator network.

A more detailed system architecture of Aurora is given in Figure 2. This figure shows the architecture of the basic Aurora run-time engine, which processes stream flows through a potentially large workflow diagram. Here, inputs from data sources and outputs from boxes are fed to the router, which forwards them either to external applications or to the storage manager to be placed on the proper queue. The storage manager is responsible for maintaining the box queues and managing the buffer. Conceptually, the scheduler picks a box for execution, ascertains what processing is required, and passes a pointer to the box description (together with a pointer to the box state) to the multi-threaded box processor. The box processor executes the appropriate operation and then forwards the output tuples to the router. The scheduler then ascertains the next processing step and the cycle repeats. The QoS evaluator continually monitors system performance and activates the load shedder when it detects an overload situation and poor system performance. The load shedder then sheds load till the performance of the system reaches an acceptable level. The catalog in Figure 2 contains information regarding the network topology, inputs, outputs, QoS information, and relevant statistics (e.g., selectivity, average box processing costs), and is essentially used by all components.

As stated above, tuples arrive at the input and are queued for processing. The scheduler selects a box with waiting tuples and executes that box 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 very relevant for the load shedding approach taken in this paper.

Aurora supports persistent storage in two different ways. First, when box queues consume more storage than available RAM, the system will spill tuples that are less likely to be needed soon to secondary storage. Second, ad hoc queries can be connected to (and disconnected from) any arc for which a connection point has been defined. A connection point stores historical portions of a stream that has flowed on the arc. For example, one could define a connection point as the last hour's worth of data that has been seen on a given arc. Any ad hoc query that connects to a connection point has access to the full stored history as well as any additional data that flows past while the query is connected.

Each output is supplied with a Quality of Service (QoS) specification. Currently, QoS is captured by three functions (1) a latency graph, (2) a value-based graph, and a (3) loss-

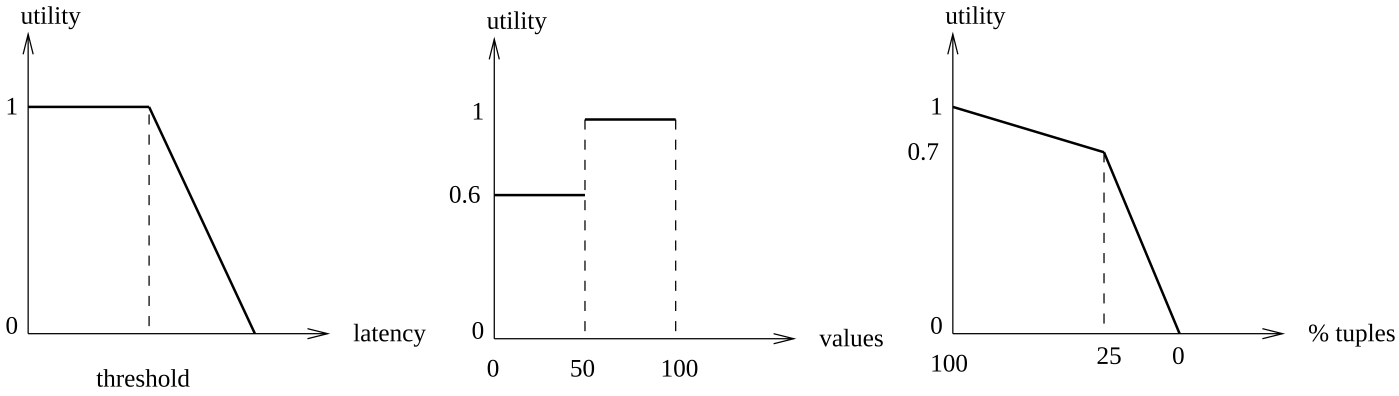

Figure 3: Quality of Service Graphs

tolerance graph as shown in Figure 3. 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. Quality metric of the loss-tolerance graph is the percentage of output tuples delivered to an application, 100% indicating zero loss. Loss-tolerance graph may either be explicitly defined for an application or be automatically derived from a defined value-based graph.

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

1. **Random drop.** This box takes a single parameter p that expresses the percentage of messages that should be dropped. This is implemented by tossing a coin that is weighted by p.
2. **Semantic drop.** This box is essentially a filter box with a predicate whose selectivity corresponds to $1-p$ and that discards tuples with the lowest utility (as given in the value-based utility graph).

