Load-Aware Shedding in Stream Processing Systems

Nicoló Rivetti
LINA / Université de Nantes, France
DIAG / Sapienza University of Rome, Italy
rivetti@dis.uniroma1.it

Yann Busnel
Crest (Ensai) / Inria
Rennes, France
yann.busnel@ensai.fr

Leonardo Querzoni
DIAG / Sapienza University of Rome, Italy
querzoni@dis.uniroma1.it

ABSTRACT

Load shedding is a technique employed by stream processing systems to handle unpredictable spikes in the input load whenever available computing resources are not adequately provisioned. A load shedder drops tuples to keep the input load below a critical threshold and thus avoid unbounded queuing and system trashing. In this paper we propose Load-Aware Shedding (LAS), a novel load shedding solution that, unlike previous works, does not rely neither on a pre-defined cost model nor on any assumption on the tuple execution duration. Leveraging *sketches*, LAS efficiently builds and maintains at runtime a cost model to estimate the execution duration of each tuple with small error bounds. This estimation enables a proactive load shedding of the input stream at any operator that aims at limiting queuing latencies while dropping as few tuples as possible. We provide a theoretical analysis proving that LAS is an (ε, δ)-approximation of the optimal online load shedder. Furthermore, through an extensive practical evaluation based on simulations and a prototype, we evaluate its impact on stream processing applications, which validate the robustness and accuracy of LAS.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

Keywords

Stream Processing, Data Streaming, Load Shedding

1. INTRODUCTION

Distributed stream processing systems (DSPS) are today considered as a mainstream technology to build architectures for the real-time analysis of big data. An application running in a DSPS is typically modeled as a directed acyclic graph (a topology) where data operators, represented by nodes, are interconnected by streams of tuples containing data to be analyzed, the directed edges. The success of such systems can be traced back to their ability to run complex applications at scale on clusters of commodity hardware.

Correctly provisioning computing resources for DSPS however is far from being a trivial task. System designers need to take into account several factors: the computational complexity of the operators, the overhead induced by the framework, and the characteristics of the input streams. This latter aspect is often the most critical, as input data streams may unpredictably change over time both in rate and in content. Over-provisioning the DSPS is not economically sensible, thus system designers are today moving toward approaches based on elastic scalability [6], where an underlying infrastructure is able to tune at runtime the available resources in response to changes in the workload characteristics. This represents a desirable solution when coupled with on-demand provisioning offered by many cloud platforms, but still comes at a cost (in terms of overhead and time for scale-up/down) and is limited to mid- to long-term fluctuations in the input load.

Bursty input load represents a problem for DSPS as it may create unpredictable bottlenecks within the system that lead to an increase in queuing latencies, pushing the system in a state where it cannot deliver the expected quality of service (typically expressed in terms of tuple completion latency). *Load shedding* is generally considered a practical approach to handle bursty traffic. It consists in dropping a subset of incoming tuples as soon as a bottleneck is detected in the system. As such, load shedding is a solution that can live in conjunction with resource shaping techniques (like elastic scaling), rather than being an alternative.

Existing load shedding solution either randomly drop tuples when bottlenecks are detected or apply a pre-defined model of the application and its input that allows them to deterministically take the best shedding decision. In any case, all the existing solutions assume that incoming tuples all impose the same computational load on the DSPS. However, such assumption (*i.e.*, same execution duration for all tuples of a stream) does not hold for many practical use cases. The tuple execution duration, in fact, may depend on the tuple content itself. This is often the case whenever the receiving operator implements a logic with branches where only a subset of the incoming tuples travels through each single branch. If the computation associated with each branch generates different loads, then the execution duration will change from tuple to tuple. A tuple with a large execution duration may delay the execution of subsequent tuples in the same stream, thus increasing queuing latencies. If further tuples are enqueues with large execution durations, this may bring to the emergence of a bottleneck.

On the basis of this simple observation, we introduce Load-Aware Shedding (LAS), a novel solution for load shedding in DSPS. LAS gets rid of the aforementioned assumptions and provides efficient shedding aimed at matching given queuing latency targets, while dropping as few tuples as possible. To reach this goal LAS leverages a smart combination of *sketch* data structures to efficiently collect at runtime information on the time needed to compute tuples. This information is used to build and maintain, at runtime, a cost model that is then exploited to take decisions on when load must be shed. LAS has been designed as a flexible solution that can be applied on a per-operator basis, thus allowing developers to target specific critical stream paths in their applications.

In summary, the contributions provided by this paper are:

- the introduction of LAS, the first solution for load shedding in DSPS that proactively drops tuples to avoid bottlenecks without requiring a predefined cost model and without any assumption on the distribution of tuples;
- a theoretical analysis of LAS that points out how it is an (ϵ, δ)-approximation of the optimal online shedding algorithm;
- an experimental evaluation that illustrates how LAS can provide predictable queuing latencies that approximate a given threshold while dropping a small fraction of the incoming tuples.

Below, the next section states the system model we consider. Afterwards, Section 3 details LAS whose behavior is then theoretically analyzed in Section 4. Section 5 reports on our experimental evaluation and Section 6 analyzes the related works. Finally Section 7 concludes the paper.

2. SYSTEM MODEL AND PROBLEM DEFINITION

We consider a distributed stream processing system (DSPS) deployed on a cluster where several computing nodes exchange data through messages sent over a network. The DSPS executes a stream processing application represented by a *topology*: a directed acyclic graph interconnecting operators, represented by vertices, with data streams (DS), represented by edges. Each topology contains at least a *source*, *i.e.,* an operator connected only through outbound DSs, and a *sink*, *i.e.,* an operator connected only through inbound DSs.

Data injected by the source is encapsulated in units called tuples and each data stream is an unbounded sequence of tuples. Without loss of generality, here we assume that each tuple t is a finite set of key/value pairs that can be customized to represent complex data structures. To simplify the discussion, in the rest of this work we deal with streams of unary tuples each representing a single non negative integer value.

For the sake of clarity, and without loss of generality, here we restrict our model to a topology with an operator LS (*load shedder*) that decides which tuples of its outbound DS σ consumed by operator O shall be dropped. Tuples in σ are drawn from a large universe $[n] = \{1, \ldots, n\}$ and are ordered, *i.e.,* $\sigma = \langle t_1, \ldots, t_m \rangle$. Therefore $[m] = 1, \ldots, m$ is the index sequence associated with the m tuples contained in the stream σ. Both m and n are unknown. We denote with f_t the unknown frequency[1] of tuple t, *i.e.,* the number of occurrences of t in σ.

We assume that the execution duration of tuple t on operator O, denoted as $w(t)$, depends on the content of the tuple t. We simplify the model assuming that w depends on a single, fixed and known attribute value of tuple t. The probability distribution of such attribute values, as well as the function w are unknown, may differ from operator to operator and may change over time. However, we assume that subsequent changes are interleaved by a large enough time frame such that an algorithm may have a reasonable amount of time to adapt. On the other hand, the input throughput of the stream may vary, even with a large magnitude, at any time.

Let $q(i)$ be the queuing latency of the i-th tuple of the stream, *i.e.,* the time spent by the i-th tuple in the inbound buffer of operator O before being processed. Let us denote as $\mathcal{D} \subseteq [m]$, the set of dropped tuples in a stream of length m, *i.e.,* dropped tuples are thus represented in $\mathcal{D}$ by their indices in the stream $[m]$. Moreover, let $d \leq m$ be the number of dropped tuples in a stream of length m, *i.e.,* $d = |\mathcal{D}|$. Then we can define the average queuing latency as: $\overline{Q}(j) = \sum_{i \in [j] \setminus \mathcal{D}} q(i)/(j-d)$ for all $j \in [m]$.

