Self-managing load shedding for data stream management systems

Thao N. Pham, Panos K. Chrysanthis, Alexandros Labrinidis
Department of Computer Science, University of Pittsburgh
{thao, panos, labrinid}@cs.pitt.edu

***Abstract*— Load shedding is an integral component in many Data Stream Management Systems, aiming at preventing the response time from exceeding a user-specified delay target under overload situations. The currently best performing load shedder determines the correct amount of load to shed by utilizing a feedback loop for correcting the statistics-based estimations. Although this load shedder outperforms previous works in controlling response time as well as minimizing data loss, it requires a manually-tuned parameter and cannot work with complex query networks containing joins, aggregations or shared operators. In this paper, we propose SEaMLeSS — SElf Managing Load Shedding for data Stream management systems, which extends and rectifies these limitations of the state-of-the-art load shedder while making it applicable for multi-tenant servers. We implement and evaluate our extensions in AQSIOS, our experimental DSMS prototype, using both synthetic and real input patterns.**

I. Introduction

Today the ubiquity of sensing devices as well as mobile and web applications generates a huge amount of data in the form of streams. Consequently, monitoring applications have been developed based on *continuous* queries (CQs) that look for interesting events over these data streams. Such CQs are efficiently processed by a Data Stream Management System (DSMS) (e.g., [1], [2], [3]), rather than a Database Management System.

An important requirement of DSMS-based monitoring applications is *near real-time* answers: it is expected that detected events (i.e., output tuples) for a specific input event (i.e., input tuple) are produced within a small period after tuple arrival, otherwise the outputs have little or no value. The length of this period, referred to as *delay target*, defines how long the outputs can be delayed and is specific to each application. However, even with the most well-provisioned DSMS, the input rates and value distribution of data streams usually fluctuate and can, at times, create an input load that is unpredictably higher than the processing speed of the DSMS, causing the DSMS to be *overloaded*. *Load shedding* has been proposed (e.g., [2], [1], [4]) to handle such cases. This method sheds data tuples to meet the quality of service requirement, i.e., delay target, assuming that data accuracy is less critical and approximated answers are acceptable.

Previous works address four basic questions related to load shedding: *when, how much, where, and what* to shed. The works on *where to shed* ([5], [6], [7], [8]) discuss how to distribute the shedding to different locations in the query network when CQs have different requirements on data accuracy. The works on *what to shed* increase the usefulness of the retained data after shedding by either considering data semantics (e.g., [5], [9], [10], [11]), or using other ways to shed load instead of completely discarding tuples (e.g., [4], [12], [13]). All these works on the *where and what* questions require that the amount of load to shed, i.e., the answer to the questions of *when and how much to shed*, is known and provided as an input.

A few previous works have addressed these first and crucial questions of *when and how much to shed* (e.g., [14], [5], [4], [15], [16]), but none of them provides a complete answer to be used efficiently in a general DSMS. The method used in Aurora [5], which is assumed in most other systems, theoretically can calculate the excess load coming into the system in every time unit. However, as shown in [14], this method cannot control the average response time of query outputs to a predefined delay target. CTRL [14] is proposed to overcome that limitation of Aurora. However, CTRL is not applicable for complex query networks consisting of joins, aggregates, or shared operators.

Both CTRL and Aurora require a manually-tuned headroom factor that represents the system processing capacity. The headroom factor is tuned off-line in both these schemes and used as constant, although in reality it is subject to changes at runtime as the system environment or the query network change. This significantly limits the applicability of these schemes in practice, especially when the DSMS is deployed on a shared, multi-tenant server.

Clearly, a good load shedder, like CTRL, should be able to bound the response time of the DSMS to the specified delay target, and like Aurora, should be applicable to all types of query networks. Unlike both CTRL and Aurora, a good load shedder should also automatically adjust the headroom factor instead of relying on a manually-tuned one. In this paper we propose a load shedder with all these three ideal properties, which provides an efficient answer for the fundamental *when and how much* questions.