The load shedding process consists of three fundamental decisions.

1. **Determining when to shed load.** A momentary spike should not be treated as an overload. The system will recover from that by itself. Load shedding is too heavyweight a process for dealing with short bursts.
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 streams can fanout to multiple outputs, 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 box, we must decide the magnitude of that drop. In the case of a random drop, this involves deciding on the percentage of dropped messages. 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 that include the input rates, and a system capacity C. 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 (% tuples) graph, and the constant H is a measure of how much headroom is desirable in the system to guard against future overloads. $U_{accuracy}(N(I))$ represents the measured utility when there is no load shedding (i.e., there are no inserted drops). In this case, the utility as measured by the loss-tolerance graph will be 1. Thus, $1 - U_{accuracy}(N'(I))$ is the loss of utility introduced by load shedding. It is this quantity that we want to minimize.

We assume that the network is very large. Thus, any algorithm that must inspect all queues dynamically will not scale. We further assume that any cycles that are recovered by the load shedder will be used sensibly by the scheduler to improve the overload situation. 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.

The next section describes our algorithms in detail.

4. THE ALGORITHMS

We have developed a scalable load shedding algorithm with two variants, one for inserting random drops (or simply, drops) and one for inserting semantic drops (i.e., filters). 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.

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 checks the current load situation at a regular preset interval to see if the load is too high or too low. The load can be too low if a previous step set a load shedding action that is now too extreme or unnecessary. We will call this the *load evaluation step*.

The load evaluation step begins by comparing the current load with the maximum system capacity. To compute the total load, we do a simple calculation that involves the current input rates, the box costs and the box selectivities. This calculation has two pieces: the load created by the current stream rates, which we call the *stream load*; and the load due to any queues that may have built up since the last load evaluation step, which we call the *queue load*. The *total load* is a summation of the two pieces.

If the total load is greater than the system capacity times a fixed headroom factor, then the system is overloaded and load must be shed by inserting drops. The headroom is a parameter that expresses reserve capacity that can be used to avoid thrashing the load shedding process. On the other hand, if the total load is less than the system capacity times the headroom factor, then load can be added 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 builds a static table called the *Load Shedding Road Map (LSRM)* that captures a sequence of plans for achieving progressively higher levels of load reduction. This means that this step 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 build on the plan generated for L. Successive rows of the table increase the amount of load that is shed. This might involve adding additional drop boxes and pushing existing drop boxes past split points.

When an overload is detected, we use the LSRM to find a plan for recovering *total load - headroom*capacity* cycles. Each row in the table has an indication of a recovery coefficient for each of the input streams that quantifies the fraction of the load that will be recovered from that path. The table also contains a plan for where to place drop boxes along the corresponding input paths. We take the stream rate for each input stream and multiply it by the corresponding recovery coefficient. The sum of these products is the actual amount of load that the plan will shed. If this sum is not enough, then we go to the next row in the table and compute its effect on the load. The desired load shedding amount is searched in this way until we reach a row which will save us enough cycles to bring the system back to a stable state. The plan stored in that row is adopted.

This process then repeats by monitoring the load again at the specified interval. If the load is ever determined to be too low, then the load shedding process is essentially run backwards. We always keep a cursor that indicates which row in the LSRM was used last. We can move upward though the rows replacing plans with less aggressive ones until we reach the desired headroom.

4.2 Load Coefficients

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 stream tuple through the network to the outputs. Consider the input stream I given in Figure 4, running through a simple continuous query of n operator boxes and producing an output stream O.

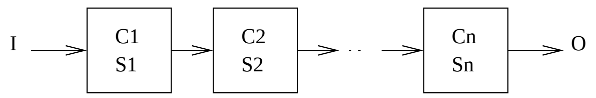

Figure 4: Calculating load coefficients for stream I

Assume that each box i has an associated cost C_i and selectivity S_i. The load coefficient for input I is computed as

$$L = \sum_{i=1}^{n} (\prod_{j=1}^{i-1} S_j) * C_i$$

for all n boxes on a path from input I to the output O. If an input has load coefficient L (in 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 N inputs, we can compute the total *stream load* as

$$\sum_{i=1}^{N} L_i * r_i.$$

Likewise, each box in the network has a pre-computed load coefficient which represents the number of cycles required to push a single tuple from that box to the outputs. We use this coefficient to calculate the contribution of each box to the total queue load. First, we define a global system parameter called $MIN\_Q\_MELT\_RATE$. 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. A box i with a load coefficient C_i and queue of length Q_i contributes to the total queue load by $MIN\_Q\_MELT\_RATE * C_i * Q_i$. The queue load of all boxes are summed to calculate the total queue load. Essentially, by having *queue load* $>$ *0*, we cause an *overshedding* in the network to melt the queues. After all the queues are melted, the load shedder may discover that there are redundant drop boxes and may remove some of them.