The goal of the load shedder is to maintain at any point in the stream the average queuing latency smaller than a given threshold τ by dropping as less tuples as possible. The quality of the shedder can be evaluated both by comparing the resulting $\overline{Q}$ against τ and by measuring the number of dropped tuples d. More formally, the load shedding problem can be defined as follows[2].

Problem 2.1 (Load Shedding). *Given a data stream $\sigma = \langle t_1, \ldots, t_m \rangle$, find the smallest set $\mathcal{D}$ such that*

$$\forall j \in [m] \setminus \mathcal{D}, \overline{Q}(j) \leq \tau.$$

3. LOAD AWARE SHEDDING

This section introduces the Load-Aware Shedding algorithm by first providing an overview, then detailing some background knowledge, and finally describing the details of its functioning.

3.1 Overview

Load-Aware Shedding (LAS) is based on a simple, yet effective, idea: if we assume to know the execution duration $w(t)$ of each tuple t on the operator, then we can foresee the queuing time for each tuple of the operator input stream and then drop all tuples that will cause the queuing latency threshold τ to be violated. However, the value of $w(t)$ is generally unknown. A possible solution to this problem is to build a static cost model for tuple execution duration and then use it to proactively shed load. However, building an accurate cost model usually requires a large amount of *a priori* knowledge on the system. Furthermore, once a model has been built, it can be hard to handle changes in the system or input stream characteristics at runtime.

[1]This definition of *frequency* is compliant with the data streaming literature.

[2]This is not the only possible definition of the load shedding problems. Other variants are briefly discussed in section 6.

LAS overcomes these issues by building and maintaining at run-time a cost model for tuple execution durations. It takes shedding decision based on the estimation $\widehat{\mathcal{C}}$ of the total execution duration of the operator: $\mathcal{C} = \sum_{i\in[m]\setminus\mathcal{D}} w(t_i)$. In order to do so, LAS computes an estimation $\hat{w}(t)$ of the execution duration $w(t)$ of each tuple t. Then, it computes the sum of the estimated execution durations of the tuples assigned to the operator, *i.e.,* $\widehat{\mathcal{C}} = \sum_{i\in[m]\setminus\mathcal{D}} \hat{w}(t)$. At the arrival of the i-th tuple, subtracting from $\widehat{\mathcal{C}}$ the (physical) time elapsed from the emission of the first tuple provides us with an estimation $\hat{q}(i)$ of the queuing latency $q(i)$ for the current tuple.

To enable this approach, LAS builds a sketch on the operator (*i.e.,* a memory efficient data structure) that will track the execution duration of the tuples it process. When a change in the stream or operator characteristics affects the tuples execution durations $w(t)$, *i.e.,* the sketch content changes, the operator will forward an updated version to the load shedder, which will than be able to (again) correctly estimate the tuples execution durations. This solution does not require any *a priori* knowledge on the stream or system, and is designed to continuously adapt to changes in the input stream or on the operator characteristics.

3.2 Background

2-Universal Hash Functions — Our algorithm uses hash functions randomly picked from a 2-universal hash functions family. A collection $\mathcal{H}$ of hash functions $h : \{1,\ldots,n\} \to \{0,\ldots,c\}$ is said to be 2-universal if for every two different items $x, y \in [n]$, for any $h \in \mathcal{H}$, $\mathbb{P}\{h(x) = h(y)\} \leq \frac{1}{c}$, which is the probability of collision obtained if the hash function assigned truly random values to any $x \in [n]$. Carter and Wegman [3] provide an efficient method to build large families of hash functions approximating the 2-universality property.

`Count Min` sketch algorithm — Cormode and Muthukrishnan have introduced in [4] the `Count Min` sketch that provides, for each item t in the input stream an (ε,δ)-additive-approximation $\hat{f}_t$ of the frequency f_t. The `Count Min` sketch consists of a two dimensional matrix $\mathcal{F}$ of size $r \times c$, where $r = \lceil \log \frac{1}{\delta} \rceil$ and $c = \lceil \frac{e}{\varepsilon} \rceil$. Each row is associated with a different 2-universal hash function $h_i : [n] \to [c]$. When the `Count Min` algorithm reads sample t from the input stream, it updates each row: $\forall i \in [r], \mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + 1$. Thus, the cell value is the sum of the frequencies of all the items mapped to that cell. Upon request of f_t estimation, the algorithm returns the smallest cell value among the cells associated with t: $\hat{f}_t = \min_{i\in[r]}\{\mathcal{F}[i, h_i(t)]\}$.

Fed with a stream of m items, the space complexity of this algorithm is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n))$ bits, while update and query time complexities are $O(\log 1/\delta)$. The `Count Min` algorithm guarantees that the following bound holds on the estimation accuracy for each item read from the input stream: $\mathbb{P}\{\mid \hat{f}_t - f_t \mid \geq \varepsilon(m - f_t)\} \leq \delta$, while $f_t \leq \hat{f}_t$ is always true.

This algorithm can be easily generalized to provide (ε,δ)-additive-approximation of point queriesd on stream of updates, *i.e.,* a stream where each item t carries a positive integer update value v_t. When the `Count Min` algorithm reads the pair $\langle t, v\rangle$ from the input stream, the update routine changes as follows: $\forall i \in [r], \mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + v$.

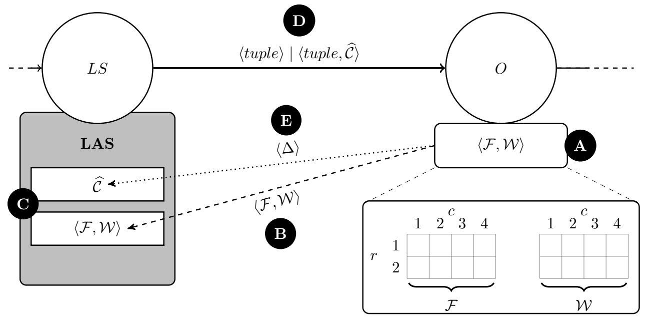

D
$\langle tuple\rangle \mid \langle tuple, \widehat{\mathcal{C}}\rangle$
LS
O
E
$\langle\Delta\rangle$
LAS
$\widehat{\mathcal{C}}$
C
$\langle\mathcal{F},\mathcal{W}\rangle$
$\langle\mathcal{F},\mathcal{W}\rangle$
B
$\langle\mathcal{F},\mathcal{W}\rangle$
A
c
1 2 3 4
1 2 3 4
r
1
2
$\mathcal{F}$
$\mathcal{W}$

Figure 1: Load-Aware Shedding design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$).

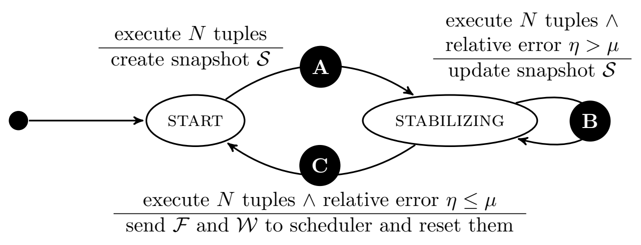
3.3 LAS design

The operator maintains two `Count Min` sketch matrices (Figure 1.A): the first one, denoted as $\mathcal{F}$, tracks the tuple frequencies f_t; the second one, denoted as $\mathcal{W}$, tracks the tuples cumulated execution durations $W_t = w(t) \times f_t$. Both `Count Min` matrices share the same sizes and hash functions. The latter is the generalized version of the `Count Min` presented in Section 3.2 where the update value is the tuple execution duration when processed by the instance (*i.e.,* $v = w(t)$). The operator will update (Listing 3.1 lines 27-30) both matrices after each tuple execution.