Contributions: We make the following contributions:

(1) By defining the concept of queued load, we improve the delay estimation model in CTRL to build SEaMLeSS, a scheme that is applicable to all types of query networks.
(2) We build into SEaMLeSS another feedback loop to automatically adjust the headroom factor at run time, and remove the need for a manually-tuned one.
(3) We evaluate SEaMLeSS in AQSIOS, our experimental DSMS, using multiple query networks and real and syn-

978-1-4673-5304-5/13/$31.00 © 2013 IEEE

thetic data. The experimental results confirm the expected advantages of SEaMLeSS over CTRL and Aurora.

Roadmap: Now that we have introduced the related work and put our contribution in perspective, in the next section we provide a more detailed background on the "when and how much" problem, followed by our proposal of SEaMLeSS in Sec. III. We describe our experiment setting and results in Sec. IV and Sec. V, respectively, and conclude in Sec. VI.

II. BACKGROUND

A. The "When and how much" problem

Definition 1: The *response time* of a tuple is the time elapsed since the tuple enters the system until the related output tuple is produced. The response time consists of processing time and queuing time.

Definition 2: The *delay target* of a query, denoted by D, is the *worst-case response time* tolerated by the stream applications using the query*.

Definition 3: The *headroom factor*, denoted by H, is the fraction of each time unit the system can spend on processing the incoming tuples. H is normally in the range of (0,1).

Definition 4: The *when and how much* problem refers to the need to determine when to apply *load shedding* and how much is the minimal amount of load to shed, so that the response time of the output tuples is upper-bounded by the delay target.

B. "When and how much" state-of-the-art

1) The Aurora approach: A first attempt to answer the "when and how much" questions, used in Aurora [5] and implied in STREAM [7], is to statistically compute the incoming load L, compare it to the system capacity L_C which estimated by a headroom factor, and shed an amount equal to L-L_C if $L > L_C$. The incoming load is computed by

$$L = \sum_i (r_i \times load\_coef_i) \tag{1}$$

where r_i denotes the input rate, i.e., the number of input tuples per time unit, of the i^{th} input stream, and $load\_coef_i$ is the *load coefficient of the stream*. The $load\_coef_i$, in the case of a flat query, i.e., no shared operator, is given by:

$$load\_coef_i = \sum_j (c_j \times \prod_{1 \le k < j} sel_k) \tag{2}$$

where c_j is the processing cost per tuple of the j^{th} operator in the path from the input stream to the corresponding output, and sel_j is its selectivity. In the case of a fan-out query plan, i.e., with shared operators, it recursively sums up the load coefficient of every sub-path along the way.

Although the Aurora approach is theoretically sound, in practice it has the following two problems:

(a) **Ad-hoc selection of headroom factor:** Aurora does not provide a method to pick the correct headroom factor. An incorrect value of the headroom factor will lead to either shedding more data unnecessarily, or failing to stop the response time from increasing beyond the delay target, neither of which is desirable.

(b) **Not delay-target-aware:** Aurora is not designed to consider a specific delay target. In fact, as pointed out in [14], Aurora is unable to bound the response time to a specific delay target, for Aurora does not use the outcome of its previous decisions as feedback to correct current decision.

2) The Control-based approach (CTRL): In [14], the authors proposed CTRL for the *when and how much* problem that primarily addressed the second shortcoming of the Aurora approach, i.e., not delay-target-aware. The CTRL approach counts the number of tuples coming in and out of the system in each period and keeps track of a *virtual queue* of tuples queued in the system. The response time (which is called *delay* in the CTRL paper) of the tuples coming to the system at the k^{th} period is then estimated by the following equation:

$$y^k = \frac{c}{H} q^{k-1} = \frac{c.T}{H} \sum_{i<k} [f_{in}^i - f_{out}^i] \tag{3}$$

where y^k is the response time at the k^{th} period, $q^{(k-1)}$ is the length of the virtual queue after the $(k-1)^{th}$ period, c is the processing cost per tuple, T is the length of the period, H is the headroom factor, f_{in} and f_{out} is the input and output rate.