In Figure 5, we provide an example query network with two continuous queries and two input streams. We present load

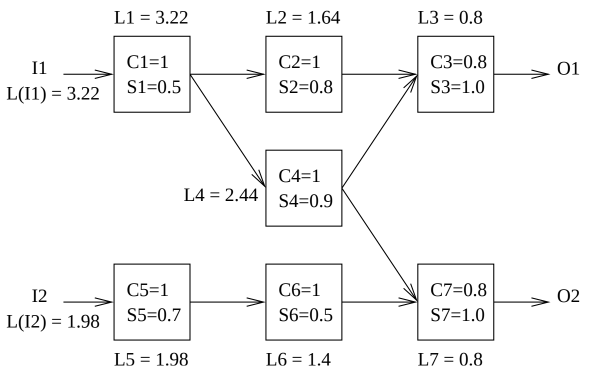

Figure 5: Example query network and with load coefficients

coefficients for each of the two input streams as well as load coefficients for each box in the network. All of these coefficients can be computed statically. 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 second, respectively, then the stream load of this network becomes $3.22*10+1.98*20 = 71.8$ cycles per second. Additionally, if it turns out that box 3 has a queue of 50 tuples and box 5 has a queue of 100 tuples given that $MIN\_Q\_MELT\_RATE = 0.1$, then the queue load of this network becomes $0.1*(0.8*50+1.98*100) = 23.8$. Thus, the total current load of the network adds up to $71.8+23.8 = 95.6$ cycles per second.

4.3 The Load Shedding Road Map (LSRM)

We statically compute a load shedding road map (LSRM) to be used at run-time when overload is detected. The LSRM is an ordered sequence of entries, each of which is a triple of the form $(CSC-list, PDC-list, Plan)$. The $Plan$ is a set of drops that will be inserted at predetermined points in the network. The $CSC-list$ is a list of input streams that will be affected by the $Plan$ along with their associated *Cycle Savings Coefficients (CRC's)*. A cycle savings coefficient can be multiplied by a stream rate to determine how many cycles will be saved along a path if the $Plan$ is adopted, as illustrated in the previous section. The $PDC-list$ is a list of *Percent-Delivered Cursors (PDC)* for the loss-tolerance QoS graphs, one for each output, indicating where the system will be running (in terms of percent of tuple delivery) if the $Plan$ is adopted.

The basic construction routine of the LSRM for both drop-based and filter-based load shedder is common. We take the original query network graph. As long as there is at least one output whose percent delivery cursor is above 0, we append a new entry to the LSRM. However, the way the new entry is created at each iteration of this loop is different for the two algorithms, as we describe below.

4.3.1 Drop-Based Load Shedder

The drop-based load shedder uses the following algorithm to create LSRM entries. First, the application(s) with the minimum negative slope on the loss-tolerance QoS graph is identified in reference to each application's current PDC. Choice of the application with the minimum negative slope guarantees that when we drop p percent of the output tuples, we drop them from the application whose utility degrades the least.

If there are any applications whose current slope is 0, we drop from each the amount of data for which the slope stays 0. Such drops would have no effect on the utility. If there are k applications which all have the minimum slope, we drop from each $min(\Delta(slope), STEP\_SIZE/k)$ load. $\Delta(slope)$ is the percentage range where the slope stays the minimum for an application. In other words, $\Delta(slope)$ is the additional percentage of tuples to drop until we reach an inflection point in the graph. STEP_SIZE is the maximum amount we want to drop from the applications in any one step. We define such an upper limit to avoid the case in which the slope of an application is constant until it reaches 0% delivery. This would cause us to shed all the load from that same application in a single step.

