DILoS: A Dynamic Integrated Load Manager and Scheduler for Continuous Queries*

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

Abstract—**In a Data Stream Management System (DSMS), continuous queries (CQs) registered by different applications inherently have different levels of importance (i.e., quality of service (QoS) and quality of data (QoD) requirements). Moreover, the shift to cloud services and the growing need for monitoring applications will inevitably lead to the establishment of data stream management cloud services. In such a multi-tenant environment, it is expected that the different applications/users will demand different QoS and QoD requirements for their CQs. In this work, we are proposing an approach that exploits the synergy between scheduling and load shedding to effectively handle different ranks of CQ classes, each associated with different QoS and QoD requirements. The proposed approach was implemented and evaluated within AQSIOS, our DSMS prototype.**

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, which are based on continuous queries (CQs) that look for interesting events over data streams, are becoming popular. These applications are made possible by a Data Stream Management System (DSMS) (e.g., [1], [2], [3]), which processes the data streams *on the fly*.

One important measure of how well a CQ is serviced by the DSMS is the response time of the query output, i.e., the gap between the time an input tuple enters the system and the time the related output is produced. Another important measure is the accuracy of the result. This concern arises when the incoming workload is higher than the system capacity and the system has to apply *load shedding* to avoid accumulated delay. These two evaluation metrics are commonly referred to as Quality of Service (QoS) and Quality of Data (QoD), respectively.

CQs registered by different applications inherently have different levels of criticality, and hence different levels of QoS and QoD. For example, consider two continuous queries: Q_1, which detects fire conditions in a forest (e.g., high temperature, low humidity, and strong wind) and Q_2, which records the average temperature and humidity for scientific observation purposes. As such, Q_1 should have higher priority than Q_2: Q_1 should have lower response time and, if the system is overloaded, it should suffer less data loss than Q_2.

Soon, a DSMS is expected to be available as a multi-tenant service on the cloud. In that case, some stream applications using the service would be willing to pay more to have a better guarantee on the service. When we also consider the inherent differences in criticality of CQs, the need for the DSMS to support priority-based QoS and QoD becomes evident.

Previous works have partially addressed these requirements, both through scheduling and load shedding ([4], [5], [6]). However, none of the previous works, including ours, have exploited the synergy of scheduling and load shedding policies in honoring the priorities of different classes of queries. An ad-hoc plug-in of a prioritized load shedder and scheduler in a system could make it hard for a DSMS to define the actual meaning of the priorities to its users. In this work, we focus on building an integrated approach in which the scheduler and load shedder work in concert to consistently satisfy the priority-based QoS and QoD requirements, as well as to efficiently exploit the system resources.

Toward this, we make the following contributions in this paper:

- We define a priority-based quality model for a DSMS, specifying both QoS and QoD agreements for different classes of queries.
- We propose *DILoS*, a dynamic, i.e., self-managed, integrated load manager and scheduler, in which we combine our two separate previous works (i.e., the adaptive load manager ALoMa [7] and the class-based scheduler CQC [5]) into a novel, unified model. In this model, the load manager is aware of the way the scheduler is enforcing the priorities and therefore can act appropriately to control overloading situations while strictly following the prioritized scheduling policy. In addition, the synergy between the load manager and the scheduler in DILoS allows the system capacity to be used more efficiently.
- We implement and evaluate DILoS in AQSIOS, our DSMS prototype. The experimental results demonstrate that DILoS, while still being consistent in preserving the classes' priorities, maximizes the utilization of the system capacity, hence significantly reducing data shedding due to overloading.

II. System Model

This work is part of the AQSIOS project, in which we build a new generation of DSMS. AQSIOS (Fig. 1) is based on the STREAM prototype source code [2], which we have extended to include new scheduling policies, such as CQC (discussed in Section III-A) and the ALoMa load manager (Section III-B).