Applying control theory on the above model, the authors develop a feedback controller to compute the additional numbers of tuples allowed, and consequently, the shedding rate, for the next period such that the response time converges quickly to the delay target. They experimentally show that CTRL can keep the response time around the target, which Aurora cannot, while CTRL sheds only 1-2% more data than Aurora. CTRL, however, has also two major shortcomings:

(a) **Manual tuning of the headroom factor:** In [14], the authors *manually* choose the headroom factor for a given query network so that the estimated delay given by Eq. 3 best matches the real response time of the output. This manual tuning is not practical, because the headroom factor can change due to events such as the starting of a new background job or a new query. If the headroom factor used is higher than the correct one, the response time will converge to a value higher than the delay taget, violating the scheme's promise of honoring the target. A headroom factor smaller than the correct one causes additional data loss unnecessarily.

(b) **Not applicable in complex query networks:** The way CTRL estimates the virtual queue based on the inflow and outflow rates does not work when the query network has join, aggregation, or shared operators, because the one to one mapping of an input to an output tuple would fail. Fig. 1 gives an example of this, with a join operator ($\bowtie_1$) and the result from the Select operator σ_2 being shared by two queries. The output flow also includes those tuples discarded along the query network (e.g., tuples that do not satisfy a selection). In this case, simply increasing the length of the virtual queue by 1 for each incoming

*Same as [14], we assume that the delay targets for all queries are either identical or the load manager will react when the first delay target is violated.

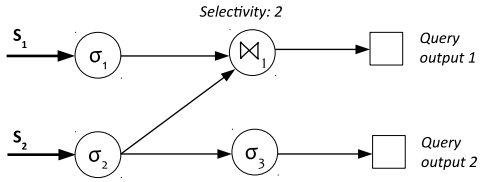

Fig. 1. A query network with joins and shared operators, for which the queuing estimation model of CTRL would not work. The selectivity of $\bowtie_1$ equals 2, meaning on average an input tuple (from either side of $\bowtie_1$) produces two output tuples.

tuple from the two sources and decreasing 1 for each tuple output or discarded would not work. The reason is, for instance, one tuple coming in stream S_2 actually goes out of the system only if it is discarded at the operator σ_2 or discarded/output along both branches downstream. Further, the selectivity of a windowed join (as $\bowtie_1$ in Fig. 1) can be greater than 1, thus one tuple coming to the join can result in multiple output tuples at the query output 1.

3) Other existing schemes: In [16], the method to estimate the excess load is similar to Aurora without considering the headroom factor (i.e., assuming that the headroom factor always equals 1). Another approach, as mentioned in [4] and [15], is to monitor the input queue(s) to decide when the system is overloaded. However, the approach in [4] did not discuss how the number of queued tuples can be used to infer whether the system is overloaded. The approach in [15] puts a threshold on the number of tuples in the queue as a signal that the system is overloaded, but it is also not specified how to select such a threshold.

III. SEAMLESS

We propose SEaMLeSS (SElf Managing Load Shedding for data Stream management systems) which achieves the three ideal properties: (1) being able to honor the delay target, (2) applicable to all types of query networks, and (3) adjusting the headroom factor automatically.

SEaMLeSS follows the design of CTRL in applying a delay estimation model to estimate the response time from the number of queued tuples, and using control theory to determine the shedding amount for the next cycle. This design allows SEaMLeSS, like CTRL, to effectively manage the response time of the DSMS to honor the delay target. However, SEaMLeSS has the following improvements over CTRL:

- Instead of simplifying the details of the queued tuples by using the virtual queue, we propose the concept of *queued load* and use that in SEaMLeSS to exactly estimate the response time without any assumption on the type of the query network. This improvement enables SEaMLess to be applicable to all types of query network, including those containing joins, aggregations or shared operators.
- SEaMLeSS uses the actually response time of the outputs as feedback to automatically adjust the headroom factor, thereby removing the need for manually-tuned one.

A. Handling complex query networks

One possible solution for overcoming CTRL's inadequacy in handling complex query networks could be to time the inflow rate and outflow rates with the number of paths sharing an operator, and the selectivities of join and aggregate operators. However, this gives an *approximation* of the length of the virtual queue in each period rather than the accurate value. The error, even though it could be negligible in a single period, accumulates to a considerable amount over time and degrades the performance of the system.