After we identify the applications and how much to drop from each, we create a drop box with the determined selectivity and place it at the end of the query plans for each of the applications. Next, we push each of the drop boxes upstream as far as possible. The more we push, the more cycle savings we get. There are multiple potential obstacles that prevents us from pushing the drop further:

- Operator commutativity properties
- Operator sharing between multiple applications (which we call *splits*)

Pushing a drop box requires that the drop operator commute with the next operator in the path. If commutativity does not hold, we need a way to account for the effect of this push on the query result semantics. So far, we have considered query plans whose operators (e.g., filter, map) can switch places with drops (both random and semantic). More complex operators like Windowed Join or Aggregate do not always have this property. Extending our techniques to include more complex queries is a topic for future research.

A *split* occurs when the output of a box fans out to more than one downstream box. Splits require a special treatment due to the fact that pushing a drop upstream past a box whose output is shared by other applications affects all other shared outputs as well. Hence, the loss of utility might be more than we intended if we push drops across split points. Our algorithm avoids this situation by allowing pushing only if it is desirable by all the applications that receive data from the split point. This is achieved by only pushing drops past splits when there are drop boxes on all sibling branches. If so, we push a drop box which is the "lowest common denominator" (the minimum drop) of all the sibling drops. For random drops, this corresponds to creating a drop box with the highest selectivity (i.e., minimum drop rate) of the siblings. The siblings' selectivities (drop rates) are updated by being divided with the maximum selectivity (minimum drop rate). As a special case, if all siblings have the same selectivity, we can remove all the sibling drop boxes after the new one is created since they become redundant (i.e., their selectivity becomes 1). When not all siblings have drop boxes, pushing stops at the split point. Otherwise, pushing continues until either an input is reached or another split point avoids further pushing.

When boxes with multiple inputs (like union) are encountered, we safely create clones of the drop box and push all the clones along each of these paths.

This algorithm returns a single LSRM entry (i.e., the triple we mentioned earlier) which contains a list of drop boxes to be inserted on the original query network with their detailed description including their selectivities and where in the query network they need to be inserted. The entry also includes the updated cursors for the loss-tolerance QoS graphs and cycle savings coefficients that this entry provides to the original query network.

4.3.2 Filter-Based Load Shedder

For semantic drops, our algorithm follows the same principles as the random drop algorithm described above. Selecting an application and deciding how much data to drop is determined exactly as in the first algorithm. Note here that, we can do this step in a uniform way since we derive the tuple-based (i.e., loss tolerance) QoS graph from a given value-based QoS graph. The details of this procedure are described below.

Once the victim application and the amount to be dropped is determined, we create a filter box rather than a drop box. This filter box must have the same selectivity as the previously determined drop amount. This requires finding a suitable filter predicate that will provide us with the required selectivity while at the same time dropping the data values with the least utility. Before we get into details of how we create this predicate, let us explain how the algorithm proceeds.

After the filter box with the proper predicate is created, we try to push it towards the inputs. Again, we face the problem of sharing at split points. The way we handle this differs from the drop-based algorithm only in the way we create a combined filter to push upstream. If all siblings have load shedding filter boxes, we create a new filter box whose predicate is the disjunction of the predicates of the sibling filters. The sibling filters are retained, and the new one is pushed across the split. Note here that we can be more intelligent in combining predicates than simply forming disjunctions by appealing to predicate-logic. For example, if predicates of the siblings are disjoint, then we do not need to create their disjunction because it is redundant. Predicates subsumed by one or more sibling predicates, do not contribute to the combined predicate.

This algorithm returns a single LSRM entry which includes a list of filter boxes to be inserted on the original query network. Each box in the list has the predicate to be used. Updates on QoS cursors and cycle savings coefficients are also recorded as described before.

Deriving Loss-tolerance QoS from Value-based 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-utility assignment graph, we can produce a percentage-utility assignment graph as described below.

Table 1: Example value intervals

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	$0.6/0.68 = 0.88$

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 that order in an array together with their u_i, f_i, and n_i values. Table 3 illustrates such an array 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 array 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 $100 - f_i * 100$ percent drops to $1 - n_i$. The utility values for the percentages in $(100, f_i * 100)$ decrease linearly.