*This work was partially supported by NSF awards IIS-0534531 and IIS-0746696 and by a VOSP Fellowship from Vietnam Government.

A query can share with others some of its operators, such as Q_1 and Q_2 in Fig. 1. In this work we assume that there are no operator sharing between queries that belong to different classes. Currently, the system is single-threaded, so all the operators in the query network are scheduled to run sequentially. The *processing delay*, or *response time*, of a tuple is the time elapsed since the tuple enters the system until the related result is output.

The stream applications that register CQs also specify the class the query belongs to. Each class has a specific priority, which determines:

- *The Quality of Service (QoS)* the class should receive. In our model, QoS is defined as the response time of the query output. The system considers two types of QoS: the *normal-state* response time and the *worst case* response time (i.e., delay target). The scheduler considers the relative priorities of the classes. When the system is not overloaded, the average response time of a class with higher priority should be less than or equal to that of another with lower priority (given the same workload). The worst case response time is used by the load manager to control the maximum response time of each class when the system is overloaded.
- *The Quality of Data (QoD) for each class*. In our model, QoD is defined as the percentage of tuples retained in the queries' output in the case the system is overloaded. Currently we do not support a hard threshold for the QoD, but instead aim to provide better QoD for the class of higher priority, i.e., QoD should be degraded according to the class priority.

Informally, if a class of queries has a priority P out of a total P_{total}, the class may use up to $\frac{P}{P_{total}}$ of the system capacity when it needs to, and any load exceeding that portion of capacity is subject to load shedding if the system cannot find any available capacity from the other classes.

We use the approach proposed in [6] to estimate the system load: the total load L per time unit is approximated based on the processing costs and selectivities of the operators, and the input rate of the stream sources. We can think of L as the total time needed to process all the tuples that come to the system in one time unit (say, a second). This total time has to be less than the time unit in order not to overload the system. Hence we can think about the system capacity as 1 time unit. However, since the system spends part of its processing capacity on other tasks such as context switching, statistics collection, etc., the actual portion of the system capacity that can be spent on tuple processing is commonly estimated by a *headroom factor*, typically in the range of (0,1). Note that if query processing can be carried out in parallel on multiple processing units, then the headroom factor can be greater than 1.

III. Background

A. CQC: a prioritized operator scheduler

In [5], we presented the CQC (Continuous Query Class) scheduler to optimize the QoS of query classes with different priorities. CQC is a two level scheduler that performs scheduling of the query classes on level 1 and of the individual operators on level 2.

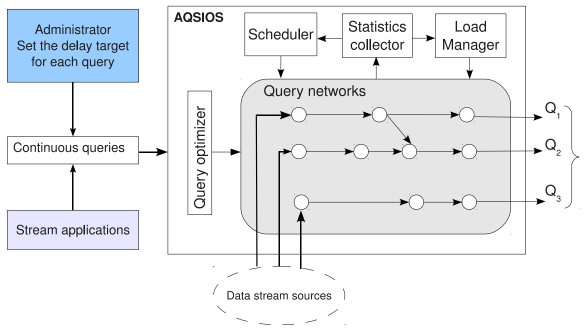


Fig. 1. AQSIOS System model

Level 2 consists of a set of HR (Highest Rate) [12] schedulers, which are designed to minimize the average response time. In particular, HR sets the priority of an operator to be its output rate. In this work, each HR scheduler is responsible for a set of operators that belong to a specific class. In AQSIOS, a source operator is considered part of a query, so each HR class scheduler is implemented in a way that distinguishes between the source and the other non-source operators and schedules them independently according to their input queues. At each scheduling point, an HR class scheduler first schedules the non-source operators and switches to the source operators only when all the other operators have empty input queues.