We propose the concept of *queued load* and use it in our solution. In a k^{th} period, SEaMLeSS estimates the queued load based on the number of tuples in the *physical queue of each operator*. Because the tuples in different queues contribute unequally to the total queued load, we consider the load coefficient of the query branch fed by each queue. In particular, up to the k^{th} period, each operator's input queue contributes to the total queued load qL^k an amount equal to the queue's length times the load coefficient of the query branch rooted at that operator, as in the following equation:

$$qL^k = \sum_i (q_{o_i}^k \times load\_coef_{o_i}) \tag{4}$$

where o_i denotes an operator in the query network, $q_{o_i}^k$ is the length of the physical input queue of o_i at the k^{th} period, and $load\_coef_{o_i}$ is the load coefficient of the query branch rooted at o_i, which is calculated following Eq. 2 in Sec. II-B1.

Assuming that the query processing task of the system is carried out sequentially and the DSMS is using a fair scheduler such as Round Robin, then a tuple coming to the system at time k has to wait for all queued tuples in the system up to time k-1. Therefore, the estimated response time for the tuples coming during the k^{th} period is given by Eq. 5, which is a modification of Eq. 3 in [14]:

$$y^k = \frac{qL^{k-1}}{H} \tag{5}$$

The Eq. 6 presents the SEaMLeSS' feedback controller, which is an adjustment of the one in [14]. In each control period this feedback controller is used to determine u^k, which is *the amount of load* that can be added to the queue in the next period without violating the delay target.

$$u^k = H \times [b_0 e^k + b_1 e^{k-1}] - a u^{k-1} \tag{6}$$

where $e^k = y^k - D$ and a, b_0, b_1 are the controller parameters. Details on the design of the controller and the derivation of these parameters can be found in [14].

In each control period of length T, the DSMS can process (i.e., take from the queues) a load of $H \times T$, so the input load that can be accepted in the next period is $v^k = u^k + H \times T$. Thus the amount of load to shed in the next period is $L^k - v^k$, where L^k is the incoming load in the next period. Since L^k has not been observed yet, it is approximated by L^{k-1}.

B. Headroom factor auto-adjustment

The number of queued tuples reflects the intermediate outcome of the shedding decision: if the shedder sheds the right load, the number of tuples in the queues should remain at a level such that the time to process these tuples does not exceed the delay target. Therefore, the number of queued tuples, in the form of a virtual queue as in CTRL or our queued load, is used as feedback to help the load shedders adjust their shedding decisions. However, the schemes cannot make the inference directly from the length of the virtual queue or the amount of queued load, but rather apply a delay estimation model over it. The delay estimation model in turn needs an estimation of the headroom factor, so that it can compute the time needed to process the queued tuples. The problem in CTRL is that there is no feedback about the correctness of the headroom factor, so it depends on a manually-tuned one.

This motivated us to add to SEaMLess another feedback loop to automatically adjust the headroom factor. Since the headroom factor is used in the delay estimation model to estimate the response time based on the number of queued tuples, the feedback that can be utilized to adjust the headroom factor should be the different between the estimated response time and the actual response time. The question is how this difference suggests the correct headroom factor.

The obvious solution of using the difference between the estimated response time (i.e., estimated delay) and the real one would not work, because this difference does not always indicate that the current headroom factor is not correct. The difference might be caused by the lag between the time of the measurement and that of the estimation. This can happens when the system is overloaded but the response time is still below the delay target. In that case, the load manager does not shed the excess load so the response time keeps increasing quickly. This is also true for the case when the system comes from an overloaded state to a non-overloaded one, causing the response time to decrease quickly. Therefore, in both of these cases, it is hard to use the difference to adjust the headroom factor. In addition, when the system is in normal state (i.e., not overloaded), the response time is small and hence factors such as system environment fluctuations and statistics errors can cause a difference that is relatively significant. Therefore, the difference between real and estimated response time during normal state is also not a good clue to adjust the headroom factor.

Because the basic goal of CTRL, and SEaMLeSS, is to keep the response time around the delay target when the system is overloaded, if the headroom factor is correct the response time should converge to the target *whenever the load is being shed*. Therefore, by monitoring the actual response time when the shedding decision is in effect and comparing it with the target, we can figure out whether the headroom factor is correct or not and how to adjust it. More specifically, a wrong value of the headroom factor causes the error in the estimated response time, which finally results in the response time converging to a value D′ that is higher or lower than the target D. The difference between the target delay D and this value D′ tells how much the headroom factor should be:

$$H_{adjusted} = H_{current} \times \frac{D}{D'}$$

where $H_{adjusted}$ is the new value of the headroom factor, and $H_{current}$ is the current one. D′ is the average real response time over a number of periods when shedding is applied.