Listing 3.1: Operator

```
 1: init do
 2:     $\mathcal{F} \leftarrow 0_{r,c}$                  ▷ zero matrices of size $r \times c$
 3:     $\mathcal{W} \leftarrow 0_{r,c}$
 4:     $\mathcal{S} \leftarrow 0_{r,c}$
 5:     $r$ hash functions $h_1,\ldots,h_r : [n] \to [c]$ from a 2-universal
    family.
 6:     $m \leftarrow 0$
 7:     $state \leftarrow$ START
 8: end init
 9: function UPDATE($tuple : t,\ execution\ time : l, request : \widehat{\mathcal{C}}$)
10:     $m \leftarrow m + 1$
11:     if $\widehat{\mathcal{C}}$ not null then
12:         $\Delta \leftarrow \mathcal{C} - \widehat{\mathcal{C}}$
13:         send $\langle\Delta\rangle$ to $LS$
14:     end if
15:     if $state =$ START $\wedge\ m \mod N = 0$ then       ▷ Figure 2.A
16:         update $\mathcal{S}$
17:         $state \leftarrow$ STABILIZING
18:     else if $state =$ STABILIZING $\wedge\ m \mod N = 0$ then
19:         if $\eta \leq \mu$ (Eq. 1) then                   ▷ Figure 2.C
20:             send $\langle\mathcal{F},\mathcal{W}\rangle$ to $LS$
21:             $state \leftarrow$ START
22:             reset $\mathcal{F}$ and $\mathcal{W}$ to $0_{r,c}$
23:         else                                          ▷ Figure 2.B
24:             update $\mathcal{S}$
25:         end if
26:     end if
27:     for $i = 1$ to $r$ do
28:         $\mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + 1$
29:         $\mathcal{W}[i, h_i(t)] \leftarrow \mathcal{W}[i, h_i(t)] + l$
30:     end for
31: end function
```

The operator is modeled as a finite state machine (Figure 2) with two states: START and STABILIZING. The



execute N tuples
create snapshot $\mathcal{S}$
execute N tuples $\wedge$ relative error $\eta > \mu$
update snapshot $\mathcal{S}$
START
STABILIZING
A
B
C
execute N tuples $\wedge$ relative error $\eta \leq \mu$
send $\mathcal{F}$ and $\mathcal{W}$ to scheduler and reset them

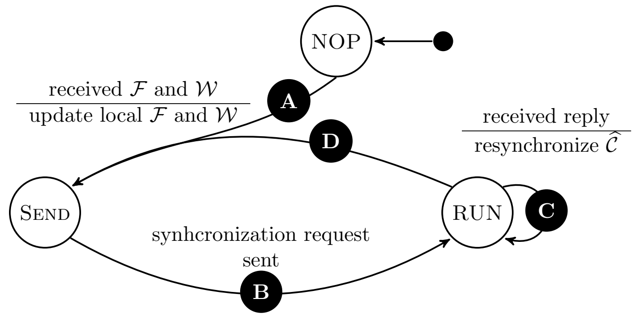
Figure 2: Operator finite state machine.

START state lasts as long as the operator has executed N tuples, where N is a user defined window size parameter. The transition to the STABILIZING state (Figure 2.A) triggers the creation of a new snapshot $\mathcal{S}$. A snapshot is a matrix of size $r \times c$ where $\forall i \in [r], j \in [c] : \mathcal{S}[i,j] = \mathcal{W}[i,j]/\mathcal{F}[i,j]$ (Listing 3.1 lines 15-17). We say that the $\mathcal{F}$ and $\mathcal{W}$ matrices are stable when the relative error η between the previous snapshot and the current one is smaller than a configurable parameter μ, *i.e.,*

$$\eta = \frac{\sum_{\forall i,j} |\mathcal{S}[i,j] - \frac{\mathcal{W}[i,j]}{\mathcal{F}[i,j])}|}{\sum_{\forall i,j} \mathcal{S}[i,j]} \leq \mu \tag{1}$$

is satisfied. Then, each time the operator has executed N tuples (Listing 3.1 lines 18-25), it checks whether Equation 1 is satisfied. **(i)** In the negative case $\mathcal{S}$ is updated (Figure 2.B). **(ii)** In the positive case the operator sends the $\mathcal{F}$ and $\mathcal{W}$ matrices to the load shedder (Figure 1.B), resets their content and moves back to the START state (Figure 2.C).

There is a delay between any change in $w(t)$ and when LS receives the updated $\mathcal{F}$ and $\mathcal{W}$ matrices. This introduces a skew in the cumulated execution duration estimated by LS. In order to compensate this skew, we introduce a synchronization mechanism that kicks in whenever the LS receives a new pair of matrices from the operator.



NOP
received $\mathcal{F}$ and $\mathcal{W}$
update local $\mathcal{F}$ and $\mathcal{W}$
A
D
received reply
resynchronize $\widehat{\mathcal{C}}$
SEND
RUN
C
synhcronization request sent
B

Figure 3: Load shedder *LS* finite state machine.

The LS (Figure 1.C) maintains the estimated cumulated execution duration of the operator $\widehat{\mathcal{C}}$ and a pairs of initially empty matrices $\langle \mathcal{F}, \mathcal{W} \rangle$. LS is modeled as a finite state machine (Figure 3) with three states: NOP, SEND and RUN. The LS executes the code reported in Listing 3.2. In particular, every time a new tuple t arrives at the LS, the function SHED is executed. The LS starts in the NOP state where no action is performed (Listing 3.2 lines 15-17). Here we assume that in this initial phase, *i.e.,* when the topology has just been deployed, no load shedding is required. When LS receives the first pair $\langle \mathcal{F}, \mathcal{W} \rangle$ of matrices (Figure 3.A), it moves into the SEND state and updates its local pair of matrices (Listing 3.2 lines 7-9). While being in the SEND states, LS sends to O the current cumulated execution duration estimation $\widehat{\mathcal{C}}$ (Figure 1.D) piggy backing it with the first tuple t that is not dropped (Listing 3.2 lines 24-26) and moves in the RUN state (Figure 3.B). This informations is used to synchronize the LS with O and remove the skew between O's cumulated execution duration $\mathcal{C}$ and the estimation $\widehat{\mathcal{C}}$ at LS. O replies to this request (Figure 1.E) with the difference $\Delta = \mathcal{C} - \widehat{\mathcal{C}}$ (Listing 3.1 lines 11-13). When the load shedder receives the synchronization reply (Figure 3.C) it updates its estimation $\widehat{\mathcal{C}} + \Delta$ (Listing 3.2 lines 11-13).

In the RUN state, the load shedder computes, for each tuple t, the estimated queuing latency $\hat{q}(i)$ as the difference between the operator estimated execution duration $\widehat{\mathcal{C}}$ and the time elapsed from the emission of the first tuple (Listing 3.2 line 18). It then checks if the estimated queuing latency for t satisfies the CHECK method (Listing 3.2 lines 19-21).

This method encapsulates the logic for checking if a desired condition on queuing latencies is violated or not. In this paper, as stated in Section 2, we aim at maintaining the average queuing latency below a threshold τ. Then, CHECK tries to add $\hat{q}$ to the current average queuing latency (Listing 3.2 lines 31). If the result is larger than τ (**i**), it simply returns *true*; otherwise (**ii**), it updates its local value for the average queuing latency and returns $false$ (Listing 3.2 lines 34-36). Note that different goals, based on the queuing latency, can be defined and encapsulated within CHECK, *e.g.,* maintain the absolute per-tuple queuing latency below τ, or maintain the average queuing latency calculated on a sliding window below τ.