Consider the simple example in Figure 6. An histogram for two value intervals and a value QoS are provided. Our goal is to generate a loss tolerance graph from these two. Table 3 presents how normalized utility is calculated for each value interval. Notice that the first value interval makes up the 40% of the values and has a normalized utility of 0.12. 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 turning point where the utility function and hence the slope changes. This leads us to the loss tolerance graph in the Figure 6.

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.

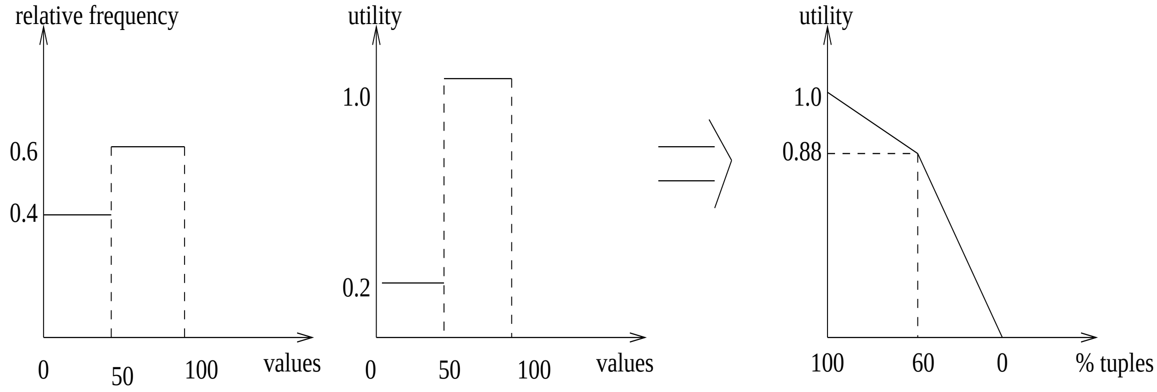

Figure 6: Derivation of loss-tolerance QoS

Creating the Predicate for Semantic Drop

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 generated from the value QoS graph, say $current\_percentage$, 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 array based on relative frequency of the intervals.

Consider the example we presented in Figure 6. Assume that the $current\_percentage$ 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 least 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. The interval to be dropped will be $0 \leq value < 25$ and the predicate for the filter box 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 $value \geq 75$.

Calculating the Utility of an Output

At any given time, we can calculate average utility obtained from each QoS graph as follows:

- Latency Utility:

  $U = \frac{\sum_{i=1}^{n} u_i}{n}$

  n: number of tuples seen so far
  u_i: utility of the i^{th} tuple

- % Tuples Utility:

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

  k: number of epochs (an epoch is the time period during which same % of the tuples are received)
  n_i: number of tuples seen in epoch i ($\sum_{i=1}^{k} n_i = n$)
  u_i: utility of each tuple during epoch i

- Value Utility:

  $U = \frac{\sum_{i=1}^{n} f'_i * u_i}{\sum_{i=1}^{n} f_i * u_i}$

  f_i: relative frequency of tuples for value interval i w/o drops
  f'_i: f_i w/ drops relative to total number of dropped + undropped tuples
  u_i: average utility for value interval i
  (values in an interval are consecutive and they have common utility function)

When there are multiple outputs, overall utility can be calculated by taking a sum. If certain outputs have higher priority, then they could be assigned heavier weights in the sum. For now, we assume equally-weighted outputs.

5. EXPERIMENTAL SETUP

5.1 The Simulator

To experiment on our algorithms, we implemented a query network simulator using the CSIM18 simulation engine. CSIM provides the ability to create multiple processes running in parallel and to define resources and facilities used by these processes. We defined our main resource, the cpu, as a CSIM facility with round robin as its 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 box in the network, which has one or more input tuples in its queue to process, with a CSIM process. A query box's executing a tuple is simulated by assigning a cost (in time units per tuple) and let the process use the cpu for that amount of time per tuple. Apart from query box processes, we also have a monitoring process, which wakes up every $MONITOR\_HOLD$ period for load inspection. Finally, we created one process to simulate each input data stream arriving at the query network. The input rates are simulated by holding the time by a specified inter-arrival time between tuple arrivals.

