The VLDB Journal (2016) 25:197–221
DOI 10.1007/s00778-015-0411-4

REGULAR PAPER

Avoiding class warfare: managing continuous queries with differentiated classes of service

Thao N. Pham[1] · Panos K. Chrysanthis[1] · Alexandros Labrinidis[1]

Received: 28 June 2014 / Revised: 22 October 2015 / Accepted: 31 October 2015 / Published online: 12 November 2015

Abstract Data stream management systems (DSMSs) offer the most effective solution for processing data streams by efficiently executing continuous queries (CQs) over the incoming data. CQs inherently have different levels of criticality and hence different levels of expected quality of service (QoS) and quality of data (QoD). Adhering to such expected QoS/QoD metrics is even more important in cases of multi-tenant data stream management services. In this work, we propose DILoS, a framework that, through priority-based scheduling and load shedding, supports differentiated QoS and QoD for multiple classes of CQs. Unlike existing works that consider scheduling and load shedding separately, DILoS is a novel unified framework that exploits the synergy between scheduling and load shedding. We also propose ALoMa, a general, adaptive load manager that DILoS is built upon. By its design, ALoMa performs better than the state-of-the-art alternatives in three dimensions: (1) it automatically tunes the headroom factor, (2) it honors the delay target, (3) it is applicable to complex query networks with shared operators. We implemented DILoS and ALoMa in our real DSMS prototype system (AQSIOS) and evaluate their performance for a variety of real and synthetic workloads. Our experimental evaluation of ALoMa verified its clear superiority over the state-of-the-art approaches. Our experimental evaluation of the DILoS framework showed that it (a) allows the scheduler and load shedder to consistently honor CQs' priorities, (b) significantly increases system capacity utilization by exploiting batch processing, and (c) enables operator sharing among query classes of different priorities while avoiding priority inversion, i.e., a lower-priority class never blocks a higher-priority one.

Keywords Data stream management system · Continuous query · Multi-tenant · Load shedding · Scheduling

1 Introduction

Motivation: Today the ubiquity of sensing devices as well as mobile and web applications continuously generates a huge amount of data which takes the form of streams. These data streams are typically high volume, often high velocity (speed), and high variability (bursty). In order to meet the near-real-time requirements of the monitoring applications and of the emerging "Big Data" applications [29], incoming data streams need to be continuously processed and analyzed. Data stream management systems (DSMSs) (e.g., [1,3,5,7,13,17,18]) have become the popular solutions to handle data streams by efficiently supporting continuous queries (CQs). CQs are stored queries that execute continuously, looking for interesting events over data streams as data arrives, *on the fly*.

CQs are registered for different purposes and inherently have different levels of criticality. For example, assume the data feed of a personal health monitoring device such as Fitbit, Microsoft Band, and Apple's iWatch. Also assume two continuous queries: CQ_1, that monitors the user's heart rate for the possibility of a heart attack due to dangerously high

This article enhances and extends preliminary work [37] that was presented in the SMDB'11 Workshop.

✉ Thao N. Pham
thao@cs.pitt.edu

Panos K. Chrysanthis
panos@cs.pitt.edu

Alexandros Labrinidis
labrinid@cs.pitt.edu

[1] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

beats per minute (as appropriate for the particular user given his/her age, physical condition and medical history), and CQ_2, that monitors the user's overall activity level (using the heart rate monitor, a pedometer and other sensors) in order to nudge him/her to remain physically active. Clearly, CQ_1 is more critical than CQ_2 and as such can demand a higher *priority* than CQ_2 in sharing the DSMS' processing capacity. Another example of CQs with different priorities is in the financial sector. Assume three CQs that monitor the transactions of credit card uses: CQ_3 is used to detect fraud (e.g., identity theft), CQ_4 is used to notify users of low credit balance remaining in their accounts, and CQ_5 is trying to find good targeted advertisements for the credit card users. Again, these three CQs have different levels of criticality with CQ_3 being more important than CQ_4 which is more important that CQ_5. A third example is one where CQs to detect a tsunami [4] would have higher priority than those that detect, understand and predict El Nino and La Nina [6]. Finally, multi-tenant DSMSs normally provide different service groups with different costs (e.g., gold, silver, and bronze), which determine the priority of the queries subscribed to each group and hence the quality guarantees, i.e., service level agreements (SLAs).

Many commercial mission critical and analytics applications require processing of queries with priorities. This demand for priority-based query processing has motivated a number of commercial database and data warehouse systems to provide it. For example, EMC Greenplum has put effort in supporting query priorities [35]. As another example, HP Vertica allows defining multiple query resource pools with different limits on memory usage, runtime priority, etc., each for a group of queries [2].

Similar demand for priority-based query processing exists for data stream management systems. Although support for CQs with different priority levels in commercial DSMSs is still limited, the priority of CQs has been discussed in research prototypes such as Aurora [7], MavStream [17], and IBM System S [5].

Problem Space: For the above reasons, we consider a DSMS (Fig. 1), which supports multiple classes of service for CQs. Each CQ submitted to this DSMS belongs to a query class that is associated with a priority. The system admits queries based on its provisioned processing capacity and the expected loads of the queries. However, due to the burstiness of data streams, the incoming workload can be, at times, higher than the system capacity, making the system *overloaded*. The two important requirements for this multiple-CQ-priority DSMS are:

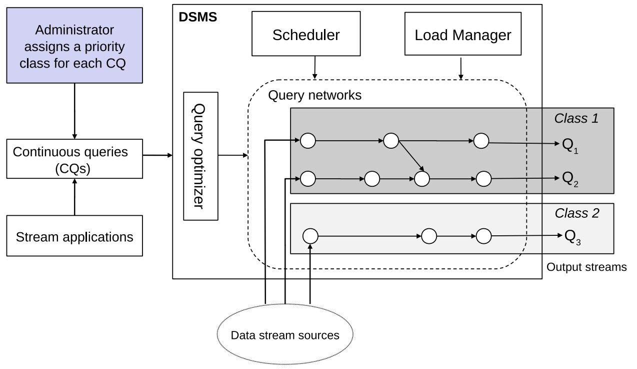

Fig. 1 AQSIOS system model

- **Guarantee an upper bound on the response time:** Most stream applications require an upper bound on the response time, which is also referred to as *Quality of Service* (QoS) in the *worst case*, or *delay target*. Each class can require a different delay target; normally, a higher-priority class requires a smaller delay target. Because of this requirement, when the DSMS is overloaded, it has to apply *load shedding*, i.e., drops an appropriate amount of data to avoid processing it further.
- **Minimize data loss with priority consideration:** With load shedding applied to honor delay targets, all classes desire as little data loss, i.e., as high *Quality of Data* (QoD), as possible. At the minimum, each CQ class expects QoD according to their priorities.

Previous works have partially addressed these requirements, either through scheduling (e.g., [10,15]) or through load shedding (e.g., [17,42]), yet these were only considered in isolation. Clearly, enforcing worst-case QoS in overload situations while providing prioritized QoD for query classes requires the participation of *both* the scheduler and the load manager (i.e., load shedder): The load manager decides how much data to drop from each class, whereas the scheduler decides how much processing time each query has, which consequently governs how much data the class can process in a period. The challenge of how to integrate scheduling and load shedding in a way to consistently honor the priorities of CQs still remains. Even if the load manager and the scheduler are both aware of the CQs' priorities and enforce policies that seem to be consistent with each other, undesired situations can still happen, as we demonstrate in the example below.

Consider a simplified example of two CQs Q_1 and Q_2, in which Q_1 and Q_2 have the same cost, yet Q_1's priority is twice as high as Q_2. Without going into the details of the scheduling and load shedding policies, let us consider a period during which the scheduler effectively executes 10 tuples of Q_1 and 5 tuples of Q_2 in every second, for a total processing capacity of 15 tuples/s. The DSMS also has a prioritized load shedder that, once detecting the excess load, will drop twice as much load from Q_2 as from Q_1. Assuming that the input rate coming to both Q_1 and Q_2 is 9 tuples/s (for a total of 18 tuples/s), the load shedder calculates the excess rate to be 3 tuples/s and, following its policy, will drop 1 tuple from the input of Q_1 and 2 tuples from the input of Q_2. We observe two problems. First, shedding 2 tuples from Q_2 is not sufficient to control Q_2's load since 7 tuples/s is still higher than Q_2's processing rate of 5 tuples/s. As such, the response time of Q_2 increases unboundedly and the system would violate any delay target set for Q_2. Second, shedding from Q_1 while it is running underloaded is a waste of the system processing capacity and unnecessarily affects Q_1's QoD.

The problems described above are due to the fact that the load manager is not aware of the way the scheduler is enforcing its priority policy and that the scheduler does not recognize the level of capacity usage of each CQ to fully utilize the system capacity. Motivated by this observation, we propose DILoS (Dynamic Integrated Load Manager and Scheduler), *a novel framework that exploits the synergy between the load manager and the scheduler to enable consistent and effective integration between the two modules in the DSMS*.

Intuitively, for our simplified example, DILoS allows the load manager to recognize that Q_2 is overloaded by 4 tuples/s and Q_1 is 1 tuple/s underloaded, so it drops 4 tuples from Q_2 and nothing from Q_1. At the same time, the load manager reports the load status of each CQ to the scheduler. Hence, in the next cycle the scheduler can choose to give the redundant CPU time from Q_1 to Q_2, enabling Q_2 to process up to 6 tuples/s. If such an adjustment is made, the load manager will reduce the shedding of Q_2 to 3 tuples, improving Q_2's QoD while fully using the system capacity.

We proposed the idea of DILoS in a short paper in the SMDB'11 workshop [37]. To the best of our knowledge, we were the first to identify and analyze the problem of integrating a priority-aware scheduler and load manager in a DSMS.

Contributions: We make the following contributions in this paper:

- We present formally and thoroughly DILoS, our novel framework that allows consistent integration between the scheduler and load manager in a DSMS to support multiple priority classes of CQs.
- We propose ALoMa, a new adaptive load management scheme that enables the realization of DILoS. ALoMa is also a general, practical DSMS load shedder that outperforms the state of the art in deciding when the DSMS is overloaded and how much load needs to be shed.
- We show how DILoS solves the congestion problem typically encountered when there is operator sharing between classes of different priority in a fully optimized query network.
- We provide a thorough experimental evaluation and analysis of DILoS in AQSIOS [21], our real DSMS prototype. The results of our evaluation, with both complex synthetic and real input rate patterns, show the robustness of DILoS and confirm that DILoS achieves the goals of (1) consistently supporting multiple levels of priorities for CQs and (2) maximizing the utilization of the system processing capacity to reduce the need for load shedding.

Roadmap: Section 2 presents the basic concepts of our assumed system model. Section 3 formally analyzes the problem and presents the overview of our proposed DILoS framework. Section 4 describes our ALoMa load manager, and Sect. 5 presents an implementation of DILoS. Section 6 shows how DILoS solves the congestion problem under inter-class sharing. We describe our experimental evaluation in Sect. 7 and discuss the possibility of incorporating different schedulers and load shedders into DILoS in Sect. 8. We review the related work in Sect. 9 and conclude in Sect. 10.

2 Background

We consider a multi-tenant DSMS (Fig. 1) in which each submitted CQ belongs to a priority class. We assume that the query class priorities have been quantified into discrete values, with higher value meaning higher priority.

Like most other DSMS architectures (e.g., [7,13,18]), our assumed DSMS has a CQ processing engine, together with a query optimizer, a scheduler, and a load manager/ shedder. Each submitted CQ is compiled and optimized into a query plan consisting of multiple relational operators (i.e., select, project, join, or aggregates), one or more source operators, and an output operator.

Each operator has one or more input queues depending on its type. Tuples produced by an operator will be placed in the input queues of the next operators downstream. Since CQs exist in the DSMS for a long time, their plans are optimized together forming a query network, in which a query can share with others some of its operators, such as Q_1 and Q_2 in Fig. 1. In such a case, the intermediate tuples produced by the shared operator will be placed in a shared input queue for the two operators downstream. The output of each CQ is continuously stored or streamed to applications.

During execution, the scheduler is responsible for assigning each operator a time slot to run. Like most other DSMS schedulers, to reduce context switching overhead, the schedulers allows *batch processing* by letting each operator process up to a predefined number of tuples in its input queue during each invocation.

Definition 1 The *response time* of a tuple is the time elapsed since the tuple enters the system until the related result is output. The response time consists of processing time and queuing time. Only tuples that are output by the query network contribute to the measuring of response time.

Definition 2 The *worst-case Quality of Service (worst-case QoS)* of a query is the *highest response time* tolerated by the stream applications using the query. In this paper, the worst-case QoS is also referred to as *delay target*, denoted by D. We assume that all queries in the same class have the same delay target.

The load manager takes action when the system gets overloaded, i.e., when the rate of the input load is higher than the processing rate of the system. In such a case, the load shedder drops a necessary amount to prevent the response time of the system from increasing unboundedly.

Definition 3 The *Quality of Data (QoD)* of a query is the percentage of output tuples retained after shedding, compared to the case with no shedding.

A good load manager should maximize QoD (i.e., minimize data loss) while controlling the response time.

3 DILoS: Dynamic integrated load manager and scheduler

3.1 DILoS as a general priority-based scheduler-load manager integration framework

At runtime, a priority-based scheduler applies its policy to assign an execution time slot for each operator in the query network. In general, the scheduler takes into account the priorities of CQ classes by given a higher-priority class a higher amount of time to execute the operators of the CQs belonging to the class.