IV. Experimental Platform

We experimentally evaluated SEaMLeSS, CTRL, and Aurora load shedders in AQSIOS, our DSMS experimental prototype.

A. DSMS platform

We developed AQSIOS [17], our experimental DSMS platform by extending the STREAM source code [2]. Our extensions include new operator implementation, optimization schemes [18], new scheduling policies [19], and load shedders.

Data streams coming to the system are read and translated to an internal representation format by source operators. In the STREAM engine, a source operator is considered part of a query and the query networks. We embed the shedding functionality to the source operators, so that they will discard an incoming tuple with a probability equal to the shedding rate determined by the load manager.

B. Experimental Settings

Query networks: We use two query networks as follows:

- *QN-complex:* is a big query network containing about 1140 operators (select, project, source and output). Some operators are shared between two to four queries. We used this big query network to create experiments that are close to real-world applications.
- *QN-flat:* is a flat query of 8 select and project operators, together with a source operator and an output operator. We add delay to the operators to increase the processing cost per tuple, so that the total cost of this query network is approximate to that of QN-complex. This QN-flat query network is similar to the one used in the CTRL paper [14]. We use this query network in our experiments to create a setting when CTRL can achieve its best performance: the simple, flat query network enables the correct calculation of the virtual queue in CTRL, even if such a query network is not representative of real applications.

Input data: We use one stream of synthetic data, denoted S_c, and one of real data S_r, as described below:

- $\mathbf{S_c}$: has a constant input rate of 200 tuples/sec, which is within the system capacity, for the first 10 seconds, and then goes to 350 tuples/sec, which overloads the system, until the end of the experiment at the 400^{th} second. S_c is used when we want to keep the input rate constant to clearly examine the effect of the factor of interest.

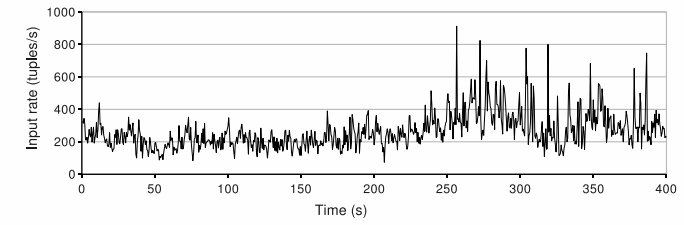

Fig. 2. Input rate of the real data S_r.

- $\mathbf{S_r}$: is a trace of TCP packets coming in and out of the Lawrence Berkeley Laboratory (Dataset LBL-PKT-4/lbl-pkt-n.tcp is publicly available at the following URL: http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html)

Parameters: We choose the delay target $D = 2s$, which is the same to that used in the CTRL paper. We use the control period $T = 0.5s$ (The CTRL paper experimentally shows that T= [250ms-1000ms] is the best range given that $D = 2s$).

In order to choose an appropriate headroom factor for CTRL, following the method used in [14] we run the CTRL's module that estimates the output delay based on the length of the virtual queue. We manually change the headroom factor used in the model and plot the estimated value together with the real one until they match with each other. This tuning gave us 0.99 as the best value of headroom factor for CTRL for the QN-flat query network. For the QN-complex query network, as anticipated, we cannot find suitable headroom factor for CTRL since the estimation of the virtual queue by CTRL is no longer correct. Thus in this case we have to run CTRL with the headroom factor obtained with the QN-flat query network, and some other values down to 0.8. For SEaMLeSS, we set the initial value of the headroom factor to 0.8.

We set the headroom adjustment period P to 30 control periods (i.e., 15s) for SEaMLeSS. A sensitivity analysis of SEaMLeSS on P are presented in Sec. V-B

V. Experimental results

In this section we report the results of our performance evaluation of SEaMLeSS compared to CTRL and Aurora, showing the advantages of SEaMLeSS in adjusting the headroom factor automatically and handling complex query network. We also analyze the sensitivity of SEaMLeSS to the headroom factor adjustment period. For all the experiments, the results reported are the average of 5 runs.