If CHECK($\hat{q}$) returns *true*, (**i**) the load shedder returns *true* as well, *i.e.,* tuple t must be dropped. Otherwise (**ii**), the operator estimated execution duration $\widehat{\mathcal{C}}$ is updated with the estimated tuple execution duration $\hat{w}(t)$, increased by a factor $1 + \varepsilon$ to mitigate potential under-estimations[3], and the load shedder returns $false$ (Listing 3.2 line 28), *i.e.,* the tuple must not be dropped. Finally, if the load shedder receives a new pair $\langle \mathcal{F}, \mathcal{W} \rangle$ of matrices (Figure 3.D), it will again update its local pair of matrices and move to the SEND state (Listing 3.2 lines 7-9).

Now we will briefly discuss the complexity of LAS.

THEOREM 3.1 (TIME COMPLEXITY OF LAS). *For each tuple read from the input stream, the time complexity of LAS for the operator and the load shedder is $\mathcal{O}(\log 1/\delta)$.*

PROOF. By Listing 3.1, for each tuple read from the input stream, the algorithm increments an entry per row of both the $\mathcal{F}$ and $\mathcal{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting update time complexity is $\mathcal{O}(\log 1/\delta)$. By Listing 3.2, for each submitted tuple, the scheduler has to retrieve the estimated execution duration for the submitted tuple. This operation requires to read entry per row of both the $\mathcal{F}$ and $\mathcal{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting query time complexity is $\mathcal{O}(\log 1/\delta)$. □

[3]This correction factor derives from the fact that $\hat{w}(t)$ is a (ε, δ)-approximation of $w(t)$ as shown in Section 4.

Listing 3.2: Load shedder

```
init do
    $\widehat{\mathcal{C}} \leftarrow 0$
    $\langle \mathcal{F}, \mathcal{W} \rangle \leftarrow \langle 0_{r,c}, 0_{r,c} \rangle$    ▷ zero matrices pair of size $r \times c$
    Same hash functions $h_1 \dots h_r$ of the operator
    $state \leftarrow$ NOP
end init
upon $\langle \mathcal{F}', \mathcal{W}' \rangle$ do    ▷ Figure 3.A and 3.D
    $state \leftarrow$ SEND
    $\langle \mathcal{F}, \mathcal{W} \rangle \leftarrow \langle \mathcal{F}', \mathcal{W}' \rangle$
end upon
upon $\langle \Delta \rangle$ do    ▷ Figure 3.C
    $\widehat{\mathcal{C}} \leftarrow \widehat{\mathcal{C}} + \Delta$
end upon
function SHED($tuple : t$)
    if $state =$ NOP then
        return $false$
    end if
    $\hat{q} \leftarrow \widehat{\mathcal{C}} -$ elapsed time from first tuple
    if CHECK($\hat{q}$) then
        return $true$
    end if
    $i \leftarrow \arg\min_{i \in [r]} \{\mathcal{F}[i, h_i(t)]\}$
    $\widehat{\mathcal{C}} \leftarrow \widehat{\mathcal{C}} + (\mathcal{W}[i, h_i(t)] / \mathcal{F}[i, h_i(t)]) \times (1 + \varepsilon)$
    if $state =$ SEND then    ▷ Figure 3.B
        piggy back $\widehat{\mathcal{C}}$ to operator on $t$
        $state \leftarrow$ RUN
    end if
    return $false$
end function
function CHECK($q$)
    if $Q/\ell > \tau$ then
        return $true$
    end if
    $Q \leftarrow Q + q$
    $\ell \leftarrow \ell + 1$
    return $false$
end function
```

THEOREM 3.2 (SPACE COMPLEXITY OF LAS). *The space complexity of LAS for the operator and load shedder is $\mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$ bits.*

PROOF. The operator stores two matrices of size $\log(\frac{1}{\delta}) \times \frac{e}{\varepsilon}$ of counters of size $\log m$. In addition, it also stores a hash function with a domain of size n. Then the space complexity of LAS on the operator is $\mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$ bits. The load shedder stores the same matrices, as well as a scalar. Then the space complexity of LAS on the load shedder is also $\mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$ bits. □

THEOREM 3.3 (COMMUNICATION COMPLEXITY OF LAS). *The communication complexity of LAS is of $\mathcal{O}\left(\frac{m}{N}\right)$ messages and $\mathcal{O}\left(\frac{m}{N}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n) + \log m\right)\right)$ bits.*

PROOF. After executing N tuples, the operator may send the $\mathcal{F}$ and $\mathcal{W}$ matrices to the load shedder. This generates a communication cost of $\mathcal{O}\left(\frac{m}{N}\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$ bits via $\mathcal{O}\left(\frac{m}{N}\right)$ messages. When the load shedder receives these matrices, the synchronization mechanism kicks in and triggers a round trip communication (half of which is piggy backed by the tuples) with the operator. The communication cost of the synchronization mechanism is $\mathcal{O}\left(\frac{m}{N}\right)$ messages and $\mathcal{O}\left(\frac{m}{N}\log m\right)$ bits. □

Note that the communication cost is low with respect to the stream size since the window size N should be chosen such that $N \gg 1$ (*e.g.,* in our tests we have $N = 1024$).

4. THEORETICAL ANALYSIS

Data streaming algorithms strongly rely on pseudo-random functions that map elements of the stream to uniformly distributed image values to keep the essential information of the input stream, regardless of the stream elements frequency distribution.

This section provides the analysis of the quality of the shedding performed by LAS in two steps. First we study the correctness and optimality of the shedding algorithm, under *full knowledge* assumption (*i.e.,* the shedding strategy is aware of the exact execution duration w_t for each tuple t). Then, in Section 4.2, we provide a probabilistic analysis of the mechanism that LAS uses to estimate the tuple execution durations. For the sake of clarity, and to abide to space limits, some of the proofs are available in a technical report paper [8].[4]

4.1 Correctness of LAS

We suppose that tuples cannot be preempted, that is they must be processed in an uninterrupted fashion on the available operator instance. As mentioned before, in this analysis we assume that the execution duration $w(t)$ is known for each tuple t. Finally, given our system model, we consider the problem of minimizing d, the number of dropped tuples, while guaranteeing that the average queuing latency $\overline{Q}(t)$ will be upper-bounded by τ, $\forall t \in \sigma$. The solution must work online, thus the decision of enqueueing or dropping a tuple has to be made only resorting to knowledge about tuples received so far in the stream.

Let OPT be the online algorithm that provides the optimal solution to Problem 2.1. We denote with $\mathcal{D}^{\sigma}_{OPT}$ (resp. d^{σ}_{OPT}) the set of dropped tuple indices (resp. the number of dropped tuples) produced by the OPT algorithm fed by stream σ (*cf.,* Section 2). We also denote with d^{σ}_{LAS} the number of dropped tuples produced by LAS introduced in Section 3.3 fed with the same stream σ.

THEOREM 4.1 (CORRECTNESS AND OPTIMALITY OF LAS). *For any σ, we have $d^{\sigma}_{\text{LAS}} = d^{\sigma}_{OPT}$ and $\forall t \in \sigma, \overline{Q}^{\sigma}_{\text{LAS}}(t) \leq \tau$.*

PROOF. Given a stream σ, consider the sets of indices of tuples dropped by respectively OPT and LAS, namely $\mathcal{D}^{\sigma}_{OPT}$ and $\mathcal{D}^{\sigma}_{\text{LAS}}$. Below, we prove by contradiction that $d^{\sigma}_{\text{LAS}} = d^{\sigma}_{OPT}$.