On level 1, a Weighted Round Robin (WRR) scheduler allocates to each query class i a quota (c_i) equal to a time slice of T_i time units. T_i is the product of the priority of query class i (P_i) and a configurable time period k divided by the sum of all the class priorities ($\sum_{j=0}^{n} P_j$). In AQSIOS, WRR does not preempt HR class schedulers, but uses a negative credit system. If any HR class scheduler i exceeds its quota (c_i), WRR deducts the excess amount from its future quotas.

B. ALoMa: An Adaptive Load Manager

ALoMa [7] is a practical load management module that can be easily incorporated within any DSMS. ALoMa does not require the manual, off-line tuning of the system and workload-dependent parameters such as the headroom factor. Also, ALoMa is generic enough to handle all types of query networks. In [7] we showed that ALoMa, while is self-tuned, can achieve equivalent or better performance compared to that of CTRL [8], the state-of-the-art manually-tuned approach. Interestingly, as we will discuss in this paper, the adaptation ability of the load manager not only absolves the system administrator from manual tuning of the headroom factor, but also plays a key role in enabling the synergy between the scheduler and the load manager.

ALoMa has two basic components that interact with each other: the *statistics-based load monitor* and the *response time monitor*. The core idea behind ALoMa is to use information about the actual response time, provided by the response time monitor, to adjust the estimation of the system capacity that is spent on processing the tuples (i.e., the headroom factor), so

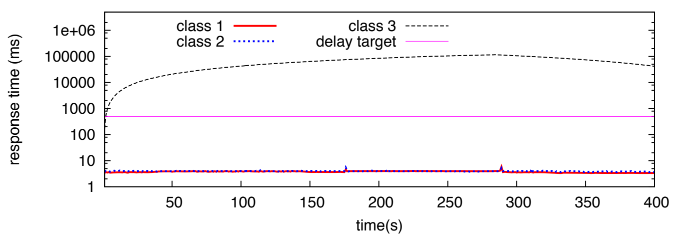


Fig. 2. Response times with S_c and no load shedder

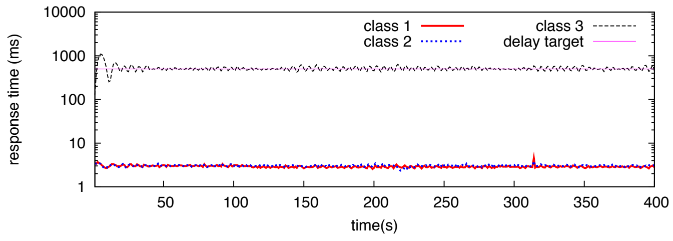


Fig. 3. Response times with S_c and one common load shedder

there is no need to manually tune it. Below we use the term *system capacity* and *headroom factor* interchangeably to refer to the part of capacity spent on processing the tuples.

The intuition that drives the ALoMa is typical of all control systems. The system starts with an initial, reasonable value of the headroom factor (for example, 0.8). Later on, if the load monitor estimates that the system is overloaded but the response time monitor does not observe any abnormality, then ALoMa decides that the system capacity should be higher. On the contrary, if the response time monitor detects that the system is overloaded but the estimated load is still less than the capacity, ALoMa decreases the capacity. When the two components agree with each other, the difference between the estimated load and the system capacity is the amount of load that needs to be removed or can be added to the system.

Let L be the current load and L_C the current system capacity when ALoMa decides that L_C should be adjusted. We adopt the approach expressed by Equation 1 to calculate how much the load manager should increase/decrease the headroom factor.

$$L_{C_{new}} = L_C + \frac{\log_2(k+1)}{k}(L - L_C) \qquad (1)$$

$$\text{where k } = \begin{cases} \frac{L-L_C}{L_C} \times 100, & \text{if } \frac{L-L_C}{L_C} \times 100 \geq 1 \\ 1, & \text{otherwise} \end{cases}$$

The equation is based on the intuition that we can be more aggressive when the gap between L and L_C is small, but should be more conservative when it is big, since the estimated capacity is expected to converge around the correct value.