A. Performance evaluation

1) Under system environment changes (Fig. 3): Selecting a correct headroom factor for CTRL is a daunting task, but despite being carefully selected, the headroom factor is not guaranteed to be correct for the whole execution time. In this experiment we illustrate this by launching background jobs while the DSMS is running. We use the input S_c and the QN-flat query network.

Fig. 3 shows the response time under CTRL, which used a fixed, manually-tuned headroom factor, and our SEaMLeSS, which automatically adjusts the headroom factor at runtime. At

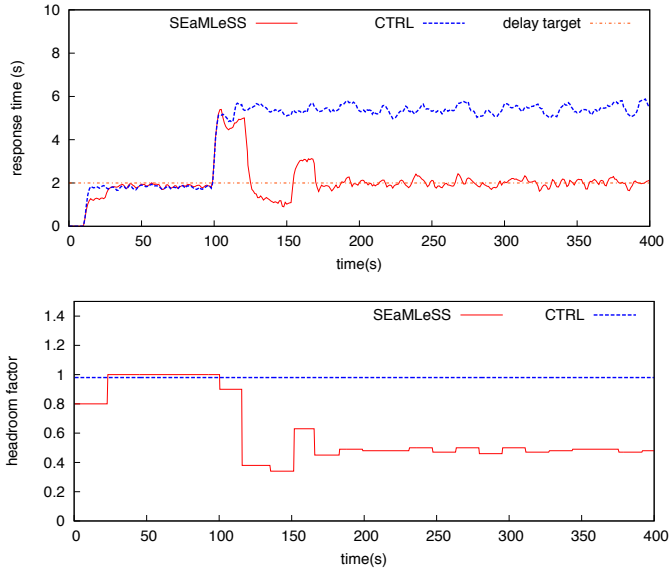

Fig. 3. Effect of environment changes on CTRL and adaptation of SEaMLeSS. Top plot shows the response time, bottom plot shows the headroom factor recognized by each scheme. Total data loss for SEaMLeSS and CTRL is 62.98% and 62.69%.

	H	Max delay violation	Average delay violation	Data loss
SEaMLeSS	auto	0.73s	0.09s	32.85%
CTRL	0.99	41.10s	23.33s	0.00%
Aurora	0.92	1.16s	0.09s	37.59%
Aurora	0.93	1.80s	0.19s	36.82%

TABLE I
DELAYS AND DATA LOSS WITH QN-COMPLEX AND S_r.

the beginning, the headroom factor tuned for CTRL is correct so it manages to keep the response time at the delay target. SEaMLeSS does not have such a well-tuned headroom factor, yet it quickly picks up the correct value and can control the response time as efficiently as CTRL. When some background jobs are launched and share the processor with the DSMS at the 100^{th} second, the headroom factor used for CTRL is no longer correct, making the response time twice as high as the delay target. SEaMLeSS, however, is able to adapt very quickly to the change, and still honor the delay target. Fig. 3 shows the headroom factor adjustment made by SEaMLeSS in response to the change in the system environment.

When the query network is flat, which is the case in this experiment, [14] has shown that CTRL outperforms Aurora. Therefore the fact that SEaMLeSS performs equivalent or better than CTRL in this experiment also means that SEaMLeSS outperforms Aurora with a flat query network.

2) With a complex query network (Fig. 4 and Table I): In this experiment we use a complex query network (QN-complex) for which CTRL's estimation is no longer correct. Since [14] does not compare CTRL's performance to Aurora for complex query networks, we include Aurora in this evaluation to confirm that SEaMLeSS also outperforms Aurora in this case.