Assume that $d^{\sigma}_{\text{LAS}} > d^{\sigma}_{OPT}$. Without loss of generality, we denote $i_1, \dots, i_{d^{\sigma}_{\text{LAS}}}$ the ordered indices in $\mathcal{D}^{\sigma}_{\text{LAS}}$, and $j_1, \dots, j_{d^{\sigma}_{OPT}}$ the ordered indices in $\mathcal{D}^{\sigma}_{OPT}$. Let us define a as the largest natural integer such that $\forall \ell \leq a, i_\ell = j_\ell$ (*i.e.,* $i_1 = j_1, \dots, i_a = j_a$). Thus, we have $i_{a+1} \neq j_{a+1}$.

- Assume that $i_{a+1} < j_{a+1}$. Then, according to Section 3.3, the i_{a+1}-th tuple of σ has been dropped by LAS as the method CHECK returned *true.* Thus, as $i_{a+1} \notin \mathcal{D}^{\sigma}_{OPT}$, the OPT run has enqueued this tuple violating the constraint τ. But this is in contradiction with the definition of OPT.

- Assume now that $i_{a+1} > j_{a+1}$. The fact that LAS does not drop the j_{a+1} tuple means that CHECK returns *false*, thus that tuple does not violate the constraint on

[4]This version of the technical report has been anonymized to comply with the double-blind review process enforced by this conference.

τ. However, as OPT is optimal, it may drop some tuples for which CHECK is *false*, just because this allows it to drop an overall lower number of tuples. Therefore, if it drops this j_{a+1} tuple, it means that OPT knows the future evolution of the stream and takes a decision on this knowledge. But, by assumption, OPT is an online algorithm, and the contradiction follows.

Then, we have that $i_{a+1} = j_{a+1}$. By induction, we iterate this reasoning for all the remaining indices from $a+1$ to d^{σ}_{OPT}. We then obtain that $\mathcal{D}^{\sigma}_{OPT} \subseteq \mathcal{D}^{\sigma}_{\text{LAS}}$.

As by assumption $d^{\sigma}_{OPT} < d^{\sigma}_{\text{LAS}}$, we have that $\exists \ell \in \mathcal{D}^{\sigma}_{\text{LAS}} \setminus \mathcal{D}^{\sigma}_{OPT}$ such that ℓ has been dropped by LAS. This means that, with the same tuple index prefix shared by OPT and LAS, the method CHECK returned *true* when evaluated on ℓ, and OPT would violate the condition on τ by enqueuing it. That leads to a contradiction. Then, $\mathcal{D}^{\sigma}_{\text{LAS}} \setminus \mathcal{D}^{\sigma}_{OPT} = \emptyset$, and $d^{\sigma}_{OPT} = d^{\sigma}_{\text{LAS}}$.

Furthermore, by construction, LAS never enqueues a tuple that violates the condition on τ because CHECK would return *true*. Consequently, $\forall t \in \sigma, \overline{Q}^{\sigma}_{\text{LAS}}(t) \leq \tau$, which concludes the proof. $\square$

4.2 Execution Duration Estimation

In this section, we analyze the approximation made on execution duration $w(t)$ for each tuple t when the assumption of full knowledge is removed. LAS uses two matrices, $\mathcal{F}$ and $\mathcal{W}$, to estimate the execution time $w(t)$ of each tuple submitted to the operator. By the `Count Min` sketch algorithm (*cf.,* Section 3.2) and Listing 3.1, we have that for any $t \in [n]$ and for each row $i \in [r]$,

$$\mathcal{F}[i][h_i(t)] = f_t + \sum_{u=1, u\neq t}^{n} f_u 1_{\{h_i(u)=h_i(t)\}}.$$

and

$$\mathcal{W}[i][h_i(t)] = f_t w(t) + \sum_{u=1, u\neq t}^{n} f_u w(u) 1_{\{h_i(u)=h_i(t)\}},$$

Let us denote respectively by $w_{\min}$ and $w_{\max}$ the minimum and the maximum execution durations. We trivially have

$$w_{\min} \leq \frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]} \leq w_{\max} \qquad (2)$$

Let assume that all the frequencies are equal[5], that is for each t, we have $f_t = m/n$. Let us define $T = \sum_{t=1}^{n} w(t)$. We then have

$$\mathbb{E}\left\{\frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]}\right\} = \frac{T - w(t)}{n-1} - \frac{c(T - n \times w(t))}{n(n-1)}\left(1 - \left(1 - \frac{1}{c}\right)^n\right)$$

From the Markov inequality we have, for every $x > 0$,

$$\Pr\left\{\frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]} \geq x\right\} \leq \frac{\mathbb{E}\left\{\frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]}\right\}}{x}.$$

By the independence of the r hash functions, we obtain

$$\Pr\left\{\min_{i=1,\dots,r} \frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]} \geq x\right\} \leq \left(\Pr\left\{\frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]} \geq x\right\}\right)^r \leq \left(\frac{\mathbb{E}\left\{\frac{\mathcal{W}[i][h_i(t)]}{\mathcal{F}[i][h_i(t)]}\right\}}{x}\right)^r.$$

[5]The experimental evaluation (*cf.,* Section 5.2) points out that uniform or lightly skewed distributions represent worst cases for our solution.

The proofs of these equations as well as some numerical applications to illustrate the accuracy are discussed in [8]. By finely tuning the parameter r to $\log(1/\delta)$, under the assumption of [8], we are then able to (ε, δ)-approximate $w(t)$ for any $t \in [n]$. Then, according to Theorem 4.1, LAS is an (ε, δ)-optimal algorithm for load shedding, as defined in Problem 2.1, over all possible data streams σ.

5. EXPERIMENTAL EVALUATION

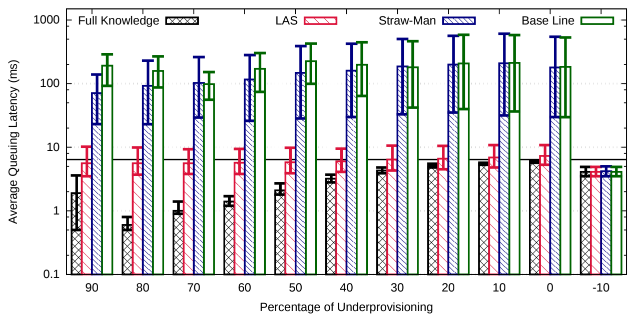
In this section we evaluate the performance obtained by using LAS to perform load shedding. We will first describe the general setting used to run the tests and will then discuss the results obtained through simulations (Section 5.2) and with a prototype of LAS integrated within Apache Storm (Section 5.3).