Definition 4 Let C_k denote the kth CQ class, with corresponding priority P_k. At the class level, in a specific time period T a *scheduling policy* can be represented by a function $f_T : f_T(P_k) = T_k$, such that $\sum_k(T_k) \leq T$, where T_k is the total time the class C_k receives during T.

Our example in Sect. 1 suggests that, in a specific period, the load manager can act consistently with the scheduler's policy if it knows (1) the current incoming workload of each class and (2) the maximum workload each class can handle (i.e., the processing capacity of the class).

We observed that, within a single class, the load management tasks are the same as what a general load manager would do for a typical DSMS without CQ priority, i.e., monitoring system load, calculating excess load based on the system processing capacity, and applying load shedding fairly for all CQs. In other words, each class can be viewed as a *virtual system*.

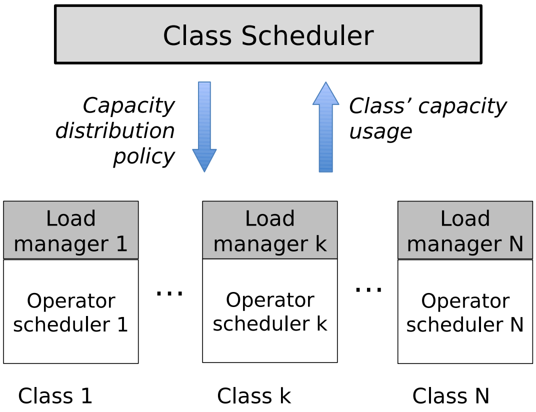

Fig. 2 Overview of the proposed DILoS framework

Based on this observation, we propose the DILoS framework in which *each class has its own load manager instance*. Each class has an incoming workload L_k and a system capacity LC_k proportional to T_k. We separate the scheduler into two levels: a *class scheduler* and a set of *local operator schedulers*. Each class C_k has its local operator scheduler, which, in each period T, schedules the operators of the CQs belonging to C^k using the assigned time T_k. The *class scheduler* schedules the CQ classes, i.e., determines the function $f_T(P_k)$ that maps the priority of C_k to T_k (capacity distribution policy). In general, the two-level scheduling can be just a logical separation: the DSMS might not explicitly have the class scheduler, in which case f_T is defined implicitly through the time the scheduler assigns for each operator of a class.

Figure 2 illustrates our DILoS framework. For simplicity, we assume for now that there is no operator sharing between classes of different priorities. We drop this assumption later in Sect. 6.

The design of DILoS allows the load manager to follow *exactly* the policy enforced by the scheduler. Within a class, the load manager instance acts as if it is managing a DSMS with all CQs having the same priority: it monitors the incoming load, detects and shed the excess load to comply with the worst-case QoS requirement of the class. The class' priority is reflected automatically: the class with higher priority is scheduled with a larger time slot (bigger processing capacity) and therefore will have a higher QoD (less data loss due to load shedding) given the same workload.

In addition, the load manager also reports the capacity usage (i.e., the ratio $\frac{L_k}{LC_k}$) of its class to the class scheduler. Based on that information, the class scheduler can consider adjusting its capacity distribution policy to better exploit the system capacity. An example of such an adjustment is taking the redundant capacity from one class and distributing it to the classes in need.

The advantage of DILOS's synergy is not only that it repairs the over-provisioning of system capacity for some classes, but it also exploits batch processing to further increase system capacity utilization. We explain further the benefit of batch processing through an experiment presented in Sect. 7.2.2.

3.2 Load management challenge

In order to successfully control the load in a CQ class, the class' load manager needs to (1) estimate the incoming load of the class and (2) detect the real system capacity of the class.

Estimate the incoming load of a class: In [42], the authors present a method to estimate the total system load L. We can apply this method with a small modification to estimate the incoming workload of each class.

Definition 5 The *incoming load of class* C_k in a time unit, denoted L_k, is given by:

$$L_k = \sum_i \left(r_i^k \times load\_coef_i^k \right) \tag{1}$$

where r_i^k denotes the input rate of the ith input stream of class C_k, and load_coef$_i^k$ is the load coefficient of the stream.

Definition 6 The *load coefficient of the* ith *input stream of class* C_k, denoted load_coef$_i^k$, in the case of a flat query (i.e., no shared operator), is given by:

$$\text{load\_coef}_i^k = \sum_j \left(c_j \times \prod_{1 \leq m < j} \text{sel}_m \right) \tag{2}$$

where c_j is the processing cost per tuple of the jth operator in the path from the input stream to the corresponding output, and sel_j is the operator's selectivity. In the case of fan-out query plans, i.e., with shared operators, it recursively sums up the load coefficient of every sub-path along the way. More information can be found in [42].

Since the input rates, costs and selectivities all change frequently at runtime, L_k need to be recalculated in every period.

Detect the real system capacity of a class: This is one of the biggest challenges in materializing DILoS. The state-of-the-art load shedders estimate the system capacity of a DSMS by using a headroom factor H, which is either assumed available or manually tuned. This is not practical since the value of the headroom factor can change during execution due to changes in the system environment, as explained in Sect. 4.1. In the case of our per-class load management, the *actual capacity portion* each class obtains (L_{C_k}) is represented by a headroom factor H_k, which is usually different from its expected value of $\frac{T_k}{T}$. This deviation is partly due to the existence of other tasks, either inside or outside the DSMS, sharing the CPU time, and partly due to the scheduling details as we will show later in our experiments. Because the existing load shedders cannot tune H automatically, when serving as a class' load manager they would also not be able to recognize the actual capacity portion that the class has. Therefore, they would not be able to successfully control the load of the class to honor its delay target.

In addition, we realize that there is also a lack of a load manager that can both strictly honor the worst-case response time and be applicable to all types of query networks. This motivates us to develop a more practical load management scheme for DSMSs in general and for DILoS in particular. We present ALoMa, our new load management scheme, in the next section.

4 Load shedding and ALoMa

In this section, we present one of the key contributions of this paper, a practical and general load manager (load shedder).

4.1 The "when and how much" problem and state of the art

The load shedding problem is typically defined by four questions: *when* to shed load, *how much* load to shed, *where* in the query network to apply load shedding, and *what* data should be shed. Among these, solutions for the two questions of "when and how much to shed" are crucial for all load shedding schemes to work correctly, while approaches for "where and what to shed" rely on a good estimation of when and how much to shed and try to reduce the impact of shedding by exploiting application-specific constrains.

It is therefore important to develop a good load manager that can provide good answers to the questions of when and how much to shed. Such a load manager is necessary for both DILoS and any general purpose DSMS. Surprisingly, few existing works have addressed these questions and none has addressed them thoroughly.

A first attempt to answer the "when and how much questions", used in Aurora [42] and implied in STREAM [14], is to compute the coming load L (based on statistics about operators' costs and selectivities), compare it to the system capacity L_C (which is estimated by a headroom factor H), and shed an amount equal to L-L_C if $L > L_C$. Although the Aurora approach is theoretically sound, in practice it has the following two problems:

1. *Ad hoc selection of headroom factor:* Aurora does not provide a method to pick the correct headroom factor and assumes one is available.

2. *Not delay-target aware:* Aurora simply assumes that the response time will be acceptable if the excess load is shed. As pointed out in [44], Aurora does not have a self-correcting mechanism to prevent the response time from exceeding a delay target.

CTRL [44] is a control-based approach proposed to addressed the second shortcoming of Aurora, 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 ith period is then estimated by the following equation, called the *delay estimation model*:

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

where y^i is the response time at the ith period, $q^{(i-1)}$ is the length of the virtual queue after the $(i-1)$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, respectively.

Applying control theory on the above model, CTRL computes the maximum number of tuples allowed to come in the next period such that the response time converges quickly to the delay target. The experimental results in [44] show that CTRL can keep the response time around the target, which the Aurora approach cannot, while shedding only 1–2 % more data than Aurora.

CTRL, however, has also two major shortcomings:

1. *Manual tuning of the headroom factor:* In [44], the authors *manually* try different values of H in Eq. 3 and pick the value such that the estimated delay best matches the real response time. This manual, offline tuning is clearly not practical since the headroom factor is not constant and can change during execution.
2. *Not applicable in complex query networks:* When the query network has shared operators, join, or aggregation operators (we call it *complex*), the one-to-one mapping of an input tuple to an output tuple, which is the way CTRL estimates the length of the virtual queue, is no longer correct. Figure 3 gives an example of such a case, where the result from the Select operator σ_2 is shared by two queries, and one of the operators is a Join ($\bowtie_1$). In this case, simply increasing the length of the virtual queue by 1 for each incoming tuple from the two sources and decreasing 1 for each tuple output or discarded would not work.

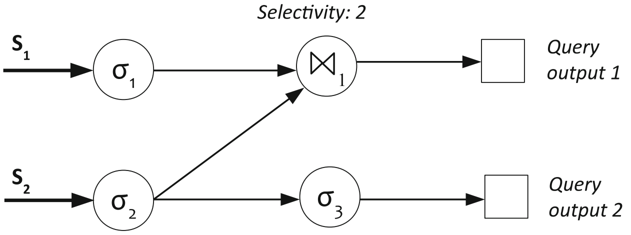

Fig. 3 A query network with joins and shared operators, for which the delay estimation model of CTRL would not work

Some other schemes have also been discussed, yet they are not as complete as Aurora and CTRL. The scheme in [30] is effectively the same as Aurora without taking into account the headroom factor (i.e., assuming that the headroom factor always equals 1). The schemes in [39] and [31], like CTRL, monitor the input queue(s) to decide when the system is overloaded, yet they do not discuss how the number of queued tuples can be used to infer whether the system is overloaded.

In order to enable the realization of DILoS as well as to provide a more practical and flexible load manager for DSMSs, we proposed a new scheme that has both the complementary strengths of CTRL and Aurora, while overcoming their weaknesses. More specifically, the new scheme aims at the following properties:

- Delay-target aware.
- No manually tuned headroom factor required.
- Applicable for all types of query networks.

4.2 ALoMa: Adaptive 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 *automatically* adjust the estimation of the system capacity (i.e., the headroom factor) based on the actual response time provided by the response time monitor. The load monitor estimates the incoming load using the method in [42] (i.e., Eq. 1 when there is only one class) and calculates the excess load. This load estimation is based on the statistics on input rates and operators' costs and selectivities, which are continuously collected in the DSMS during execution.

The system starts with some initial value of the headroom factor that might be reasonable (for example, 0.8). Later on, if the load monitor estimates that the system is overloaded but the response time monitor still observes normal response time, ALoMa decides that the system capacity should be higher. On the contrary, if the response time monitor detects that the response time is already higher than the delay target but the incoming load is still less than the estimated capacity, ALoMa decreases the estimated capacity. When the two components agree with each other, the difference between the estimated load and the system capacity is the amount of

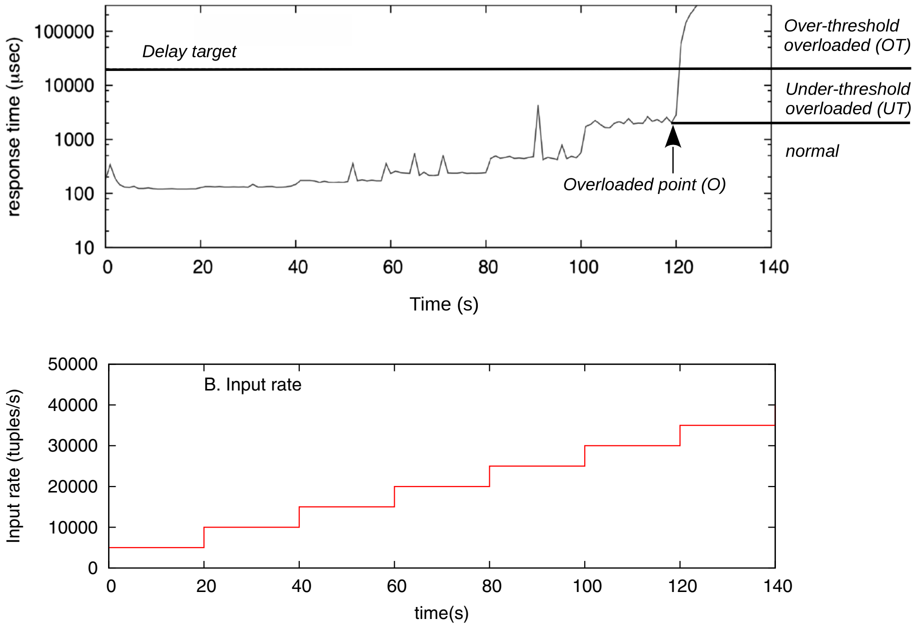

Fig. 4 Response time (**a**—*top plot*) with increasing input rate (**b**—*bottom plot*) and its imply on system's load state

load that needs to be removed or can be added to the system. Next we explain the intuition behind ALoMa's decisions.

4.2.1 *Observing the response time*

One important part of developing ALoMa was to identify what the response time implies about the system's load status, so we studied the response time of the system (Fig. 4a) in response to step changes of the input rate (Fig. 4b). All experiments were carried out on AQSIOS, our experimental DSMS prototype described in Sect. 2. Note that the Y-axis in Fig. 4a is in log scale. The input rate starts from 5000 tuples/s and increases by 5000 tuples/s after every 20 s.

