# EXPLOITING THE SYNERGY BETWEEN SCHEDULING AND LOAD SHEDDING TO FACILITATE DIFFERENTIATED LEVELS OF SERVICE FOR CONTINUOUS QUERIES

by

**Thao Nguyen Pham**

B.Sc. in Information Technology, HCM University of Science, 2004

M.Sc. in Computer Science, University of Pittsburgh, 2014

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

**Doctor of Philosophy**

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Thao Nguyen Pham

It was defended on

April 8th 2016

and approved by

Panos K. Chrysanthis, Professor, University of Pittsburgh

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

Adam J. Lee, Associate Professor, University of Pittsburgh

Christos Faloutsos, Professor, Carnegie Mellon University

Dissertation Advisors: Panos K. Chrysanthis, Professor, University of Pittsburgh,

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

**EXPLOITING THE SYNERGY BETWEEN SCHEDULING AND LOAD SHEDDING TO FACILITATE DIFFERENTIATED LEVELS OF SERVICE FOR CONTINUOUS QUERIES**

Thao Nguyen Pham, PhD

University of Pittsburgh, 2016

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 dissertation, we propose DILoS, a framework that supports differentiated QoS and QoD for multiple classes of CQs by tightly integrating priority-based scheduling and load shedding. Unlike existing works that consider scheduling and load shedding separately, DILoS is a novel unified framework that exploits the synergy between them. For the realization of DILoS, we propose ALoMa and SEaMLeSS, two general, adaptive load managers. Our load managers can also be used standalone and outperform the state-of-the-art in three dimensions: (1) they automatically tune the headroom factor, (2) they honor the delay target, and (3) they are applicable to complex query networks with shared operators.

We implemented DILoS, ALoMa and SEaMLeSS in our real DSMS prototype system (AQSIOS) and systematically evaluate their performance using real and synthetic workloads. Our experimental evaluation of ALoMa and SEaMLeSS verified their advantages over the state-of-the-art approaches. Our evaluation of DILoS 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.

To further support differentiated QoS and QoD for CQs in distributed DSMSs, we propose ARMaDILoS, a conceptual framework for large scale adaptive resource management using DILoS. A fundamental component in ARMaDILoS is CQ migration. For this reason, we propose and implement UniMiCo, a protocol to migrate CQs without interrupting the execution of the queries. Our experiments showed that UniMiCo produced correct outputs and did not introduce any hiccup in the response time of the queries.

**Keyswords** Data stream, continuous query, scheduling, load shedding.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ALGORITHMS

## LIST OF EQUATIONS

## PREFACE

Looking back, I feel that the thing I treasure the most during this journey is not this dissertation, but the love and support I have had throughout the years...

I am in deep gratitude to my advisor, Panos K. Chrysanthis. To me Panos is a whole-hearted mentor who has always been available for me with his support, advices, and encouragement. He is also a role model for me about devotion and integrity - whether that is when he is doing research, preparing his teaching material, or reviewing a peer's paper. Panos, I thank you for all your care, understanding and trust, and for the countless late nights you worked to give me feedbacks on my work.

I would like to thank Alexandros Labrinidis, who has been my very supportive co-advisor. Not only did Alex provide me with countless feedbacks and lots of technical helps, he has been a friend who shared with us, his graduate students, many fun activities and memorable moments.

I would like to thank Adam J. Lee, whose role in my years at PITT has been beyond a committee member. I greatly appreciated Adam's careful review of my work and many helpful feedbacks. I was touched when he shared with me his experience as a new parent to help me prepare for the arrival of my baby. I also admired Adam's enthusiasm in his class that I attended, which made the class so well-organized, lively, and helpful.

I would also like to thank Christos Faloutsos for serving in my thesis committee and providing me with many feedbacks to improve my work.

I thank the staff and faculty of the Computer Science Department, especially Kathy Allport, Karen Dicks, Nancy Kreuzer, Keena Walker, Sangyeun Cho, and Daniel Mosse for their help, sharing and advices.

I thank my labmates and classmates: Roxana Gheorghiu, Shenoda Guirguis, Lory Al

Moakar, Rakan Maddah, Santiago Bock, Ruhsary Rexit, Vyasa Sai, Di Bao, Alex Connor, Musfiq Rahman, Sriranjani Mandayam, and many others, for all the collaborations, discussions, sharing and laughers we had together. Thanks for the late night puzzles in the lab, for the badminton games, and for the frozen-yogurt breaks. Thanks for giving me the opportunity to learn about many different cultures.

I thank my dear friend, Hoang Tran, who shared with me every joy and sadness since the start of this journey. Hoang, you would be there for me when I needed, no matter how busy you were. I will never forget what you told me, that busy or not was all about setting priorities.

I thank my other Vietnamese friends, Ha Nguyen, Anh Le, Ngan Nguyen, Son Le, Thuy Bui, Phuong Pham, who gave me the feelings of home away from home, and cheered me up with your warm friendship.

I am grateful to my mom and dad for their unconditional love. Through all the hardship of our family, they never stopped believing that their shy and slow little girl could fly up high, and I am deeply thankful for that.

I thank my husband, Tuan Nguyen, for his love and support, for putting our son to bed every night and taking him out every weekend so I could finish this dissertation. I thank my little son, Nam Nguyen, for entering my life and bringing a whole new meaning to it. Tuan and Nam, thank you for sharing this life with me, through all the ups and downs...

I would like to acknowledge that the work in this dissertation has been supported in part by NSF(IIS-0534531, CAREER IIS-0746696), a gift from EMC/Greenplum, a VOSP fellowship from Vietnam and an Andrew Mellon Predoctoral Fellowship.

# 1.0 INTRODUCTION

## 1.1 MOTIVATION

Today the ubiquity of sensing devices as well as mobile and web applications continuously generate 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 [48], incoming data streams need to be continuously processed and analyzed. Data stream management systems (DSMSs) (e.g., [13, 19, 27, 29, 10, 6, 5]) 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, Apple's iWatch, etc. Also assume two continuous queries: $CQ_1$, that monitors the user's heart rate for the possibility of a heart attack due to abnormally low or high 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 tar-

geted 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 than $CQ_5$. A third example is one where CQs to detect a tsunami [7] would have higher priority than those that detect, understand and predict El Nino and La Nina [11].

In contrast to single-application DSMS, which is dedicated to a specific application, multi-tenant DSMSs host multiple applications and normally provide different service groups with different costs (e.g., gold, silver, bronze etc.). The differentiated service groups determine the priority of the queries subscribed to each group and hence the quality guarantees, i.e., service level agreements (SLAs).

Clearly, it is important to support priority-based query processing in DSMS. CQ's priority has been discussed in research prototypes such as Aurora [13], MavStream [27], and IBM System S [10], and AQSIOS [32].

## 1.2 PROBLEM STATEMENT

For the above reasons, we consider a DSMS (Figure 1) that 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:

- **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 further cost of processing it.

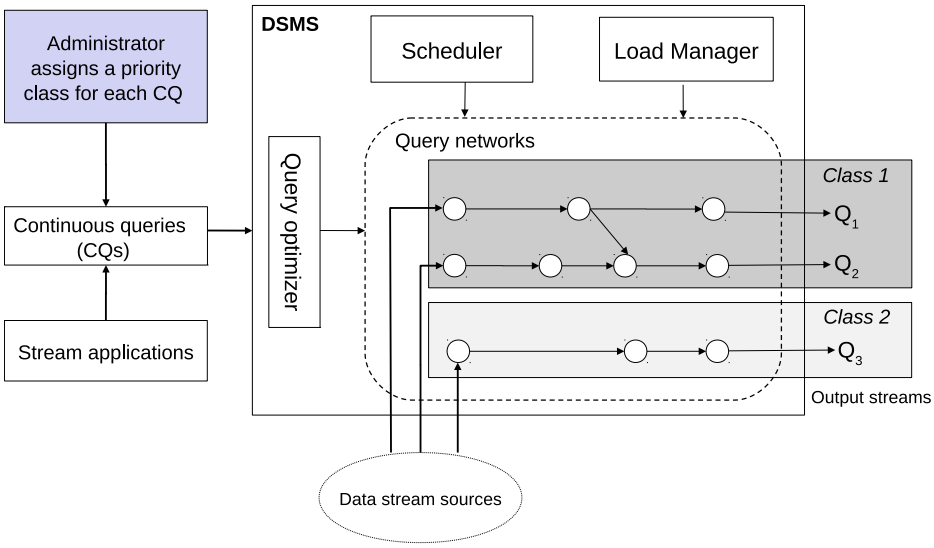


Figure 1: AQSIOS System model.

- **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., [25, 58]) or through load shedding (e.g., [74, 27]), 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.

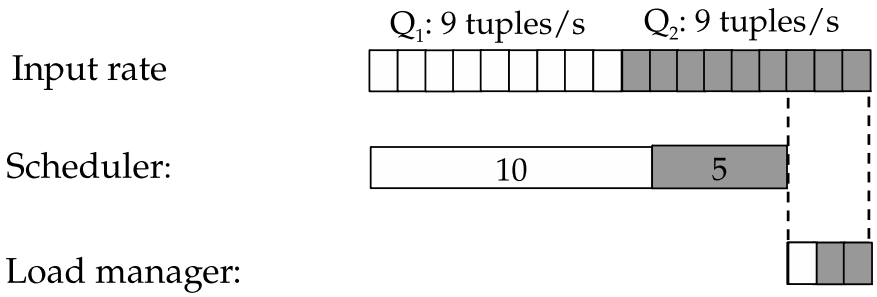


Figure 2: Motivation example (Example 1).

**Example 1.** 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 $Q_2$. We illustrate this example in Figure 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.

Our hypothesis is that *the proper cooperation between the scheduler and the load manager would consistently provide differentiated levels of services for CQs, while using the system capacity more effectively.*

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.

## 1.3 APPROACH

### 1.3.1 Scheduler-load manager synergy framework

We propose DILoS (**D**ynamic **I**ntegrated **L**oad Manager and **S**cheduler) [67, 66], 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 (Example 1), 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.

Our experimental evaluations, with both complex synthetic and real input rate patterns, show the robustness of DILoS and confirm that DILoS achieves the following two basic goals:

- Consistently supporting multiple levels of priorities for CQs.
- Maximizing the utilization of the system processing capacity to reduce the need for load shedding.

### 1.3.2 Adaptive load managers

The implementation of DILoS requires a load manager that is capable of recognizing the scheduler's policy and acts accordingly. Given that state-of-the-art load shedders do not fulfill this requirements, we propose two adaptive load managers, namely SEaMLeSS [65] and ALoMa [66]. Besides enabling the realization of DILoS, SEaMLeSS and ALoMa are also

general, adaptive load managers that perform better than the state-of-the-art alternatives in three dimensions:

- Automatically tune the headroom factor.
- Honor the delay target.
- Applicable to complex query networks with shared operators.

In realizing DILoS, we choose ALoMa because, compare to SEaMLeSS, ALoMa has the advantages of not depending on the fairness of the DSMS scheduler, and easier to be implemented in different DSMSs, as we explain in Sections 4.4 and 5.3.2.4.

### 1.3.3 Large-scale adaptive resource management using DILoS

The elasticity brought by modern cloud infrastructure provides further solution for DSMSs to handle their highly-variable workload, in addition to load shedding: the system can scale out to deal with overwhelming or prolonged overloading, and scale in when the load is light. The stated problem still persists in such a cloud-based DSMS, but with additional challenges, as both the processing capacity and query network now span across multiple nodes.

We outline ARMaDILoS (**A**daptive **R**esource **Ma**nagement using **DILoS**), a conceptual framework for adaptive resource management in cloud DSMSs. ARMaDILoS aims to extend the stated goals of consistently honoring CQs' priorities and increasing system capacity usage in a multi-node cluster. The framework has DILoS as a local workload management unit which, when combined with similar units from other nodes, support a global workload management that considers priority-based capacity distribution across the whole system.

A fundamental component in ARMaDILoS is CQ migration. Although the full implementation of ARMaDILoS is beyond the scope of this dissertation, we propose and implement UniMiCo (**Uni**nterruptible **Mi**gration of **Co**ntinuous Queries), a protocol to migrate CQs because it plays a key role in the framework. UniMiCo [64] supports CQ migration without the need to migrate the states of stateful operators and does not cause any down time in CQ processing. Such a migration scheme is vital for the success of an elastic, cloud-based DSMS model, as the stream applications usually expect CQ's output in near real-time and hence cannot accept query migration that adds latency by stopping and resuming a CQ.

The protocol has been designed in a general way to handle both time-based and tuple-based window. Moreover, it allows migrating a query with multiple stateful operators, each of which could have a different window specification.

## 1.4 CONTRIBUTIONS

This dissertation makes the following contributions:

- **DILoS**, a novel framework that allows consistent integration between the scheduler and load manager in a DSMS to support multiple priority classes of CQs. DILoS also solves the congestion problem typically encountered when there is operator sharing between classes of different priority in a fully optimized query network
- **ALoMa** and **SEaMLeSS**, two new general, practical DSMS load shedders that outperform the state-of-the-art in deciding when the DSMS is overloaded and how much load needs to be shed. At the same time, they are adaptive load management schemes that enables the realization of DILoS.
- **UniMiCo**, an interruptible migration protocol for CQs that does not cause any down time in CQ processing.
- **Prototype implementation**: All the proposed schemes, namely DILoS, ALoMa, SEaMLeSS, and UniMiCo are implemented on AQSIOS [32], our real DSMS prototype. DILoS with ALoMa has been released with AQSIOS 2.0 [4], providing a basic experimental platform for future work on DSMS resource management.

We provide thorough experimental evaluation of the proposed approaches, and compare their performance to the state-of-the-art when applicable (i.e., for ALoMa and SEaMLeSS).

## 1.5 OUTLINE

Chapter 2 presents the background on our assumed DSMS and studies the related work on workload and resource management in DSMSs. Chapter 3 formally analyzes the problem

and presents the overview of our proposed DILoS framework. Chapters 4 and 5 describe our work on SEaMLeSS and ALoMa, respectively. Chapter 6 presents an implementation and evaluation of DILoS using ALoMa, with a discussion on the possibility of incorporating different schedulers and load shedders. We ourline ARMaDILoS and present UniMiCo in Chapter 7, and then conclude in Chapter 8.

# 2.0 SYSTEM MODEL AND RELATED WORK

In this chapter, we first present background on DSMS and CQ processing, with a focus on our assumed system model and experimental platform. We then discuss in more detail the state-of-the-art of DSMS resource management, including scheduling, load shedding, memory management, and workload distribution and balancing.

## 2.1 SYSTEM MODEL

### 2.1.1 AQSIOS

Like most other DSMS architectures (e.g., [13, 19, 29]), our assumed DSMS (Figure 1) has a *CQ processing engine*, together with a *query optimizer*, a *scheduler*, and a *load manager/ shedder*. Users register CQs which are executed as data arrives. The DSMS connects to one or more stream sources, which feed data tuples continuously to the CQs.

We consider a multi-tenant DSMS 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.

AQSIOS [32] is our DSMS prototyped based on the above system model. AQSIOS is inherited from STREAM source code [19], written in C/C++. Extensions made by the ADMT Lab at the University of Pittsburgh include new operator implementation [45], optimization strategy [46], new scheduling policies [71, 57], and all the schemes proposed in this dissertation.

AQSIOS is the platform for all of our experimental evaluation presented in this dis-

sertation. Like STREAM, AQSIOS accepts queries written in CQL language [19]. System parameters such as memory pool size, scheduling method, load shedding scheme, and number of CQ classes and their priorities can be specified in a configuration file prior to execution. Currently, all the query processing in AQSIOS, including scheduling and load management tasks, is single threaded (i.e., they are scheduled to run sequentially). AQSIOS reads data stream from files, simulating each tuple's arrival time based on the timestamp of the tuple. Output tuples are also written to files.

Toward an implementation of ArMaDILoS, our resource management framework for distributed DSMSs, AQSIOS is extended with communication threads, which reports the capacity usage of each class to the coordinator, receives requests from the coordinator, and communicates with the other AQSIOS node during a CQ migration (Figure 33). These threads run in parallel with the main thread which executes CQs. Note that we use this version of AQSIOS only in the experimental evaluation of UniMiCo, our CQ migration protocol (Chapter 7).

### 2.1.2 CQ processing

Each submitted CQ is compiled and optimized into a query plan consisting of multiple relational operators (e.g., select, project, join, or aggregates). In addition, the query plan also consists of one or more *source operator* and an *output operator*. A source operator accepts tuples from a corresponding stream source and transforms the tuple into internal representation format so that it can be processed by the subsequent operators. The output operator converts the output tuples from internal format back to a form understood by stream applications, and either writes the output tuples to a file or database or streams them to applications.

A continuous query plan can be conceptualized as a data flow tree [13, 20], where the nodes are operators that process tuples and edges represent the flow of tuples from one operator to another. An edge from operator $O_i$ to operator $O_j$ means that the output of $O_i$ is an input to $O_j$. Each operator has an *input queue* where input tuples are buffered until they are processed. 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. In AQSIOS, all the queues have a fixed size which can be set in the configuration file prior to execution. If the input queue of an operator is full, the corresponding upstream operator has to pause its processing, waiting for the tuples in the queue to be consumed by the downstream operators.

AQSIOS supports *batch processing*, which allows each operator to process up to a certain number of input tuples in the operator's turn, if the tuples are available in its input queue. This reduces the context switching overhead and hence reduce the processing cost per tuple. The batch size is set at a reasonable value (50 in our experiments), such that the total time to process each batch is much lower than the worst-case response time (i.e. delay target).

Multiple queries with common sub-expressions can be partially merged together to eliminate the repetition of similar operations [69]. For example, in Figure 1 the segment containing the first two operators of $Q_1$ is shared with $Q_2$. 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.

In a query, each operator $O_i$ is associated with two parameters: processing cost and selectivity, as defined below:

**Definition 1.** *Processing cost* ($c_i$) is the amount of time needed for $O_i$ to process an input tuple.

**Definition 2.** *Selectivity* ($sel_i$) is the number of tuples produced after processing one input tuple. $sel_i$ is less than or equal to 1 for a filter operator and it could be greater than 1 for a join operator.

### 2.1.3 Quality metrics

For each CQ in the DSMS, we define below the quality metrics for the CQ:

**Definition 3.** The *response time* of a tuple is the time elapsed between when the tuple enters the system until it is output. For a tuple $t_i$, let $ta_i$ and $to_i$ denote the arrival and output time of the tuple, respectively, then the response time $y_i$ of $t_i$ is calculated as:

$$y_i = to_i - ta_i$$

Only tuples that are output by the query (i.e, not being filtered) contribute to the measuring of response time.

**Definition 4.** 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 dissertation, 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.

Different works have been proposed targeting the QoS (i.e., response time) of a DSMS, among which are query optimization (e.g., [78, 17, 46, 60]) and operator scheduling (e.g., [71, 20]). While query optimization can help the system to handle a higher incoming load, it can not guarantee that the DSMS will be free from overload situations. Scheduling policies in general are able to reduce the average response time of the query outputs since they optimize for queuing time, however they cannot control the response time once the system has got into an overload state (i.e., when the rate of the input load is higher than the processing rate of the system). In such cases, the load manager may shed a necessary amount of load to prevent the response time of the system from increasing unboundedly, which raises the need for a metric on the quality of data.

**Definition 5.** The *Quality of Data (QoD)* of a query is the percentage of output tuples retained after load shedding, compared to the case with no shedding. Let $Ns$ and $N$ be the number of tuples with and without shedding, respectively, QoD is calculated as follows:

$$QoD(\%) = \frac{Ns}{N} \times 100$$

**Conjecture:** A good load manager should maximize QoD (i.e., minimize data loss) while controlling the response time to the specified delay target.

Note that Definition 5 for QoD is only meaningful for CQs in which shedding of input tuples results in proportional loss of output tuples (e.g., CQs consist of select, project, and join operators). For CQs with aggregations, shedding of input tuples results in inaccuracy of output tuples other than loss of them. In such cases, we use shedding rate as a comparative evaluation for QoD.

## 2.2 RELATED WORK: RESOURCE MANAGEMENT IN DSMS

### 2.2.1 Scheduling

Scheduling of CQs in a DSMS focuses on time-sharing the system resources among the query operators. During execution, the scheduler is responsible for assigning each operator a time slot to run, deciding the order to execute the operators in the query network. While Round Robin (RR) has been used as the default scheduling policy in many prototype DSMS systems such as [19, 13], there are many other proposals for scheduling the execution of CQs in a DSMS with the objective of optimizing certain performance goals such as minimizing latency ([25, 71]) or minimizing memory requirements ([18]). A hybrid approach that balances both memory and latency optimization has also been considered [28].

Related to our work on multi-class CQ processing are the works in [13, 25, 27], which consider latency-based QoS functions for each query, and in [79, 52, 81] 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 [58], we proposed another scheduling scheme, called CQC (**C**ontinuous **Q**uery **C**lass scheduler), 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 [57]. None of these works on priority-based schedulers considers the integration with a load shedder to handle overload situations.