5.2 Workload Generation

5.2.1 Query Networks

We generate our test networks randomly, starting from the output ends. First, we decide on how many applications we want to deliver data to (which is also equal to the number of queries in the network), say N. Then we decide the number of boxes each query will have by choosing a column size Q_COLS. We then create a two-dimensional grid of boxes of size Q_COLSx$(2^{Q\text{-}COLS} * N)$. Each column of this grid is a level of boxes from which we choose a random box to connect to the previous level. Starting from an application, when we pass through all the columns, we reach to the input end and create a stream. We can adjust the relative proportions of box types in the network by changing their probabilities of being chosen. We can also control the amount of sharing in the network by using an arc_reuse_factor. If $arc\_reuse\_factor = 0$, then there are no splits in the network whereas for $arc\_reuse\_factor = 10$, all arcs are shared. We use this factor to decide, at each level of the grid, whether we choose a previously unused box from the grid or we reuse an old one.

For each filter box, we should also generate a filter predicate and assign its selectivity. Assuming a uniform data workload enables us to estimate selectivities. The predicates are simple comparison predicates of the form $value > constant$. In addition to filter boxes, we use union boxes where a union box simply combines tuples coming from two input streams into one output stream. Each box type has a fixed cost. We assume that load shedding boxes (i.e., drops) are relatively cheaper than others.

5.2.2 Streams

We use streams of integers whose values are chosen randomly from the range $[0, 100]$. We can generate streams both with a constant inter-arrival time and also with changing inter-arrival times each of whose duration can be adjusted.

5.2.3 Quality of Service

Our main QoS graph is the value-based utility graph. Given that and the output histograms, we can generate the loss-tolerance graph. We use two value intervals to assign utilities to. Interval [0, 50] has a lower utility than interval [51, 100], which always has utility 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 utility interval. To obtain histogram information, we always run the network once without load shedding and then use the resulting data interval frequencies in the following runs. We also generate delay-based QoS graphs by choosing a random utility and percentage point and having two pieces of linear functions that intersect at this point.

5.3 Parameters

Table 2 lists the parameters we used in the simulation.

Table 2: Simulation parameter settings

Parameter	Value
MONITOR_HOLD	10
DROP_COST	0.01
LS_FILTER_COST	0.01
FILTER_COST	0.1
UNION_COST	0.1
STEP_SIZE	0.1
NUM_ELEMENTS	1000
CPU_SLICE	0.01
CAPACITY	100
HEAD_ROOM	1.0
Q_COLS	5
NUM_QUERIES	5
Q_ROWS	160

6. EXPERIMENTS

6.1 Admission Control Algorithms

In addition to our drop-based algorithm, *drop-ls*, and the value-based algorithm, *filter-ls*, 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, $\Delta(L)$, is detected, this algorithm randomly selects one input stream and sheds sufficient load on that stream to compensate for $\Delta(L)$. 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.

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.

6.2 Experimental Data

Figure 7 shows a sample network we used in our experiments. This network is generated with no splits and has a mix of 20% merge boxes and 80% filter boxes.

The interval data for each application is shown in Table 3. We have one row per application in this table (f_i denotes the frequency of interval i). Interval 2 has a utility of 1.0 for all applications, while interval 1 has a utility of u_1, which is set differently for each application. These utility values are created using a Zipf distribution with a skew parameter of 0.5.

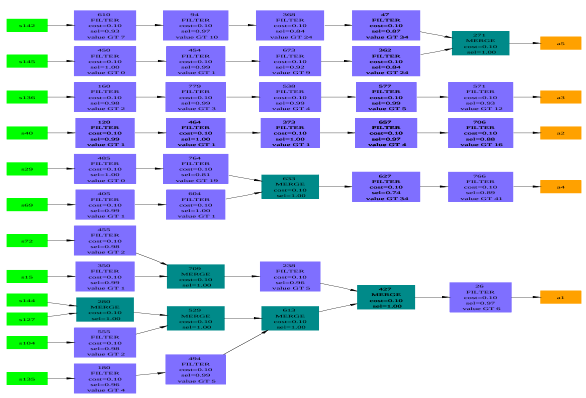

Figure 7: Sample query network

Table 4 shows the mean input rates used to overload the simulated system. With a rate of 0.2 tuples per time unit, the system is 14.225% over its capacity. When we we pump

Table 3: Interval Data

Application	f_1	f_2	u_1
1	0.47	0.53	0.29
2	0.42	0.58	0.05
3	0.46	0.54	0.12
4	0.15	0.85	0.32
5	0.28	0.72	0.90

Table 4: Load levels used

Mean Rate	Load
0.2	+14.225%
0.4	+128.42%7
0.6	+242.542%
0.8	+356.928%
1.0	+471.941%

up the rates to 1 tuple per unit time, we model an extreme overload scenario where the system is approximately 400% overloaded (i.e., the system needs five times its capacity to deal with the input stream rates).