From the 0th to 20th second, the response time remains at around 120μs. One can think that this 120μs reflects the processing cost per tuple and that the system will be overloaded with an input rate greater than 1 tuple/150μs (about 8300 tuples/s). However, we can observe that during the next 20 s when the input rate reaches the value of 10,000 tuples/s the response time jumps to a higher value, but it remains constant during that 20-s period. This trend continues in all of the other 20-s periods before the 120th second. This means there is no accumulation of queuing delay over time, and the system is not overloaded until the input rate exceeds 35,000 tuples/s.

This phenomenon is due to batch processing. As the input rate increases, more tuples are waiting every time an operator gets executed, so it can process more tuples in a batch (up to a predefined batch size) and reduce the processing cost per tuple. Therefore, the system can endure input rates that are higher than the anticipated one. Figure 5 confirms our explanation by showing a huge fluctuation of the processing cost *per tuple* as the input rate changes (we circle some of the points where the cost decreases significantly as the input rate comes to a peak). On the other hand, this decrease in processing cost results in higher response time since every tuple has to wait for the others in the same batch.

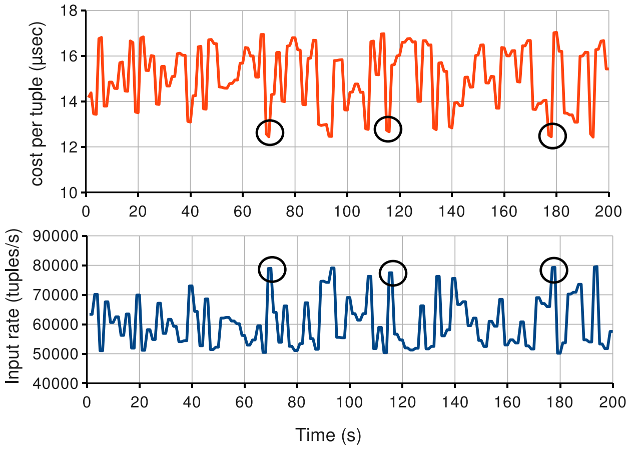

Fig. 5 Cost fluctuation in response to changes of input rate, measured on the AQSIOS system

Note that there are some occasional overshoots in the response time. This is due to events such as operating system interrupts and can occur randomly at any point during the execution time.

When the input rate exceeds 35,000 tuples/s in Fig. 4b, the corresponding response time in Fig. 4a goes up dramatically due to the accumulated queuing time and the system can be considered to be overloaded. If the user-specified delay target D (the horizontal line in Fig. 4a) is higher than the response time before this overloading point, which is usually the case in practice, the system can be allowed to run in an overloaded state as long as the response time is still below the target.

Let O denote the point after which the system starts to be overloaded (i.e, the 120th second in Fig. 4). Based on the above observation, we can map the response time to the following three load states of the DSMS, each one requiring a different action from the load manager:

- *Normal*: the system is not overloaded, the response time is below or equal to the response time at the O point.
- *Under-threshold overloaded (UT)*: the system is overloaded so the queuing time starts accumulating, the response time is greater than that at the O point but still less than the delay target.
- *Over-threshold overloaded (OT)*: the system is overloaded and the response time is higher than the delay target.

We explain later at the end of Sect. 4.2.3 how we find the O point in practice.

4.2.2 Increasing and decreasing the capacity

When ALoMa decides that the estimated headroom factor H should be increased, a straightforward answer is to set L_C (i.e., H) equal to L, since the system can withstand the load of L without being overloaded.

However, consider the case when a high input rate is measured at time t to calculate the load L. At that time, it is possible that the response time is still that of those tuples coming at a much lower rate from the previous period. So ALoMa would then make a mistake by setting L_C equal to L. The dynamic nature of ALoMa enables it to quickly correct the mistake, but a less aggressive solution will improve its performance.

Given that the system environment is fairly stable, the headroom factor usually fluctuates with small amplitudes and big, sudden changes just happen once in a while. Therefore, when the gap between L and L_C is small, we can be more aggressive in moving L_C toward L (i.e., when the gap is small enough, we can set L_C equal L). In such cases, the impact of a mistake due to not-up-to-date statistics, if any, is also small. On the other hand, if the gap is big, we should be more conservative and move L_C by a smaller fraction of the gap, because the disagreement of the two components (which leads to the decision to adjust L_C) is more likely to be caused by the not-up-to-date statistics and the impact of an error could be big.

We codify the above ideas into Eq. 4. Note that when the gap between L_C and L gets bigger, this formula moves L_C by a bigger *absolute* amount, but the ratio of that amount to the gap is smaller.

$$L_{C_{new}} = L_C \pm \frac{\log_2(z+1)}{z}|L - L_C|$$

$$\text{where z } = \begin{cases} \frac{|L-L_C|}{L_C} \cdot 100 \text{ if } & \frac{|L-L_C|}{L_C} \cdot 100 \geq 1 \\ 1 & \text{otherwise} \end{cases} \qquad (4)$$

4.2.3 The ALoMa algorithm

The pseudocode in Algorithm 1 shows the skeleton of ALoMa. Periodically, the load monitor recomputes the current incoming load L, and the response time monitor determines the current load state of the system (lines 2, 3).

Algorithm 1 ALoMa

```
1: BEGIN
2: L := load_monitor.compute_current_load()
3: state := response_time_monitor.detect_current_state()
4: if $L > L_C$ then
5:   if $state = normal$ then
6:     Increase $L_C$
7:   else if $state = OT$ then
8:     Shed $(L - L_C)$ more load
9:   else {$state = UT$}
10:    if ($shedding\ is\ being\ applied$)
       and ($response\_time \leq previous\_response\_time$) then
11:      Increase $L_C$
12:      Reduce shed amount by x%
13:    end if
14:  end if
15: else {$L \leq L_C$}
16:  if $state = OT$ then
17:    if ($response\_time \geq previous\_response\_time$) then
18:      Decrease $L_C$
19:      Shed x% more load
20:    end if
21:  else
22:    if $shedding\ is\ being\ applied$ then
23:      Reduce shed amount by $(L_C - L)$
24:    end if
25:  end if
26: end if
27: END
```

Load rate $L >$ estimated capacity L_C There are three cases to consider when the current load rate L is greater than the estimated capacity L_C.

- If the state reported by the response time monitor is *normal*, then the estimated capacity L_C is increased following Eq. 4 (lines 5, 6).
- If the state is *OT*, ALoMa sheds an additional amount equal to the difference between L and L_C, because the two components are agreeing with each other (lines 7, 8).
- If the state is *UT*, ALoMa further checks if load shedding is being applied and the response time is not increasing (line 10). If true, ALoMa is shedding more than necessary and so it decides to increase L_C (line 11). Also, because the system at this time tends to be able to endure a load

higher than L, although it is not clear how much higher, ALoMa tries to reduce the shed load by $x\%$ (line 12). ALoMa learns the result of this trial in the next cycle and if the same situation is observed, it increases x. The algorithm starts with $x = 1\,\%$, which is the minimum increase/decrease in the shedding amount that we used in the system. x is increased by the binary logarithm of k, which is the number of times the situation has been observed in a sequence. More specifically, x is given by Eq. 5

$$x = 1 + \log_2(k) \tag{5}$$

Load rate $L \leq$ estimated capacity L_C When the current load rate L is smaller than or equal to the estimated system capacity L_C, we only need to consider whether or not the delay target is violated (i.e., the system is in OT state).

- If the system is in OT state (line 16), ALoMa continues to check whether the response time is not decreasing (line 17). If this is true, the estimated capacity L_C needs to be decreased toward L following Eq. 4 (line 18), since it is likely higher than the correct value. Also, the fact that the response time is higher than the delay target and is not decreasing means that ALoMa needs to shed more data to bring the response time back to the target. However, since the load now is smaller than the estimated capacity, it is not clear how much more data should be shed. We also approach this by trying to drop an additional $x\%$ (line 19), with x started as $1\,\%$ and increased following Eq. 5.
- If the system is not in OT state, which means the two components are agreeing with each other, ALoMa reduces $(L_C - L)$ from the current shedding amount being applied, if any.

One question in this algorithm is how to recognize the precise O point to distinguish the normal state from the UT state which is, unfortunately, impossible in practice. However, in the design of ALoMa, the only purpose of recognizing the UT state is to know whether or not to increase the estimated capacity *early* (lines 5, 6). Therefore, a rough estimation of this point is sufficient: The response time monitor signals that the system is in UT state whenever the response time doubles the smallest response time it observed so far. It is not a problem if this estimated point is a little higher than the actual value, because once the system enters the overloaded state, the response time increases very quickly and exceeds this higher value no later than it does the correct one. Thus, the load manager can stop increasing the estimated capacity just in time. It is also fine if the estimation point is lower than the real one, as there is a provision for the estimated capacity to be increased when the system is overloaded, should it be smaller than the real one (line 11). We can periodically refresh the smallest response time by doubling the current value and updating it with the smallest observed one since then.

Note that we are assuming a feasible delay target which is higher than the O point. However, the algorithm still holds if the delay target is smaller than the O point but still higher than the response time when the system is very lightly loaded (e.g., before the 20th second in Fig. 4, which approximates the processing cost per tuple). In such a case, the UT state will never happen, and the system capacity is not fully used. If the delay target is smaller than the lightly loaded response time, the load shedder cannot honor it unless shedding everything. But this means the original provisioned capacity is not sufficient and no load shedder can deal with it.

4.2.4 Summary of ALoMa's properties

Advantages: As confirmed later through experiments, ALoMa achieves the stated goals, i.e., being applicable to all types of query networks and able to honor the delay target without requiring any manually tuned headroom factor. In addition, ALoMa offers another advantage (compared to CTRL): ALoMa does not assume the fairness of the operator scheduler while CTRL does. This fact comes directly from the design of each approach: CTRL estimates the response time based on the number of tuples in the virtual queue, while ALoMa bases its decision on the real response time at each output. Therefore, if the scheduler is priority-based, CTRL will only control the average response time across all queries to the delay target, so the query with lower priority might suffer from a response time much higher than the delay target. ALoMa, on the other hand, will make sure that the response times of all queries honor the delay target. We summarize in Table 1 the properties of ALoMa compared to the two state-of-the-art approaches, i.e., CTRL and Aurora.

Overhead: At every load management cycle, ALoMa needs to (1) recompute the total load of the system and (2) adjust

Table 1 ALoMa's properties compared to the state of the art

	ALoMa	Aurora [VLDB'03]	CTRL [VLDB'06]
Automatically tune headroom factor	√		
Honor delay target	√		√
Applicable to complex query networks (including shared operators)	√	√	
Independent of scheduler's fairness	√	√	

the headroom factor and calculate the amount of load to drop. The time complexity of (1) is $O(Op)$, where Op is the number of operators in the query network, and the cost of (2) is a small constant (a few numeric calculations). ALoMa, as well as CTRL and Aurora, uses the statistics on response time and operator costs and selectivities, which has time complexity of O(T*Op) where T is the number of incoming tuples. However, a typical DSMS system would still need to collect these statistics for a variety of purposes such as scheduling, query optimizing, and performance auditing. Therefore, it is reasonable to exclude these costs from ALoMa's overhead.

Worst-case: As with any adaptive technique, the worst-case scenario of ALoMa is when the headroom factor (i.e., its adaptivity object) goes up and down very frequently, causing a value of the headroom factor to become stale before ALoMa has even learned it. Such an unstable environment would be hostile to any adaptive load management techniques.

The worst-case workload for ALoMa, as well as any load management scheme, is when the system is so overloaded that it calls for 100 % shedding (we know the system still needs to spend some CPU cycles on dropped tuples). If such a case persists, load shedding is no longer a sufficient solution and the system has to be either scaled out or re-provisioned.

5 DILoS implementation

In this section, we present an implementation of DILoS that uses ALoMa, our proposed adaptive load manager.

5.1 Load manager

We create one instance of ALoMa to be the local load manager of each class. ALoMa's self-tuning ability allows the ALoMa instance to automatically recognize the *actual* capacity portion L_{Ck} (represented by H_k) that the corresponding class obtains. Consequently, each ALoMa instance manages to control the load of its class as if it is managing a virtual system. After calculating the load that exceeds the capacity portion of the class, the ALoMa instance sheds this excess load from the class by specifying the calculated shedding rate uniformly across the source operators of the class. Figure 6 illustrates this implementation, in which the dark operators are the source operators with a load shedder embedded.

5.2 Scheduler

In this implementation, we use a two-level, class-based DSMS scheduler proposed in [10], called CQC. As indicated in Sect. 3.1, although the physical separation of the scheduler into two levels is not required in our general DILoS framework, it is easier for an actual two-level scheduler to develop a capacity redistribution policy.