### 2.2.2 Load shedding

Load shedding has been proposed in many DSMS architectures as a method to handle overloading [19, 13, 68, 50]. We define below some basic concepts commonly used in the load shedding problem:

**Definition 6.** The *incoming load*, denoted by $L$, is the amount of time needed to process all the tuples that come to the system per time unit (say, a second). $L$ is proportional to the processing cost of the whole query network and the input rate of the input stream.

**Definition 7.** The *system capacity*, denoted by $L_C$, represents the fraction of each time unit the system can spend on processing the incoming tuples. Since the DSMS might share the CPU with some other processes and also spends part of its processing time on other tasks such as context switching, statistics collection, etc., this fraction of time for tuple processing is normally less than 1 and is approximated by a a *headroom factor*, $H$, which is typically in the range of (0,1).

**Definition 8.** *Overload* refers to a state of the DSMS (or a class of CQs) at which the *incoming load* $L$ to the system is higher than the *system capacity* $L_C$. In such a situation, the queuing time accumulates over time, causing the response time to increase unboundedly, exceeding the specified the delay target.

In [74], Tatbul et al. articulate four basic questions for a load shedder: *when, how much, where and what* to shed. This dissertation focuses on the first two questions: when and how much.

The works in [74, 73, 21] 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. [59] 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 [74, 31, 30, 36] to increase the usefulness of the query results after shedding. Also related to this question, in [68, 61, 42, 41, 43] the authors propose methods to shed load other than simply discarding tuples from a query network. In [68], dropped tuples are routed to a lightweight *shadow plan* that produces approximated results. The work in [61] is customized for spatio-temporal data streams, in which a dropped tuple is approximated by the mean value of the cluster it belongs to. In [41, 43] the system load is shed by selecting only subsets of the windows to perform the joins. In [42] the DSMS delegates the load shedding task to the source filters, which apply varying amounts of shedding to different regions of the data space. [75] considers a whole window, not a single tuple, as the shedding unit.

There are a few previous works addressing the questions of *when and how much* to shed ([76, 74, 68, 50]), the first questions to be answered by any load shedding module. Compared to the other existing schemes that address these questions, CTRL [76] and Aurora [74] are the most mature schemes in term of reacting on time to overload situations as well as minimizing data loss. Compared to each other, CTRL and Aurora have complementary strengths: CTRL is able to control the response time to the delay target, while Aurora is able to handle complex query networks (with join, aggregate and shared operators). None of them, however, has both of these two necessary abilities, i.e., delay-target awareness and applicable to all types of query network. In addition, both of these schemes depend on a manually-tuned headroom factor, which is subject to change during execution and requires constant monitoring and human intervention. In this dissertation, we propose ALoMa and SEaMLeSS, two adaptive load managers that have all these required properties of a practical, general-purpose load shedding scheme. ALoMa and SEaMLeSS enable us to build our proposed scheduler-load manager integration framework. Compared to each other, ALoMa is more flexible because it does not require the fairness of the scheduling policy as SEaMLeSS does.

*Admission control* can be viewed as a more proactive way of load shedding: the system decides to drop some of the queries rather than the data. 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 [80], the goal is to maximize the utilization of the system and the overall importance of the CQs, whereas in [56], the goal is profit maximization, strategyproofness and sybil immunity even at the expense of system utilization. Admission control is not considered in this dissertation. However, like the works on where and what to shed, admission control can be used in combination with a load shedder such as ALoMa, which gives the answer to the questions of when and how much to shed.

Combination approaches have been proposed in different settings. For example, [44] combines admission control and load shedding (i.e., update shedding and query shedding) in a mobile CQ setting. In [40], the authors model both load shedding and resource allocation as a dual optimization problem, formally solves the problem and illustrates the solution using a simulation. This work 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 [74] or maximal tolerable relative error [50, 27]. 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, unlike our DILoS framework, none of these load shedders can provide feedback to the scheduler to improve scheduling decisions.

In [80], 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, this work considers job admission rather than load shedding and does not provide any guarantees on QoS and QoD for different ranks as our scheme does.

### 2.2.3 Memory management

Constrained by the near-realtime requirement of monitoring application, a DSMS normally executes the CQs over the incoming data stream in memory, limiting disk I/O overhead. Memory usage in a DSMS mainly falls into two categories: operator state and buffer space for queued input and intermediate results.

Because a data stream is infinite, the memory required to maintain the states of some stateful operators might be unbounded. As characterized in [15], a set of CQs can be computed using bounded memory (e.g., selection, duplicate-preserving projection, min/max), while the other requires memory that grows linearly with the input size (e.g., join, duplicate-eliminating projection, most aggregation with group-by). Because a data stream is infinite, most DSMS employ techniques such as sliding window (e.g., [19, 13]), punctuation ([77]) or heartbeat [19] to divide the stream into overlapping, finite data sets, over which the continuous queries are evaluated. Such techniques help to bound the memory used to maintain the states of stateful operators, in addition to helping to produce timely outputs.

There have been several works trying to optimize the memory used for CQ processing, targeting both categories of memory consumption. Adaptive query processing techniques are

usually used to adjust in-memory states to cope with memory limitation. Example of these are [39], in which part of the state of a join operator is adaptively pushed to disk yet trying to hide disk I/O latency, and [24], in which window size and slide are increased/decreased according to the load state of the system. On the other hand, scheduling strategies are proposed to minimize the queue size, including [20], [22], and [28]. The general idea of these schedulers is to execute first the operator that can reduce the most its input size. In [55], the authors present preliminary study on the implication of different types of memory on the performance of CQs.

Our assumed DSMS system implements sliding windows, which is the most generic and commonly used method. For the work in this dissertation, we assume that the available memory is sufficient for all computing needs, hence no CQ priorities need to be considered in memory allocation. The memory manager simply allocates memory blocks whenever there are requests from the operators to maintain their states or to accommodate waiting tuples.

### 2.2.4 Workload distribution and balancing

**2.2.4.1 CQ migration** Flux [70] was one of the early attempts to introduce a monitoring and load detection operator in a query network, and provided a state migration protocol to move CQs across different machines. Fernandez et al. [26] presented a solution in which backup Virtual Machines (VMs) are used in a distributed network of VMs for periodically storing state. In the event of load imbalance, CQs are migrated by receiving the state from the backup VMs, and resume execution by the time the full state has been transferred, along with incremental changes. Recently, Lin et al. [54] discussed an operation migration mechanism, which also follows the state migration paradigm.

The efficiency of the migration mechanism is crucial, and no system downtime is accepted since it translates to loss of data (hence the term "live" in previous work). Further, performing a migration imposes additional load to a machine, which can sometimes make matters worse and prolong an overloaded situation.

Our approach on CQ migration shares the basic idea of the *Window Recreation Protocol* (WRP) presented in [47]. In WRP, an operator's state for the migrating window is recon-

structed at the target node without the need for state transfer. However, WRP can handle the migration of a sub-query with only one stateful operator, and considers only time-based windows. In contrast, UniMiCo's protocol has been designed in a general way to handle both time-based and tuple-based window. Moreover, it allows migrating a query with multiple stateful operators, each of which could have a different window specification. Finally, unlike WRP, UniMiCo does not need to involve the upstream data source in synchronizing the migration point.

**2.2.4.2 Other works on large-scale DSMSs** There have been a number of previous works on workload distribution and balancing for database systems on the cloud. Since data partitioning and replication is the key achieving scalability for cloud OLTP databases, many previous works focus on data migration techniques that avoid service downtime and reduce latency (e.g., [38, 35, 23]), and new data storage structures associated with "good enough" consistency level that reduce the data synchronization overhead among the partitions (e.g., [37, 34]). For OLAP databases on the cloud, the focus is more on parallelizing the processing of complex analytical queries, with the map-reduce paradigm receiving a big attention (e.g., [33, 14])

A cloud DSMS shares some characteristics with an OLAP database system, where queries are read-only, long-running and are the focus of workload partitioning. However, while the load of an analytical query is basically stable during its execution, the load of a continuous query can fluctuate considerably due to the fluctuation in arrival rate of the incoming streams or the value distribution of the incoming data. Therefore, while a workload distribution plan for an analytical query can be fixed with the exception of node failure, that of a continuous query network has to be re-evaluated on the fly and query/operator migration might need to happen frequently for load balancing purposes.

There have been previous works on finding a query network deployment that is resilient to workload fluctuation at run-time [82, 72, 53]. Other works are on strategies to parallelize continuous queries across multiple nodes, such as [83, 47]. The work in [26] also integrates fault tolerance and scaling out of stream operators. In [63] and [62], the authors consider the problem of splitting CQ processing between DSMS server and client mobile devices, with

the goal of minimize power consumption. These work have different goals from ours, as they do not target priority-based adaptive resource management.

In [80], the authors discuss job admission in IBM System S, assuming a highly-overloaded DSMS where load shedding is not sufficient. In this system, in every epoch continuous stream processing jobs are considered for admission and assigning to a set of serving nodes. The rank of the jobs is taken into account in the admission decision, yet other factors are also considered to optimize for the total importance of the selected jobs. This system is relevant to our proposed work on a multi-node deployment of DILoS with respect to a dynamic workload distribution plan. However, the system model is different: we assume that the system is sufficiently provisioned so overloading just happens occasionally, and load shedding is able to handle it. We also aim at an explicit priority guarantee for each class of queries before optimizing for some overall metrics. In [84], the author also discusses a dynamic operator placement scheme to proactively balancing load among nodes. The goal of the scheme, however, is different from that of our proposed work: the scheme aims at minimizing worst-case relative performance among the CQs, i.e., aims at providing the same quality of service for all CQs, while we aim at providing differentiated services for different class of CQs and maximizing capacity usage.

Other commercial stream processing systems such as Storm [3], Spark [9], Flink [2], Samza [8] also support distributed and scalable processing of CQs. However, those systems do not scale automatically as what our ARMaDILoS aims for, but instead require users to monitor and add or remove the executors themselves. Amazon Kinesis [1] is a multi-tenant, cloud-based stream processing system, which charges a registered stream a monetary cost based on the throughput the stream requires. Yet the required throughput has to be specified upfront, and explicitly increased and decreased by the stream's owner at runtime. Amazon Kinesis does not apply load shedding or dynamic scaling to cope with the fluctuation of the stream load.

A few previous works [51, 49] has proposed dynamic, automatic scaling of cloud DSMS to cope with variability in input load. However these systems are still at an early stage with simplified system models. Unlike ARMaDILoS, they do not consider CQ priorities.

## 2.3 SUMMARY

In this chapter we presented the background on our assumed DSMS and studied the related work on workload and resource management in DSMSs, including scheduling, load shedding, memory management, and, for cloud DSMSs, workload distribution and load balancing. We also discussed why previous works could not solve the problem stated in Section 1.2.

Workload and resource management is a common problem in many systems including DSMS, real-time database, networking, and web services. 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. In the scope of this dissertation, our discussion focuses on workload management in DSMS, which are the most closely-related to our work.

# 3.0 DILOS: DYNAMIC INTEGRATED LOAD MANAGER AND SCHEDULER

In this chapter we formally analyze the problem of integrating priority-based scheduler and load manager and present the basic idea of DILoS [67, 66], our framework for the cooperation between the load manager and scheduler in a DSMS. We also point out the load management challenges in realizing DILoS, which motivate us to seek for new adaptive load managers.

## 3.1 DILOS AS A GENERAL PRIORITY-BASED SCHEDULER AND 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 9.** *Scheduling policy* Let $C_k$ denote the $k^{th}$ 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 $f_T : P_k \mapsto T_k$, such that $\sum_k(T_k) \leq T$, where $T_k$ is the total time the class $C_k$ receives during T.

Example 1 in Section 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).

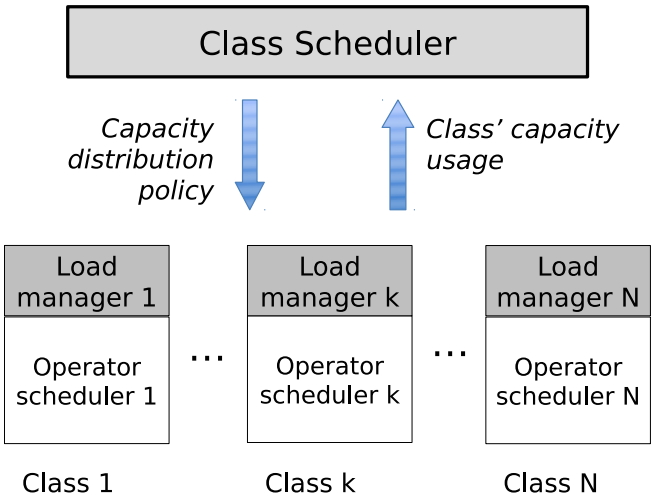


Figure 3: Overview of the proposed DILoS framework.

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*.

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 $L_{Ck}$ 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 3 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 Section 3.2.

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 $\frac{L_k}{L_{Ck}}$) 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 Section 6.2.2.

## 3.2 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.

### 3.2.1 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 4 illustrates this, in which a query of class 1 (higher

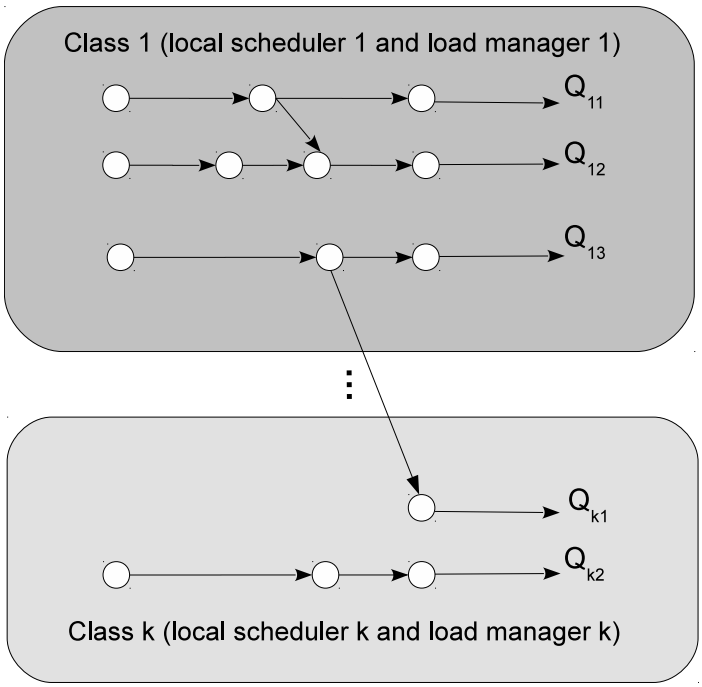


Figure 4: Inter-class sharing in DILoS, with class 1 (high priority) sharing a segment with class k (lower priority).

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.

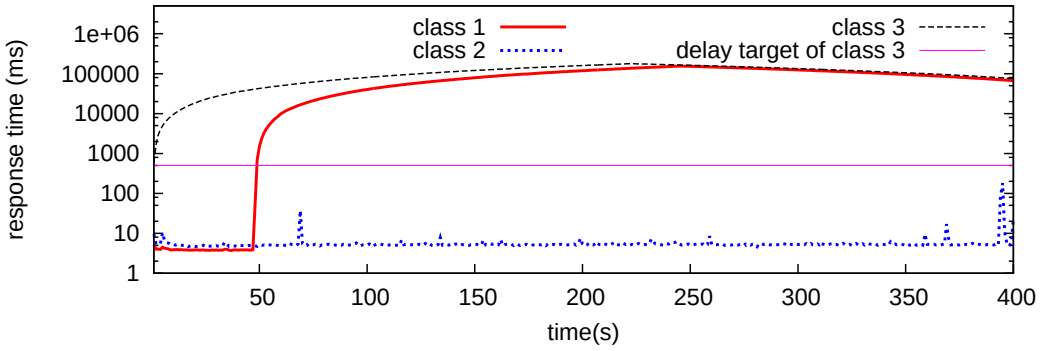


Figure 5: Congestion problem: Class 1 sharing a query segment with class 3 and is affected by the congestion in class 3.

We demonstrate this problem in Figure 5, with an experiment using three classes, with class 1 having the highest priority and class 3 the lowest. The delay targets of class 1, 2, and 3 are 300ms, 400ms, and 500ms, respectively. In this experiment, we enable the class 1 and class 3 to share a query network segment consisting of two source operators and two select operators. Because class 3 is overloaded, the response times of the class 1's queries that are shared with class 3 are affected by the congestion and increase dramatically after a certain period. Class 2, although having lower priority than class 1, is not affected because it does not share any query segment with class 3.

### 3.2.2 Handling inter-class sharing in DILoS

Interestingly, the aforementioned problem on priority inversion can be solved with an appropriate employment of load management based on the following observation: as long as the low-priority class can keep up with the incoming workload including the input fed by the shared segment, i.e., does not become overloaded, there will be no congestion of intermediate tuples at the end of the shared segment, i.e., the shared queue.

**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 per-class load manager enabled in DILoS, 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.

## 3.3 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 [74], 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 10.** The *incoming load of class* $C_k$ in a time unit, denoted $L_k$, is given by:

$$L_k = \sum_i (r_i^k \times load\_coef_i^k) \tag{3.1}$$

where $r_i^k$ denotes the input rate of the $i^{th}$ input stream of class $C_k$, and $load\_coef_i^k$ is the load coefficient of the stream.

**Definition 11.** The *load coefficient of the $i^{th}$ 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:

$$load\_coef_i^k = \sum_j (c_j \times \prod_{1 \leq m < j} sel_m) \tag{3.2}$$

where $c_j$ is the processing cost per tuple of the $j^{th}$ operator in the path from the input stream to the corresponding output, and $sel_j$ is 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 [74].

Since the input rates, costs and selectivities all change frequently at runtime, $L_k$ needs to be recalculated periodically.

**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 Section 3.3.1. In the case of our per-class load management, the *actual capacity portion* each class obtains ($L_{Ck}$) 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, as we discuss below.

### 3.3.1 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", proposed in Aurora [74], 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 [76], Aurora does not have a self-correcting mechanism to prevent the response time from exceeding a delay target.

CTRL [76] 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 $i^{th}$ period is then estimated by the following equation, called the *delay estimation model*:

$$y^i = \frac{c}{H} q^{i-1} = \frac{c.T}{H} \sum_{j<i} [f_{in}^j - f_{out}^j] \tag{3.3}$$

where $y^i$ is the response time at the $i^{th}$ 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 [76] 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 [76], the authors *manually* try different values of H in Eq. 3.3 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 6 gives an example of such a case, where the result from the Select operator $\sigma_2$ 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.

Some other schemes have also been discussed, yet they are not as complete as Aurora and CTRL. The scheme in [50] 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 [68] and [51], 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.

The above limitation of the state-of-the-art load shedders motivates us to develop more practical load management schemes for DSMSs in general and for DILoS in particular. We

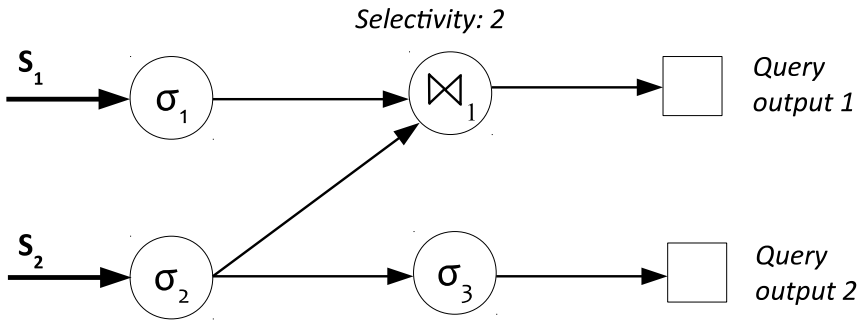


Figure 6: A query network with joins and shared operators, for which the delay estimation model of CTRL would not work.

propose two new schemes, namely SEaMLeSS and ALoMa, that have both the complementary strengths of CTRL and Aurora, while overcoming their weaknesses. More specifically, our new schemes aim at the following properties:

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

We will present in detail both SEaMLeSS in Chapter 4 and ALoMa in Chapter 5.

## 3.4 SUMMARY

In this chapter we presented DILoS as a general framework for the cooperation between the load manager and scheduler in a DSMS. We discussed why the state-of-the-art load shedders is not sufficient to realizing DILoS: they are not able to recognize the system capacity, and either fails to control the response time (Aurora) or not applicable to complex query network (CTRL). This motivates us to propose our adaptive load managers, ALoMa and SEaMLeSS, which we presents in the next chapters.

# 4.0 SEAMLESS

In this chapter we present SEaMLeSS(**SE**lf **M**anaging **L**oad **S**hedding for data **S**tream management systems) [65], which is our first attempt to build a load managers that achieves the three desired properties of a practical load manager, namely 1) honoring the delay target, 2) not requiring any manual-tuned headroom factor, and 3) applicable for all types of query networks. We first outline the basic idea of SEaMLeSS, followed by details of how SEaMLeSS handles complex query networks and automatically adjusts the headroom factor. We present experimental evaluations and finally summarize SEaMLeSS's advantages and limitations.