IV. Scheduler - Load Manager Synergy

A. Why Sheduler - Load Manager Synergy?

When the load is heavy, in a DSMS with a prioritized scheduler we observe that the lowest priority classes receive very bad QoS: their response times quickly exceed their delay targets and keep increasing unboundedly if load shedding is not applied. In other words, these classes are overloaded while the other classes, with higher priority, are not.

In this section, we demonstrate, through a set of experimental results, why a simple combination of a scheduler and a load manager is not as effective as a unified model that integrates the two and makes them aware of each other. These experiments, running in our AQSIOS prototype, use a query network that consists of three classes of queries of the same size, whose priorities are 6, 3 and 1 and delay targets are 300ms, 400ms and 500ms, respectively. We use the queries and input data stream S_c described in Section V.

TABLE I
Average response time and data loss, with S_c and one common load shedder

	Class 1	Class 2	Class 3
Response time (ms)	3.00	3.13	517.07
Data loss (%)	11.42	11.43	11.60

Fig. 2 shows the response time of the three classes when there is no load shedding. We also plot the delay target of class 3 (in this set of experiments the other classes never reach their delay targets so they are not plotted). We see that the response time of class 3 exceeds its delay target (500ms) by a huge amount, while those of the other classes stay at around 3ms. Note that the x-axis is the timestamp of the input tuple related to the output, which explains the decrease in the response time of class 3 at the end of the experiment: due to the huge waiting time, by the time these tuples of class 3 get processed the other two classes have already finished processing all of their input tuples, leaving the full system to class 3.

To satisfy the worst-case response time constraint, one idea is to apply load shedding to keep the response time of class 3 at or below its delay target. We tried this idea by enabling our ALoMa load manager in AQSIOS. The load manager, being unaware of the prioritized scheduling policy, calculates the excess load and applies the shedding to all the inputs of all classes *evenly*. Fig. 3 shows the detailed response times of the three classes under this scheme, and Table I summarizes the average response time and data loss for each class.

Although the load manager successfully controls the response time of class 3 to satisfy the QoS on worst case response time, it does not honor the priorities of the classes with respect to QoD: the three classes lose the same amount of data, and class 1 and class 2 suffer from data loss even though they are not overloaded. One might think of a solution in which load shedding applies only to those queries whose response times exceed the delay target. However, if, for example, both class 2 and class 3 suffer from overloading (i.e., their response times exceed their delay targets), such a scheme would not recognize how much to shed from class 2, and how much from class 3. Therefore, for any thorough solution, the load manager should be able to take into account the priorities of the classes.

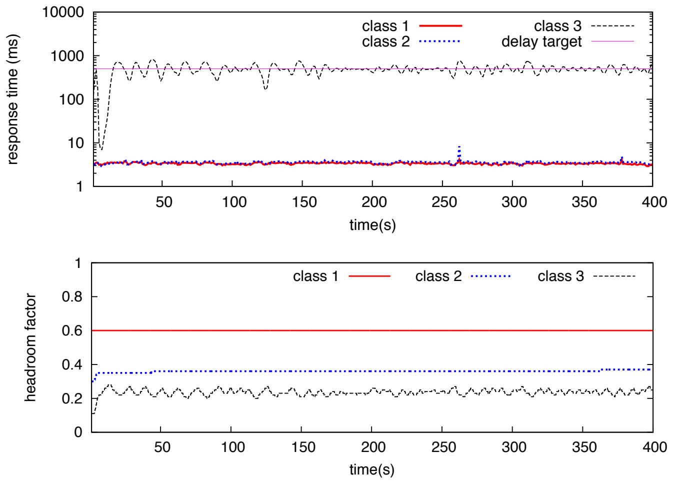


Fig. 4. Response times and estimated headroom factors, with S_c and one load shedder per class

TABLE II
AVERAGE RESPONSE TIME AND DATA LOSS, WITH S_c AND ONE LOAD SHEDDER PER CLASS

	Class 1	Class 2	Class 3
Response time (ms)	3.55	3.75	492.84
Data loss (%)	0	0	35.95

B. Scheduling-Aware Load Manager

Instead of building a separate algorithm for the load manager, we propose an approach that allows it to follow the scheduler's policy in honoring the classes' priorities. We believe (and justify later in this paper) that an integrated approach for the scheduler and the load manager enable us to further optimize the usage of the system capacity while still being consistent in providing prioritized QoS and QoD.