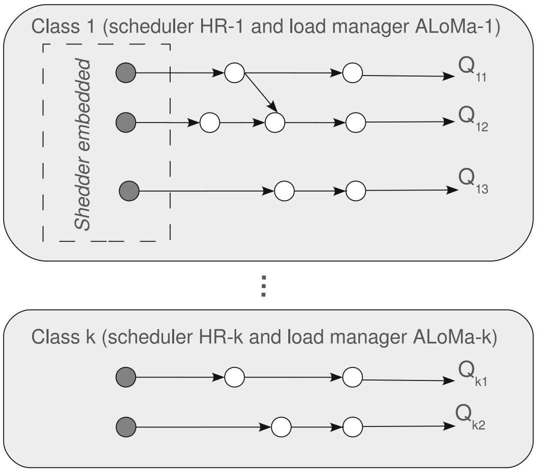
Class 1 (scheduler HR-1 and load manager ALoMa-1)
Shedder embedded
Q_{11}
Q_{12}
Q_{13}
⋮
Class k (scheduler HR-k and load manager ALoMa-k)
Q_{k1}
Q_{k2}

Fig. 6 Per-class load management with ALoMa without inter-class sharing

CQC is a class-based scheduler that supports CQ classes with different priorities, essentially giving more execution time to the class of higher priority. At the class level, a *Weighted Round Robin (WRR)* scheduler allocates to each query class C_k a time quota T_k such that $T_k = \frac{P_k}{\sum_i (P_i)} \times T$. At the operator level, there is a set of slightly modified HR (Highest Rate) [40] schedulers. Each modified HR scheduler is in charge of the set of operators that belong to a specific class. The modified HR scheduler aims to preserve the goal of the original priority-based HR scheduler to minimize the average response time, yet eliminates starvation within a class. More details on CQC can be found at [10].

5.3 Capacity redistribution

After every period, each ALoMa instance reports to the class scheduler the capacity usage $u_k = \frac{L_k}{L_{Ck}}$ of the class. In order for the scheduler to adjust its decisions based on each class' capacity usage, we extend its policy to incorporate capacity redistribution. Intuitively, the class scheduler recognizes the available capacity from classes that are running underloaded and distributes this capacity to the classes that are overloaded following a "highest priority first" rule. Specifically, for each class C_k the scheduler calculates: demand$_k$, which is the additional percentage of the system capacity the class needs in order to process all of its current load without shedding, and $supply_k$, which is the percentage of the system capacity the class can share with others without itself being overloaded.

Let u_k denote the capacity usage of class C_k, and L_{Ck} and $L^0_{Ck} = \frac{P_k}{\sum_i (P_i)}$ denote its current capacity and its initial expected capacity portion, respectively. Values demand$_k$ and supply$_k$ are computed as follows:

$$\text{demand}_k = \begin{cases} (u_k - 1) \times L_{Ck} & \text{if} \quad u_k < 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{supply}_k = \begin{cases} (1 - u_k) \times L_{Ck} - 5\,\% \times L^0_{Ck} & \text{if} \quad u_i < 1 - \frac{5\% \times L^0_{Ck}}{L_{Ck}} \\ 0 & \text{otherwise} \end{cases}$$

Note that in order 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. This small amount of 5 % of a class' original capacity is reserved so that the often small perturbations of input load do not overload a class and lead to a new capacity re-distribution. Using a higher percentage would increase the stability of the capacity distribution and decrease the possibility of a class having to shed tuples when input load suddenly increases. Yet, a higher percentage means the system capacity is not used as fully. Other customizations for this trade-off can be trivially incorporated into DILoS (e.g., higher percentage may be used for critical classes).

The scheduler calculates $budget = \sum_k \text{supply}_k$, and redistributes the system capacity as follows:

1. For a class k, after the redistribution, either demand$_k$ is satisfied (is 0) or it has at least its 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 remaining budget, after satisfying all demands, is returned to the classes whose quotas are less than their original quotas. This proceeds from the highest to the lowest class.

The capacity portion of each class resulted from this redistribution, denoted $L^{\text{new}}_{C_k}$, is the expected capacity portion of the class in the next period. As such, the scheduler calculates the time quota T^{new}_k for the next period as $T^{\text{new}}_k = \frac{L^{\text{new}}_{C_k}}{L_{Ck}} \times T_k$. The sum of time quotas should 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 given in Eq. 6. This new value set by the scheduler does not need to be perfectly accurate because the load manager is able to automatically adjust it.

$$H^{\text{new}}_k = \frac{T^{\text{new}}_k}{T_k} \times H_k \tag{6}$$

5.4 Overhead of DILoS

The overall overhead of DILoS includes the cost of the statistics collection and the cost of redistributing the system capacity among classes. As discussed in Sect. 4.2.4, a typical DSMS system needs to collect these statistics for a variety of purposes. Therefore, the mere cost added by DILoS is the cost of redistributing the system capacity among the classes. This cost actually depends on the specific policy incorporated. For the specific implementation presented in this paper, the redistributing requires one pass to compute demand$_i$ and supply$_i$, and another pass to distribute the total budget. Therefore, this process has time complexity of O(C) where C is the number of priority classes. Because C usually ranges from a few to tens, and the redistributing only happen once after several scheduling cycles, this cost is negligible. In fact, as shown in our experiments, this extra cost of DILoS is obscured by the benefit it brings: significantly more data can be processed (i.e., much less shedding).

6 Inter-class sharing in DILoS

In a fully optimized query network, there can be sharing between classes of different priority. We explain in this section the congestion problem caused by this inter-class sharing and show how DILoS solves this problem.

6.1 Congestion problem

Given a prioritized scheduler such as CQC, intuitively the shared segment between a query of high priority and a query of lower priority should remain in the high-priority class in order not to affect its performance. Figure 7 illustrates this, in which a query of class 1 (higher priority) shares a segment with a query of class k (lower priority), and the shared segment remains in class 1.

However, this still could lead to a situation when the performance of the high-priority query is negatively affected, which is due to the congestion at the end of the shared segment. The intermediate tuples produced by the shared segment are placed in a shared queue for the downstream operators to read from. While the downstream operator belonging to the high-priority class can consume these tuples fast enough to keep up with the production rate, the operators belonging to the low-priority class, however, are much slower. Therefore, the intermediate tuples accumulate and once they fill the queue, the upstream segment has to stop processing and wait, causing the corresponding high-priority queries also to be blocked. Note that this problem persists even if each downstream operator has its own input queue for the intermediate tuples instead of using a shared queue: the upstream shared segment still needs to postpone its processing if one of the queues becomes full.

6.2 Handling inter-class sharing in DILoS

Interestingly, this problem can be solved with an appropriate employment of load management: as long as the low-priority class is not overloaded, i.e., it can keep up with the incoming workload including the input fed by the shared segment, there will be no congestion of intermediate tuples at the end of the shared segment.

Claim 1 *If the load manager manages to keep the response time of the low-priority class to its delay target, the number of tuples accumulating in the shared queue is no higher than the ratio R between the delay target of the low-priority class and the average processing cost per tuple at that low-priority class.*

Proof (By contradiction) Let d be the delay target and c the average processing cost per tuple at the lower-priority class. We define $R = d/c$.

Assume that the load manager satisfies the delay target d and there are $S > R$ tuples accumulating in the shared queue.

It is known that the response time of an output tuple is equal to its processing time plus its waiting time. With S tuples in the shared queue (waiting to be processed by the low-priority class), the waiting time of a new tuple entering the queue, which is to be processed by that class, is going to be $S * c$ and its response time t is going to be greater than $S*c$. Since $S > R$ and $R = d/c$, then $t > d$. This contradicts the fact that the load manager can control the response time of the class to be no more than the delay target ($t <= d$). □

A direct consequence of Claim 1 is that, with ALoMacon enabled, which can guarantee the delay target, as long as the shared queue size is big enough to contain R tuples, the high-priority class is not affected by the congestion problem. This is a reasonable assumption for the queue size, since this ratio is normally within tens to hundreds (in our setup it is around 25–50) and can be either estimated in advance or dynamically extended during execution. Clearly, with the seamless integration of ALoMa, DILoS inherently solves the congestion problem that exists with any class-based scheduler, allowing inter-class sharing for a more optimized query network.

When there is sharing between a higher-priority class and a lower-priority class, the ALoMa instance which is in charge of the lower-priority class views the first operator(s) in the class after the shared segment as the source operator(s) of the class, so the shared segment is excluded from the lower-priority class from a load management perspective. In our current implementation in this paper, we embed load shedding into the source operators, which means this operator also has a shedder embedded. Figure 7 illustrates this method, in which the shared segment is moved completely to the higher-priority class (class 1), while the load manager of the low-priority class (class k) behaves as if query Q_{k1} starts from the dark operator after the shared segment.

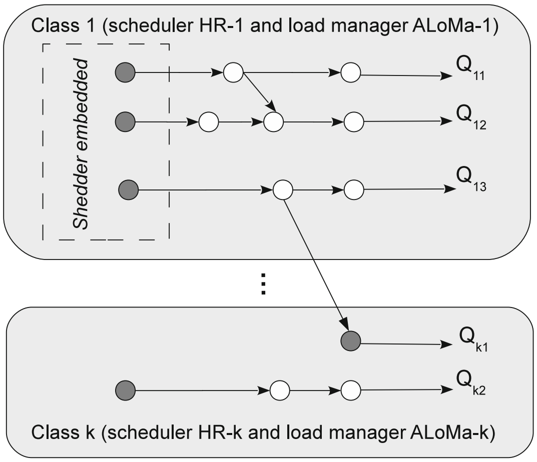
Class 1 (scheduler HR-1 and load manager ALoMa-1)
Shedder embedded
Q11
Q12
Q13
Qk1
Qk2
Class k (scheduler HR-k and load manager ALoMa-k)

Fig. 7 per-class load manager, with class 1 (high priority) sharing a segment with class k (lower priority)

Such a sharing can be trivially applied to more complicated cases when a segment is shared among several classes: The shared segment will belong to the highest priority class and all the load managers of the other classes will consider the corresponding first operators after the shared segment as sources of their classes.

The above approach for inter-class sharing guarantees the original benefit of the high-priority class: Sharing should not affect its performance negatively. At the same time, although it does not appear to benefit directly from the sharing, there is a potential advantage for it: when the load of the lower-priority class becomes lighter thanks to sharing, it can have some redundant capacity to share with the high-priority class when necessary.

The effect on the lower-priority class, however, is twofold. It is clear that when the high-priority class has enough capacity to process all of its incoming load, the lower-priority class takes advantage of the shared processing to reduce its own incoming load. However, once the high-priority class becomes overloaded, it will apply the shedding at all of its sources, including the shared ones, which results in the loss of QoD for the low-priority class even if the class is not overloaded. We believe that such a case is rare, for the higher-priority class should be provisioned with higher capacity (relative to its load) than lower-priority ones. We can also apply differentiated shedding between shared and not shared segments.

This discussion about handling inter-class sharing assumes that the load shedder randomly drops tuples. If a semantic load shedder is used, it assumes that all the classes sharing a query segment consider the same semantics for the tuples coming into the segment (i.e., there is no case when, for example, a tuple is important to a higher-priority class but not important to a lower-priority class).

7 Experimental evaluation

We evaluated our proposed schemes (the ALoMa load manager and an implementation of DILoS) in AQSIOS, our DSMS experimental platform developed on top of the STREAM code base [13]. We divide our experiments into two sets: one evaluating ALoMa, our adaptive load manager, as a stand-alone DSMS load manager in comparison with the state of the art, and the other evaluating the DILoS framework (with ALoMa as the local load management in each class). All experiments were run 5 times, and we report the averages.

7.1 ALoMa evaluation

7.1.1 Experiment settings

We experimentally evaluated the performance of ALoMa compared with two existing schemes, namely CTRL [44] and Aurora [42] (Sect. 4.1).

Query networks: We use three query networks as described below:

- *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 [44][1]. We use this query network in our experiments to create a setting where CTRL can achieve its best performance. The simple, flat query network enables the correct calculation of the virtual queue in CTRL, even though such a query network is not representative of real applications.
- *QN-complex:* is a big query network containing 1140 operators. The query network contains 60 identical groups of 4 queries, with select, project, source and input operators. The queries in the same group read data from the same stream source. We intentionally let the queries in each group share some operators with each other, which creates a case where CTRL is not applicable, as analyzed in Sect. 4.1.
- *QN-long*: This query network contains long queries (i.e., queries having many operators). A representative query in this network is presented in CQL syntax [13] below[2], with S, T, U, V, W and M being the six stream sources:

```
SELECT l. avg(m) FROM
ISTREAM
( SELECT S.l AS l,
         (S.m + T.m + U.m + V.m + W.m + X.m)/6 AS m
  FROM S[Range 10 seconds],
       T[Range 10 seconds],
       U[Range 10 seconds],
       V[Range 10 seconds],
       W[Range 10 seconds],
       X[Range 10 seconds]
  WHERE S.l = T.l and T.l = U.l and U.l = V.l
        and V.l = W.l and W.l = X.l
) [Rows 10]
GROUP BY l
HAVING avg(m) < 40.0;
```

  Effectively, the query has five Joins and five Range windows, one Relation-to-stream operator (ISTREAM), one Group-aggregate and one Row window, and one Select. In addition, the query has five Stream sources and one Output operator, for a total of 20 operators. There are five groups in the query network, each containing 4 queries with multiple levels of sharing. More specifically, two of the queries in each group share with each other the segment from stream sources up to the group-aggregate, while sharing with the other two queries the stream sources and the first range window join.

Input data: We use two streams of synthetic data, denoted SA_c and SA_{step}, and one of real data SA_r. We generated the input tuples for each source beforehand and stored them in a file. Each tuple has a timestamp, which indicates the time the tuple will arrive at the system during execution (relative to the experiment's start time) and reflects the input rate.