In the following graphs, each point is the mean of result obtained from five independent runs of the simulated system. Note that the results did not show any sensitivity to varying data value distributions.

6.3 Tuple Utility Loss

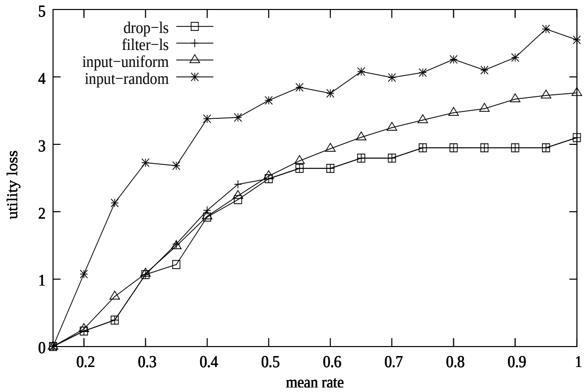

Figure 8: Tuple Utility Loss vs Mean Rate

Our first experiment quantifies the loss in *tuple-QoS* for different load shedding schemes for varying overload levels. As shown in Figure 8 and Figure 9 (the figures share the same source data; the latter focuses on the lower input rates), all algorithms are clearly negatively affected by increasing input rates. Because the system has fixed capacity, the percentage of the tuples that needs to be dropped increases with increasing input rates, decreasing the utility of the system.

We observe that the two QoS-driven algorithms perform the same, which is not surprising as they make their decisions based on the same tuple-QoS graph (generated from the same value-QoS graph). Hence, both algorithms make identical shedding decisions. The input-random algorithm performs poorly compared to others, incurring relatively higher utility losses for all input rates. An interesting result is that, for smaller input rates, the input-uniform algorithm performs comparably to the QoS-driven algorithms even though it does not utilize any QoS information. 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 8, this situation changes and we start observing the benefits of the QoS-driven algorithms over the input-uniform algorithm.

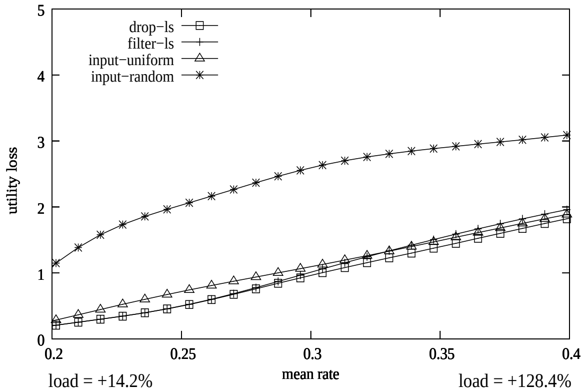

Figure 9: Tuple Utility Loss vs Mean Rate (focused)

6.4 Value Utility Loss

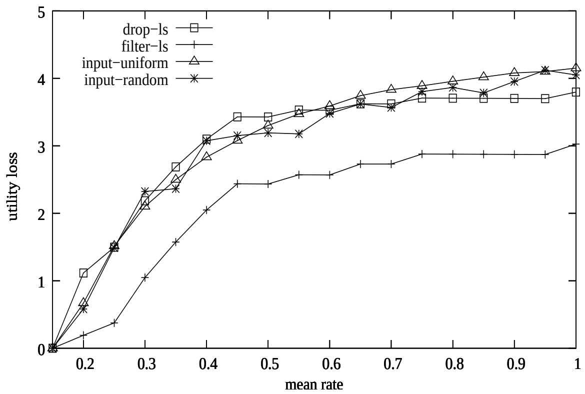

Figure 10: Value Utility Loss vs Mean Rate

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

Figures 10 and 11 clearly demonstrate that the semantic drop algorithm significantly outperforms the other approaches. Furthermore, the difference tends to increase with increasing input rates. Note that drop-ls does not seem to provide any benefit over the input-based algorithms in this case. In fact, it seems to perform somewhat worse than the others. This interesting behavior is a result of drop-ls's bias towards shedding certain high frequency, high utility values, a bias which is not present in the other algorithms that drop tuples randomly. It would thus be more fair to use the tuple utility