With a prioritized scheduling policy like CQC, each query class can be considered to be receiving a portion of the system capacity that is roughly proportional to the class' priority. Then, the ability of ALoMa to automatically recognize the headroom factor solves the problem: all we need to do is to create a separate instance of ALoMa for each class. At run time each instance can automatically recognize the portion of the capacity that the class actually has, and use it to calculate the excess load that will be shed from the class. This implies that each load manager instance detects and follows the priority enforced by the scheduler.

The top plot in Fig. 4 shows the detailed response time of the three classes under this scheduling-aware load management scheme, in which the response times of the classes are all well controlled and Table II gives the summary of response time and data loss for each class. As expected, with this approach, only class 3, which is the one that is overloaded, experiences load shedding.

The bottom part of Fig. 4 plots the headroom factors estimated by each load manager of each class. At the beginning of the experiment, we initialize the headroom factors for class 1, 2, and 3 to be 0.6, 0.3, and 0.1 respectively, which are proportional to their priorities. However, we observed that at run time the headroom factor for class 3 actually went up to around 0.25, and that of class 3 slightly higher than 0.3. This is due to the policy of CQC: if a class finishes executing all tuples in its queues, the scheduler lets the next class in the round run, even though the former class has not used up its quota. Thus, when a class is lightly loaded (class 1 in this case), part of its assigned capacity is automatically given to the other classes. Note that the headroom factor of class 1, however, does not change and still remains at the initial value because the load manager does not have the necessary signals to decrease it – the load manager decreases the headroom factor of a class internally only when the class is under overload situation. This observation motivated our design of DiLoS to explicitly control the distribution of the excess capacity of a class to the others and consequently, the load managers can use this capacity distribution information to early adjust the headroom factors.

C. DILoS: Scheduler-Load Manager synergy

DILoS is designed with two principles in mind that strongly integrate the scheduler and load manager:

- Distributing the available capacity (e.g., from class 1) to the classes in need based on their priorities.
- Exploiting batch processing to use the system capacity more efficiently.

The second principle is originated from our previous work [7], in which we observed that the higher the number of tuples an operator can process in a batch, the lower the processing cost per tuple. CQC, like most other DSMS schedulers, allows an operator to process up to a certain number of tuples in each scheduling round. However, if the workload is much less than the processing capacity (as in the case of class 1), there are very few tuples waiting in an operator's input queue, so it cannot take advantage of the assigned quota to reduce the processing cost. By explicitly reducing the capacity portion of the lightly-loaded class, we can increase the number of tuples its operators process in batch and hence, the class can fit in the smaller capacity without being overloaded, sharing more capacity to the other classes.

Since each ALoMa instance can estimate the current load as well as the capacity of the class it is in charge of, it can keep track of the average capacity usage of the class, i.e., the ratio between the load and the capacity. This capacity usage of each class is periodically reported to the scheduler. Based on that, for each class *i* the scheduler calculates:

- $demand_i$: the additional percentage of the system capacity the class needs in order to process all of its current load without shedding.
- $supply_i$: the percentage of the system capacity the class can share with others, without itself being overloaded.