## 4.1 OVERVIEW

SEaMLeSS follows the design of CTRL [76] in applying a delay estimation model to estimate the response time from the number of queued tuples, and using control theory to determine the shedding amount for the next cycle (Section 3.3.1). This design allows SEaMLeSS, like CTRL, to effectively manage the response time of the DSMS to honor the delay target. However, SEaMLeSS has the following improvements over CTRL:

- Instead of simplifying the details of the queued tuples by using the virtual queue, we propose the concept of *queued load* and use that in SEaMLeSS to estimate the response time without any assumption on the type of the query network. This improvement enables SEaMLess to be applicable to all types of query networks including those containing joins, aggregations or shared operators.

- SEaMLeSS uses the actual response time of the outputs as feedback to automatically adjust the headroom factor, thereby removing the need for manually-tuned one.

In the next section, we will present in detail the implementation of SEaMLeSS and show how the above two properties are realized.

## 4.2 IMPLEMENTATION

### 4.2.1 Handling complex query networks

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

$$qL^k = \sum_i (q_{o_i}^k \times load\_coef_{o_i}) \tag{4.1}$$

where $o_i$ denotes an operator in the query network, $q_{o_i}^k$ is the length of the physical input queue of $o_i$ at the $k^{th}$ period, and $load\_coef_{o_i}$ is the load coefficient of the query branch rooted at $o_i$, which is calculated following Eq. 3.2 in Section 3.3.1.

Assuming that the query processing task of the system is carried out sequentially and the DSMS is using a fair scheduler such as Round Robin, then a tuple coming to the system at time k has to wait for all queued tuples in the system up to time k-1. Therefore, the estimated response time for the tuples coming during the $k^{th}$ period is given by Eq. 4.2, which is a modification of Eq. 3.3 in [76]:

$$y^k = \frac{qL^{k-1}}{H} \tag{4.2}$$

The Eq. 4.3 presents the SEaMLeSS' feedback controller, which is an adjustment of the one in [76]. In each control period, this feedback controller is used to determine $u^k$, which is *the amount of load* that can be added to the queue in the next period without violating the delay target.

$$u^k = H \times [b_0 e^k + b_1 e^{k-1}] - a u^{k-1} \tag{4.3}$$

where $e^k = y^k - D$ and $a$, $b_0$, $b_1$ are the controller parameters. Details on the design of the controller and the derivation of these parameters can be found in [76].

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

### 4.2.2 Headroom factor auto-adjustment

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

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

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

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

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

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

## 4.3 EXPERIMENTAL EVALUATION

### 4.3.1 Experiment settings

We evaluated SEaMLeSS in AQSIOS, our real DSMS platform. In this section we present two sets of experiments, one confirming the effect of off-tuned headroom factor on the two state-of-the-art schemes (i.e., Aurora and CTRL), and one evaluating SEaMLeSS. All experiments were run 5 times and we report the averages, ensuring statistically significant results.

**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 [76] [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 1,140 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 Section 3.3.1.

**Input data**: We use two streams of synthetic data, denoted $S_c$ and $S_{step}$, and one of real data $S_r$. 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.

- $S_c$**:** has a constant input rate of 200 tuples/s, which is within the system capacity, for

[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.

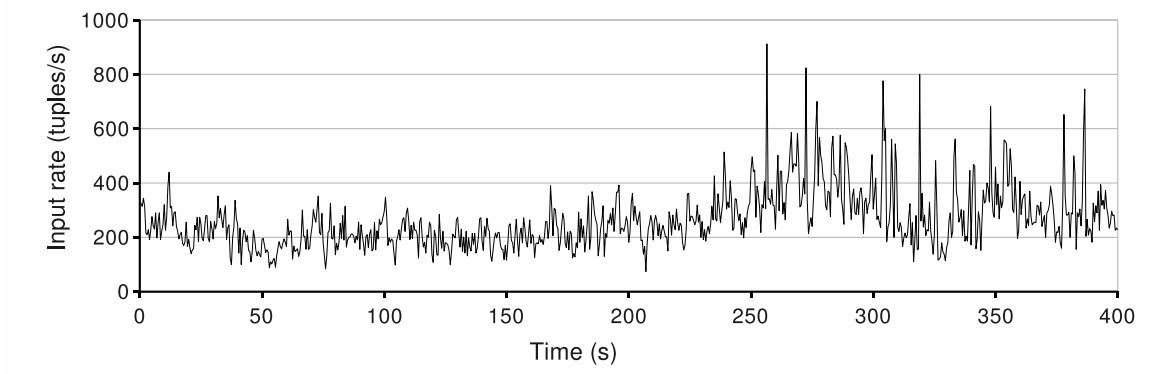


Figure 7: Input rate of the real data in $S_r$ and $SD_r$.

the first 10 seconds, and then goes to 350 tuples/s, which overloads the system, until the end of the experiment at the $400^{th}$ second. $S_c$ is used when we want to keep the input rate constant to clearly examine the effect of the factor of interest.

- $S_r$**:** is a trace of TCP packets between the Lawrence Berkeley Laboratory and the rest of the world[2]. Figure 7 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 [250ms-1000ms] 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 [76] 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

[2] 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.

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.

### 4.3.2 Effect of incorrectly-tuned headroom factor on Aurora and CTRL

In this section we experimentally verify our observations in Section 3.3.1 about the dependence of Aurora's and CTRL's performance on the selection of the headroom factor, and study how much the effect of an incorrect selection would be. We use the QN-flat query network, together with the $S_c$ input streams.

**4.3.2.1 Effect of incorrect headroom factor on Aurora** It is actually difficult to determine a "correct" headroom factor for Aurora, as it does not have any feasible method to select one. The right value should be the one that prevents the response time from exceeding the delay target, while, compared to other values that can do so, minimizing data loss. A correct headroom factor for CTRL does not guarantee to work for Aurora, due to all the difference in the estimation of excess load.

We show in Figure 8 the detailed response time under Aurora with different headroom factor values[3], and Table 1 summarizes the average response time and data loss. We can observe that Aurora is extremely sensitive to the headroom factor: a difference of 5% in the headroom factor can create a huge change in the violation of delay target (up to more than 300% ), and significant difference in data loss (up to 7.7%). In fact, even when the headroom factor is just a little higher than the correct one, such as 0.99 in this case, Aurora can no longer stop overloading and therefore, the response time keeps increasing although the input rate remains constant.

[3]In the current prototype, which is single threaded, a correct headroom factor cannot be higher than 1. However, when the system is parallelized to exploit the multi-core infrastructure, the headroom factor, in theory, can approach the number of cores being used.

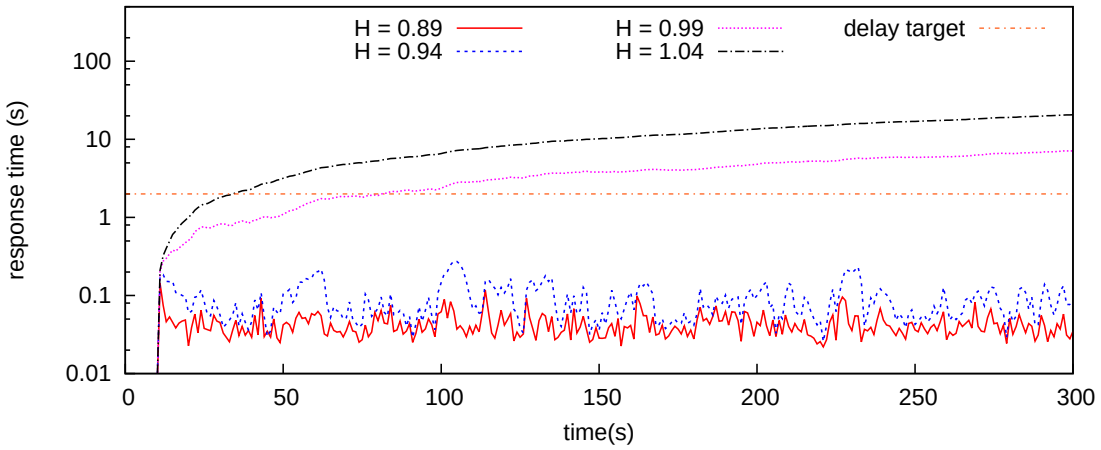


Figure 8: Effect of headroom factor tuning on Aurora, with constant input rate $S_c$.

**4.3.2.2 Effect of incorrect headroom factor on CTRL** We first show how CTRL performs, under a constant input rate, with values for the headroom factor higher or smaller than the correct, manually-tuned one (0.99 in this case). The detailed response time is shown in Figure 9 and the average response time and data loss are shown in Table 2.

In this case, while the delay violation increases significantly when the headroom factor is higher than the correct one, the data loss is not much higher if the headroom factor is lower than it should be (about 0.1% when the headroom factor is 5% lower). This can

Table 1: Effect of headroom factor tuning on average delay violation and data loss under Aurora, with $S_c$.

| Headroom factor | Delay violation | Data loss |
|---|---|---|
| 0.89 | 0.00 sec | 45.44% |
| 0.94 | 0.00 sec | 41.89% |
| 0.99 | 4.60 sec | 38.80% |
| 1.04 | 17.48 sec | 36.90% |

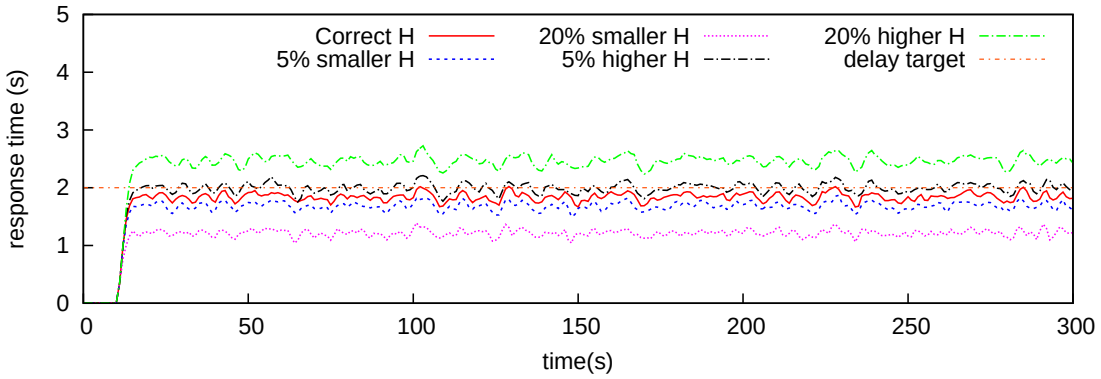


Figure 9: Effect of incorrect tuning of headroom factor on CTRL, with constant input rate $S_c$.

be explained: the difference is only at the time the load shedder starts shedding (i.e., if the headroom factor is smaller the scheme will start shedding earlier and hence lose more data). In the later period when the system remains overloaded, the fact that the response time stops at a constant value means the incoming workload is exactly equal to the system capacity, that is, the shedding rates are the same although a different headroom factor is used. The difference in data loss will be more significant if the workload goes back and forth from normal state to overload state.

We can see that, using the length of the virtual queue as feedback, CTRL manages to reduce the effect of an incorrectly chosen headroom factor so that it is not as severe to CTRL as it is to Aurora. However, such an effect is still not desirable: the load shedder still either violates the delay target or drops more data than necessary.

### 4.3.3 SEaMLeSS evaluation

**4.3.3.1 Under system environment changes** Selecting a correct headroom factor for CTRL is a daunting task, but despite being carefully selected, the headroom factor is not guaranteed to be correct for the whole execution time. In fact, it is virtually guaranteed not to be correct for the the whole execution time. In this experiment we illustrate this by

Table 2: Effect of headroom factor tuning on average delay violation and data loss under CTRL, with $S_c$.

| Headroom factor | Delay violation | Data loss |
|---|---|---|
| 20% lower | 0.00 sec | 40.11% |
| 5% lower | 0.00 sec | 40.03% |
| Correct (0.99) | 0.00 sec | 39.99% |
| 5% higher | 0.03 sec | 39.96% |
| 20% higher | 0.47 sec | 39.91% |

launching background jobs while the DSMS is running. We use the input $S_c$ and the QN-flat query network.

Figure 16 shows the response time under CTRL, which used a fixed, manually-tuned headroom factor, and our SEaMLeSS, which automatically adjusts the headroom factor at runtime. At the beginning, the headroom factor tuned for CTRL is correct so it manages to keep the response time at the delay target. SEaMLeSS does not have such a well-tuned headroom factor, yet it quickly picks up the correct value and can control the response time as efficiently as CTRL. When some background jobs are launched and share the processor with the DSMS at the $100^{th}$ second, the headroom factor used for CTRL is no longer correct, making the response time twice as high as the delay target. SEaMLeSS, however, is able to adapt very quickly to the change, and still honor the delay target. Figure 16 shows the headroom factor adjustment made by SEaMLeSS in response to the change in the system environment.

When the query network is flat, which is the case in this experiment, [76] has shown that CTRL outperforms Aurora. Therefore the fact that SEaMLeSS performs equivalent or better than CTRL in this experiment also means that SEaMLeSS outperforms Aurora with a flat query network.

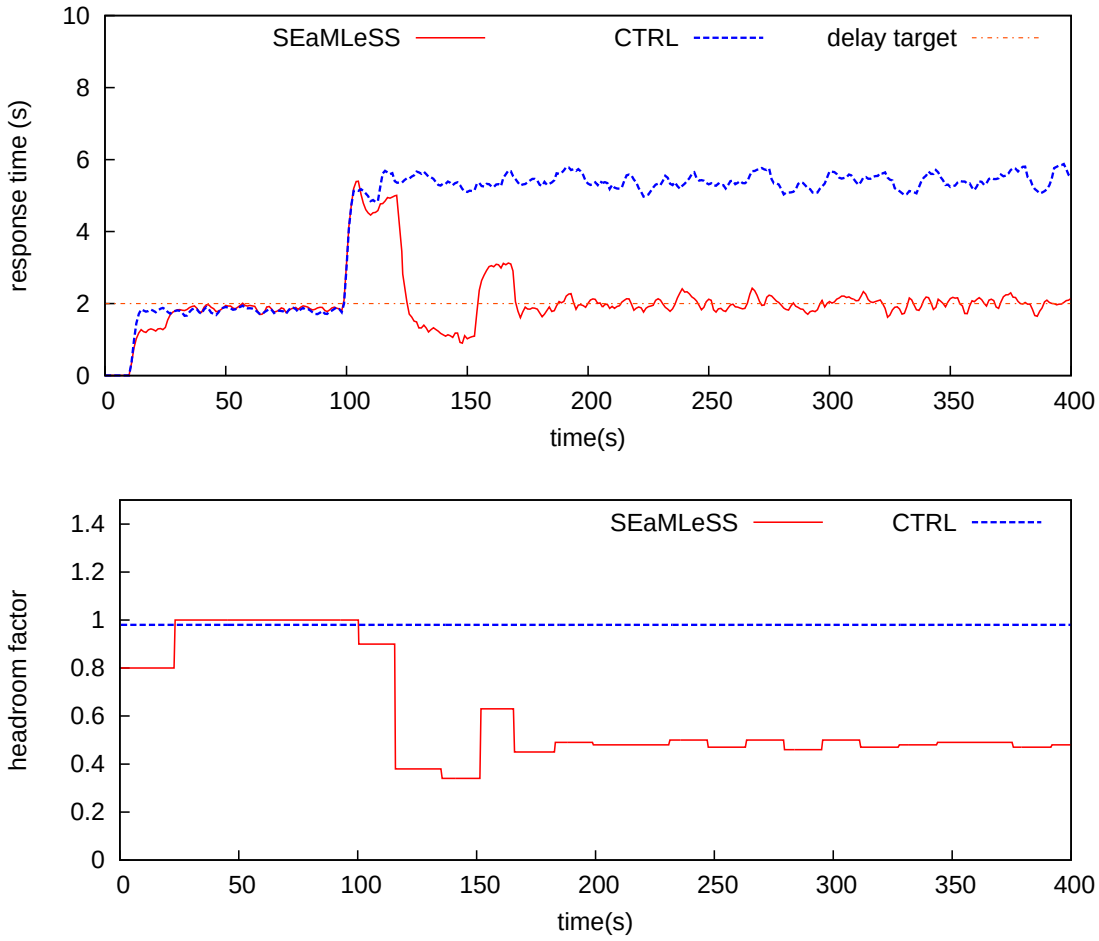


Figure 10: Effect of environment changes on CTRL and adaptation of SEaMLeSS. Top plot shows the response time, bottom plot shows the headroom factor recognized by each scheme. Total data loss for SEaMLeSS and CTRL is 62.98% and 62.69%.

**4.3.3.2 With a complex query network** In this experiment we use a complex query network (QN-complex) for which CTRL's estimation is no longer correct. Since [76] does not compare CTRL's performance to Aurora for complex query networks, we include Aurora in this evaluation to confirm that SEaMLeSS also outperforms Aurora in this case. Because the Aurora scheme does not suggest a way to pick a correct value for the headroom factor, we ran it with a range of possible values. However, in this setup no value of the headroom factor could enable it to perform equivalently to SEaMLeSS. If the headroom factor is too

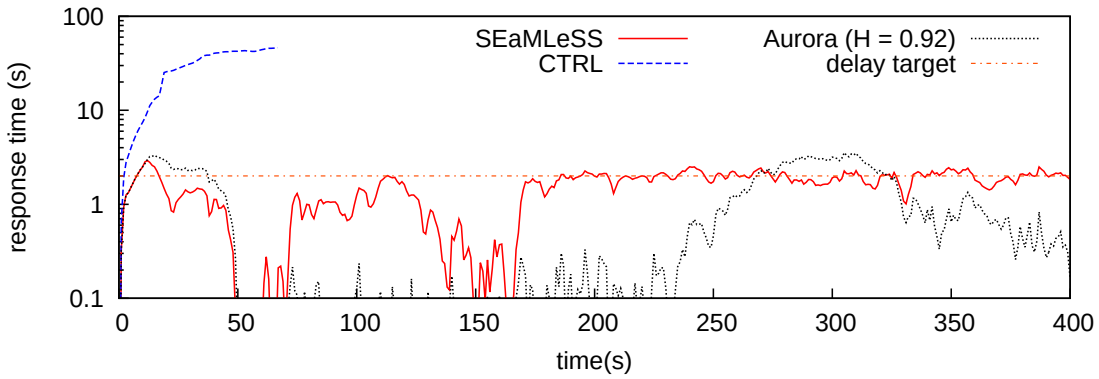


Figure 11: Response times with QN-complex and $S_r$. The X-axis plots the input timestamps, showing that within the specified experiment time the system under CTRL was only able to process tuples coming in the first 66 seconds. Note that the Y-axis is in logarithmic scale.

small, the response time is kept well below the target at all time by dropping much more data unnecessarily. When the headroom factor equals 0.92 (Figure 11), the average delay violation of Aurora is roughly the same as SEaMLeSS but Aurora drops considerably more data (Table 3). Increasing the headroom factor to 0.93 makes the delay violation to be significantly higher (due to the higher peak in the response time) and the data loss is still higher than SEaMLeSS. This is consistent with the properties of Aurora analyzed in [76]: the Aurora method is not aware of the delay target and cannot recover from a previous wrong decision since it does not look at its outcomes.

Table 3: Delays and data loss with QN-complex and $S_r$.

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

The method given by CTRL to tune the headroom factor cannot be applied with the complex query network: no matter how we change the value of the headroom factor, the delay estimated by CTRL does not match the real output delay. Because the query network contains a shared operator, an input tuple actually corresponds to several tuples in the output flow. CTRL cannot recognize this mapping and hence it miscalculates the length of the virtual queue. We still tried to run CTRL with the headroom factor equal 0.99 (i.e., the value we tuned for QN-flat). As we show in Figure 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 420s), the system with CTRL has only been able to process input tuples coming in the first 66s (out of 400s). We tried some other values of the headroom factor from 0.8 - 0.99 as well, but they do not make any observable difference to the performance of CTRL compared to that in this case.

**4.3.3.3 Sensitivity analysis** In this experiment we show the sensitivity level of SEaMLeSS to the headroom adjustment period, denoted P.

We ran SEaMLeSS's headroom adjustment algorithm varying P from 1 to 60 control periods with the experiment presented in Section 4.3.3.1, in which the headroom factor changes significantly at the $100^{th}$ second. We expect that when P is large, it takes SEaMLeSS longer to adjust the headroom factor but it is more stable. When P is smaller SEaMLeSS starts adjusting earlier but it tends to make more inaccurate adjustments and hence becomes less robust against fluctuation caused by system events.

The sensitivity analysis shows that in this case SEaMLeSS works best (in term of both delay violation and data loss) with P in the range of [20-40]. To provide more insight, we show in Figure 12 the three cases with P equals 1, 30 and 60. As expected, when P = 1 the adjustment decision is much less accurate so it has to adjust it many times before getting to the appropriate value. And its response time afterward also fluctuates more than the others. With P=60 the load shedder has to wait for a long time unnecessarily before adjusting the headroom factor.