- SA_c: has a constant input rate of 200 tuples/s, which is within the system capacity, for the first 10 s, and then goes to 350 tuples/s, which overloads the system, until the end of the experiment at the 400th second. SA_c is used when we want to keep the input rate constant to clearly examine the effect of the factor of interest.
- SA_{step}: has an initial constant input rate of 200 tuples/s for the first 10 s, then goes up to a higher level every 40 s until the system is so overloaded that load shedding can no longer control the response time. We use this input to test a worst-case situation.

[1] In fact, the CTRL paper does not even use real operators: It used only delay operators to simulate an operator with a certain processing cost and selectivity. The Aurora paper uses only a simulation for its experiment, not a real DSMS.

[2] Note that because STREAM (inherited by AQSIOS) does not support everything in the CQL syntax, we had to split the query into several virtual queries in the actual script.

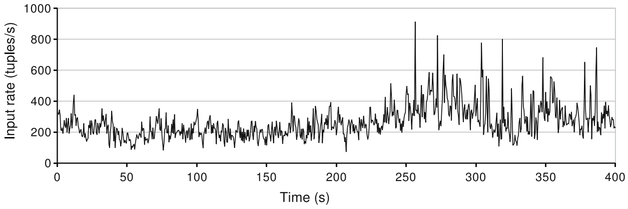

Fig. 8 Input rate of the real data in SA_r and SD_r

- SA_r: is a trace of TCP packets between the Lawrence Berkeley Laboratory and the rest of the world[3]. Figure 8 shows the input rate of this stream. This input rate allows us to evaluate the performance of our scheme, compared to the others, with the fluctuations of a real-world data stream. Note that this real input rate pattern is the same as that of the input used in the CTRL paper.

We use a uniform distribution for the values of the tuples in order to fix the selectivities of the select operators and make sure they are not the cause for the cost fluctuation.
Parameters: We choose the values for the delay target $D = 2s$, which are the same to that used in the CTRL paper. We use the control period $T = 0.5s$ (CTRL paper experimentally shows that [250–1000 ms] is the best range for T given that $D = 2s$).

In order to choose an appropriate headroom factor for CTRL, we follow the method used in [44] and 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 one another. 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 and QN-long query network, as anticipated, it is impossible for us to find a suitable headroom factor for CTRL since the estimation of the virtual queue by CTRL is no longer correct. Therefore, in this case, we have to run CTRL with the headroom factor obtained with the QN-flat query network, as well as some other values down to 0.8. For ALoMa, we set the initial value of the headroom factor to 0.8.

7.1.2 *ALoMa vs CTRL under CTRL's ideal setting (Fig. 9; Table 2)*

In this experiment, we use the flat query network QN-flat so that all the calculations of CTRL's delay estimation model are correct. In addition, we manually tune its headroom factor and keep the system environment unchanged during execution, so that the tuned value remains accurate (even though this is unrealistic for real systems). The real input SA_r is used for the experiment. We run ALoMa under the same setting, but *without the manual tuning of the headroom factor*.

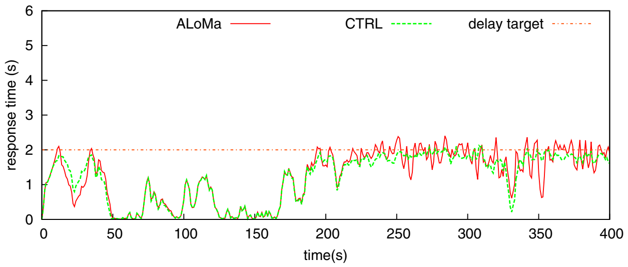

Fig. 9 Response times with QN-flat and SA_r

Table 2 Average delay and data loss, with QN-flat and SA_r for CTRL with optimal, manually tuned headroom factor

	Average delay violation (s)	Max delay violation (s)	Data loss (%)
ALoMa	0.05	0.62	21.36
CTRL	0.01	0.35	21.41

Figure 9 shows the response time of the output under ALoMa and CTRL. Table 2 summarizes the average delay violation, the maximum violation observed, and the data loss under each scheme. ALoMa has higher maximum violation, and from Fig. 9 we can observe that the response time fluctuates more under ALoMa than under CTRL. This, however, is expected, since ALoMa has to make multiple adjustments of the headroom factor on the fly, while CTRL has the headroom factor manually pre-tuned. Nevertheless, ALoMa manages to honor the delay target, closely to what CTRL does. The average delay violation under ALoMa is slightly bigger than CTRL but is still very small (0.05 s compared to the delay target of 2s)

Clearly, ALoMa achieves performance very close to that of CTRL under CTRL's ideal setting, *even though ALoMa makes all the headroom factor adjustment automatically, without requiring any manually tuned value as CTRL does.*

7.1.3 *ALoMa verss CTRL under system environment changes (Fig. 10)*

Despite being carefully selected, a specific value of the headroom factor is not guaranteed to be correct for the whole execution time. In fact, it is virtually guaranteed not to be correct for the whole execution time. To illustrate this, we launch two background jobs while the DSMS is running. In order to clearly show the effect of the system environment change, we use the input SA_c with constant input rate.

Figure 10 shows the response time of the system under CTRL, which uses a fixed, manually tuned headroom factor, and our proposed scheme ALoMa, which automatically

[3] 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.

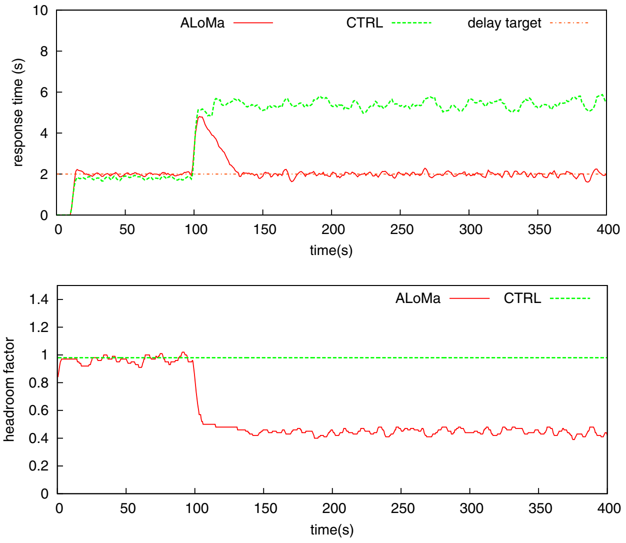

Fig. 10 Effect of environment changes on CTRL and adaptation of ALoMa. Top plot shows the response time; bottom plot shows the headroom factor recognized by each scheme. Total data loss for ALoMa and CTRL is 62.98 and 62.69 %

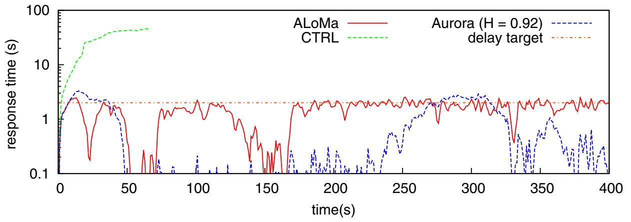

Fig. 11 Response times with QN-complex and SA_r. Note that the X-axis plots the input timestamps, showing that within the specified experiment time the system under CTRL was only able to process tuples coming in the first 66 s

Table 3 Delays and data loss with QN-complex and SA_r

	H	Max delay violation (s)	Average delay violation (s)	Data loss (%)
ALoMa	Auto	0.75	0.06	32.41
CTRL	0.99	41.10	23.33	0.00
Aurora	0.92	1.16	0.09	37.59
Aurora	0.93	1.80	0.19	36.82

adjusts the headroom factor at runtime. We can observe that as the background jobs are launched and share the processor with the DSMS at the 100th second the headroom factor used for CTRL is no longer correct, making the response time to be twice as high as the delay target. ALoMa, however, is able to adapt very quickly to the change as expected and still honor the delay target despite the change of the environment. The data loss with ALoMa, in this case, is similar to that with CTRL. For more insight, Fig. 10 also shows the headroom factor adjustment made by ALoMa in response to the change in the system environment.

7.1.4 ALoMa versus CTRL and Aurora with a complex query network (Fig. 11; Table 3)

The CTRL paper [44] shows that CTRL outperforms Aurora in the experiments with flat query networks, as does ALoMa, since ALoMa performs equivalently to CTRL as shown above. Because [44] does not show CTRL's performance compared to Aurora for complex query networks, we include Aurora in this evaluation to confirm that our scheme also outperforms Aurora in this case.

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 ALoMa. If the headroom factor is too small, the response time is kept well below the target at all times by dropping much more data unnecessarily. When the headroom factor equals 0.92 (Fig. 11), the average delay violation of Aurora is roughly the same as ALoMa, but Aurora drops 5 % more data (Table 3). Increasing the headroom factor to 0.93 makes the delay violation significantly higher (due to the higher peak in the response time), while the data loss is still higher than ALoMa. This is consistent with the properties of Aurora analyzed in [44]: The Aurora method is not aware of the delay target and cannot recover from its previous wrong decision since it does not look at its outcomes.

The method given by CTRL to tune the headroom factor cannot be applied any more with the complex query network: No matter how we change the value of the headroom factor, the delay estimated by CTRL does not match with the real output delay. Because the query network contains shared operators, 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. 11, 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 420 s), the system with CTRL has only been able to process input tuples coming in the first 66 s (out of 400 s). We tried some other values of the headroom factor from 0.8 to 0.99 as well, but they do not make any difference to the performance of CTRL in this case.

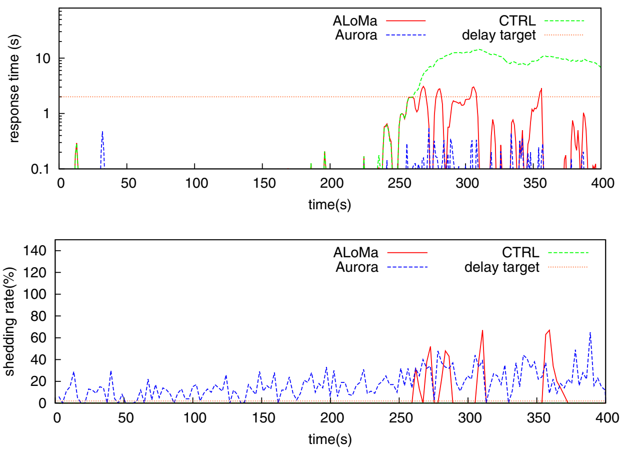

Fig. 12 Performance of ALoMa, CTRL and Aurora with QN-long and SA_r. *Top plot* is the response time and *bottom plot* is the shedding rate

7.1.5 ALoMa versus CRTL and Aurora with a query network containing long queries (Fig. 12)

In this experiment, we use QN-long to confirm that ALoMa is applicable for query network containing long queries with all basic types of operators and with multiple levels of operator sharing. We use the real input rate pattern SA_r for all of the stream sources. Note that because there are five range window joins in each query, the effective overshoots in the input load is actually much higher than the overshoots in the individual input load shown in Fig. 8. The reason is that the increase in input rate increases the number of tuples in each window, causing the selectivity of the range window join to increase. Figure 12 shows the response time under the 3 schemes.

In general, ALoMa can control the response time well at the delay target. We observe four points when the delay target is violated, of which the highest violation is 1.08 s. These violations correspond to the very high overshoots in the input load. However, ALoMa was able to cope with them by increasing the shedding rate from 0 to almost 70 %.

CTRL, as expected, cannot control the response time because it cannot correctly estimate the length of the virtual queue of a complex query network. We show Aurora's performance just for completeness, as without being aware of the delay target its performance for a certain workload is very unpredictable. In this experiment, with headroom factor set to 0.92, it happens that it drops more than necessary, as shown in the bottom plot of Fig. 12.

7.1.6 Worst-case scenarios (Figs. 13 and 14)

In this set of experiments, we illustrate the worst-case scenarios explained in Sect. 4.2.4. We use query network QN-flat, so that CTRL is applicable.

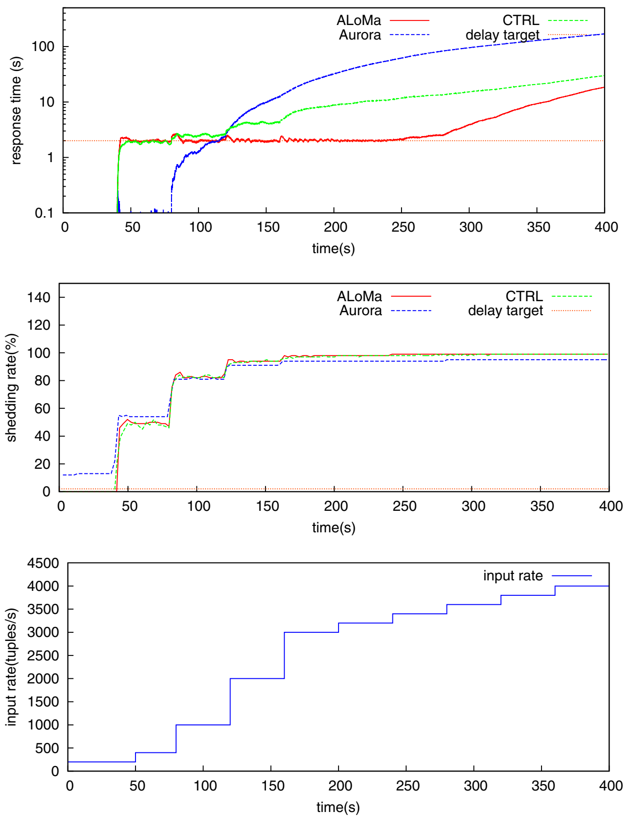

Fig. 13 Performance of ALoMa, CTRL and Aurora with workload increasing to worst-case situation. *Top plot* is the response time, *middle plot* is the shedding rate and *bottom plot* is the input rate

In the first setup, we use the input SA_{step} to push the input workload from no overload (200 tuples/s) to extreme overload. As expected, as the input load reaches a certain point, none of the schemes can any longer control the response time to the delay target even though they drop almost 100 % (we set maximum shedding rate for all the schemes at 99 %, so that we can retain some output tuples). This is because the system still spends some CPU cycles on a dropped tuple to read it from the stream source and to decide whether to drop it. When the input load is too big, this cost alone is enough to overload the system. Figure 13 (top plot) shows the response time of the system under each scheme, corresponding to the input rate plotted in the bottom plot. The middle plot shows the shedding rate under each scheme.