More specifically, let u_i denotes the capacity usage of the class, p_i and p_{0i} (both in %) denote its current and original capacity portion, respectively, then $demand_i$ and $supply_i$ are computed as follows:

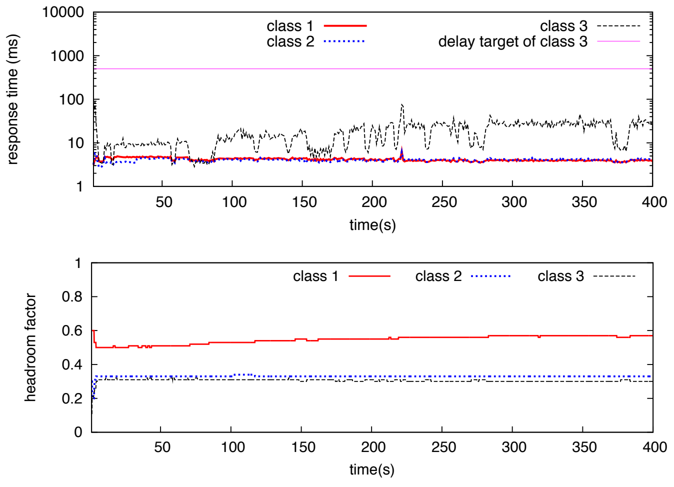


Fig. 5. Response times and estimated headrrom factors, with S_c and DILoS

$$demand_i = \begin{cases} (u_i - 1) \times p_i, & \text{if } u_i < 1 \\ 0, & \text{otherwise} \end{cases}$$

$$supply_i = \begin{cases} (1 - u_i) \times p_i - 0.05 \times p_{0_i}, & \text{if } u_i < 1 - \frac{0.05 \times p_{0_i}}{p_i} \\ 0, & \text{otherwise} \end{cases}$$

Note that to increase the system stability, the scheduler does not take all of the estimated redundant capacity from a class, but conservatively leaves 5% of its original capacity portion.

The scheduler calculates $budget = \sum_i supply_i$, and redistributes the system capacity based on the following criteria:

1) For a class i, after the redistribution either $demand_i$ is satisfied (is 0) or it has at least it original capacity (i.e., original quota).
2) If the original priority of class i is higher than class j, then $demand_i$ must be satisfied using the available budget before $demand_j$.
3) Any available budget, after satisfying all demands, is given back to the classes whose quotas are less than their original quotas. This distribution goes from the class of highest original priority to the class of lowest one.
4) The sum of quotas does not change before and after the redistribution.

In order to help each load manager to quickly adapt to the new value of the capacity portion, the scheduler also changes the headroom factor of each load manager, as in Equation 2. This new value set by the scheduler does not need to be perfectly accurate because the load manager is able to automatically adjust it.

$$headroom_{new} = \frac{quota_{new}}{quota_{old}} \times headroom_{old} \tag{2}$$

The top plot in Fig. 5 shows the detailed response time of the three classes when the system implements our proposed synergy between the scheduler and the load manager instances, and Table III shows the average response time and data loss. Interestingly, none of the classes has to shed data any longer,

TABLE III
AVERAGE RESPONSE TIME AND DATA LOSS, WITH S_c AND DILOS

	Class 1	Class 2	Class 3
Response time (ms)	4.28	4.38	42.95
Data loss (%)	0	0	0

meaning DILoS actually uses the system capacity much more efficiently. The response times of class 1 and class 2 increase by around 1ms, because the synergy forces the operators in these classes to process more tuples in each batch so each tuple has to wait for a longer time. However, we are still consistent in providing better QoS for the class of higher priority: class 1 is still the one with the smallest average response time. Note that the fluctuation in the response time of class 3 is normal due to the fluctuation of the environment and the fact that its capacity usage is around 100%. The bottom plot in Fig. 5 shows that class 3 actually receives more of the system capacity now, which explains why it does not need to drop any data.