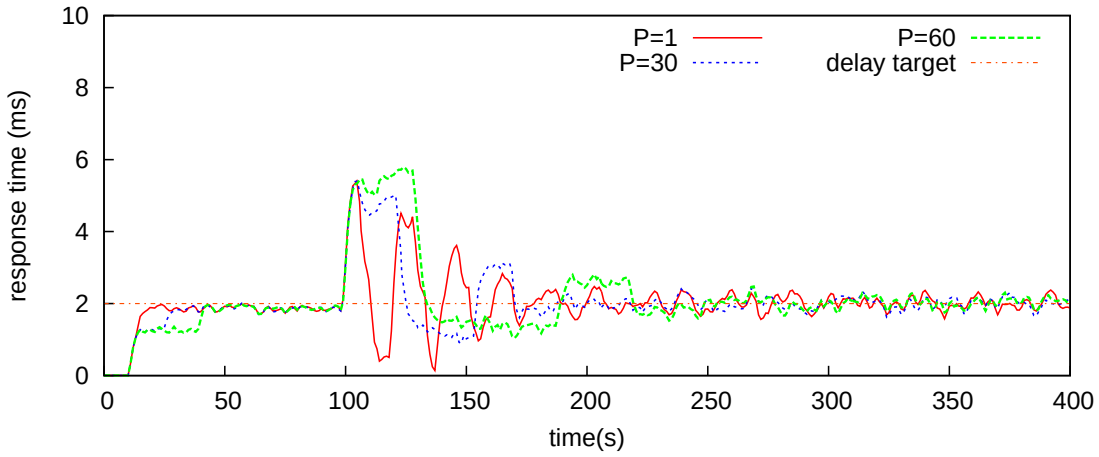


Figure 12: Effect of different headroom adjustment periods on SEaMLeSS.

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

## 4.4 SEAMLESS'S LIMITATION

SEaMLeSS overcomes two major drawbacks of CTRL and has all of the stated desired properties for an adaptive load manager. However, SEaMLeSS still has some limitations as discussed below.

First, SEaMLeSS depends on the delay estimation model. Although the model has been extended to capture complex query networks, it still only works under a *fair* scheduling policy in which the waiting time of new coming tuples depends only on the total queued load. For priority-based schedulers, SEaMLeSS would exhibit a problem: the response time of the low-priority queries can be much higher than the promised worst-case response time. We present in Section 5.3.2.4 an experiment showing how SEaMLeSS performs under a weighted Round Robin scheduler.

Second, SEaMLeSS needs to know the number of tuples waiting in the input queues of all CQs. This requires a model where an independent thread runs in parallel with the query processing engine to receive the input tuples, count them and inform the load shedder. Such a thread must not be overloaded (otherwise it cannot keep up with the incoming tuples to count them), meaning that it cannot share the same CPU with the thread that processes the query network. Although it was not mentioned explicitly before, this is also a problem with CTRL's applicability.

AQSIOS does not have such an independent counting thread available. In AQSIOS, the number of tuples that have come to the system can be counted by the source operators when it reads the tuples. However during overloading, the source operators (which are part of the query processing thread) cannot keep up with the input rate to count the incoming tuples (since they are also overloaded and have not processed up to the last tuples coming yet). Therefore, in our experiments we had to prepare additional information for SEaMLeSS (and CTRL) to simulate the case when such information is available. Specifically, we provide these schemes with a file specifying the number of new tuples coming in every load management cycle. The scheme reads the file and calculates the numbers of tuples that have come up to the current time, even though the tuples have not been read by the source operator (i.e., the tuples are waiting at the input queues of the sources). Such a file is prepared in advance based on the input rate of the data that will be use for the experiment[4].

## 4.5 SUMMARY

In this chapter we presented SEaMLeSS, our first attempt for an adaptive load manager. The experiments show that SEaMLeSS outperforms the state-of-the-art with respect to the requirements for a practical load manager: being able to honor the delay target, applicable to all types of query networks, and not assuming a manually-tuned headroom factor.

[4]For a fair performance comparison, in all the experiments we also force AQSIOS operating under other load shedders to read the file and do the same calculation, although Aurora and ALoMa does not use that information.

We also pointed our some limitations of SEaMLeSS, which motivated us to propose ALoMa. ALoMa, same as SEaMLeSS, satisfies the three desired properties of a practical load manager, but overcomes SEaMLeSS's limitation.

# 5.0 ALOMA

In this chapter we present ALoMa (**A**daptive **Lo**ad **Ma**nager) [66], our second adaptive load manager. We start with the general idea of ALoMa, then go into details of the algorithms. We present the experimental evaluation and conclude with a summary of ALoMa's properties compared to SEaMLeSS and the state-of-the-art load shedders.

## 5.1 OVERVIEW

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 [74] (i.e., Eq. 3.1 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 load that needs to be removed or can be added to the system. Next we explain the intuition behind ALoMa's decisions.

## 5.2 IMPLEMENTATION

### 5.2.1 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 (Figure 13A) in response to step changes of the input rate (Figure 13B). All experiments were carried out on AQSIOS, our experimental DSMS prototype described in Section 2. Note that the Y-axis in Figure 13A is in log-scale. The input rate starts from 5,000 tuples/s and increases by 5,000 tuples/s after every 20 seconds.

From time $t = 0s$ to $t = 20s$ the response time remains at around 120$\mu$s. One can think that this 120$\mu$s reflects the processing cost per tuple and that the system will be overloaded with an input rate greater than 1 tuple/150$\mu$s (about 8300 tuples/s). However, we can observe that during the next 20 seconds 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 second period. This trend continues in all of the other 20-second periods before $t = 120s$. 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 14 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.

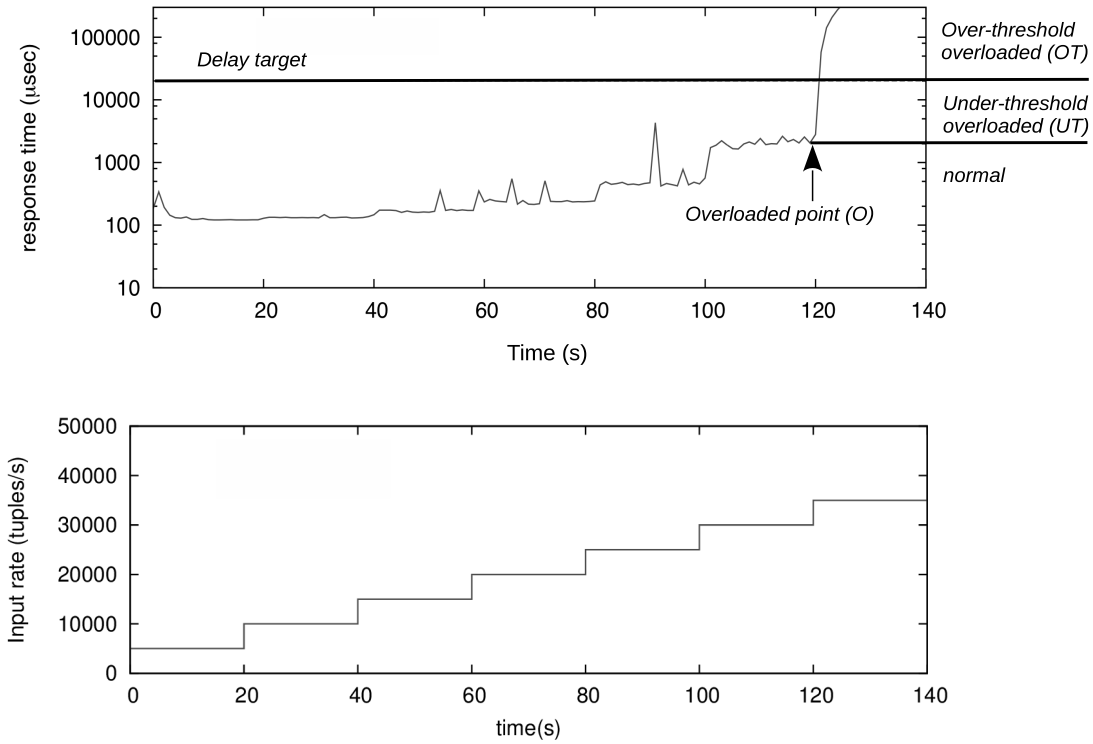


Figure 13: Response time (top plot) with increasing input rate (bottom plot) and its imply on system's load state.

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 Figure 13B, the corresponding response time in Figure 13A 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 Figure 13A) 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 $120^{th}$ second in Figure 13). Based on the above observation, we can map the response time to the

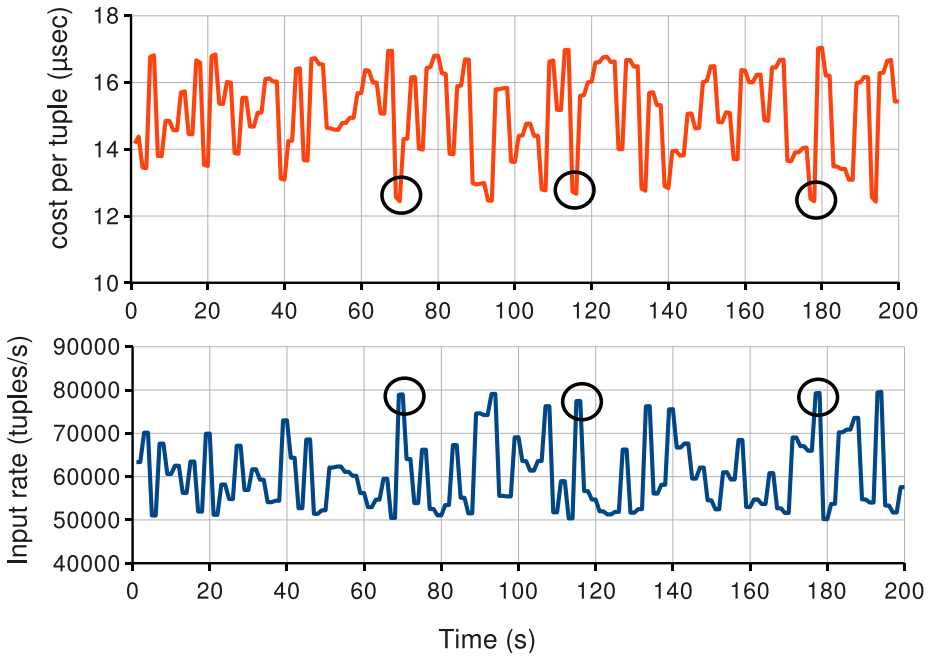


Figure 14: Cost fluctuation in response to changes of input rate, measured on the AQSIOS system.

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 Section 5.2.3 how we find the O point in practice.

### 5.2.2 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. 5.1. 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.

$$L_{C_{new}} = L_C \pm \frac{\log_2(z+1)}{z}|L - L_C| \tag{5.1}$$

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

### 5.2.3 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).

**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. 5.1 (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. 5.2:

$$x = 1 + \log_2(k) \tag{5.2}$$

**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. 5.1 (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. 5.2.

- 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 $20^{th}$ second in Figure 13, 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.