Interestingly, the three schemes have different points at which they can no longer control the response time, with ALoMa's point being the farthest to the right. We observed that, when the input rate is very high (beyond 1000 tuples/s in this experiment), the headroom factor decreases when the rate increases. ALoMa's adaptivity allows it to cope with this change, whereas CTRL and Aurora failed to cope with it. Thus, a value of the headroom factor that works well for CRTL and Aurora at the beginning becomes incorrect, causing the two schemes to lose control of response time

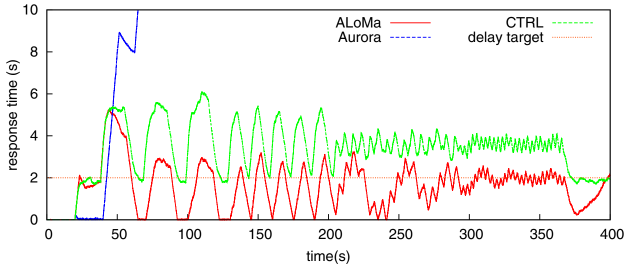

Fig. 14 Response time under ALoMa, CTRL and Aurora with background job coming and leaving at different frequencies

early. Our explanation for this decrease in the headroom factor is that when the input rate significantly increases, batch processing kicks in lowering the cost of processing each tuple. Therefore, some fixed costs (e.g., scheduling, statistics collection) become *relatively* bigger compared to the processing cost per tuple. However, we think this phenomenon depends greatly on the detailed implementation of each system, so it can be different across different DSMSs.

In the second setup, we use the constant input SA_c as in the experiment in Sect. 7.1.3. After the experiment has run for the first 10 s, we kick off a background job which stays for 10 s, then leaves for 10 s and comes back for another 10 s. The pattern is repeated for about 60 s, then switches to a pattern of 5-second stay and 5-second leave for another 60 s, then 60 s of 2-second stay and 2-second leave and finally 60 s of 1-second stay and 1-second leave. This creates situations where the change in the headroom factor happens suddenly yet does not stay long enough for ALoMa to adapt. Figure 14 shows the response time under all three schemes.

We can see that there are points at which the response time under ALoMa drops close to 0. Ideally, with this constant rate the response time should be kept at the delay target (i.e., the maximum allowed), so as to minimize the data lost. However, the process of adjusting the headroom factor takes time. When the background job leaves, ALoMa needs a few seconds to adjust the headroom factor back to the original, bigger value, so during that transition time it drops more data than necessary. Interestingly, when the frequency of coming and leaving of the background job becomes very high (i.e., every one second in this experiment), ALoMa's performance becomes better, because by the time the job leaves, ALoMa is not too far from decreasing the headroom factor so it just needs a short time to move it back up.

CTRL does not recognize the change in the headroom factor so the response time under it fluctuates above the delay target. Aurora loses its control of the response time beginning with the very first appearance of the background job, as it does not consider any kind of feedback from the outcome of its decision and hence has no way to recover.

7.2 DILoS evaluation

7.2.1 Experimental settings

Query network: We use two query networks *QN-A* and *QN-B*:

- **QN-A:** A query network that consists of three classes of queries:
  - Class 1: Priority 6 (highest), with delay target 300 ms.
  - Class 2: Priority 3 (second highest), with delay target 400 ms.
  - Class 3: Priority 1 (lowest), with delay target 500 ms.

  All three classes have the same set of 11 queries, consisting of five aggregates, two window joins, and four selects. These types of operators would appear in a typical monitoring continuous query, for example those in the Linear Road Benchmark [11].
- **QN-B:** The same as QN-A except that we triple the size of the first class so that, when using the real input trace for the first class, the resulting workload is heavy enough to create some load impact in the system.

Input data: We use two streams of synthetic input patterns, denoted SD_c, SD_p, and one using real input traces, SD_r, as described below:

- $\mathbf{SD_c}$: All the input streams coming to the three classes have a constant input rate of 950 tuples/s, which, together with the query network QN-A, creates a total load that is slightly higher than the total system capacity. The simple pattern of this input allows us to easily analyze the behavior of each scheme.
- $\mathbf{SD_p}$: The input rate (per control period) of classes 2 and 3 follows a Pareto distribution in the range of [800–1300] and [300–800], respectively, with skewness equal to 1. 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 the range for its input rate distribution (also Pareto) after every 50-second period (Fig. 15 sketches the changes of the range) in order to vary the amount of

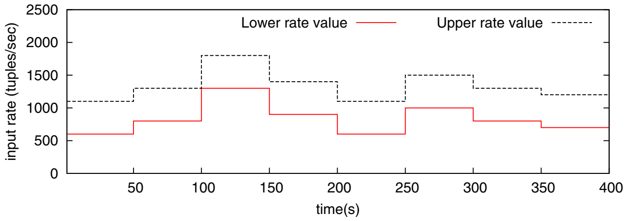

Fig. 15 Input rate ranges for class 1—input setup SD_p

Table 4 DILoS' advantages shown through average response time and data loss

	Response time (ms)			Data loss (%)		
	Class 1	Class 2	Class 3	Class 1	Class 2	Class 3
No load manager	5.25	7.22	117,132.74	0	0	0
Common load manager	4.01	4.74	513.71	42.19	42.15	42.24
Separate load manager	4.91	7.21	492.16	0	0	85.37
DILoS (Full synergy)	8.90	34.18	487.04	0	0	24.43
DILoS with inter-class sharing	9.05	36.54	482.53	0	0	14.70

excess capacity it can share with the other classes. The query segment that can be shared with class 3, however, has the same input rate as class 3, so that we can keep the entire workload of class 3 to be at the same level during the experiment).

- $\mathbf{SD_r}$: The same input rate patterns as in SD_p are used for class 2 and 3, while the input rate of class 1 is the real trace used in SA_r (Fig. 8).

Parameters: For all experiments, we set 150 ms to be the *load management cycle*. In [44], the authors report the appropriate load management cycle to be around one-fourth to half of the delay target, and we had a similar experience. We set the *capacity redistribution cycle* (i.e., the cycle at which the scheduler considers redistributing the system capacity for each class) to be 10 load management cycles (i.e., 1.5 s). We report the sensitivity analysis on the length of this capacity redistribution cycle in Sect. 7.2.4.

7.2.2 Confirming the advantages of DILoS

In these experiments, we run the query network QN-A with the constant input rate SD_c in five cases: (1) when there is no load manger, (2) when there is one common load manager for the whole system, (3) when one ALoMa load manager instance is created for each CQ class, (4) when the scheduler uses the feedback from the load manager to adjust its scheduling decisions, in the complete DILoS framework and (5) when operator sharing is enabled in the DILoS framework, allowing class 1 and class 3 to share a query segment. Table 4 summarizes the response time and data loss of the three class in each of these cases.

When there is no load manager, class 3 is overloaded, and, as a result, its response time (117,132.74 ms) exceeds its delay target (500 ms) by three orders of magnitude. With one common load shedder, which is the case for all the state-of-the-art systems, the load shedder is oblivious to the priority enforcement of the scheduler. Thus, although the load manager successfully controls the response time of class 3 to satisfy the worst-case QoS, 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.

When one load manager instance is created for each CQ class, the load manager can follow exactly the priority enforcement of the scheduler. As a result, only class 3, which is the one that is overloaded, experiences load shedding of 85.37 %. Not only that, the observed data loss for class 3 is actually less than the total data loss for the three classes in the case of a common load shedder.

Under a complete DILoS framework when the scheduler use the feedback from the load manager instances, its effectiveness is clear: The data loss is reduced by more than 70 % compared to the case with no synergy (24.43 vs 85.37 % data loss for class 3 as in Table 4)[4]. Given 13 stream sources used by class 3, each with the input rate of 950 tuples/s, this decrease in data loss means approximately 7526 more tuples are processed *per second*. At the same time, the response times of the three classes are well controlled, and the overall goal is preserved: DILoS is still consistent in providing better QoS and QoD for the class of higher priority. When inter-class sharing is supported in DILoS more data is saved (14.70 vs 24.43 %)[5], while the performance of the higher-priority class 1 is not affected by the lower-priority class 3. Figure 18 shows the response time of the three classes under a complete DILoS framework with inter-class sharing.

Understand the benefit of the synergy: One might think that the advantage of DILoS' full synergy in reducing data loss is only due to the fact that it repairs the over-provisioning of system capacity for some classes. This benefit is true for the global scheduler that strictly fixes the CPU time allocation. However, DILoS actually achieves more than merely repairing the over-provisioning: *it exploits batch processing to further increase system capacity utilization.*

Figure 16 plots the headroom factor (i.e., the capacity portion) estimated by each load manager of each class when an ALoMa instance is created to manage the load in each

[4] We have observed in some experiments (not shown in this paper), that the reduction in data loss under DILoS can reach up to 100 %, i.e., completely eliminating the need for shedding.

[5] Since the three classes have the same amount of data, total data loss of the three classes is calculated by $\frac{\sum_{1 \le i \le 3}[\text{dataloss}_i])}{3}$

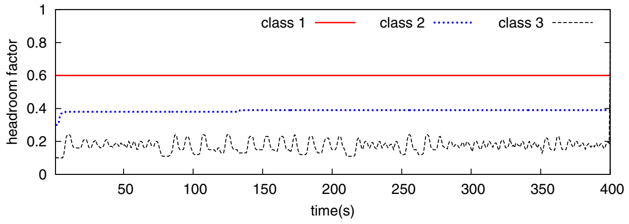

Fig. 16 Headroom factor estimated, with SD_c, QN-A, and one ALoMa instance per class

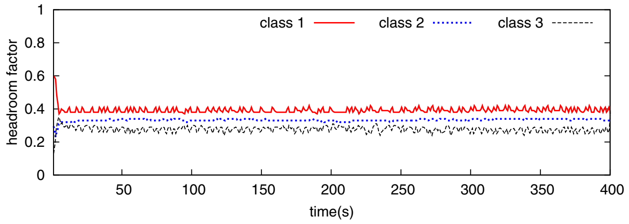

Fig. 17 Headroom factor estimated, with SD_c, QN-A, and DILOS' full synergy

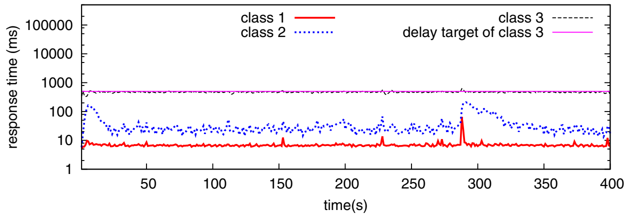

Fig. 18 Response times with SD_c, QN-A, DILoS, and inter-class sharing

class, but the scheduler does not use the feedback from these ALoMa instances to adjust its decision. At the beginning of the experiment, we initialize the headroom factors for classes 1, 2, and 3 by their expected values, i.e., 0.6, 0.3, and 0.1, respectively. However, we observed that the headroom factor of classes 2 and 3, estimated by the load manager at runtime, was above their expected values of 0.3 and 0.1, respectively. This phenomenon 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 without waiting for the former class to use up its quota (waiting for new tuples). Thus, when a class is very lightly loaded (class 1 in this case), part of its assigned capacity is automatically given to the other classes[6]. Thus, CQC by itself already allows implicit capacity sharing, and the system capacity seems to have been used fully.

However, Fig. 17 shows that class 3 actually receives even more system capacity when the full synergy is used (i.e., the scheduler uses feedback from the ALoMa instances to adjust its decisions, which explains why it does not need to drop as much data. Where does the "extra" capacity come from? The answer is from batch processing. We have known that the higher the number of tuples an operator can process in a batch, the lower the processing cost per tuple. 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 allowed batching to reduce the processing cost. By explicitly reducing the capacity portion of the lightly loaded class, DILoS effectively increases the number of tuples its operators process in batch and reduces the processing cost per tuple. Therefore, the class can fit in the smaller capacity without being overloaded, sharing more capacity with the other classes.