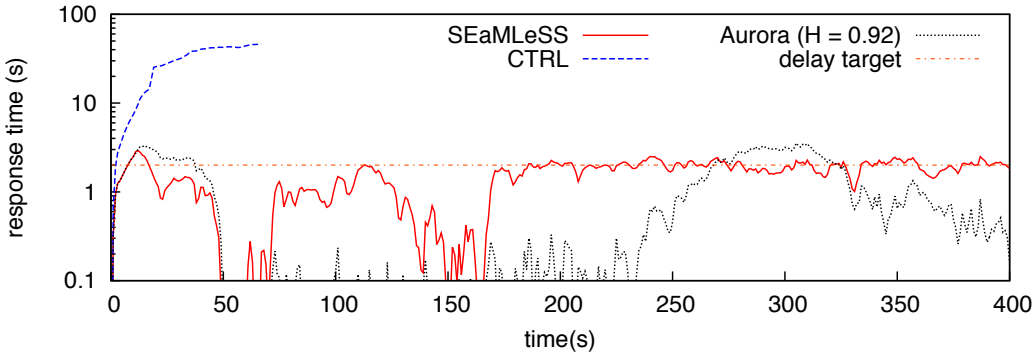

Fig. 4. Response times with QN-complex and S_r. The X-axis plots the input timestamps, showing that within the specified experiment time the system under CTRL was only able to process tuples coming in the first 66 seconds. Note that the Y-axis is in logarithmic scale.

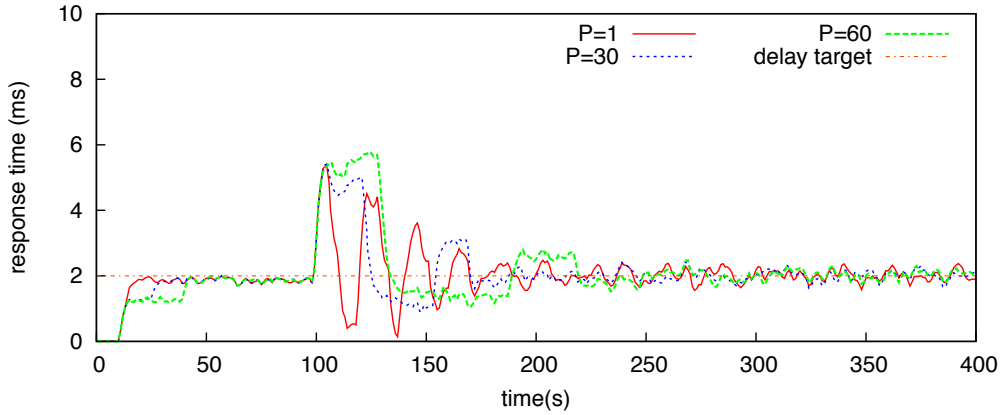

Fig. 5. Effect of different headroom adjustment periods on SEaMLeSS.

Since the Aurora scheme does not suggest a way to pick a correct value for the headroom factor, we ran it with a range of possible values. However, in this setup no value of the headroom factor could enable it to perform equivalently to SEaMLeSS. If the headroom factor is too small, the response time is kept well below the target at all time by dropping much more data unnecessarily. When the headroom factor equals 0.92 (Fig. 4), the average delay violation of Aurora is roughly the same as SEaMLeSS but Aurora drops considerably more data (Table I). Increasing the headroom factor to 0.93 makes the delay violation to be significantly higher (due to the higher peak in the response time) and the data loss is still higher than SEaMLeSS. This is consistent with the properties of Aurora analyzed in [14]: the Aurora method is not aware of the delay target and cannot recover from a previous wrong decision since it does not look at its outcomes.

The method given by CTRL to tune the headroom factor cannot be applied with the complex query network: no matter how we change the value of the headroom factor, the delay estimated by CTRL does not match the real output delay. Because the query network contains a shared operator, an input tuple actually corresponds to several tuples in the output flow. CTRL cannot recognize this mapping and hence it miscalculates the length of the virtual queue. We still tried to run CTRL with the headroom factor equal 0.99 (i.e., the value we tuned for QN-flat). As we show in Fig. 4, CTRL totally fails to control the response time: it does not realize that the system is overloaded and does not apply any shedding, letting the response time of the query output exceed the delay target quickly (the Y-axis is in log scale). As a result, when the experiment stops (for all schemes, we let the experiment run for 420s), the system with CTRL has only been able to process input tuples coming in the first 66s (out of 400s). We tried some other values of the headroom factor from 0.8 - 0.99 as well, but they do not make any observable difference to the performance of CTRL compared to that in this case.

B. Sensitivity analysis (Fig. 5)

In this section we show the sensitivity level of SEaMLeSS to the headroom adjustment period, denoted P.