5.1 Setup

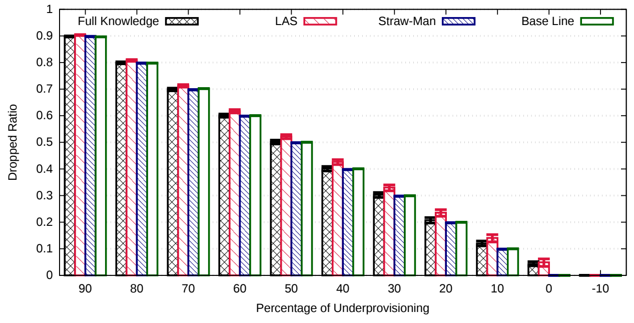
Datasets — In our tests we consider both synthetic and real datasets. Synthetic datasets are built as streams of integer values (items) representing the values of the tuple attribute driving the execution duration when processed on the operator. We consider streams of $m = 32,768$ tuples, each containing a value chosen among $n = 4,096$ distinct items. Streams have been generated using the Uniform and Zipfian distributions with different values of $\alpha \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0\}$, denoted respectively as Zipf-0.5, Zipf-1.0, Zipf-1.5, Zipf-2.0, Zipf-2.5, and Zipf-3.0. We define w_n as the number of distinct execution duration values that the tuples can have. These w_n values are selected at *constant* distance in the interval $[w_{min}, w_{max}]$. We ran experiments with $w_n\{1, 2, \cdots, 64\}$, however, due to space constraints, we only report results for $w_n = 64$, and with $w_{max} \in \{0.1, 0.2 \cdots, 51.2\}$ milliseconds. Tests performed with different values for w_n did not show unexpected deviations from what is reported in this section. Unless otherwise specified, the frequency distribution is Zipf-1.0 and the stream parameters are set to $w_n = 64$, $w_{min} = 0.1$ ms and $w_{max} = 6.4$ ms; this means that the $w_n = 64$ execution durations are picked in the set $\{0.1, 0.2, \cdots, 6.4\}$ milliseconds.

Let $\overline{W}$ be the average execution duration of the stream tuples, then the stream maximum theoretical input throughput sustainable by the setup is equal to $1/\overline{W}$. When fed with an input throughput smaller than $1/\overline{W}$ the system will be over-provisioned (*i.e.,* possible underutilization of computing resources). Conversely, an input throughput larger than $1/\overline{W}$ will result in an underprovisioned system. We refer to the ratio between the maximum theoretical input throughput and the actual input throughput as the percentage of underprovisioning that, unless otherwise stated, was set to 25%.

In order to generate 100 different streams, we randomize the association between the w_n execution duration values and the n distinct items: for each of the w_n execution duration values we pick *uniformly* at random n/w_n different values in $[n]$ that will be associated to that execution duration value. This means that the 100 different streams we use in our tests do not share the same association between execution duration and item as well as the association between frequency and execution duration (thus each stream has also a different average execution duration $\overline{W}$). Each of these permutations has been run with 50 different seeds to randomize the stream ordering and the generation of the



(a) Average queuing latency $\overline{Q}$



(b) Dropped ratio α

Figure 4: LAS performance varying the amount of underprovisioning.

hash functions used by LAS. This means that each single experiment reports the mean outcome of 5,000 independent runs.

We considered two types of constraints defined on the queuing latency:

ABS(τ): requires that the queuing latency for each tuple is at most τ milliseconds: $\forall i \in [m] \setminus D, q(i) \leq \tau$

AVG(τ): requires that the total average queuing latency does not exceeds τ milliseconds: $\forall i \in [m] \setminus D, \overline{Q}(i) \leq \tau$

While not being a realistic requirement, the straightforwardness of the ABS(τ) constraint allowed us to grasp a better insight of the algorithms mechanisms. However, in this section we only show results for the AVG(6.4) constraint as is it a much more sensible requirement with respect to a real setting.

The LAS operator window size parameter N, the tolerance parameter μ and the number of rows of the $\mathcal{F}$ and $\mathcal{W}$ matrices δ are respectively set to $N = 1024$, $\mu = 0.05$ and $\delta = 0.1$ (*i.e.,* $r = 4$ rows). By default, the LAS precision parameter (*i.e.,* , the number of columns of the $\mathcal{F}$ and $\mathcal{W}$ matrices) is set to $\varepsilon = 0.05$ (*i.e.,* $c = 54$ columns), however in one of the test we evaluated LAS performance using several values: $\varepsilon \in [0.001, 1.0]$.

For the real data, we used a dataset containing a stream of preprocessed tweets related to the 2014 European elections. Among other information, the tweets are enriched with a field *mention* containing the *entities* mentioned in the tweet. These entities can be easily classified into *politicians*, *media* and *others*. We consider the first 500,000 tweets, mentioning roughly $n = 35,000$ distinct entities and where the most frequent entity has an empirical probability of occurrence equal to 0.065.

Tested Algorithms —We compare LAS performance against three other algorithms:

Base Line The Base Line algorithm takes as input the percentage of under-provisioning and drops at random an equivalent fraction of tuples from the stream.

Straw-Man The Straw-Man algorithm uses the same shedding strategy of LAS, however it uses the average execution duration $\overline{W}$ as the estimated execution duration $\hat{w}(t)$ for each tuple t.

Full Knowledge The Full Knowledge algorithm uses the same shedding strategy of LAS, however it feeds it with the exact execution duration w_t for each tuple t.

Evaluation Metrics —The evaluation metrics we provide, when applicable, are

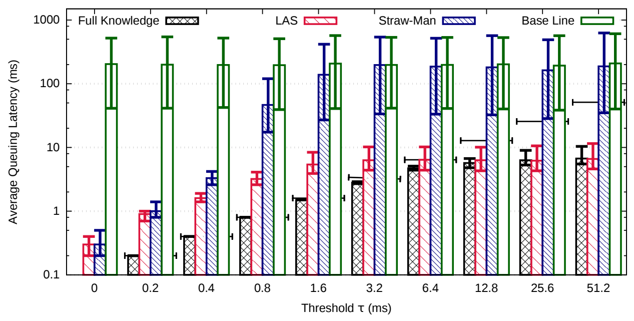
- the dropped ratio $\alpha = d/m$.
- the ratio of tuples dropped by algorithm alg with respect to Base Line: $\lambda = (d^{alg} - d^{\text{Base Line}})/d^{\text{Base Line}}$. In the following we refer this metric as shedding ratio.
- the average queuing latency $\overline{Q} = \sum_{i\in[m]\setminus\mathcal{D}} q(i)/(m - d)$.
- the average completion latency, *i.e.,* the average time it takes for a tuple from the moment it is injected by the source in the topology, till the moment operator O concludes its processing.

Whenever applicable we provide the maximum, mean and minimum figures over the 5,000 runs.

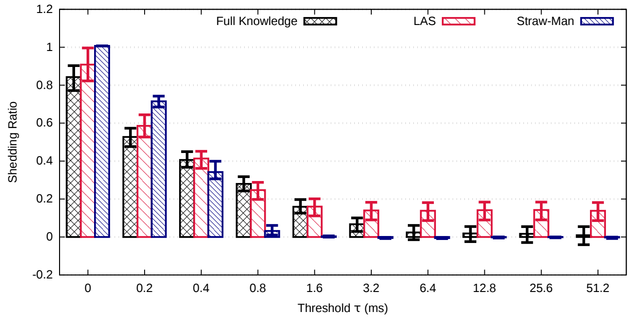
5.2 Simulation Results

In this section we analyze, through a simulator built adhoc for this study, the sensitivity of LAS while varying several characteristics of the input load. The simulator faithfully simulates the execution of LAS and the other algorithms and simulates the execution of each tuple t on O doing busy waiting for $w(t)$ milliseconds.