### 5.2.4 Overhead and worst case

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

**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 ≤ previous_response_time) then
11:         Increase L_C
12:         Reduce shed amount by x%
13:      end if
14:   end if
15: else {L ≤ L_C}
16:   if state = OT then
17:      if (response_time ≥ 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
```

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.

## 5.3 EXPERIMENTAL EVALUATION

### 5.3.1 Experiment settings

We evaluated ALoMa in AQSIOS along the same lines as SEaMLeSS (Section 4.3). We use ALoMa to realize our DILoS framework due to its flexibility, so we perform a more extensive evaluation of it. Again, all experiments were run 5 times and we report the averages.

**Query networks**: We use the query networks *QN-flat* and *QN-complex* (Section 4.3.1), which are also used for SEaMLeSS's evaluation. In addition, we use another query network named *QN-long*, which contains long queries (i.e., queries having many operators). A repre-

sentative query in this network is presented in CQL syntax [19] below[1], 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**: Besides reusing the two streams $S_c$ and $S_r$ (Section 4.3.1), we use an additional stream called $S_{step}$. $S_{step}$ has an initial constant input rate of 200 tuples/s for the first 10 seconds, then goes up to a higher level every 40 seconds until the system is so overloaded

[1]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.

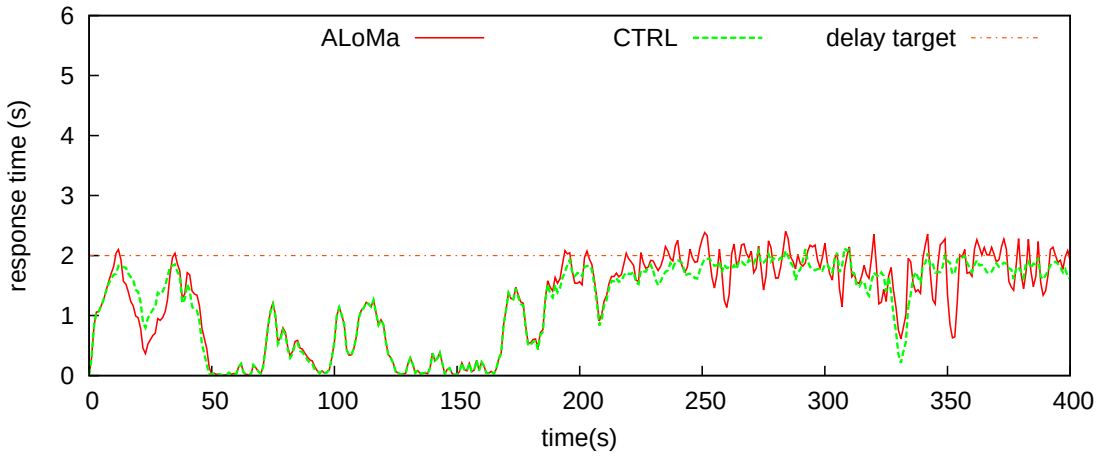


Figure 15: Response times with QN-flat and $S_r$.

that load shedding can no longer control the response time. We use this input to test a worst case situation.

**Parameters:** We use the same parameters as in the experimental evaluation of SEaMLeSS (Section 4.3.1). We set the initial value of the headroom factor for ALoMa to be 0.8.

### 5.3.2 Experiment results

**5.3.2.1 ALoMa vs CTRL under CTRL's ideal setting** 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 $S_r$ is used for the experiment. We run ALoMa under the same setting, but *without the manual tuning of the headroom factor*.

Figure 15 shows the response time of the output under ALoMa and CTRL. Table 4 summarizes the average delay violation, the maximum violation observed, and the data loss under each scheme. ALoMa has higher maximum violation, and from Figure 15 we can observe that the response time fluctuates more under ALoMa than under CTRL. This,

Table 4: Average delay and data loss, with QN-flat and $S_r$ for CTRL with optimal, manually-tuned headroom factor.

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

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 (0.05s 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.*

**5.3.2.2 ALoMa vs CTRL under system environment changes** As shown in Section 4.3.3.1, a specific value of the headroom factor is not guaranteed to be correct for the whole execution time. In this experiment we repeat the setup in Section 4.3.3.1 with two background jobs launched while the DSMS is running. In order to clearly show the effect of the system environment change, we again use the input $S_c$ with constant input rate.

Figure 16 shows that ALoMa 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, Figure 16 also shows the headroom factor adjustment made by ALoMa in response to the change in the system environment. Figure 16 again shows that CTRL, which uses a fixed, manually-tuned headroom factor can no longer control the response time to the delay target when the background jobs are launched and share the processor with the DSMS at the time $t = 100s$.

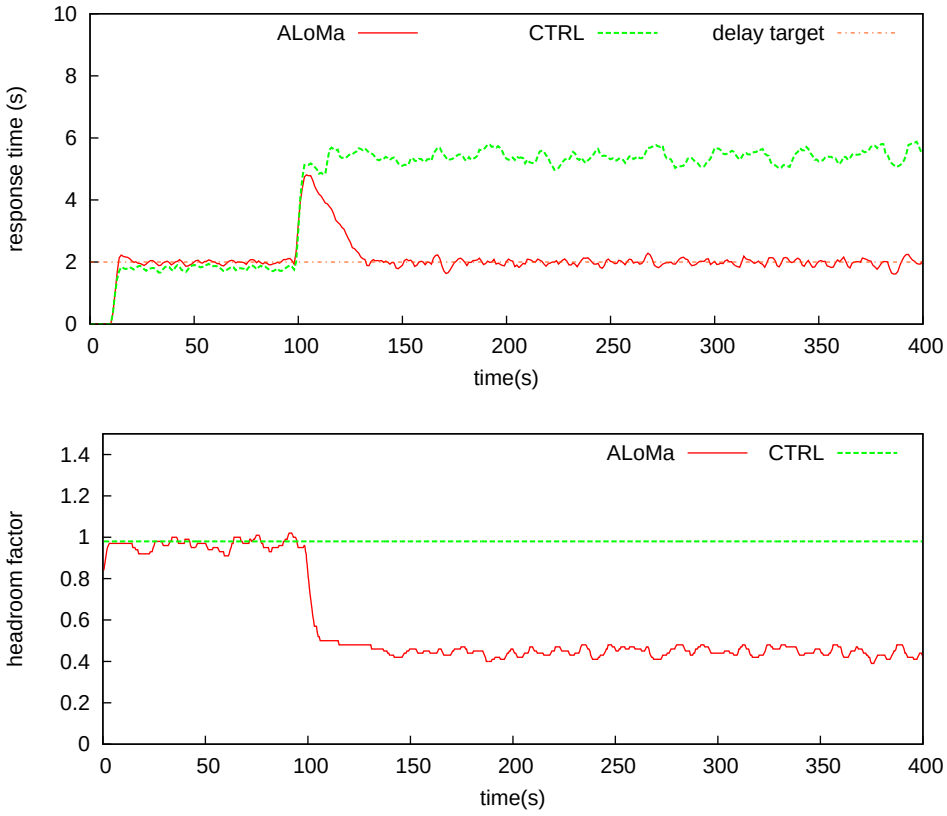


Figure 16: Effect of environment changes on CTRL and adaptation of ALoMa. Top plot shows the response time, bottom plot shows the headroom factor recognized by each scheme. Total data loss for ALoMa and CTRL is 62.98% and 62.69%.

#### 5.3.2.3 ALoMa vs CTRL and Aurora with a complex query network

With the same experiment setup as that in Section 4.3.3.2, we show ALoMa's performance compared with CTRL and Aurora using a complex query network QN-complex and real input data $S_r$. We observe similar result: no value of the headroom factor could enable Aurora to perform equivalently to ALoMa, while CTRL's performance is completely off as it's delay estimation model does not work with complex query network (in this case, the query network contains shared operators). ALoMa, while not required any pre-tuned headroom factor, performs well in controlling the response time to the delay target and minimizing the data loss.

Table 5: Delays and data loss with QN-complex and $S_r$.

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

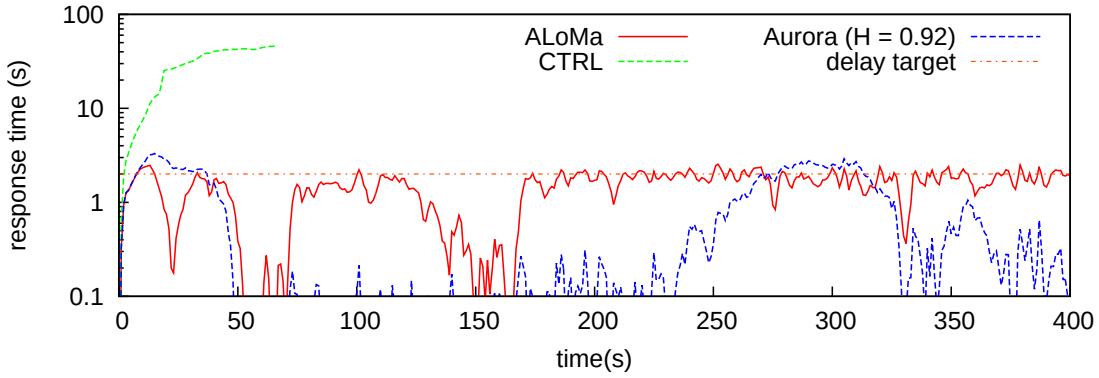


Figure 17: Response times with QN-complex and $S_r$. Note that the X-axis plots the input timestamps, showing that within the specified experiment time the system under CTRL was only able to process tuples coming in the first 66 seconds.

**5.3.2.4 ALoMa vs SEaMLeSS under a priority-based scheduler** In this section, we demonstrate our remarks in Section 4.4 that the delay estimation model is dependent on the specific operator scheduling policy and will not work appropriately under an unfair scheduler.

We used the QN-complex query network for this experiment. We implemented in AQSIOS a simple weighted Round Robin scheduler which, in each cycle, gives half of the queries (i.e., the operators in these queries) a scheduling time quota of 50% bigger than that for the other half. We call the two halves the high-priority and the low-priority, respectively.

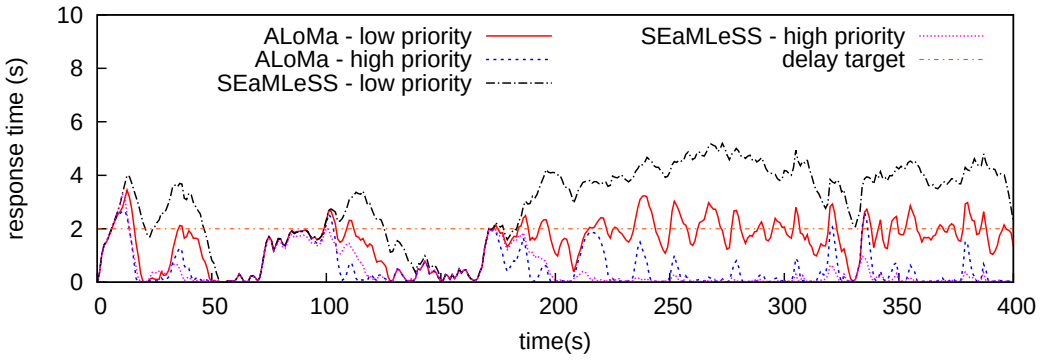


Figure 18: ALoMa vs SEaMLeSS under weighted RR scheduler.

Table 6: Average data loss and delay violation of ALoMa and SEaMLeSS under a weighted RR scheduler, with $S_c$.

| | Delay violation-low priority | Delay violation - high priority | Data loss |
|---|---|---|---|
| ALoMa | 0.16 sec | 0 sec | 33.92% |
| SEaMLeSS | 2.12 sec | 0 sec | 32.80% |

For both schemes, we show the detailed response time and delay violation separately for the low-priority and high-priority queries in Figure 18 and Table 6. While ALoMa manages to control the worst-case response time (i.e., response time of the low-priority queries when overloaded) to be around the delay target, SEaMLeSS fails to do that for the low-priority queries. This is because the delay estimation model does not incorporate any information about the priority and assumes that the processing cost and the length of the queues are the only factors that determine the output delay. In this case, since the higher-priority queries can consume their tuples faster, an incoming tuple that goes through a high priority path will have a significantly less waiting time. ALoMa, although still oblivious to the priority of the queries, still is able to handle this situation because it is monitoring the response time directly and therefore is independent of the scheduling details.

Our experimental results do not suggest any clear relationship between the relative priorities and the relative response times of the queries. In addition, we observe that high-priority

queries usually do not exploit batch processing as much as the lower-priority ones, resulting in higher processing cost. Therefore, incorporating scheduling priority to the delay estimation model is clearly not trivial, even for the simple priority-based scheduler as the one we used in this experiment.

**5.3.2.5 ALoMa vs CRTL and Aurora with long queries** 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 $S_r$ 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 Figure 7. 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 19 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 1.08s. 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 Figure 19.

**5.3.2.6 Worst-case scenarios** In this set of experiments we illustrate the worst case scenarios explained in Section 5.4. We use query network QN-flat, so that CTRL is applicable.

In the first setup, we use the input $S_{step}$ 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

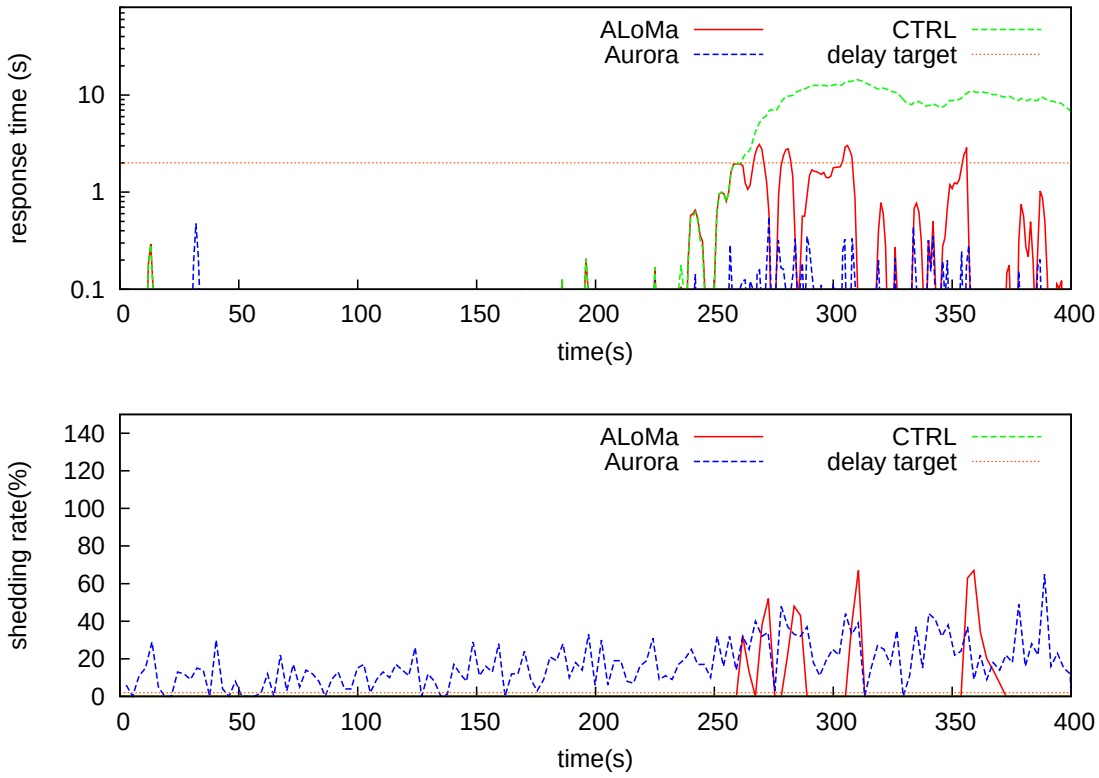


Figure 19: Performance of ALoMa, CTRL and Aurora with QN-long and $S_r$. Top plot is the response time and bottom plot is the shedding rate.

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 20 (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

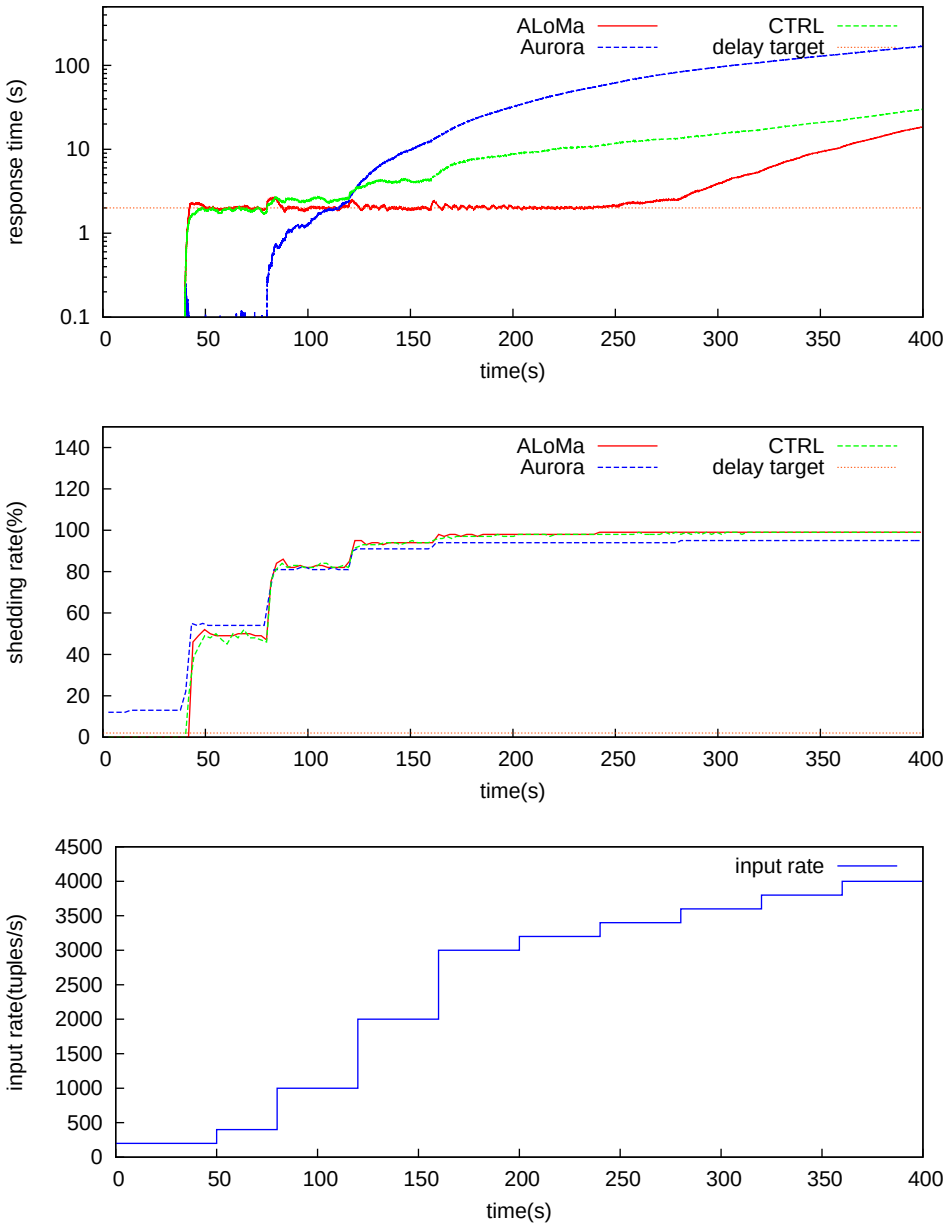


Figure 20: Performance of ALoMa, CTRL and Aurora with workload increasing to worst case situation. Top plot is the response time, middle plot is the shedding rate and bottom plot is the input rate.

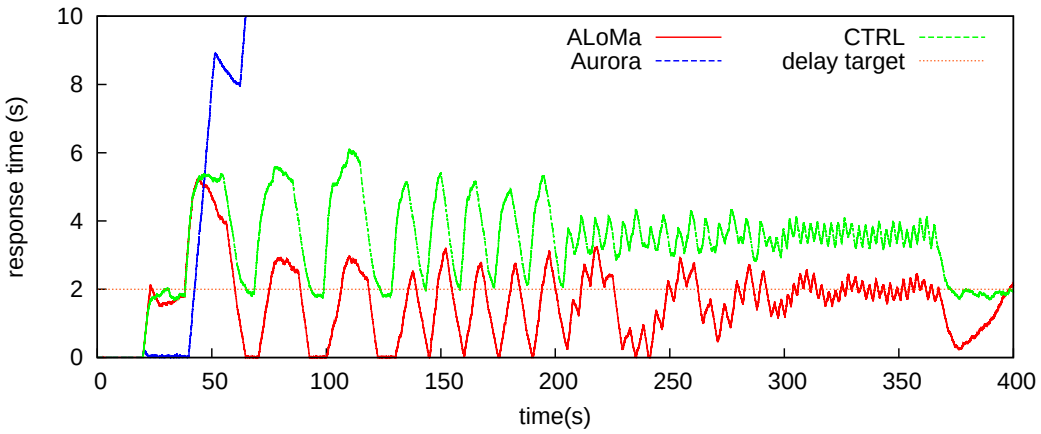


Figure 21: Response time under ALoMa, CTRL and Aurora with background job coming and leaving at different frequencies.

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 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 $S_c$ as in the experiment in Section 5.3.2.2. After the experiment has run for the first 10 seconds, we kick off a background job which stays for 10 seconds, then leaves for 10 seconds and comes back for another 10 seconds. The pattern is repeated for about 60 seconds, then switches to a pattern of 5-second stay and 5-second leave for another 60 seconds, then 60 seconds of 2-second stay and 2-second leave and finally 60 seconds 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 21 shows the response time under all three schemes.

We can see that there are points at which the response time under ALoMa drops close

Table 7: ALoMa's properties compared to the state-of-the-art.

| | ALoMa [VLDBJ'16] | SEaMLeSS [SMDB'13] | 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 | √ | | √ | |

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.

## 5.4 SUMMARY

As confirmed 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 SEaMLeSS: ALoMa does not assume the fairness of the operator scheduler while SEaMLeSS does. With respect to implementation, ALoMa also provides more flexibility as it does not require a separate, not-overloaded thread to count the number of tuples in the input queues of the CQs.

We summarize in Table 7 the properties of ALoMa compared to SEaMLeSS and the two state-of-the-art approaches, i.e., CTRL and Aurora.

With ALoMa, the realization of DILoS is now feasible. We present in the next chapter an implementation of DILoS using ALoMa.

# 6.0 DILOS IMPLEMENTATION AND EVALUATION

In this chapter, we present an implementation of DILoS (**D**ynamic **I**ntegrated **L**oad Manager and **S**cheduler). This implementation of DILoS uses ALoMa, our proposed adaptive load manager. We also describe the experimental results evaluating DILoS performance and discuss the extensibility of DILoS.

## 6.1 DILOS IMPLEMENTATION

Recall that in Chapter 3 (Section 3.1), we proposed the DILoS framework in which we separate the scheduler into two levels: class-level and operator-level scheduling. Each class is effectively a virtual DSMS and has its own load manager instance. The system capacity of each class is adjusted periodically by the class-level scheduler based on the priority and the capacity usage of each class. For our specific implementation in this dissertation, we discuss about the specific two-level scheduler, the per-class load manager, and the capacity redistribution policy.

### 6.1.1 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

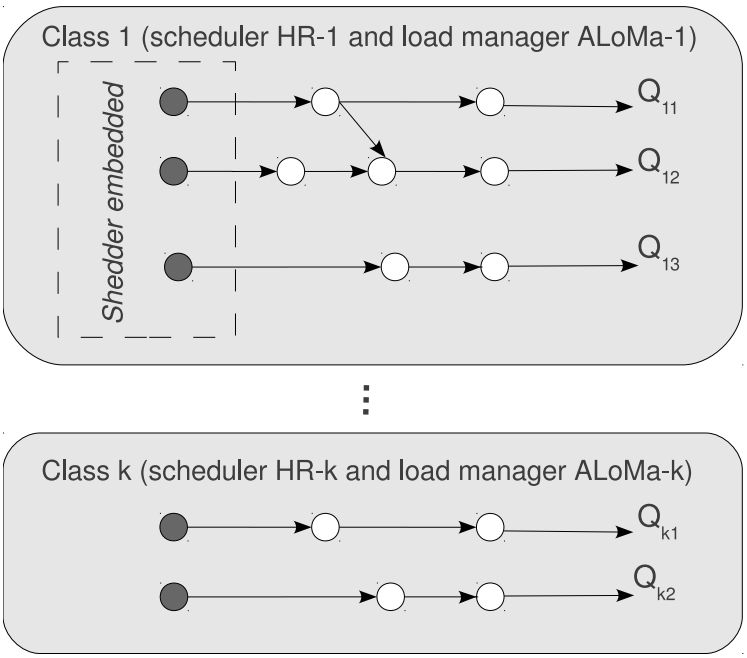


Figure 22: Per-class load management with ALoMa without inter-class sharing.

sheds this excess load from the class by specifying the calculated shedding rate uniformly across the source operators of the class. Figure 22 illustrates this implementation, in which the dark operators are the source operators with a load shedder embedded.

### 6.1.2 Scheduler

In this implementation, we use a two-level, class-based DSMS scheduler proposed in [58], called CQC. As indicated in Section 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.

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) [71] schedulers. Each modified HR scheduler is in charge of the set of operators that be-

long 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 [58].

### 6.1.3 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:

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

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

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 $budget = \sum_k supply_k$, 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 $L_{C_k}^{new}$, is the expected capacity portion of the class in the next period. As such, the scheduler calculates the time quota $T_k^{new}$ for the next period as $T_k^{new} = \frac{L_{C_k}^{new}}{L_{C\,k}} \times T_k$. 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 in Eq. 6.1. This new value set by the scheduler does not need to be perfectly accurate because the load manager is able to automatically adjust it.

$$H_k^{new} = \frac{T_k^{new}}{T_k} \times H_k \tag{6.1}$$

### 6.1.4 Handling inter-class sharing

In Section 3.2, we have explained the congestion problem that exists with any class-based scheduler. We have also proved that, with an appropriate load manager such as ALoMa being the per-class load manager, which can control the response time of the class, DILoS inherently solves the congestion problem that exists with any class-based scheduler, allowing inter-class sharing for a more optimized query network.

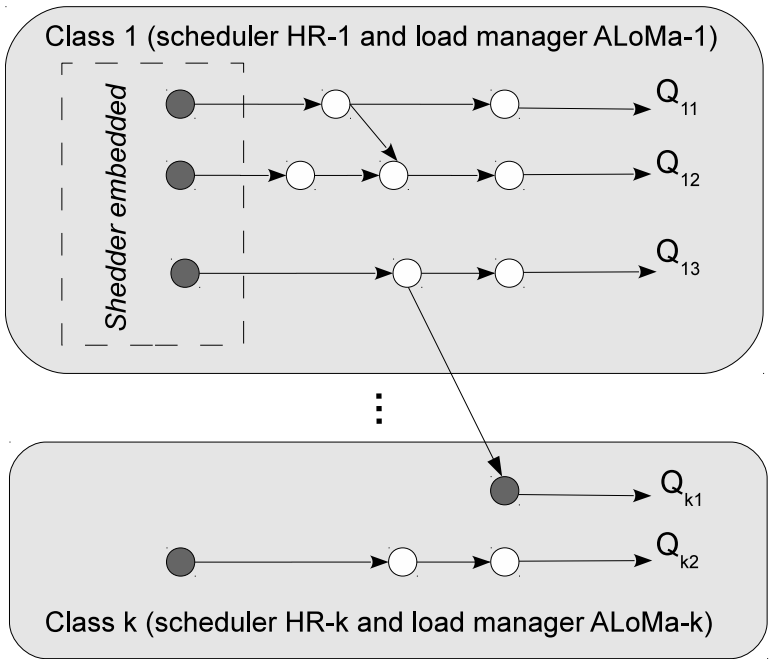


Figure 23: Per-class load manager, with class 1 (high priority) sharing a segment with class k (lower priority).

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, we embed load shedding into the source operators, which means this operator also has a shedder embedded. Figure 23 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.

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 (e.g., [74, 31]) 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).

### 6.1.5 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 Section 5.4, a typical DSMS system needs to collect these statistics for a variety of purposes such as optimization and scheduling. 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 disseration, the redistributing requires one pass to compute $demand_i$ and $supply_i$, and another pass to distribute the total budget. 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).

## 6.2 EVALUATION

In this section, we first describe our experimental settings, and then discuss the experiment results showing the advantages and robustness of DILoS.