We ran SEaMLeSS's headroom adjustment algorithm varying P from 1 to 60 control periods with the experiment presented in Sec. V-A1, in which the headroom factor changes significantly at the 100^{th} second. We expect that when P is large, it takes SEaMLeSS longer to adjust the headroom factor but it is more stable. When P is smaller SEaMLeSS starts adjusting earlier but it tends to make more inaccurate adjustments and hence becomes less robust against fluctuation caused by system events.

The sensitivity analysis shows that in this case SEaMLeSS works best (in term of both delay violation and data loss) with P in the range of [20-40]. To provide more insight, we show in Fig. 5 the three cases with P equals 1, 30 and 60. As expected, when P = 1 the adjustment decision is much less accurate so it has to adjust it many times before getting to the appropriate value. And its response time afterward also fluctuates more than the others. With P=60 the load shedder has to wait for a long time unnecessarily before adjusting the headroom factor.

In other experiments when we keep the headroom factor unchanged during the execution time, there is no considerable difference for P in [20-60] (the auto-adjustment of the headroom factor at the beginning is too small to observe the effect of P), and we have shown the results with P = 30.

VI. Conclusions

Motivated by the shortcomings of the state-of-the-art load shedders, namely CTRL and Aurora, we have proposed SEaMLeSS to efficiently answer the "when and how much" questions of the DSMS load shedding problem. SEaMLeSS, like CTRL, is able to bound the response time of the system to a specified delay target and outperforms Aurora. Yet SEaMLeSS overcomes CTRL's limitations: it is applicable to all types of query networks and does not require a manually tuned headroom factor. We confirmed the addvantages of SEaMLeSS over CTRL and Aurora through an extensive set of experiments on AQSIOS, using both real and synthetic input streams.

Acknowledgments: We would like to thank Profs. Tu and Prabhakar for providing us the code from their CTRL paper. This work was partially supported by NSF awards IIS-0534531 and IIS-0746696, by an Andrew Mellon Fellowship, and a gift from EMC/Greenplum.

References

[1] D. J. Abadi et al., "Aurora: a new model and architecture for data stream management," in *VLDB Journal*, 12(2): 120–139, 2003.
[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *PODS '02*.
[3] S. Chandrasekaran et al., "Telegraphcq: continuous dataflow processing," in *SIGMOD '03*.
[4] F. Reiss and J. M. Hellerstein, "Data triage: An adaptive architecture for load shedding in telegraphcq," in *ICDE '05*.
[5] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *VLDB '03*.
[6] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: efficient load shedding techniques for distributed stream processing," in *VLDB '07*.
[7] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE '04*.
[8] B. Mozafari and C. Zaniolo, "Optimal load shedding with aggregates and mining queries," in *ICDE '10*.
[9] Y. Chi, H. Wang, and P. S. Yu, "Loadstar: load shedding in data stream mining," in *VLDB '05*.
[10] J. H. Chang and H.-C. M. Kum, "Frequency-based load shedding over a data stream of tuples," *Inf. Sci.*, 179(21): 3733–3744, 2009.
[11] R. Dash and L. Fegaras, "Synopsis based load shedding in xml streams," in *EDBT/ICDT '09 Workshops*.
[12] R. V. Nehme and E. A. Rundensteiner, "Clustersheddy: load shedding using moving clusters over spatio-temporal data streams," in *DASFAA'07*.
[13] B. Gedik, K.-L. Wu, and P. S. Yu, "Efficient construction of compact shedding filters for data stream processing," in *ICDE '08*.
[14] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao, "Load shedding in stream databases: a control-based approach," in *VLDB '06*.
[15] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, "Balancing load in stream processing with the cloud," in *ICDEW' 11*.
[16] B. Kendai and S. Chakravarthy, "Load shedding in mavstream: Analysis, implementation, and evaluation," in *BNCOD '08*.
[17] P. K. Chrysanthis, "Aqsios - next generation data stream management system," *CONET Newsletter*, June 2010.
[18] S. Guirguis, M. Sharaf, P. Chrysanthis, and A. Labrinidis, "Three-level processing of multiple aggregate continuous queries," in *ICDE'12*.
[19] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Algorithms and metrics for processing multiple heterogeneous continuous queries," *ACM TODS*, 33(1): 5.1–5.44, 2008.