Input Throughput — Figure 4 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio α (right) as a function of the percentage of under-provisioning ranging from 90% to -10% (*i.e.,* the system is 10% overprovisioned with respect to the average input throughput). As expected, in this latter case all algorithms perform at the same level as load shedding is superfluous. In all the other cases both Base Line and Straw-Man do not shed enough load and induce a huge amount of exceeding queuing latency. On the other hand, LAS average queuing latency is quite close to the required value of $\tau = 6.4$ milliseconds, even if this threshold is violated in some of the tests. Finally, Full Knowledge always abide to the constraint and is even able to produce a much lower average queuing latency while dropping no more tuples that the competing solutions. Comparing the two plots we can clearly see that the resulting average queuing latency is strongly linked to which tuples are dropped. In particular, Base Line and Straw-Man shed the same amount of tuples, LAS slightly more and Full Knowledge is in the middle. This

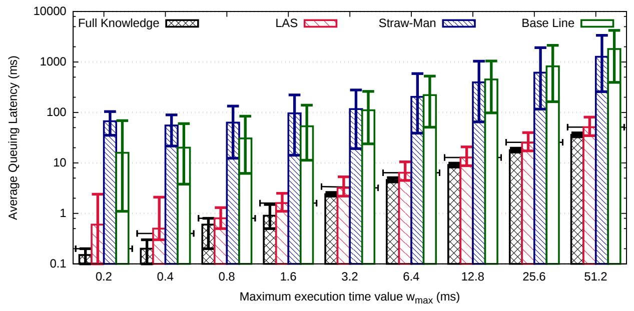


(a) Average queuing latency $\overline{Q}$

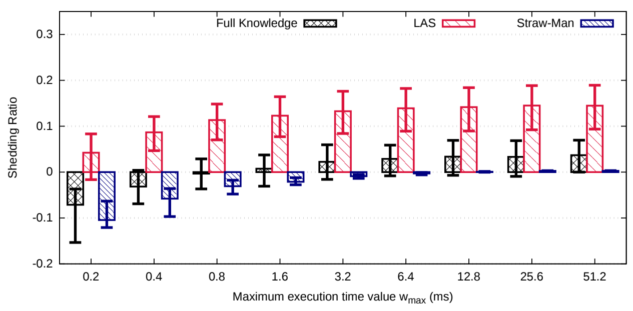


(b) Shedding ratio λ

Figure 5: LAS performance varying the threshold τ.



(a) Average queuing latency $\overline{Q}$



(b) Shedding ratio λ

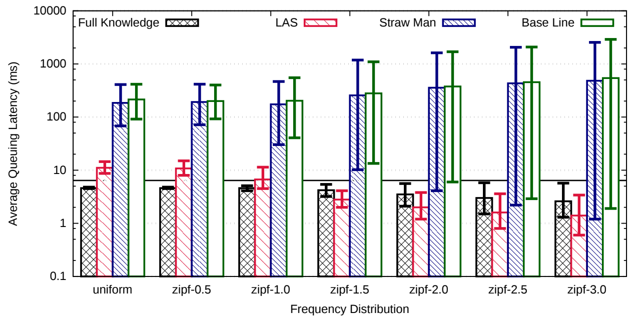
Figure 6: LAS performance varying the maximum execution duration value w_{max}.

result corroborates our initial claim that dropping tuples on the basis of the load they impose allows to design more effective load shedding strategies.

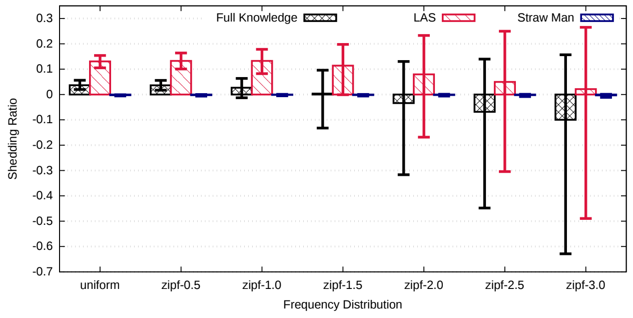
Threshold τ — Figure 5 shows the average queuing latency $\overline{Q}$ (left) and shedding ratio λ (right) as a function of the τ threshold. Notice that with $\tau = 0$ we do not allow any queuing, while with $\tau = 6.4$ we allow at least a queuing latency equal to the maximum execution duration w_{max}. In other words, we believe that with $\tau < 6.4$ the constraint is strongly conservative, thus representing a difficult scenario for any load shedding solution. Since Base Line does not take into account the latency constraint τ it always drops the same amount of tuples and achieves a constant average queueing latency. For this reason Figure 5b reports the shedding ratio λ achieved by Full Knowledge, LAS and Straw-Man with respect to Base Line. The horizontal segments in Figure 5a represent the distinct values for τ. As the graph shows Full Knowledge always perfectly approaches the latency threshold, but for $\tau = 12.8$ where it is slightly smaller. Straw-Man performs reasonably well when the threshold is very small, but this is a consequence of the fact that it drops a large number of tuples when compared with Base Line as can be seen by Figure 5b. However, as τ becomes larger (*i.e.,* $\tau \geq 0.8$) Straw-Man average queuing latency quickly grows and approaches the one from Base Line as it starts to drop the same amount of tuples. LAS, in the same setting performs largely better, with the average queuing latency that for large values of τ approaches the one provided by Full Knowledge. While delivering these performance LAS drops a slightly larger amount of tuples with respect to Full Knowledge, to account for the approximation in calculating tuple execution durations.

Maximum execution duration value w_{max} — Figure 6 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio λ (right) as a function of the maximum execution duration value w_{max}. Notice that in this test we varied the value for τ setting it equal to w_{max}. Accordingly, Figure 6a shows horizontal lines that mark the different thresholds τ. As the two graphs show, the behavior for LAS is rather consistent while varying w_{max}; this means that LAS can be employed in widely different settings where the load imposed by tuples in the operator is not easily predictable. The price paid for this flexibility is in the shedding ratio that, as shown in Figure 6b is always positive.

Frequency Probability Distributions — Figure 7 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio λ (right) as a function of the input frequency distribution. As Figure 7a shows Straw-Man and Base Line perform invariably bad with any distribution. The span between the best and worst performance per run increases as we move from a uniform distribution to more skewed distributions as the latter may present extreme cases where tuple latencies match their frequencies in a way that is particularly favorable or unfavorable for these two solutions. Conversely, LAS performance improve the more the frequency distribution is

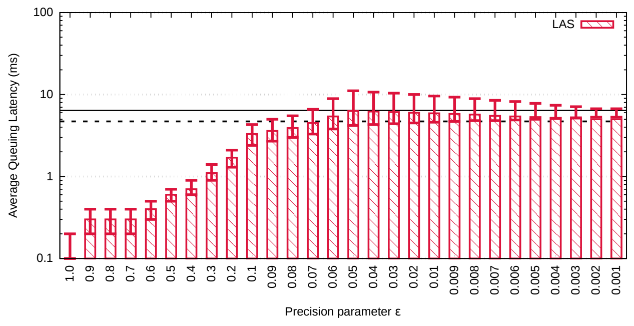


(a) Average queuing latency $\overline{Q}$

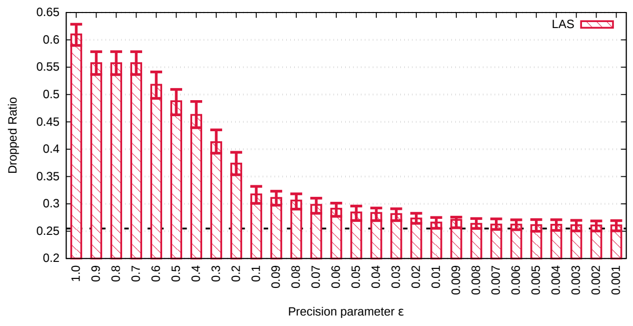


(b) Shedding ratio λ

Figure 7: LAS performance varying the frequency probability distributions.