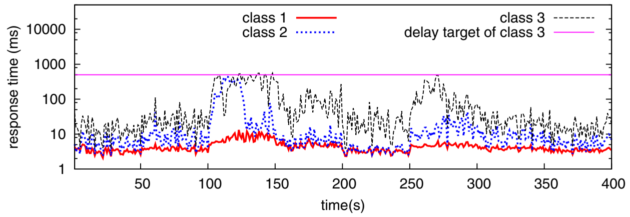

Fig. 19 Response times with SD_p, QN-A, and DILoS (with sharing)

We can observe that the response time of classes 1 and 2 increases. This is a side effect of batch processing: These classes are forced to process more tuples in each batch, so each tuple has to wait for a longer time. We believe this side effect is not an issue given that the response times of the three classes still meet their QoS requirement.

7.2.3 Asserting DILoS robustness

Because there is no previous work with an equivalent model to compare our work with, we evaluated DILoS with more challenging input rate patterns, both real and synthetic, in order to assert its robustness. More specifically, we tested how fast our scheme can react to sudden changes of input rate and whether the benefit of the synergy still exists in such cases.

QN-A and SD$_p$ (Figs. 19, 20; Tables 5, 6): This set of experiments simulates situations where the load level of class 1 (the highest priority) changes dramatically after a certain period, aiming to test whether DILoS reacts fast enough to sudden changes in the load of the class that is sharing its redundant capacity with others. Also, at a given load level, the input rate (of all the three classes) is still not constant but fluctuates following a Pareto distribution with sudden high peaks. We show the response times of the three classes under DILoS with inter-class sharing in Fig. 19. In Fig. 20, we show the changes in the capacity portion of each class, which is reflected through the headroom factor estimated by each load manager instance, and the corresponding changes in the shedding rates.

[6] Note that in this case, the estimated headroom factor of class 1 is not adjusted and still remains at the initial value because the load manager does not have the necessary signals to decrease it.

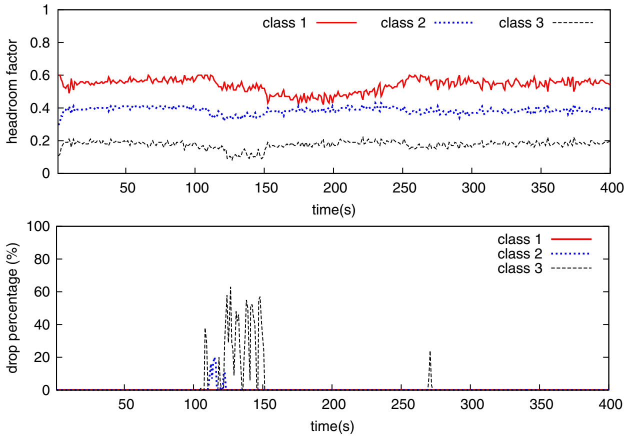

Fig. 20 Estimated headroom factors (*top*) and shedding rates (*bottom*), with SD_p, QN-A, and DILoS (with sharing)

Table 5 Average response time (ms) with SD_p and QN-A

	Class 1	Class 2	Class 3
No synergy (& no sharing)	5.30	15.13	176.37
DILoS without sharing	6.47	43.98	84.21
DILoS with sharing	5.94	38.04	72.73

Table 6 Average data loss (%) with SD_p and QN-A

	Class 1	Class 2	Class 3
No synergy (& no sharing)	0	0	8.53
DILoS without sharing	0	0.23	2.30
DILoS with sharing	0	0.16	1.42

We observe that when the load of class 1 is low, DILoS enables the global scheduler to distribute the excess capacity from class 1 to the other classes, allowing them to shed less. However, as soon as the load of class 1 increases (e.g., at the 100th second), DILoS returns to class 1 all or part of its original capacity, so that its performance, as specified by its class priority, is preserved.

In Tables 5 and 6, we compare DILoS' average response time and data loss with those two alternatives (i.e., DILoS without sharing and the scheme without the synergy). Clearly, the synergy between the scheduler and load shedder exploits better the system capacity and saves considerably more data (2.3 vs 8.53 % of data loss of class 3). As expected, the response times of class 1 and class 2 increase under the synergy due to the side effect of batch processing, but they are all well below their delay target. The higher-priority class still receives the better QoS, which complies to the implemented policy. The average response time of class 3 is smaller under the synergy, because there are more periods during which the class is not overloaded and its response time is much smaller than its delay target.

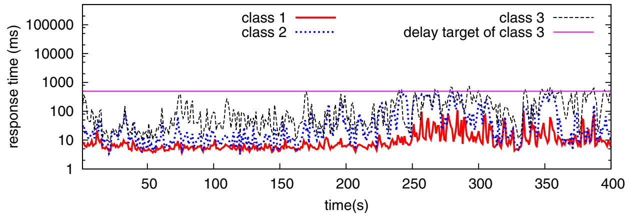

Fig. 21 Response times with SD_r, QN-B, and DILoS (with sharing)

In this experiment, class 2 incurs a data loss of 0.2 % under DILoS, although its expected data loss should be 0 %. This reveals an inherent aspect of any statistics-based module, including those used by DILoS to enforce explicit capacity redistribution: They might need some cycles of adjustment before they can pick up the right decision. This occurs when the input rate fluctuates considerably after each load management cycle (recall that in S_p although the upper and lower bounds of the input rate are kept constant for class 2, the input rate of each load management cycle follows a Pareto distribution within the two bounds). In such a case, the lag of the statistics-based decision causes small additional shedding in some time windows. The additional data loss, however, is very small and often not observed, because it is obscured by the normal fluctuations in the system.

The results also show the benefit of sharing in saving data and confirm that with appropriate load management the sharing does not affect the QoS and QoD of the higher-priority class.

QN-B and SD$_\mathbf{r}$ (Figs. 21, 22; Tables 7, 8): In this set of experiments, we replace the synthetic input rate pattern by SD_r with the real trace for class 1 (Fig. 8). This real input rate pattern has two challenging periods when the rate keeps increasing with sudden, very high peaks.

We show the response time of the three classes under DILoS with inter-class sharing in Fig. 21. In order to understand better the behavior of the load manager under each of the three classes, we also plot the headroom factors and shedding percentages in Fig. 22 (the top and the middle plot, respectively). For convenience, at the bottom of this figure we repeat the real input rate pattern used for class 1. As expected, when the input rate of class 1 increases (e.g., from the 250th to the 300th second), the excess capacity the class can give to the other classes decreases. This has the clearest effect on the lowest priority class 3, causing this class to drop a lot more data during that period.

In the first 250 s, none of the classes are overloaded, and the recognized headroom factors might be higher than the true values because of the implicit redistribution of the system capacity when some of the classes have very light load, as mentioned in Sect. 7.2.2. The load manager recognizes the correct headroom factor when the load of some of the classes reaches their capacities and the explicit redistribu-

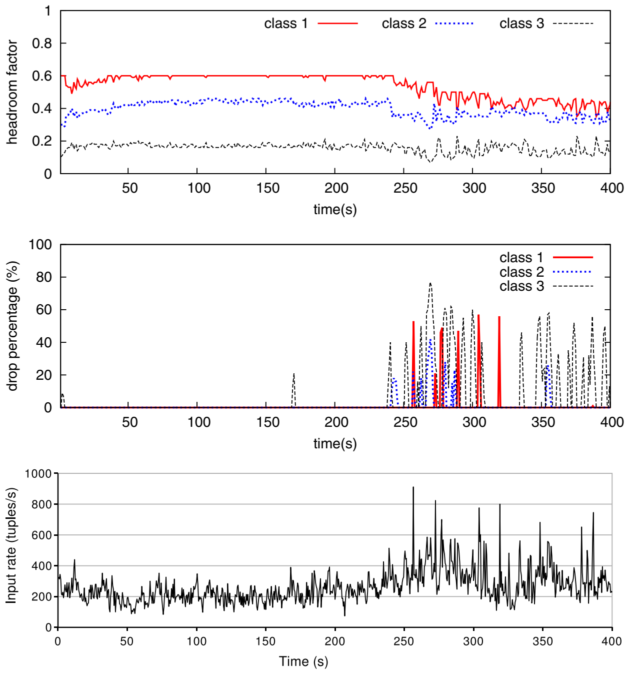

Fig. 22 Estimated headroom factors (*top*) and shedding rates (*middle*) in response to the input rate of class 1 (*bottom*), with SD_r, QN-A, and DILoS (with sharing)

Table 7 Average response time (ms) with SD_r and QN-B

	Class 1	Class 2	Class 3
No synergy (& no sharing)	22.31	68.23	300.91
DILoS without sharing	25.69	76.86	122.66
DILoS with sharing	25.03	70.29	127.28

Table 8 Average data loss (%) with SD_r and QN-B

	Class 1	Class 2	Class 3
No synergy (& no sharing)	0.01	0.79	21.67
DILoS without sharing	0.46	0.68	8.70
DILoS with sharing	0.44	0.82	6.54

tion happens, which is the case during the high-load period (after the 250th second).

Tables 7 and 8 compare the average response time and data loss for all cases. In this experiment, while synergy still brings significant benefit in terms of exploiting system capacity (much more data are saved: 3.28 vs 7.49 % of total data loss), it also incurs a trade-off: the data loss of class 1 under the two cases with synergy is higher compared to the case without synergy. As shown in Fig. 22, the shedding of class 1 corresponds to the sudden high peaks of input rate during the high-load period. As in the previous experiment, this is due to inherent lag of the statistics-based decision. Specifically, since class 1 passed its excess capacity to the others, its remaining capacity became rather tight, hence a sudden, huge increase in the input rate caused overloading, and subsequently, load shedding, before the scheduler could recognize and correct the situation.

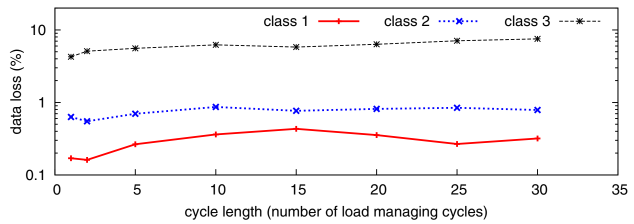

Fig. 23 Data loss at different lengths of the capacity redistribution cycles

We believe this trade-off is acceptable given that the increase in the shedding rate of class 1 (0.45 %) is much smaller compared to the total data saved (12.97 % for class 3 and 4.21 % overall). This happens only in very extreme situations and is eventually corrected. In practice, if a class is highly critical and such a trade-off cannot be tolerated, one can develop a capacity redistribution policy that includes a limit on the shared usage of the class' capacity (while still allowing the class to use redundant capacity from other classes and allowing the normal capacity redistribution among the other classes).

These results also confirm that the proposed approach for inter-class sharing saves more data for class 3 while leaving class 1, i.e., the higher-priority class sharing a query segment with class 3, unaffected.

7.2.4 Sensitivity analysis

In this section, we report the sensitivity of the system performance to the length of this capacity redistribution cycle (CRC for short).

We show in Fig. 23 the system performance in terms of average data loss per class at different values of CRC, under the S_r input rate pattern which we expect the CRC to have the biggest impact. Note that the y-axis is in logarithmic scale. We observe that the data loss of class 1 (and the other two) is smallest when CRC is equal to 1 or 2 load management cycles (i.e., 150–300ms). This is because the system can react faster with sudden changes of the input rates and in the system environment. However, the difference across all values is rather small, suggesting that the long-term performance of the system is somewhat stable to a wide range of CRC values.

As mentioned in Sect. 7.2.1, for all the above experiments we let the scheduler consider redistributing the system capacity after every 10 load management cycles (i.e., 1.5 s). To better evaluate the framework, we avoid using the best-picked value (2 load management cycles in this case) and instead use one that gives average performance.

8 Extensibility of DILoS

As a framework with two-level integrated scheduling and load managing, DILoS enables easy incorporation of different scheduling and load shedding schemes at both the class and operator level.

At the class level, different capacity allocation and redistribution policies can be adopted. For example,

- *Absolute priority for higher-priority class.* A higher-ranked class can use all of the available system capacity if needed before a lower-ranked class is considered. A hybrid policy between absolute and relative priority is also possible: The first class might use up the whole system capacity if needed, but any remaining capacity is distributed to the other classes proportionally by their priorities.
- *Relative priority with workload consideration.* The current policy in CQC guarantees better QoD for a class of higher priority compared to a lower-priority one only if the higher class has the same or less load than the lower one. With the support of DILoS, a stricter guarantee is possible: The higher class will receive either maximum QoD (i.e., no data loss) or better QoD than the lower class, regardless of the relative workloads of the two. Since the global scheduler receives feedback from the load manager about the capacity utilization of each class, it can recognize any violation of such policy and fix it by moving the necessary capacity from the lower class to the higher one.

At the operator level (i.e., in within a class), different load shedders and operator schedulers can be used. Any operator or query-based scheduling policy can be easily plugged in as a local scheduler inside a class without affecting the benefit brought by DILoS. We have verified this through an experiment with round robin as the local scheduler (result not shown due to space limitation).