V. Experimental Evaluation

As discussed above, we experimentally evaluated the performance of DILoS in AQSIOS. In all the experiments reported in this paper, we set 150ms to be the load management cycle, and the scheduler considers redistributing the system capacity for each class after 10 load management cycles (i.e., 1.5 seconds). We ran each experiment five times; the numbers reported are the average. We use a query network that consists of three classes of queries, whose priorities are 6, 3 and 1 and delay targets are 300ms, 400ms and 500ms, respectively. All the three classes have the same set of 11 queries, consisting of five aggregates, two window joins, and four selects.

We use the following two setups for the input data:

- $\mathbf{S_c}$: All the input streams coming to the three classes have constant input rate of 1300 tuples/sec, which, together with the query network setting, creates a total load that approximates the total system capacity.
- $\mathbf{S_{steps}}$: The input streams for classes 2 and 3 have a constant input rate of 1500 tuples/sec and 900 tuples/sec, respectively. These input rates are expected to overload the classes if they are limited to their originally assigned capacity portions. For class 1, which is the class of highest priority, we change its input rate after every 50 second period (Fig. 6) in order to vary the amount of excess capacity it can share with the other classes.

The S_c input setup has been used in the set of experiments presented in Section IV. The simple pattern of this input allowed us to easily examine the behavior of each scheme and build up our understanding. In this section we show the experimental results using the input S_{steps} to demonstrate that DILoS efficiently redistributes the system capacity to different classes when the input rate suddenly changes significantly.

We show the response times of the three classes under DILoS in Fig. 7, and summarize the average response time and data loss in Table IV. In Fig. 8 we show the changes in the capacity portion of each class, which is reflected through

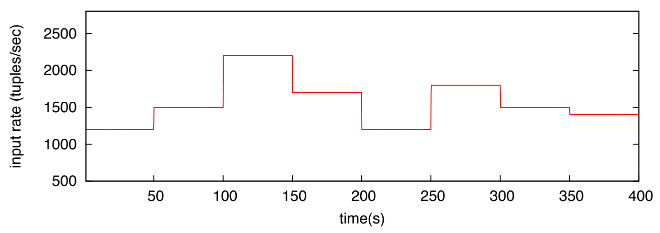


Fig. 6. Input rates for class 1 - input setup S_{steps}

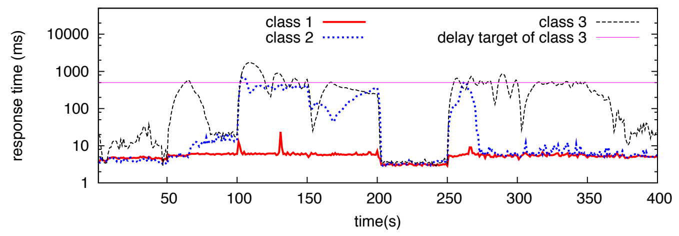


Fig. 7. Response times with S_{steps} and DILoS

the headroom factor estimated by each load manager instance, and the corresponding changes in the shedding rates.

We observe that when the load of class 1 is low, DILoS distributes the excess capacity from class 1 to the other two classes, enabling them to shed less. However, as soon as the load of class 1 increases (e.g., at the 100^{th} second), DILoS gives back to this class all or part of its original capacity so that its performance, as specified by its class priority, is preserved. (In other experiments, not reported due to space limitations, we got similar results for classes with different set of queries.)

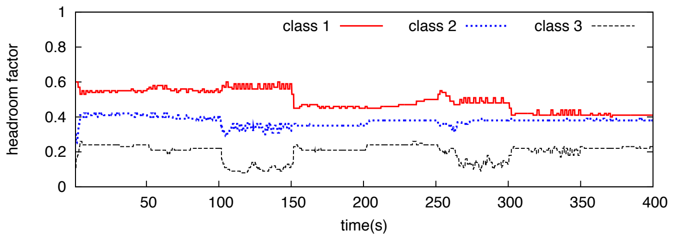
VI. Related work

Load shedding has been proposed in many DSMS architectures as a method to handle overloading (e.g., [6], [9], [10], [11], [8]) but few of them consider the priority of the CQs. CQ priorities have been implicitly considered through loss-tolerance QoS (i.e., QoD) graphs [6], or maximal tolerable relative error [11]. However, the emphasis of these approaches is on load shedding: the load shedder is unaware of the scheduler and does not provide feedback to improve it.