### 6.2.1 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 300ms.
  - Class 2: Priority 3 (second highest), with delay target 400ms.
  - Class 3: Priority 1 (lowest), with delay target 500ms.

  By assigning priorities 6, 3 and 1 to classes 1, 2 and 3, respectively, the CQC scheduler (Section 6.1.2) will allocate 60%, 30% and 10% of capacity to the corresponding classes. 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 [16].

- **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:

- **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

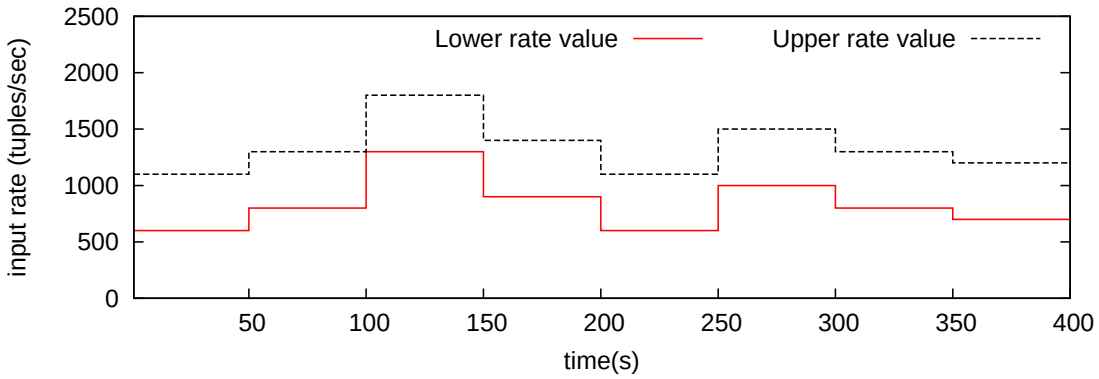


Figure 24: Input rate ranges for class 1 - input setup $SD_p$.

slightly higher than the total system capacity. The simple pattern of this input allows us to easily analyze the behavior of each scheme.

- **SD$_\mathbf{p}$**: 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 (Figure 24 sketches the changes of the range) in order to vary the amount of 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).

- **SD$_\mathbf{r}$**: 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 $S_r$ (Figure 7).

**Parameters:** For all experiments, we set 150ms to be the *load management cycle*. In [76], 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., 1.5s). We report the sensitivity analysis on the length of this capacity redistribution cycle in Section 6.2.4.

All experiments were run 5 times and we report the averages.

### 6.2.2 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 8 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 (117,132.74ms) exceeds its delay target (500ms) 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 8)[1]. Given 13 stream sources used by class 3, each with the input rate of 950 tuples/s, this

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

Table 8: 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 | 117132.74 | 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 |

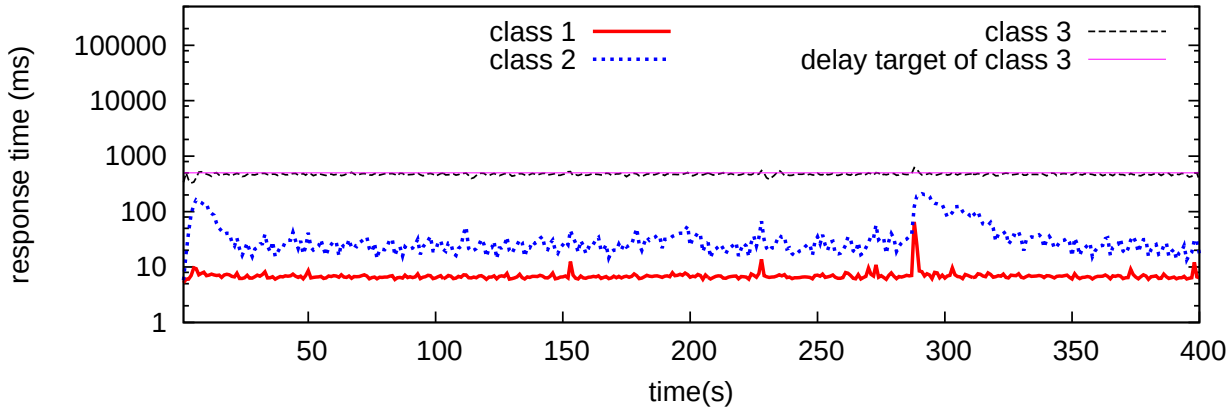


Figure 25: Response times with $SD_c$, QN-A, DILoS, and inter-class sharing.

decrease in data loss means approximately 7,526 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%)[2], while the performance of the higher-priority class 1 is not affected by the lower-

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

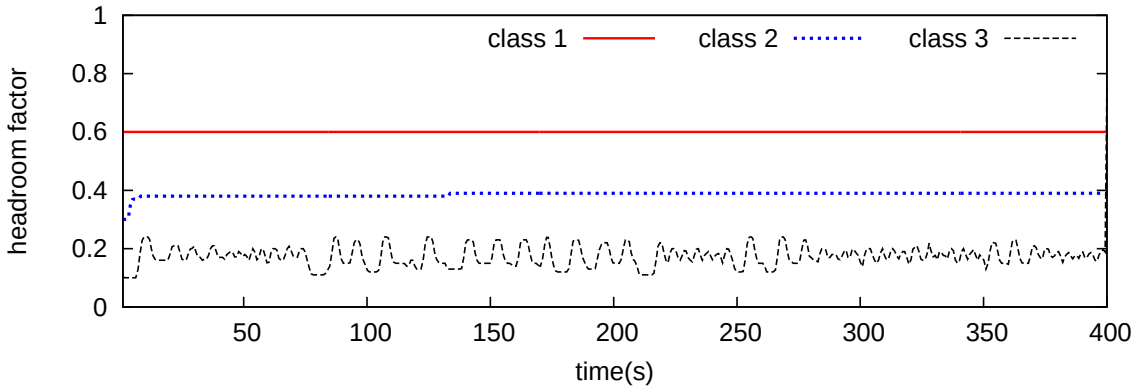


Figure 26: Headroom factor estimated, with $SD_c$, QN-A, and one ALoMa instance per class.

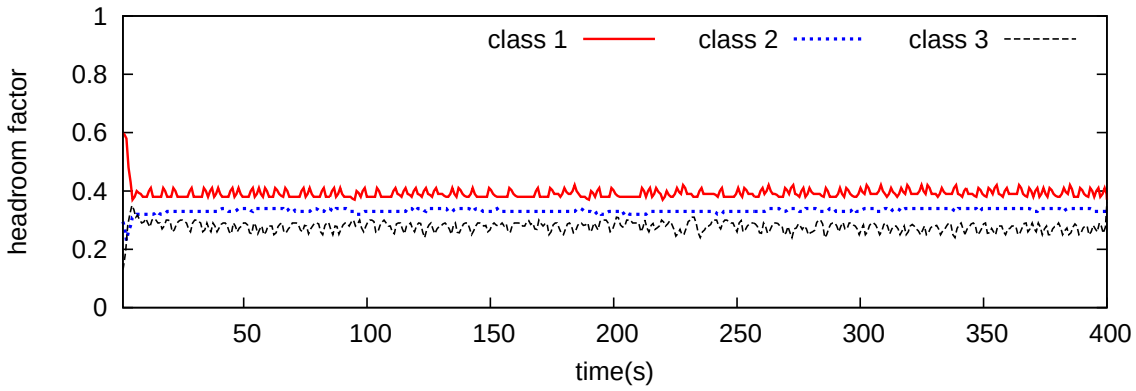


Figure 27: Headroom factor estimated, with $SD_c$, QN-A, and DILoS' full synergy.

priority class 3. Figure 25 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 26 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 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, were 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[3]. Thus, CQC by itself already allows implicit capacity sharing, and the system capacity seems to have been used fully.

However, Figure 27 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.

We can observe that the response time of class 1 and 2 increase. 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.

[3]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.

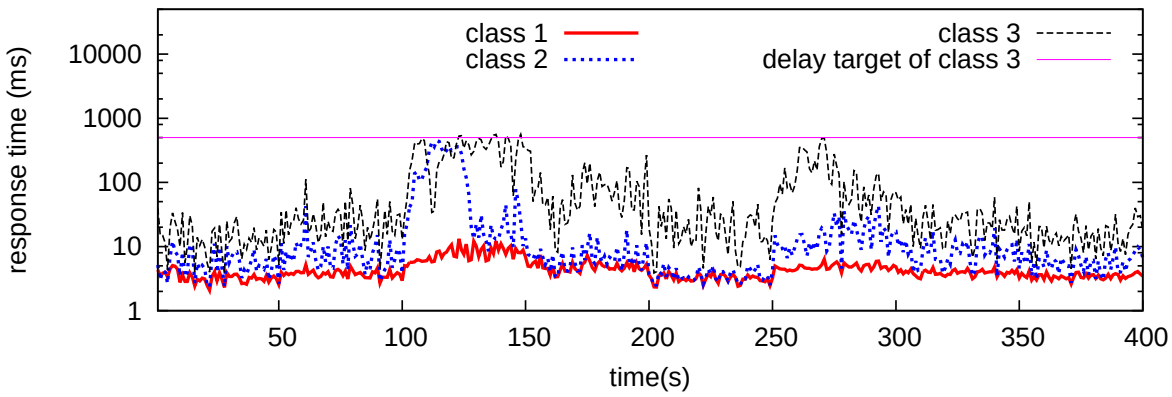


Figure 28: Response times with $SD_p$, QN-A, and DILoS (with sharing).

### 6.2.3 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.

**6.2.3.1 QN-A and $SD_p$** This set of experiments simulate situations where the load level of class 1 (the highest priority) changes dramatically after a certain period, aiming to test if 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 Figure 28. In Figure 29 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.

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.

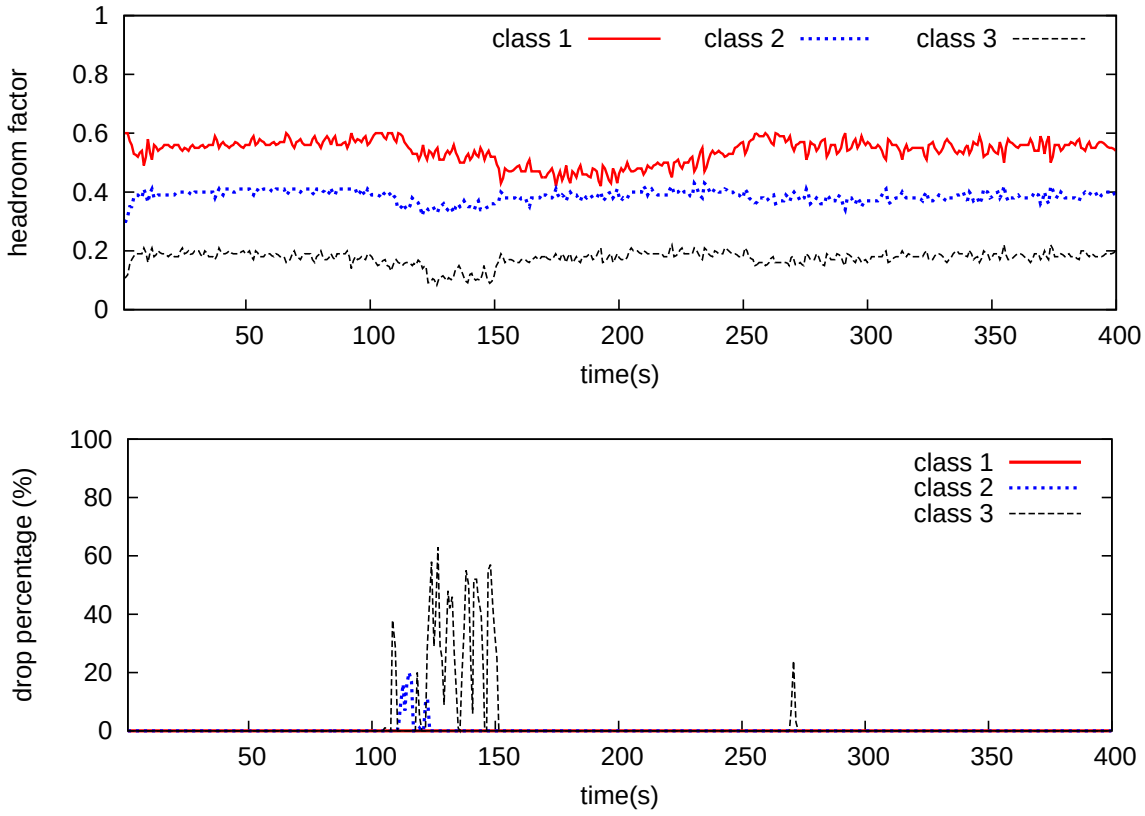


Figure 29: Estimated headroom factors (top) and shedding rates (bottom), with $SD_p$, QN-A, and DILoS (with sharing).

However, as soon as the load of class 1 increases (e.g., at time $t = 100s$), 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 Table 9 and 10 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

Table 9: 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 10: 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 |

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.

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 confirms that with appropriate load management the sharing does not affect the QoS and QoD of the higher priority class.

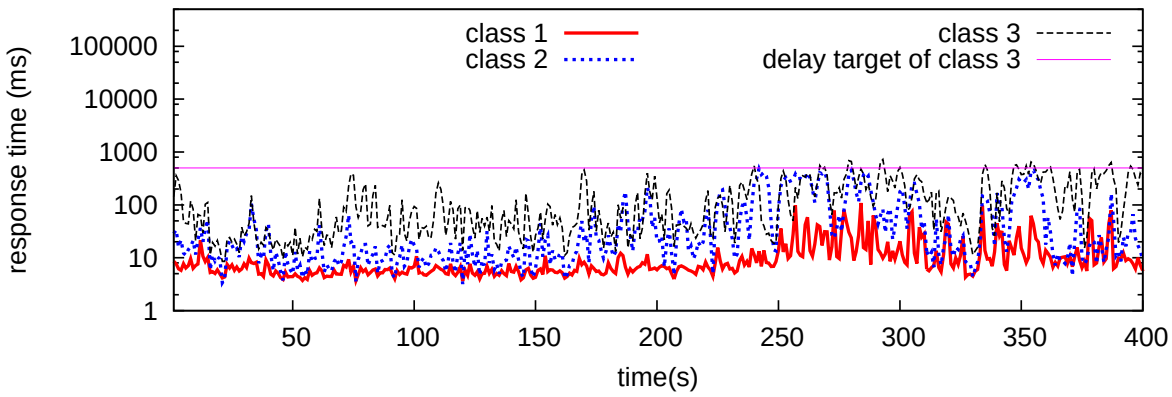


Figure 30: Response times with $SD_r$, QN-B, and DILoS (with sharing).

**6.2.3.2 QN-B and $SD_r$** In this set of experiments we replace the synthetic input rate pattern by $SD_r$ with the real trace for class 1 (Figure 7). 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 Figure 30. 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 Figure 31 (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 time $t = 250s$ to $t = 300s$), 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 seconds, 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 Section 6.2.2. The load manager recognizes the correct headroom factor when the load of some of the classes reaches their capacities and the explicit redistribution happens, which is the case during the high-load period (after the $250^{th}$ second).

Tables 11 and 12 compare the average response time and data loss for all cases. In

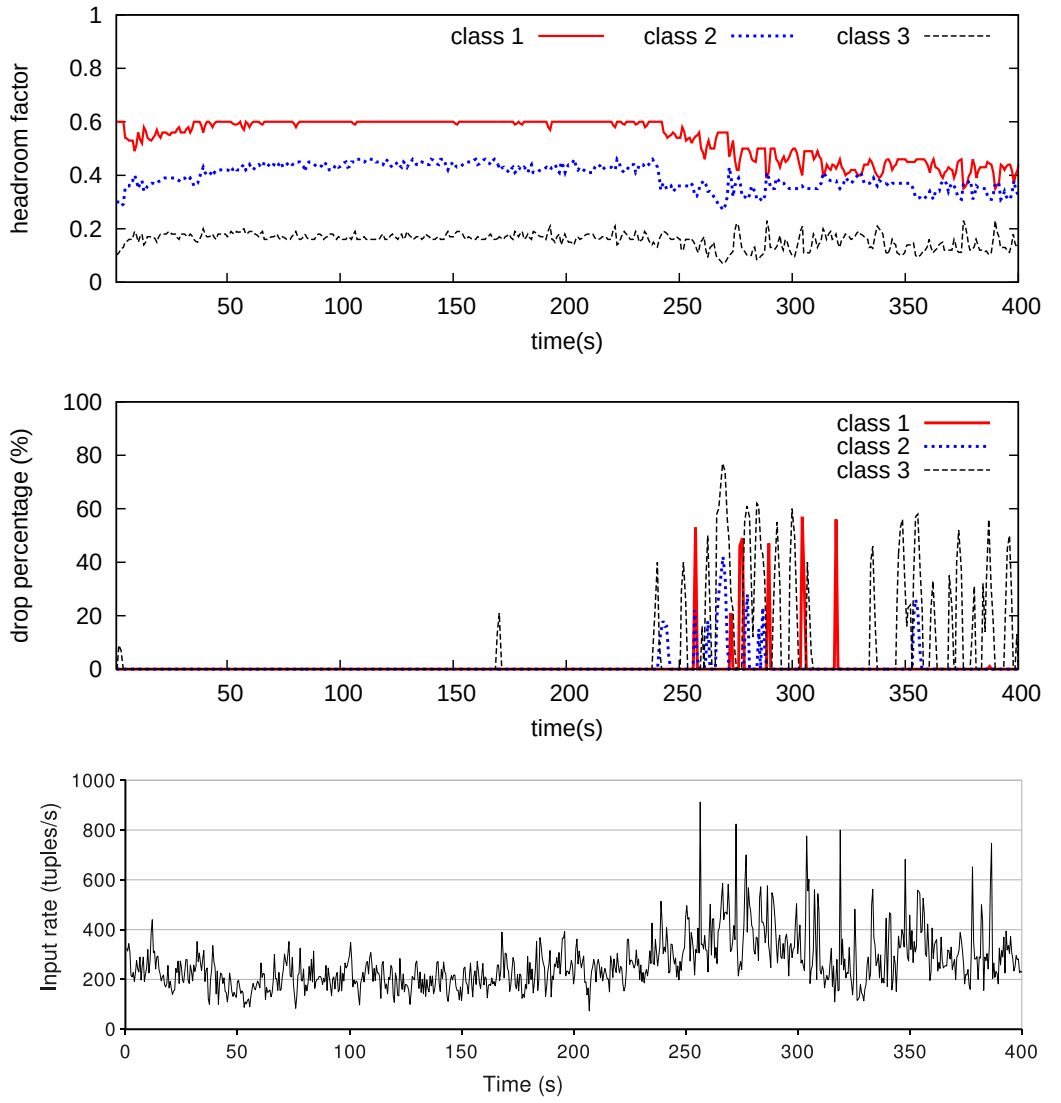


Figure 31: Estimated headroom factors (top) and shedding rates (middle) in response to the input rate of class 1 (bottom), with $SD_r$, QN-A, and DILoS (with sharing).

this experiment, while synergy still brings significant benefit in terms of exploiting system capacity (much more data is 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 Figure 31, 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

Table 11: 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 12: 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 |

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.

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.

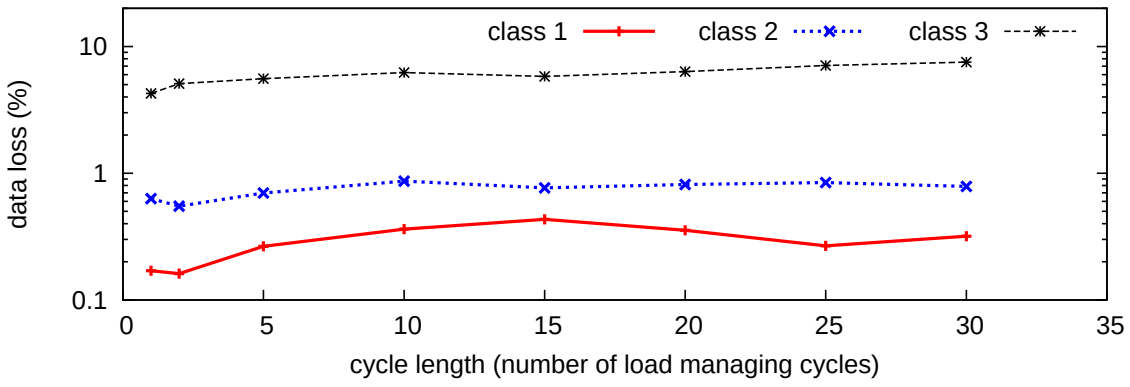


Figure 32: Data loss at different lengths of the capacity redistribution cycles.

### 6.2.4 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 Figure 32 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) are smallest when CRC is equal to 1 or 2 load management cycles (i.e., 150ms - 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 Section 6.2.1, for all the above experiments we let the scheduler consider redistributing the system capacity after every 10 load management cycles (i.e., 1.5s). 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.

## 6.3 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.

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. Recall that 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., [30, 36, 74]) and determining where in the query network to shed data to minimize semantic loss (e.g., [74, 21]), 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.

## 6.4 SUMMARY

In this chapter we presented an implementation of the DILoS framework using CQC and ALoMa, our new adaptive load manager. Through experimental results of this implementation, we confirmed the significant benefit of DILoS, which exploits the synergy between the scheduler and load managers and supports the hypothesis of this dissertation. We also discussed the extensibility of DILoS, making it clear that DILoS is a general framework, which can integrate many different schedulers, load shedders, as well as different priority-based capacity redistribution policies.

# 7.0 LARGE-SCALE ADAPTIVE RESOURCE MANAGEMENT USING DILOS

Modern cloud infrastructure makes possible a further solution for the overloading problem in DSMSs: a system can scale-up under heavy load, and scale-down during idle periods. Note that this solution does not replace load shedding entirely: load shedding is still necessary to deal with mild, short-term overloading at each node in a DSMS cluster.

In this chapter, we outline our conceptual framework named ARMaDILoS (**A**daptive **R**esource **Ma**nagement using **DILoS**), which aims at providing a priority-based resource management model for DSMS deployed on a multi-node cluster. We then present UniMiCo (**Uni**nterruptible **Mi**gration of **Co**ntinuous Queries), a protocol that allows smooth migration of a CQ from one node to another, which is a key step in implementing ARMaDILoS. We conclude with a discussion on technical questions that need to be addressed in our future work to fully realizing ARMaDILoS.

## 7.1 ARMADILOS

ARMaDILoS is formed around our hypothesis that when deployed on a cloud-based, multiple-node infrastructure, the DILoS framework can supports a global workload management that considers priority-based capacity distribution across the whole system.

We assume a system consisting of multiple shared-nothing nodes, connected by reliable, high-speed network. We propose a system model in which one node serves the role of the coordinator, while the other are peers and each one of them runs one instance of a DSMS,

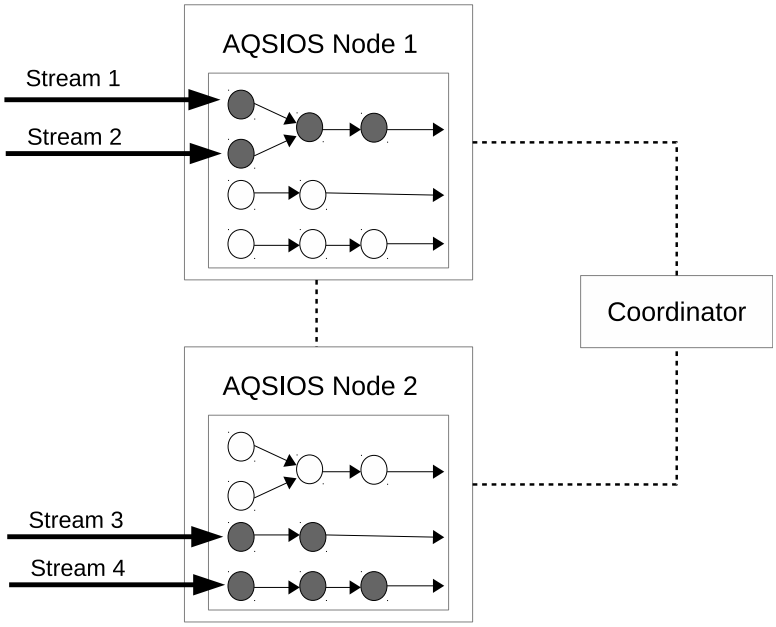


Figure 33: ARMaDILoS system model

such as AQSIOS, our standalone DSMS with necessary extension to support communication with the coordinator and with the other peers.

CQs are registered with the coordinator which optimizes them into a query network. Each AQSIOS node keeps a copy of the whole query network, but, only a subset of it is active on the node. A node only connects to the stream sources that are necessary for the active queries in the node. Data streams, coming from (possibly) different sources, are received by the source operators, which are the most upstream operators in a CQ.

Figure 33 is an example of our system model with two AQSIOS nodes. The CQs comprised of dark operators are those active at the node. The dash lines represent network connection among the nodes.

The AQSIOS instance on each node runs DILoS as a local workload mamagement unit. Periodically, each DILoS instance reports to the coordinator the workload of the node of which it is in charge. The report contains information about the load of each CQ, as well as how the local system capacity is distributed for each priority class. The coordinator implements a priority-based global workload management policy which, based on the workload

reports from the AQSIOS nodes, decides when to move a subset of CQs from one node to another. The coordinator can also ask a DILoS instance to adjust the local priorities of a CQ class. The coordinator's policy aims at providing global priorities of the CQ classes and maximizing total resource usage.

In such a framework, it is crucial to have a lightweight protocol to support the migration of workload (in the forms of operators, query segments or queries) from one node to another in the cluster. The protocol should not cause any interruption in the execution of the CQ and should have negligible cost, otherwise it can make matters worse and prolong an overloaded situation.

The implementation of ARMaDILoS is beyond the scope of this dissertation. However, because ARMaDILoS cannot be realized without an efficient CQ migration protocol, we proposed and implemented UniMiCo, an interruptible migration protocol for CQs, as part of a proof of concept for the feasibility of ARMaDILoS. We present UniMiCo in the next section.

## 7.2 UNIMICO

UniMiCo is our migration protocol implemented as a first step toward the above large-scale adaptive resource management. UniMiCo has the ability to (i) migrate stateful CQs without the need to transfer any state, (ii) do the previous in a "live" fashion (i.e. no downtime). UniMiCo's protocol has been designed in a general way to handle both time-based and tuple-based window.

In this dissertation we consider the whole query as the migration unit. However, the protocol can also be used to migrate only a segment of a CQ: the operator(s) right before the migrated segment becomes the stream source(s) for that segment. and their downstream operators act as source(s) in the corresponding CQs. We assume that there is no operator sharing between the query to be migrated and the rest of the query network. However, in the present of operator sharing, the first step is to decouple the migrating CQ from any shared operators and treat it as an independent CQ.

Below, for ease of exposition, we first present a background on window-based operators in CQs, then describe the basic idea of UniMiCo followed by its details. We also present some preliminary results showing the correctness and efficiency of the protocol.

### 7.2.1 Window-based operators

There are two types of operators in a CQ: *stateless* and *stateful* operators. A stateless operator, such as selection ($\sigma$), produces an output tuple based solely on the current input tuple. Conversely, a stateful operator, such as join or aggregation, needs to refer to values from previous input tuples. Due to the fact that input streams are infinite, DSMSs use either *tumbling* or *sliding windows*, to limit the state of operators. Sliding windows allow the output to be continuously computed based on the most recent "portion" of the stream data. In addition, a sliding window is specified through a length (or range) $l$, and a slide $s$, which can be either time interval or tuple count. These two types of windows are called *time-based* and *tuple-based windows*, respectively [19].

While most DSMSs embed the window definition into the corresponding stateful operator, some systems treat it as a separate operator (e.g., [19]). In this paper, when the semantic of the stateful operator is not important, we refer only to the window aspect of it as if the window is a separate operator. UniMiCo works the same way no matter whether the window operator is physically merged to the corresponding aggregate/join operator or not.