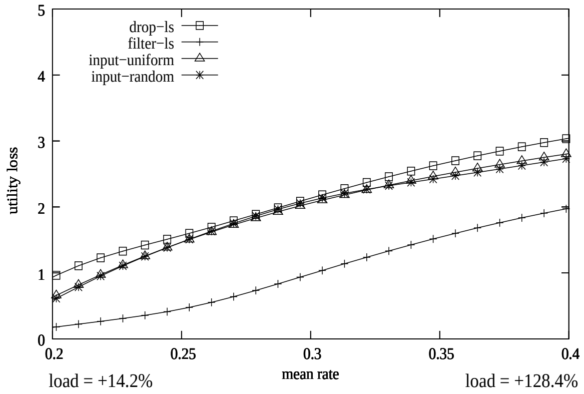

Figure 11: Value Utility Loss vs Mean Rate (focused)

loss (as opposed to value utility) to compare the non-value-based algorithms as we did in the previous experiment.

6.5 Effect of Utility Skew

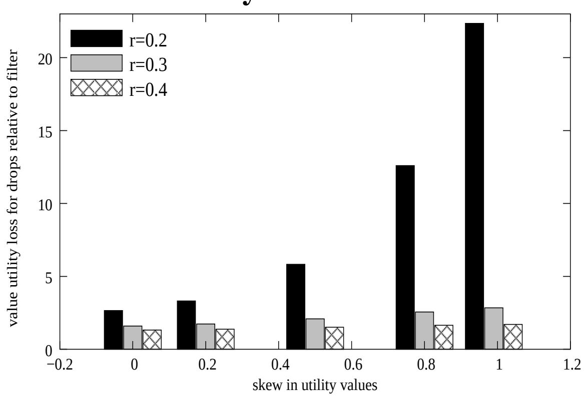

Figure 12: Value Utility Loss for Drops relative to Filter vs Skew in Utility

In the scenarios we considered so far, the utility values for different tuple values were uniformly distributed. This choice implicitly favors non-value-based algorithm that are essentially value-unaware.

In order to characterize the impact of skewed utility distributions, we devise a scenario where we inject skew to the utility values for data interval 1, which always has a utility value less than that of data interval 2 (see Table 4). 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 occurrance. Hence, we expect that with high skew, value-based dropping will perform much better than the randomized drop-based approach as 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 12 illustrates our results. On the y-axis, we show the ratio of the value utility loss coming from the drop algorithm to that coming from the filter 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, filter algorithm will too have to start dropping tuples from the higher utility region.

7. RELATED WORK

The congestion control problem in data networks [11] 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 [6, 8]. 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.

Recently, there has been increased attention to overload management for Internet services. SEDA [10] is an innovative service architecture where overload management is integrated into system design and architecture as a fundamental primitive. SEDA applications consist of a network of event-driven stages connected by explicit queues, much like the Aurora model. SEDA uses resource controllers to automatically adjust the resource use of each of its stages primarily by adjusting the number of threads executing within a stage and the number of events processed by each invocation of an event handler within that stage. Aside from the obvious differences in the processing models, Aurora sheds load by dropping data based on application-specific QoS specifications.

Aurora load shedding is also related to approximate query answering, data reduction, and summary techniques (e.g., [7, 3, 1]. 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. [4] describes histogram-based techniques to compute approximate answers correlated aggregation queries. [5] describes wavelet-based techniques to compute data summaries, which are then used to provide approximate answers to ad-hoc queries. [2] 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 [9]. The unique aspect of our approach is that our sampling is driven by QoS specifications that relate result tuple values to utility.

To our knowledge, scaling the system against extreme data rates through load shedding is a feature that has not yet been formally addressed by any data stream management architecture.

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 a variant of admission control in which denial of new input is spread evenly across all inputs. 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.

9. REFERENCES

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

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

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

[5] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 79–88, Roma, Italy, September 2001.

[6] 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.

[7] 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.

[8] 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.

[9] 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: First Biennial Conference on Innovative Data Systems Research*, 2003. to appear.

[10] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

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