Also, there are many proposals for scheduling the execution of CQs in a DSMS with the objective of optimizing certain performance goals such as minimizing latency ([4], [12]) or minimizing memory requirements ([13]). Related to our work on multi-class CQ scheduling are the works in [1], [4], [11] which consider latency-based QoS functions for each query, and in [14], [15], [16] which schedule real time CQs where each CQ has a deadline. These schemes, however, try to optimize the overall benefit of the system rather than explicitly guarantee the benefit of each class according to its priority. Also, none of them considers the synergy between the scheduler and the load manager as in our work.

VII. Conclusions

In this paper, we proposed to integrate the scheduler and load manager in a DSMS and implemented DILoS in our AQSIOS DSMS prototype to efficiently support priority-based QoS and QoD for continuous queries. Through both analyses and experiments on AQSIOS, we have shown that the self-managed synergy between the scheduler and load manager in DILoS can significantly increase the utilization of the system capacity while still being consistent in enforcing the QoS and QoD of different classes of queries.



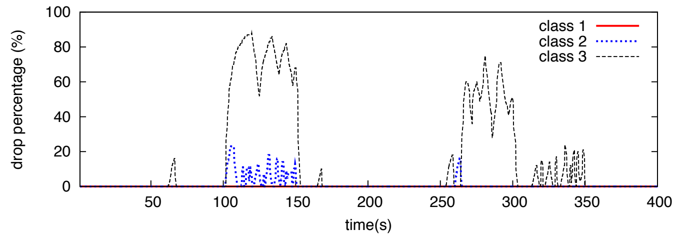


Fig. 8. Shedding and estimated headroom factors, with S_{steps} and DiLoS

TABLE IV
Average response time and data loss with S_{steps} and DILoS

	Class 1	Class 2	Class 3
Response time (ms)	6.67	96.77	226.05
Data loss (%)	0	5.19	50.97

References

[1] D. J. Abadi et al., “Aurora: a new model and architecture for data stream management,” in *VLDB Journal*, 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] D. Carney et al., “Operator scheduling in a data stream manager,” in *VLDB ’03*.
[5] L. A. Moakar et al., “Class-based continuous query scheduling for data streams,” in *DMSN ’09*.
[6] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, “Load shedding in a data stream manager,” in *VLDB ’03*.
[7] T. Pham, P. Chrysanthis, and A. Labrinidis, “An adaptive load manager for the AQSIOS stream engine,” *Technical Report TR-10-175*, 2010.
[8] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao, “Load shedding in stream databases: a control-based approach,” in *VLDB ’06*.
[9] B. Babcock, M. Datar, and R. Motwani, “Load shedding for aggregation queries over data streams,” in *ICDE ’04*.
[10] F. Reiss and J. M. Hellerstein, “Data triage: An adaptive architecture for load shedding in telegraphcq,” in *ICDE ’05*.
[11] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing, 2009.
[12] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, K. Pruhs, “Algorithms and metrics for processing multiple heterogeneous continuous queries,” *ACM TODS*, 2008.
[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, “Operator scheduling in data stream systems,” *The VLDB Journal*, 2004.
[14] Y. Wei, S. H. Son, and J. A. Stankovic, “Rtstream: Real-time query processing for data streams,” in *ISORC ’06*.
[15] D. Kulkarni, C. V. Ravishankar, M. Cherniack, “Real-time load-adaptive processing of continuous queries over data streams,” in *DEBS’ 08*.
[16] S. Wu, Y. Lv, G. Yu, Y. Gu, and X. Li, “A qos-guaranteeing scheduling algorithm for continuous queries over streams,” in *APWeb/WAIM ’07*.