### 7.2.2 Overview of UniMiCo

The key goal of UniMiCo is to avoid transferring state during the migration of a CQ containing stateful operators. To achieve this, UniMiCo migrates a CQ at a window boundary, meaning that the originating node continues processing until it completes the last in-progress window, while the target node starts processing from the first tuple of the next window. Given that two consecutive sliding windows overlap, the tuples belonging to the overlap of the two windows are processed by both the originating and the target nodes. This way, the state of the operator is reconstructed at the target node so there is no need to migrate it.

We illustrate this strategy in Figure 34. In this example, the sliding window of a stateful

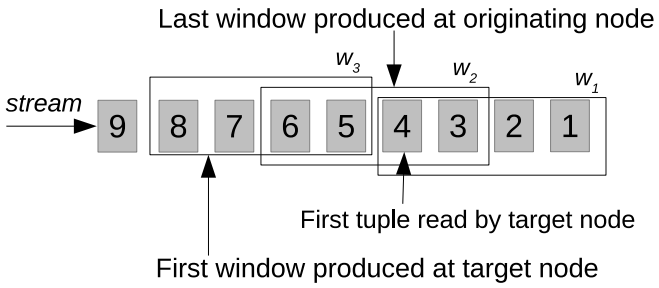


Figure 34: UniMiCo's migration strategy

operator (e.g., aggregate) has a size of 4 seconds and a slide of 2 seconds, with input rate 1 tuple/second. The number in each stream tuple is its timestamp, which is assumed to monotonically increase over time (i.e. *in-order* processing of tuples). By the time the migration process starts, the most recent window produced is $w_1$, whose start timestamp is 1. In addition, the first tuple received by the target node after it connects to the stream has a timestamp of 4. UniMiCo determines that (1) the originating node will continue processing until $w_2$ expires, which happens to be the last window with start timestamp less than 4, and (2) the corresponding CQ at the target node will start processing tuples with timestamp greater or equal to 5 ($w_3$).

### 7.2.3 Migration timestamp

The migration timestamp marks a CQ hand-off from the originating to the target node. It is used to synchronize the stop of the last window at the originating node and the start of the next window at the target node.

**Definition 12.** The *migration timestamp* is the start timestamp of the last window to be processed at the originating node.

In the example in Figure 34, the start timestamp of $w_2$, which is 3, is the migration timestamp.

### 7.2.4 Calculating the migration timestamp

The exact calculation of the migration timestamp depends on the implementation details of the window operation. In this section we present how to calculate the migration timestamp on both time-based and tuple-based cases. In all the equations below, $s$ denotes the slide of the window.

**Time-based, single-input window:** Assuming a time-based window of length $l$ and slide $s$, let $ts_{start}$ denote the timestamp of the first input tuple the stream source at target node was able to read after connecting to the stream. Furthermore, $ts_{last\_w}$ is the timestamp of the most recent window processed. The migration timestamp, denoted $ts_{mi}$ is calculated as follows (note that now $s$ is in number of tuples):

$$\begin{aligned} ts_{mi} &= \begin{cases} ts_{last\_w} & \text{if } ts_{start} \leq ts_{last\_w} \\ ts_{start} - \delta \text{ otherwise} \end{cases} \\ \text{where } \delta &= \begin{cases} s \quad \text{if } (ts_{start} - ts_{last\_w})\%s = 0 \\ (ts_{start} - ts_{last\_w})\%s \text{ otherwise} \end{cases} \end{aligned} \tag{7.1}$$

**Tuple-based, single-input window:** For tuple-based windows, the calculation is the same in the case when $ts_{start} \leq ts_{last\_w}$. When $ts_{start} > ts_{last\_w}$, UniMiCo needs to wait until a tuple $t$ comes to the window operator, whose timestamp is equal or greater than $ts_{start}$. This way, UniMiCo is aware of the number of tuples with timestamps between $ts_{last\_w}$ and $ts_{start}$ (let that number be $N$). The migration timestamp can be calculated by the following equation:

$$\begin{aligned} ts_{mi} &= timestamp(\delta^{th} \text{ tuple preceding } t) \\ \text{where } \delta &= \begin{cases} s \text{ if } (N+1)\%s = 0 \\ (N+1)\%s \text{ otherwise} \end{cases} \end{aligned} \tag{7.2}$$

**Multiple-input window:** The most popular example of window-based operator with multiple inputs is a binary join. For time-based windows, Equation 7.1 can be used, with $ts_{start} = max(ts_{start_i})$, where $ts_{start_i}$ is the timestamp of the first input tuple the stream

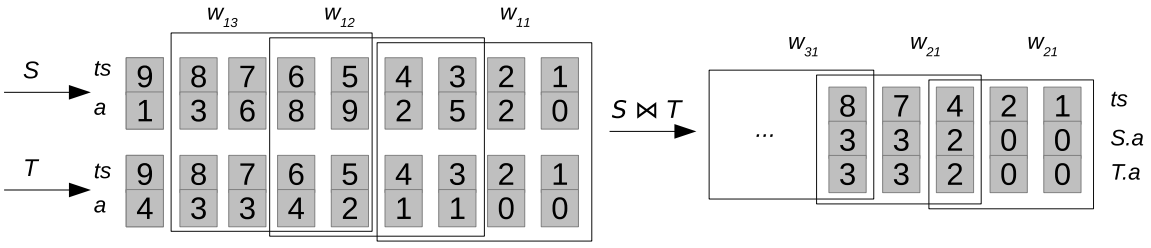


Figure 35: Calculating migration timestamp with two consecutive windows

source $i$ at target node was able to read. For tuple-based window, the number of tuples $N_i$ coming between $ts_{start_i}$ and $ts_{last\_w}$ is calculated separately for each input $i$. Afterwards, Equation 7.2 is applied with $N = max(N_i)$.

**Multiple window operators:** A CQ can have multiple window-based operators with different window specifications (i.e., length and slide), such as a query with an aggregation on top of a join. For these cases, we introduce the concept of the controlling window operator.

**Definition 13.** The controlling window operator is the closest window operator to the output of the CQ. The controlling window operator handles the calculation of the migration timestamp, as well as controlling the start and stop of the migrated query at the *target* and originating nodes.

For simplicity, we assume that the timestamp of an output tuple of a window-based operator is the *earliest* timestamp of input tuples involved in the calculation of that output tuple (we discuss later how this assumption can be relaxed). When the aforementioned condition holds, we know that all the original input tuples, contributing to the result produced by the farthest window of start timestamp $ts$, have timestamps greater than or equal to $ts$. Therefore, only the farthest window operator (i.e., the controlling window operator) in the CQ needs to be involved, and the calculation is the same as in the case of single window. Note that the previous assumption is not required for the controlling window operator.

Figure 35 shows an example of a CQ consisting of two window-based operators: a binary join, whose window has length of 4 seconds and slide 2 seconds, followed by an aggregation,

whose window has length of 3 tuples and size of 2 tuples. For each tuple, its timestamp is shown on the upper and its join key on the bottom parts. For the controlling window, the most recent window being produced is $w_{21}$, whose start timestamp is 1 (i.e., $ts_{last\_w} = 1$). In addition, assume that out of the two first tuples read from S and T by the target node, the latest timestamp $ts_{start}$ equals 5. In this case, the migration timestamp is calculated as if there is only the controlling window operator (i.e, the aggregation) with two inputs S and T. Because the controlling window operator is tuple-based, UniMiCo has to wait until tuple $t$ of timestamp 7 arrives to know that there are 3 tuples whose timestamps are between 1 and 5, i.e., $N = 3$. Applying the calculation from Equation 7.2 for the case of tuple-based window, UniMiCo decides that the migration timestamp is that of the tuple preceding $t$, which is 4. In other words, the last window produced at target is $w_{21}$.

When the previous condition on output tuples' timestamps of preceding window operators does not hold, $ts_{start}$ is measured as the timestamp of the first tuple arriving at the controlling window operator on the target node. Recall that when this condition holds, $ts_{start}$ is the timestamp of the first tuple coming to the source operator, i.e., it can be captured earlier. With the new $ts_{start}$, all of the above calculations of the migration timestamp are still applicable. Note that in this case if $ts_{start}$ is smaller than $ts_{last\_w}$, there will be some wasted processing at the target to process tuples from source up to the controlling window between $ts_{start}$ and $ts_{last\_w}$. Because migration happens when the target is lightly loaded, it is expected that processing at the target node will be at least as fast as that at the originating node, hence the wasted processing, if any, would be small.

### 7.2.5 Stopping and resuming continuous queries

**7.2.5.1 Stopping the query at the originating node** Once the migration timestamp is determined, stopping the query at the originating node is relatively straightforward: all operators in the CQ continue to process normally until they receive the signal from the controlling window operator to deactivate themselves. This happens when the controlling window operator has consumed its last window, i.e., the window started with the migration timestamp.

When the controlling window operator is associated with a join, a minor adjustment is needed in order to avoid duplicate outputs between the *originating* and target nodes. Normally, when there is a match between a tuple $t$ of one input and $t'$ of the other, the join tuple $tt'$ is produced only once, even if both $t$ and $t'$ fall in the overlap of two (or more) consecutive windows. If we start migrating from one of the windows, the join tuple $tt'$ will be produced once at the originating node, and again at the target node. In the latter case, the production of a duplicate tuple is avoided by suppressing the production of the join result at the originating node. Note that when two matching tuples have their timestamps in the window overlap, the previous adjustment is needed only if the join is the last window-based operator in the query. In the event that a join is followed by another window operator, the duplicated intermediate output $tt'$ is needed, as it is an input for the subsequent window at the target node.

**7.2.5.2 Starting the query at target node** All the operators of the migrated CQ can be activated at the target node, as soon as the migration is initialized. However, full activation is made feasible by controlling the flow of tuples based on the migration timestamp. That process behaves differently on time-based and tuple-based windows, as we describe below.

**Time-based controlling window operator:** If the CQ has a time-based controlling window operator, the stream source operator(s) calculate(s) the activation timestamp as migration timestamp increased with the slide of the window. Then, the stream source operator discards any input tuples, which carry timestamps less than the activation timestamp. In addition, it starts producing tuples with timestamp equal to or greater than the activation timestamp. With tuples being outputted from the stream source(s), the query is fully activated.

**Tuple-based controlling window operator:** In this case, the stream source operator(s) start(s) producing results from tuples with timestamps greater than the migration timestamp. But, the controlling window operator will discard all first $(s - \delta)$ tuples, where $s$ is the slide of the window and $\delta$ is calculated from Equation 7.2 by the originating node.

**Algorithm 2** UniMiCo protocol at target node

1: **BEGIN**
2: Receive(originating_node, *migrate*(Q))
3: **for** $i = 0; i <$ Q.num_streams$; i++$ **do**
4: connect(Q.streams[i])
5: $ts_{start}$[i] = read(Q.streams[i])
6: **end for**
7: Send(originating_node, $ts_{start}$)
8: Receive(originating_node,$ts_{mi}$)
9: Resume Q based on $ts_{mi}$
10: **END**

**Algorithm 3** UniMiCo protocol at originating node

1: **INPUT**: Query Q to be migrated
2: **BEGIN**
3: Send(target_node, *migrate*(Q)
4: Receive(target_node, $ts_{start}$)
5: $ts_{mi}$ = calculate_migration_timestamp
6: Send(target_node, $ts_{mi}$)
7: Finish_processing(Q, $ts_{mi}$)
8: **END**

For both types of windows, if the output timestamp of the preceding window-based operator is not the window's start timestamp, the controlling window operator has the single authority that decides when to output tuples. Thus, the source operator cannot do any early filtering.

Algorithms 2 and 3 give the outline of the UniMiCo protocol executed at *target* and originating node, respectively.

## 7.3 EXPERIMENTAL EVALUATION OF UNIMICO

### 7.3.1 Experiment settings

We implemented and evaluated UniMiCo in a distributed setup of AQSIOS, our DSMS prototype. Inherited from STREAM, the window operator is a separate operator, which receives stream tuples as input, and injects *minus* tuples to the stream to mark the boundary of a window [12]. Windows can have either time-based or tuple-based length, but the window slide is always 1 tuple. Window-based operators, such as join or aggregation, will rely on those minus tuples to perform their window-based processing. More information on window-based operators in STREAM can be found in [12].

Figure 36 show an example snapshot of data tuples output from a window operator in AQSIOS/STREAM. Each line correspond to a tuple, with its columns separated by the colons. The first field in quare brackets is the timestamp of the tuple, followed by the sign (plus or minus) and then the value(s). A plus tuple is a real "inserted" tuple, whereas a minus tuple is just a marker for a completion of the window started by the corresponding plus tuple. In this snapshot, the window has the length of 3 tuples and slide of 1 tuple. So the $4^{th}$ tuple is a minus tuple marking the end of the window started by the first tuple (i.e., $t_1$). Note that for illustration purpose this snapshot starts from the beginning of a stream, so we see all the first three plus tuples coming before the first minus tuple appears.

With the separation of the window operator, each input to a join operators can have a window of different length and type. In the scope of this work, we assume that join inputs have windows of the same length and the same type, so UniMiCo treats them as a single multiple-input window operator.

We run each experiment between two AQSIOS nodes[1]. In order to evaluate the correctness and efficiency of UniMiCo, we ran each query twice, one with the migration and one without it. Afterwards, we compared the query's outputs and response times around the migration point. All settings are the same between the two runs.

[1]Note that the coordinator is not necessary for UniMiCo migration protocol. The coordinator in AQSIOS facilitates our on-going project on workload balancing.

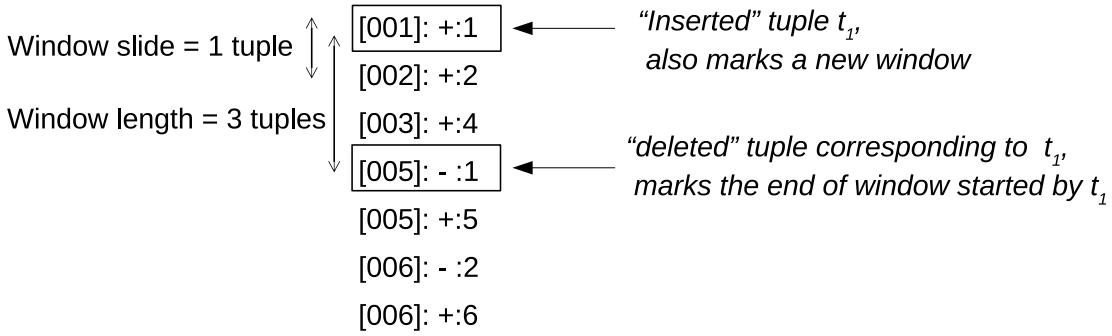


Figure 36: Example of a output tuples from a window operator in AQSIOS/STREAM

### 7.3.2 Experiment results

We run two types of experiments, one with simple CQs consisting of a single window operator and another with a complex CQ consisting of two window operators. In either case we have not included any non-stateful operators since they do not have any impact on the migration.

**7.3.2.1 Simple CQ migration:** We used UniMiCo to migrate a simple continuous query with a join operator (Q1), and another query with an aggregate operator (Q2). We show the two queries written in CQL [19] below:

```
Q1: SELECT *
    FROM S [Range 10 seconds],
         T [Range 10 seconds]
    WHERE S.1 = T.1;

Q2: SELECT sum(m)
    FROM S [Rows 5];
```

Figures 37 and 39 show the result of Q1 and Q2 around the migration point, respectively In Figure 37, the top plot is the result under migration, in which the rows above the dash line are the last output tuples at the originating node, and those below the dash line are the first output tuples at the target node. The bottom plots show the result without migration,

*Output with migration*

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
--------------------------------------------------
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

*Output without migration*

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

Figure 37: Result of Q1 around the migration point. Top plot is result with migration and bottom plot is result without migration.

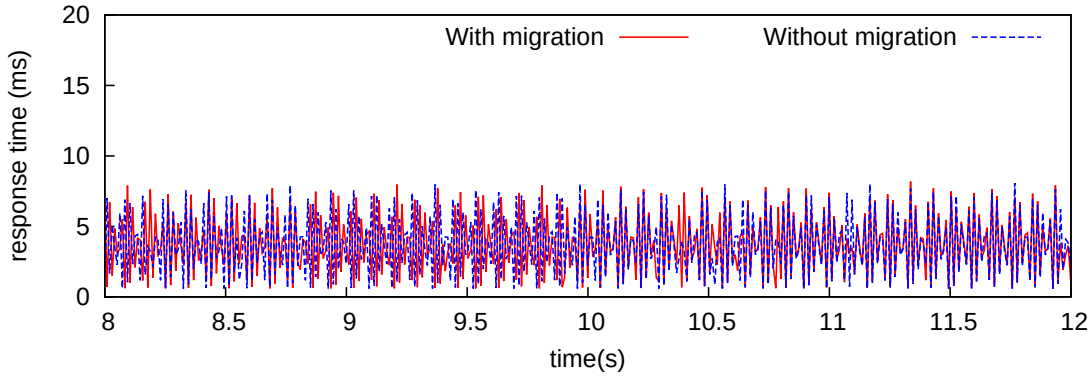


Figure 38: Response time of Q1 around the migration point of time $t = 10s$ second. The lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not introduce any noticeable delay.

| *Output with migration* | *Output without migration* |
|---|---|
| [10054323000]:+:92 | |
| [10054323000]:-:46 | [10054323000]:+:92 |
| [10054323000]:+:87 | [10054323000]:-:46 |
| [10054323000]:-:92 | [10054323000]:+:87 |
| ---------- | [10054323000]:-:92 |
| [10055771000]:+:102 | [10055771000]:+:102 |
| [10055771000]:-:87 | [10055771000]:-:87 |
| [10055771000]:+:99 | [10055771000]:+:99 |
| [10055771000]:-:102 | [10055771000]:-:102 |

Figure 39: Result of Q2 around the migration point. Left plot is result with migration and right plot is result without migration.

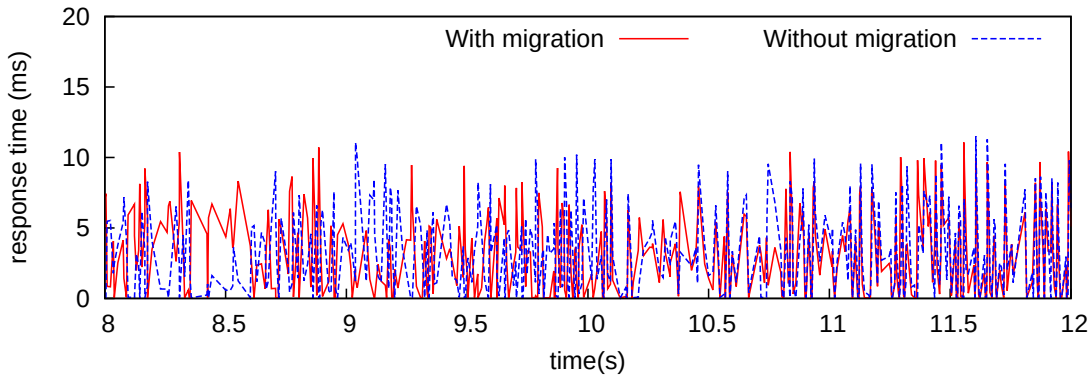


Figure 40: Response time of Q2 around the migration point of time $t = 10s$. The lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not introduce any noticeable delay.

which is exactly the same as the concatenation of the two parts of the top plot. Similar observations can be made in Figure 39 for Q2, except that the result with migration is on the left and that without migration is on the right. As one can see, the correctness of the output is maintained by using UniMiCo, and its protocol succeeds in performing the hand-off without losing any data.

Figures 38 and 40 show the response time of queries Q1 and Q2 two seconds before and after the migration point of time $t = 10s$. As can seen in both figures, there are no

| *Output with migration* | *Output without migration* |
|---|---|
| [10022574000]:+:109 | |
| [10022574000]:-:104 | [10022574000]:+:109 |
| [10022574000]:+:104 | [10022574000]:-:104 |
| [10022574000]:-:109 | [10022574000]:+:104 |
| --- | [10022574000]:-:109 |
| [10028529000]:+:107 | [10028529000]:+:107 |
| [10028529000]:-:104 | [10028529000]:-:104 |
| [10028529000]:+:70 | [10028529000]:+:70 |
| [10028529000]:-:107 | [10028529000]:-:107 |

Figure 41: Result of the complex query Q3 around the migration point. Left plot is result with migration and right plot is result without migration.

noticeable hiccups in the response time of the queries throughout the migration. For Q1, the average and standard deviation of the response time in this period without migration is 3.751ms and 3.99ms, respectively, while under migration they are 3.750ms and 3.97ms. For Q2, the corresponding numbers are 3.155ms and 3.923ms without migration, and 3.101ms and 3.836ms with migration. The difference in both cases is negligible.

**7.3.2.2 Complex CQ migration:** In this experiment we migrate a more complex query, Q3, with both join and aggregate operators, each use a different window definition as below:

```
Q3: SELECT sum(S.m)
    FROM ISTREAM
         (SELECT *
          FROM S [Range 10 seconds],
               T [Range 10 seconds]
          WHERE S.1 = T.1
          )[ROWS 5];
```

In this case, the last window, which is the tuple-based window of size 5 (i.e., [ROWS 5]) associated with the aggregation, plays the role of the *controlling window*.

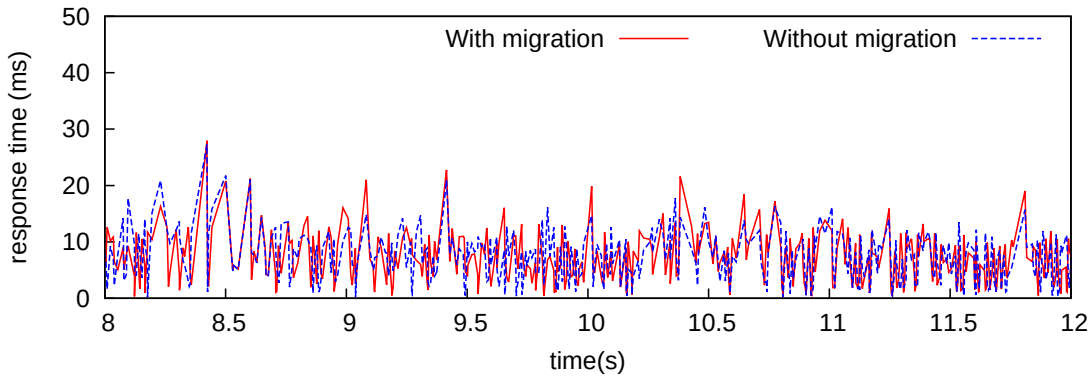


Figure 42: Response time of the complex query Q3 around the migration point of time $t = 10s$. The lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not introduce any noticeable delay

Figure 41 shows the output tuples and Figure 42 shows the response time of the query Q3 around the migration point, compared with the run when there is no migration. Similar to the cases of the simple queries, the query output is preserved and the cost of migration is not noticeable. The average and standard deviation of the response time without migration are 6.568ms and 6.133ms respectively, while those with migration are 6.658ms and 6.217ms.

## 7.4 SUMMARY

In this chapter we sketched our proposed framework, namely ARMaDILoS, for a large-scale adaptive resource management using DILoS. We proposed and evaluated UniMiCo, our lightweight, uninterruptible CQ migration protocol which serves as a key step toward an implementation of ARMaDILoS. UniMiCo itself is also a general CQ migration protocol that can be used in any multi-node DSMSs. Preliminary experimental results showed that UniMiCo could migrate CQs correctly from one node to another, while did not introduce any noticable changes in the response time of the migrated CQs.

# 8.0 CONCLUSIONS

## 8.1 SUMMARY OF CONTRIBUTION

This dissertation targets at solutions to the problem that can arise in a DSMS when a priority-based scheduler and load manager do not cooperate properly with each other in order to honor the priorities of CQs, which are specified by the user or application. That is, separately the policies can make inconsistent decisions, leaving the system in undesired situations such as failing to control the workload for some CQs while shedding more data than necessary from some other CQs. Furthermore, the system capacity might not be fully used, causing more data to be lost during heavy-load periods.

In this dissertation after analyzing the above problem, we proposed 1) DILoS, a novel framework that supports seamless integration between DSMS priority-based scheduler and load manager, 2) ALoMa and SEaMLeSS, two adaptive load managers which enables the realization of DILoS and outperform the state-of-the-art in determining when and how much load to shed, and 3) UniMiCo, an interruptible migration protocol for CQs. We also propose ARMaDILoS, a conceptual design of an adaptive resource management framework for cloud DSMSs, in which DILoS and UniMiCo are among the key components. We implemented and experimentally evaluated DILoS, ALoMa, SEaMLeSS, and UniMiCo in AQSIOS, our real DSMS prototype. The experiment results confirmed that the proposed schemes achieved their stated goals.

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 priority-based scheduler (e.g., CQC); (2) the scheduler can now better exploit the system

capacity and reduce load shedding by adjusting its decision using feedback from the load manager; 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*. 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 four key 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 and (4) it works with both fair and priority-based operator schedulers. SEaMLeSS was our initial effort and performs as well as ALoMa in terms of the first three dimensions. However SEaMLeSS is not independent of the fairness of the operator schedulers and implementation-wise it poses more constraint on the host DSMS (i.e., requires a separate not-overloaded thread to count the number of queued input tuples).

UniMiCo is a protocol that allows CQ migration without state transferring for stateful, window-based operators and without any downtime for the CQ. UniMiCo supports both time-based and tuple-based sliding window, and allows the migrated CQ to have multiple stateful operators with different window specification.

The success of DILoS, which facilitates the synergy between the scheduler and load manager in our new framework, confirms our hypothesis that *the synergy between the scheduler and the load manager would consistently provide differentiated levels of services for CQs, while using the system capacity more effectively.*

## 8.2 INTELLECTUAL MERIT

With DILoS, we pointed out that it is necessary for the different resource management modules in a system to work in synergy, which is missing in the state-of-the-art. That

would not only ensure consistent policies, but also promise better resource usage. DILoS itself is extensible: it is not a specific scheduling/load shedding policy, but instead a general framework for a priority-based DSMS scheduler and load shedder to cooperate to achieve some overall goal. As such, different load shedder can be plugged in (as long as it has the ability to automatically recognize the system capacity), and both the class-level and operator-level scheduling policies can be changed depending on the specific goal of each stream system.

Our adaptive load shedders, namely SEaMLeSS and ALoMa, while aiming at a real implementation of DILoS, also make an important contribution to the work of load shedding in DSMSs. The question of "when and how much to shed" is a crucial question that every load shedder has to answer before going any further, yet has not been solved thoroughly by the state-of-the-art. Our load shedders has filled in the gap, and can be used in complement to works that focus on the other question of the load shedding problem, i.e., what and where to shed.

Although UniMiCo shares the basic idea of state recreation with WRP [47], UniMiCo is the first that fully covers both tuple and sliding windows, as well as supports multiple stateful operators in the migrated CQ. UniMiCo therefore can be used as a general CQ migration protocol in any cloud-based DSMS, regardless whether that DSMS follows our proposed system model.

## 8.3 FUTURE WORK

Clearly, DILoS and ARMaDILoS are foundation steps towards efficient support for differentiated levels of service for CQs and there are many future extensions, especially in light of the constant advances in computer systems.

### 8.3.1 DILoS

DILoS has been implemented and evaluated on a single-thread DSMS. The obvious next step is to implement DILoS on a parallelized DSMS, which can utilize multiple CPUs on a

server in processing CQs. In such DSMSs, the processing of CQs is split up into multiple threads (each thread might or might not correspond to an operator, depending on the specific strategy of the execution engine). We expect that DILoS is also beneficial in such multi-core deployment. The load manager would still be able to recognize the capacity of the class it is in charge, which could be greater than 1. However, because there are multiple CPU, the scheduling task becomes challenging. The scheduler needs a proper strategy to schedule processing threads of the CQ classes on the multiple CPUs available, so that the priority of each class is honored. Also, capacity redistribution might result in moving certain processing threads from one CPU to another, incurring overheads such as cache misses.

Another important future work to extend DILoS is to combine ALoMa with a semantic load shedder (e.g., [74, 31, 30, 36]). ALoMa's decision on the amount of load to shed would serve as a required input for a semantic load shedder. However, semantic dropping is different from random dropping in that semantic dropping can result in an *effective shedded load* greater or smaller than the amount determined by the load shedder. This is because semantic dropping changes the selectivities of the downstream operators. ALoMa's adaptivity should allow it to cope with this issue, yet a more robust approach would be to force the effective shedded load to be the same as the amount decided by ALoMa.

### 8.3.2 ARMaDILoS

While DILoS and UniMiCo play the key roles in the proposed ARMaDILoS system model, there are still various important issues that need to be addressed in a full implementation:

1. Design a solid global workload distribution policy which takes into account the priorities of the CQ classes: Although ARMaDILoS can accept different policies, it is important to implement a reasonable one to evaluate the framework.
2. Deciding when to migrate, where to migrate to, and what queries to migrate: The coordinator needs to take into account the priority of the CQ classes (e.g., a lower-priority class might be the first candidate to be moved), the migrating cost (e.g., stateless CQs might be cheaper to move than stafeful CQs), and the potential benefit (e.g., the ability for the migrated query to share some operators with the query network at destination).

Also, because migration always has some cost, the coordinator needs to avoid migrating CQs just as a react to a brief spike in the load.

3. Improve overall system utilization: when the workload of the system is small, the coordinator might consider putting some nodes to sleep to save power.

The above issues are targets for future work to realize ARMaDILoS.

## 8.4 BROADER IMPACT

The work in this dissertation would enhance a DSMS's ability to provide differentiated levels of service for CQs, which is crucial because it helps guaranteeing that critical queries run fastest and received the most accurate results even when the DSMS is highly loaded. This is meaningful in many contexts, including health care (e.g, detection of emergency health problem), environmental surveillance (e.g., detection of wildfire, earthquake etc.), and financial market (e.g., spot trend changes).

The main technical contribution of the dissertation (DILoS and ALoMa) has been made available in a release of AQSIOS [4], which provides a basic experimental platform for further research on CQ processing in DSMSs.

## BIBLIOGRAPHY


[1] Amazon kinesis. https://aws.amazon.com/kinesis.

[2] Apache flink. http://flink.apache.org.

[3] Apache storm. http://storm.apache.org.

[4] Aqsios software release version 2.0. http://db.cs.pitt.edu/group/projects/aqsios/2.0/.

[5] Esper. http://esper.codehaus.org.

[6] Microsoft StreamInSight. https://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx.

[7] Pacific tsunami warning center. http://ptwc.weather.gov/.

[8] Samza. http://samza.apache.org.

[9] Spark. http://spark.apache.org.

[10] System S - Stream Computing at IBM Research. http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534.

[11] Tropical Asmosphere Ocean Project. http://www.pmel.noaa.gov/tao/.

[12] A. Arasu et al. Stream: The stanford data stream management system. Technical report, Stanford InfoLab, 2004.

[13] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[14] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[15] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, 2004.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases*, pages 480–491. VLDB Endowment, 2004.

[17] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMoD Record*, 29(2):261–272, 2000.

[18] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB JournalThe International Journal on Very Large Data Bases*, 13(4):333–353, 2004.

[19] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[20] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 253–264. ACM, 2003.

[21] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of 20th International Conference on Data Engineering*, pages 350–361. IEEE, 2004.

[22] Y. Bai and C. Zaniolo. Minimizing latency and memory in dsms: a unified approach to quasi-optimal scheduling. In *Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 58–67. ACM, 2008.

[23] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy. "cut me some slack": latency-aware live migration for databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 432–443, New York, NY, USA, 2012. ACM.

[24] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel. An approach to adaptive memory management in data stream systems. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 137–137. IEEE, 2006.

[25] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 838–849. VLDB Endowment, 2003.

[26] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[27] S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer, 2009.

[28] S. Chakravarthy and V. Pajjuri. Scheduling strategies and their evaluation in a data stream management system. *Flexible and Efficient Information Handling*, 4042:220–231, 2006.

[29] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[30] J. H. Chang and H.-C. M. Kum. Frequency-based load shedding over a data stream of tuples. *Information Sciences*, 179(21):3733–3744, 2009.

[31] Y. Chi, H. Wang, and P. S. Yu. Loadstar: load shedding in data stream mining. In *Proceedings of the 31st international conference on Very large data bases*, pages 1302–1305. VLDB Endowment, 2005.

[32] P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, June 2010.

[33] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21. USENIX Association, 2010.

[34] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the 34th international conference on Very large data bases*, 1(2):1277–1288, 2008.

[35] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, 2011.

[36] R. Dash and L. Fegaras. Synopsis based load shedding in xml streams. In *Proceedings of the 2009 EDBT/ICDT Workshops*, pages 93–98. ACM, 2009.

[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[38] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 301–312. ACM, 2011.

[39] F. Farag, M. Hammad, and R. Alhajj. Adaptive query processing in data stream management systems under limited memory resources. In *Proceedings of the 3rd workshop on Ph.D. students in information and knowledge management*, pages 9–16. ACM, 2010.

[40] H. Feng, Z. Liu, C. H. Xia, and L. Zhang. Load shedding and distributed resource control of stream processing networks. *Performance Evaluation*, 64(9):1102–1120, 2007.

[41] B. Gedik, K.-L. Wu, S. Y. Philip, and L. Liu. Cpu load shedding for binary stream joins. *Knowledge and Information Systems*, 13(3):271–303, 2007.

[42] B. Gedik, K.-L. Wu, and P. S. Yu. Efficient construction of compact shedding filters for data stream processing. In *IEEE 24th International Conference on Data Engineering*, pages 396–405. IEEE, 2008.

[43] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1363–1380, 2007.

[44] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Mobiqual: Qos-aware load shedding in mobile cq systems. In *Proceedings of the 24th IEEE International Conference on Data Engineering*, pages 1121–1130. IEEE, 2008.

[45] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1515–1524. ACM, 2011.

[46] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011.

[47] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, (12):2351–2365, 2012.

[48] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[49] N. R. Katsipoulakis, C. Thoma, E. A. Gratta, A. Labrinidis, A. J. Lee, and P. K. Chrysanthis. Ce-storm: Confidential elastic processing of data streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 859–864, 2015.

[50] B. Kendai and S. Chakravarthy. Load shedding in mavstream: Analysis, implementation, and evaluation. In *Sharing Data, Information and Knowledge*, pages 100–112. Springer, 2008.

[51] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In *6th IEEE Workshop on Self Managing Database Systems*, pages 16–21. IEEE, 2011.

[52] D. Kulkarni, C. V. Ravishankar, and M. Cherniack. Real-time, load-adaptive processing of continuous queries over data streams. In *Proceedings of the second international conference on Distributed event-based systems*, pages 277–288. ACM, 2008.

[53] C. Lei and E. A. Rundensteiner. Robust distributed query processing for streaming data. *ACM Transactions on Database Systems*, 39(2):17, 2014.

[54] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 811–825. ACM, 2015.

[55] C. Mafrica, J. Johnson, S. Bock, T. N. Pham, B. R. Childers, P. K. Chrysanthis, and A. Labrinidis. Stream query processing on emerging memory architectures. In *Proceedings of the 4th IEEE Non-Volatile Memory Systems and Applications Symposium*, 2015.

[56] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs. Admission control mechanisms for continuous queries in the cloud. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, pages 409–412, 2010.

[57] L. A. Moakar, A. Labrinidis, and P. K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *7th IEEE Workshop on Self Managing Database Systems*, pages 289–294. IEEE, 2012.

[58] L. A. Moakar, T. N. Pham, P. Neophytou, P. K. Chrysanthis, A. Labrinidis, and M. Sharaf. Class-based continuous query scheduling for data streams. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, pages 1–6. ACM, 2009.

[59] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *Proceddings of the 26th IEEE International Conference on Data Engineering*, pages 76–88. IEEE, 2010.

[60] K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 852–863. IEEE, 2011.

[61] R. V. Nehme and E. A. Rundensteiner. Clustersheddy: load shedding using moving clusters over spatio-temporal data streams. In *Advances in Databases: Concepts, Systems and Applications*, pages 637–651. Springer, 2007.

[62] P. Neophytou, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Power-aware operator placement and broadcasting of continuous query results. In *Proc. of the ACM International Workshop on Data Engineering for Mobile and Wireless Data Access*, pages 1–8, 2010.

[63] P. Neophytou, J. Szwedko, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimizing the energy consumption of continuous query processing with mobile clients. In *Proc. of the 12th International IEEE Conference on Mobile Data Management*, number 1, pages 98–103, 2011.

[64] T. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis. Uninterruptible migration of continuous queries without operator state migration. Under submission to SIGMOD Record.

[65] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Self-managing load shedding for data stream management systems. In *8th IEEE Workshop on Self Managing Database Systems*, pages 70–76. IEEE, 2013.

[66] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Avoiding class warfare: Managing continuous queries with differentiated classes of service. *The VLDB JournalThe International Journal on Very Large Data Bases*, 25(2):197–221, 2016. 11/12/2015 published on-line.

[67] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *6th IEEE Workshop on Self Managing Database Systems (SMDB 2011)*, pages 10–15. IEEE, 2011.

[68] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *Proceedings of the 21st International Conference on Data Engineering*, pages 155–156. IEEE, 2005.

[69] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[70] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering*, pages 25–36. IEEE, 2003.

[71] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems*, 33(1):5.1–5.44, 2008.

[72] I. Stanoi, G. Mihaila, C. Lang, and T. Palpanas. Whitewater: distributed processing of fast streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1214–1226, 2007.

[73] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[74] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases*, pages 309–320. VLDB Endowment, 2003.

[75] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases*, pages 799–810. VLDB Endowment, 2006.

[76] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006.

[77] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[78] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd international conference on Very large data bases*, pages 619–630. VLDB Endowment, 2006.

[79] Y. Wei, S. H. Son, and J. A. Stankovic. RTSTREAM: Real-Time Query Processing for Data Streams. pages 141–150. IEEE Computer Society, 2006.

[80] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware 2008*, pages 306–325. Springer, 2008.

[81] S. Wu, Y. Lv, G. Yu, Y. Gu, and X. Li. A qos-guaranteeing scheduling algorithm for continuous queries over streams. In *Advances in Data and Web Management*, pages 522–533. Springer, 2007.

[82] Y. Xing, J. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, pages 775–786. VLDB Endowment, 2006.

[83] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. In *36th International Conference on Very Large Data Bases*, 2011.

[84] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *Proceedings of the 2006 Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, pages 54–71.