An important part of DILoS is the capability of the load manager to automatically recognize exactly the system capacity each class is receiving, which ALoMa satisfies. However, ALoMa only focuses on the question of detecting when the system is overloaded and how much the excess load is. Regarding the other common questions related to load shedding, i.e., what to shed and where to shed, ALoMa uses a general, domain-independent method of applying random dropping evenly from the input of all queries in the class. Other works on these questions, such as those considering semantic dropping (e.g., [19,22,42]) and determining where in the query network to shed data to minimize semantic loss (e.g., [14,42]), can be trivially plugged in to replace the basic method ALoMa is using. Note that all these schemes need to know when and how much load to shed, which is answered by ALoMa. For example, assuming that semantic shedding is desired for a class of 2 CQs, Q_1 and Q_2, each of which has input tuples containing integer keys in [1–10]. For Q_1, output with keys [9–10] is more important than those in [1–8], while for Q_2 those in [1–2] is more important than the others. When ALoMa determines that, say, 20 % of the current load needs to be shed, the semantic shedder will take that 20 % as input for its algorithm. Correspondingly, the semantic shedder decides that for Q_1, it drops $\frac{10}{8} \times 20\,\%$ tuples with keys in [1-8], while for Q_2 it drops $\frac{10}{8} \times 20\,\%$ tuples with keys in [3–10], keeping the whole important range (assuming a uniform distribution of the keys). Note that this assumes queries which have different semantic on incoming tuples, as in the case of Q_1 and Q_2 in this example, do not share operators with each other.

9 Related work

Overloading is a common problem in many systems including DSMS, real-time database, networking, and web services. Common approaches for this problem can be divided into three categories: *resource allocation*, *admission control*, and *load shedding*. Although for each approach the basic ideas are shared across systems, every system has its own model and constraints, which determine the details of the approach. For example, measuring bandwidth load in networking is very different from measuring workload in data stream systems; in data stream management systems, the objects of load management are both incoming data and query operators, as opposed to network packets. In the scope of this paper, our discussion focuses on workload management in DSMS, which are the most closely related to our work.

Resource allocation in DSMS, in general, decides how the system resources (CPU cycles, memory, etc.) should be assigned to each query/operator in each period of time. Included in this category are workload distribution/ balancing and scheduling. *Workload distribution and balancing* either distributes and rebalances the workload on the fly over the available processing nodes (e.g., [28,31,46]), or finds a query network deployment that is resilient to workload fluctuation at run time (e.g., [33]). Castro Fernandez et al. [16] also integrates fault tolerance and scaling out of stream operators. *Scheduling* of CQs in a DSMS focuses more on time sharing the system resources among the query operators, aiming at optimizing certain performance goals such as minimizing latency ([15,40]) or minimizing memory requirements ([12]). Related to our work on multi-class CQ scheduling are the works in [7,15,17] which consider latency-based QoS functions for each query, and in [32,45,47] which schedule real-time CQs where each CQ has a deadline. These schemes try to optimize the overall benefit of the system rather than explicitly guarantee the benefit of each class according to its priority. In our previous work [10], we proposed another scheduling scheme, called CQC, in which each query belongs to a class of a specific relative priority, and the benefit of each class according to its priority. CQC was later extended in [9]. None of these works on priority-based schedulers considers the integration with a load shedder to handle overload situations.

Load shedding has been proposed in many DSMS architectures as a method to handle overloading [7,13,30,39]. [42] articulates four basic questions for a load shedder: *when, how much, where and what* to shed.

The works in [14,41,42] mainly focus on the question of *where to shed*, i.e., given an amount of excess load, which positions in the query network should drop how much of the load, such that the loss of quality of data is minimized. [34] basically considers the same problem, but the model is for aggregates and mining queries and aims at deciding the shedding ratio for each of the keys of the queries.

The question of *what* to shed has been addressed in many of previous works in load shedding. Instead of randomly dropping tuples, semantic models are used in [19,20,22,42] to increase the usefulness of the query results after shedding. Also related to this question, in [24,25,27,36,39] the authors propose methods to shed load other than simply discarding tuples from a query network. In [39], dropped tuples are routed to a lightweight *shadow plan* that produces approximated results. The work in [36] is customized for spatiotemporal data streams, in which a dropped tuple is approximated by the mean value of the cluster it belongs to. In [24,27], the system load is shed by selecting only subsets of the windows to perform the joins. In [25], the DSMS delegates the load shedding task to the source filters, which apply varying amounts of shedding to different regions of the data space. In [43], Tatbul and Zdonik consider a whole window, not a single tuple, as the shedding unit.

A few works have addressed the question of when to shed load and how much load to shed ([14,30,31,39,42, 44]). However, as analyzed in Sect. 4.1, they have shortcomings that limits their applicability in practice, among which is the requirement of a manually tuned estimation of the system capacity. In our previous work [38], we provided another load shedding solution that can avoid this manual tuning, but this solution is not complete since its approach depends on the fairness of the operator scheduler. Therefore, in this paper we proposed another approach, ALoMa, which follows a different design to achieve the same goal without making any assumption on the scheduler's fairness.

Admission control can be viewed as a more drastic way of load shedding: The system decides to drop some of the queries rather than the data, as in load shedding. Typically, admission control schemes select a subset of CQs to run every period of time or epoch based on some optimization objective. For example, in [46], the goal is to maximize the utilization of the system and the overall importance of the CQs, whereas in [8], the goal is profit maximization, strategy-proofness and sybil immunity even at the expense of system utilization.

Combinations of the above approaches have been proposed in different settings. For example, [26] combines admission control and load shedding (i.e., update shedding and query shedding) in a mobile CQ setting.

In [23], the authors model both load shedding and resource allocation as a dual optimization problem, formally solve the problem, and illustrate the solution using a simulation. The paper does not consider query priorities in both resource allocation and load shedding and assumes a known system capacity (i.e., resource budget).

Few of the previous works on load shedding have considered the priority of the CQs. CQ priorities have been implicitly considered through loss-tolerance QoS (i.e., QoD) graphs [42] or maximal tolerable relative error [17,30]. However, the emphasis of these approaches is on load shedding: The load shedder is unaware of the priorities the scheduler is enforcing, and there is no unified priority model which a load manager and a scheduler can together support consistently. As a result, none of these load shedders can provide feedback to the scheduler to improve scheduling decisions.

In [46], the authors consider the problem of resource allocation and job admission for DSMS deployed on multiple nodes, taking into account the rank of the jobs. This work also aims at maximizing resource utilization and giving higher admission priority to jobs with higher rank. However, the paper considers job admission rather than load shedding and does not provide any guarantee on QoS and QoD for different rank as our scheme does.

As opposed to the above works, DILoS, which was initially proposed in a 6-page workshop paper [37] and expanded and enhanced in this paper, consistently combines priority-based scheduling and load shedding to provide a guarantee on QoS and class-consistent QoD.

10 Conclusions

In this paper, we presented DILoS, a novel framework to support priority-based CQ processing, and ALoMa, an adaptive load manager utilized in DILoS implementation.

The success of DILoS, which facilitates the synergy between the scheduler and load manager in our new framework, strengthens our hypothesis that further optimization of QoS and QoD for a DSMS can be achieved by exploiting the synergy of the scheduler and load shedder. We have shown, through analysis and experimental evaluation on AQSIOS, a real DSMS prototype, that the synergy developed in DILoS brings three basic benefits: (1) the integration enables the load manager to honor query class' priorities in a consistent way with a two-level, class-based scheduler (e.g., CQC); (2) by adjusting its decision using feedback from the load manager, the scheduler can now better exploit the system capacity and reduce load shedding; and (3) the proper employment of the load manager helps to release the congestion problem in the class-based scheduler to allow the sharing of processing among queries of different classes, thereby enhancing even more the ability of the system to meet the QoD and QoS specifications.

ALoMa is a general and practical DSMS load manager that effectively determines *when and how much to shed* and it can be used in conjunction with any statistical or semantic scheme that determines *where and what to shed*. Our experimental evaluation of ALoMa verified its clear superiority over the state-of-the-art load managers in three key dimensions: (1) it automatically tunes the headroom factor, (2) it honors the delay target, and (3) it is applicable to complex query networks with shared operators.

Acknowledgments Our thanks to the anonymous reviewers for their insightful comments and Mark Silvis and Eric Gratta for their help with copyediting. This work was supported in part by NSF awards IIS-0534531, IIS-0746696, OIA-1028162, an Andrew Mellon Predoctoral Fellowship and EMC/ Greenplum.

References

1. Esper. http://esper.codehaus.org
2. HP Vertica Best Practices: Resource Management. http://www.vertica.com/2015/02/19/hp-vertica-best-practices-resource-management
3. Microsoft StreamInSight. https://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx
4. Pacific tsunami warning center. http://ptwc.weather.gov/
5. System S - Stream Computing at IBM Research. http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534
6. Tropical Atmosphere Ocean Project. http://www.pmel.noaa.gov/tao/
7. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. In: VLDBJ '03
8. Al Moakar, L., Chrysanthis, P. K., Chung, C., Guirguis, S., Labrinidis, A., Neophytou, P., Pruhs, K.: Admission control mechanisms for continuous queries in the cloud. In: ICDE'10
9. Al Moakar, L., Labrinidis, A., Chrysanthis, P. K.: Adaptive class-based scheduling of continuous queries. In: SMDB '12
10. Al Moakar, L., Pham, T. N., Neophytou, P., Chrysanthis, P. K., Labrinidis, A., Sharaf, M.: Class-based continuous query scheduling for data streams. In: DMSN '09
11. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: VLDB' 04
12. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. In: VLDBJ '04
13. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02
14. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: ICDE '04
15. Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: VLDB' 03
16. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: SIGMOD'13
17. Chakravarthy, S., Jiang, Q.: Stream Data Processing: A Quality of Service Perspective Modeling. Load Shedding, and Complex Event Processing. Springer, Scheduling (2009)
18. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., Shah, M. A.: TelegraphCQ: continuous dataflow processing. In: SIGMOD '03
19. Chang, J.H., Kum, H.-C.M.: Frequency-based load shedding over a data stream of tuples. Inf. Sci. **179**(21), 3733–3744 (2009)
20. Chi, Y., Wang, H., Yu, P. S.: Loadstar: load shedding in data stream mining. In: VLDB '05
21. Chrysanthis, P. K.: AQSIOS—Next Generation Data Stream Management System. CONET Newsletter, June 2010
22. Dash, R., Fegaras, L.: Synopsis based load shedding in XML streams. In: EDBT/ICDT '09 Workshops
23. Feng, H., Liu, Z., Xia, C. H., Zhang, L.: Load shedding and distributed resource control of stream processing networks. In: Performance Evaluation (2007)
24. Gedik, B., Wu, K.-L., Yu, P., Liu, L.: GrubJoin: An Adaptive, Multi-Way. Windowed Stream Join with Time Correlation-Aware CPU Load Shedding, TKDE (2007)
25. Gedik, B., Wu, K.-L., Yu, P. S.: Efficient construction of compact shedding filters for data stream processing. In: ICDE '08
26. Gedik, B., Wu, K.-L., Yu, P. S., Liu, L.: Mobiqual: Qos-aware load shedding in mobile CQ systems. In: ICDE '08
27. Gedik, B., Wu, K.-L., Yu, P.S., Liu, L.: CPU load shedding for binary stream joins. Knowl. Inf. Syst. **13**(3), 271–303 (2007)
28. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. In: IEEE TPDS, 2012
29. Jagadish, H.V., Gehrke, J., Labrinidis, A., Papakonstantinou, Y., Patel, J.M., Ramakrishnan, R., Shahabi, C.: Big data and its technical challenges. CACM **57**(7), 86–94 (2014)
30. Kendai, B., Chakravarthy, S.: Load shedding in MavStream: analysis, implementation, and evaluation. In: BNCOD '08
31. Kleiminger, W., Kalyvianaki, E., Pietzuch, P.: Balancing load in stream processing with the cloud. In: ICDEW' 11
32. Kulkarni, D., Ravishankar, C. V., Cherniack, M.: Real-time load-adaptive processing of continuous queries over data streams. In: DEBS' 08
33. Lei, C., Rundensteiner, E. A.: Robust distributed query processing for streaming data. In: ACM TODS, 2014
34. Mozafari, B., Zaniolo, C.: Optimal load shedding with aggregates and mining queries. In: ICDE '10
35. Narayanan, S., Waas, F.: Dynamic prioritization of database queries. In: ICDE '11
36. Nehme, R. V., Rundensteiner, E. A.: Clustersheddy: load shedding using moving clusters over spatio-temporal data streams. In: DASFAA'07

37. Pham, T. N., Al Moakar, L., Chrysanthis, P. K., Labrinidis, A.: DILoS: a dynamic integrated load manager and scheduler for continuous queries. In: SMDB '11
38. Pham, T. N., Chrysanthis, P. K., Labrinidis, A.: Self-managing load shedding for data stream management systems. In: SMDB '13
39. Reiss, F., Hellerstein, J. M.: Data triage: an adaptive architecture for load shedding in telegraphCQ. In: ICDE '05
40. Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A., Pruhs, K.: Algorithms and metrics for processing multiple heterogeneous continuous queries. In: ACM TODS, 2008
41. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying FIT: efficient load shedding techniques for distributed stream processing. In: VLDB '07
42. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: VLDB '03
43. Tatbul, N., Zdonik, S.: Window-aware load shedding for aggregation queries over data streams. In: VLDB '06
44. Tu, Y.-C., Liu, S., Prabhakar, S., Yao, B.: Load shedding in stream databases: a control-based approach. In: VLDB '06
45. Wei, Y., Son, S. H., Stankovic, J. A.: RTSTREAM: real-time query processing for data streams. In: ISORC' 06
46. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer, L.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Middleware' 08
47. Wu, S., Lv, Y., Yu, G., Gu, Y., Li, X.: A QoS-guaranteeing scheduling algorithm for continuous queries over streams. In: APWeb/WAIM' 07