(a) Average queuing latency $\overline{Q}$



(b) Dropped ratio α

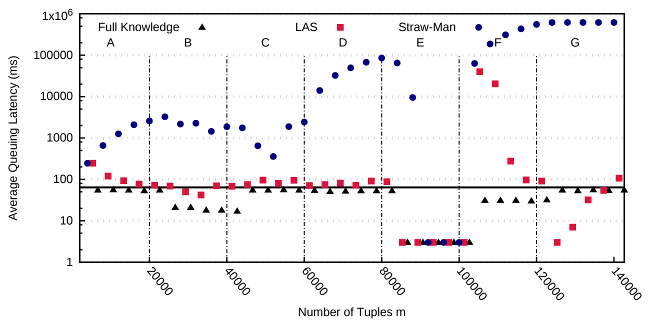
Figure 8: LAS performance varying the precision parameter ε.

skewed. This result stems from the fact that the sketch data structures used to trace tuple execution durations perform at their best on strongly skewed distribution, rather than on uniform ones. This result is confirmed by looking at the shedding ratio (Figure 7b) that decreases, on average, as the value of α for the distribution increases.

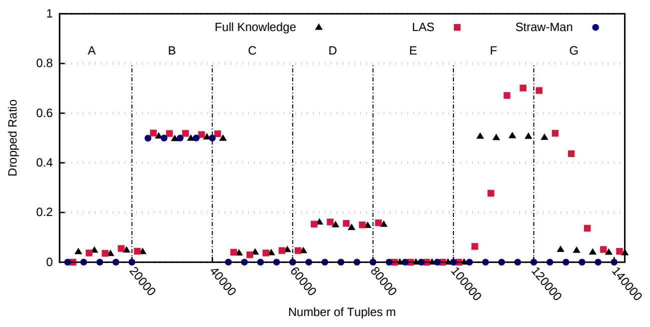
Precision parameter ε — Figure 8 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio α (right) as a function of the precision parameter ε. This parameter controls the trade-off between the precision and the space complexity of the sketches maintained by LAS. As a consequence it has an impact on LAS performance. In particular, for large values of ε (left side of the graph), the sketch data structures are extremely small, thus the estimation $\hat{w}(t)$ is extremely unreliable. The corrective factor $1 + \varepsilon$ (see Listing 3.2 line 23) in this case is so large that it pushes LAS to largely overestimate the execution duration of each tuple. As a consequence LAS drops a large number of tuples while delivering average queuing latencies that are close to 0. By decreasing the value of ε (*i.e.,* $\varepsilon \leq 0.1$), sketches become larger and their estimation more reliable. In this configuration LAS performs at its best delivering average queuing latencies that are always below or equal to the threshold $\tau = 6.4$ while dropping a smaller number of tuples. The dotted lines in both graphs represent the performance of Full Knowledge and are provided as a reference.

Time Series — Figure 9 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio α (right) as the stream unfolds (x-axis). Both metrics are computed on a jumping window of 4.000 tuples, *i.e.,* each dot represent the mean queuing latency $\overline{Q}$ or the dropped ratio α computed on the previous 4.000 tuples. Notice that the points for Straw-Man, LAS and Full Knowledge related to a same value of the x-axis are artificially shifted to improve readability. In this test we set $\tau = 64$ milliseconds. The input stream is made of 140.000 tuples and is divided in phases, from a A through G, each lasting 20.000 tuples. At the beginning of each phase we inject an abrupt change in the input stream throughput and distribution, as well as in $w(t)$ as follows:

phase A : the input throughput is set in accordance with the provisioning (*i.e.,* 0% underprovisioning);
phase B : the input throughput is increased to induce 50% of underprovisioning;
phase C : same as phase A;
phase D : we swap the most frequent tuple 0 with a less frequent tuple t such that $w(t) = w_{max}$, inducing an abrupt change in the tuple values frequency distribution and in the average execution duration $\overline{W}$;
phase E : the input throughput is reduced to induce 50% of overprovisionig;
phase F : the input throughput is increased back to 0% underprovisioning and we also double the execution duration $w(t)$ for each tuple, simulating a change in the operator resource availability;
phase G : same as phase A.

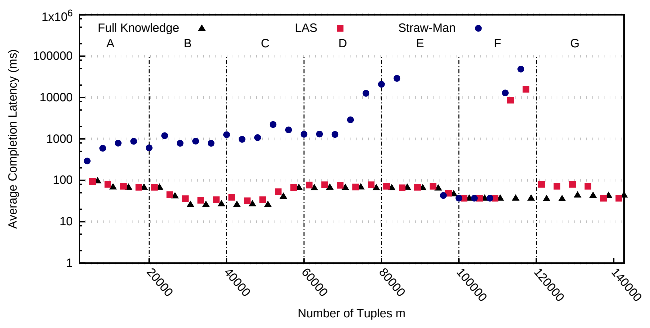


(a) Average queuing latency $\overline{Q}$

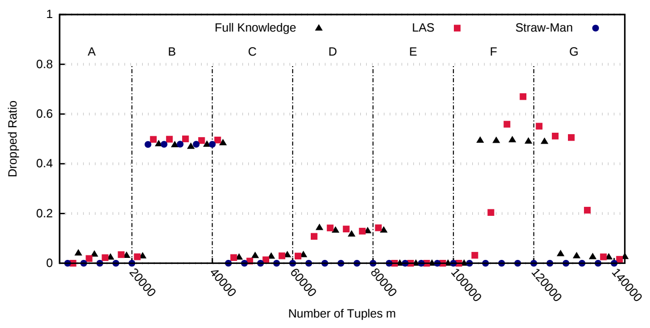


(b) Dropped ratio α

Figure 9: Simulator time-series.



(a) Average completion latency



(b) Dropped tuples d

Figure 10: Prototype time-series

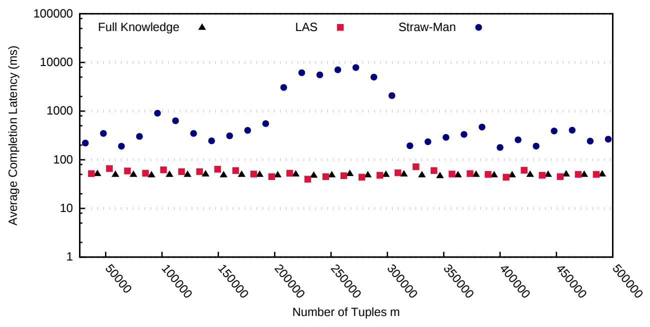
As the graphs show, during phase A the queuing latencies of LAS and Straw-Man diverge: while LAS quickly approaches the performance provided by Full Knowledge, Straw-Man average queuing latencies quickly grow. In the same timespan, both Full Knowledge and LAS drop slightly more tuples than Straw-Man. All the three solutions correctly manage phase B: their average queuing latencies see slight changes, while, correctly, they start to drop larger amounts of tuples to compensate for the increased input throughput. The transition to phase C brings the system back in the initial configuration, while in phase D the change in the tuple frequency distribution is managed very differently by each solution: both Full Knowledge and LAS compensate this change by starting to drop more tuples, but still maintaining the average queuing latency close to the desired threshold τ. Conversely, Straw-Man can't handle such change, and its performance incur a strong deterioration as it drops still the same amount of tuples. In phase E the system is strongly overprovisioned, and, as it was expected, all three solution perform equally well as no tuple needs to be dropped. The transition to phase F is extremely abrupt as the input throughput is brought back to the equivalent of 0% of underprovisioning, but the cost to handle each tuple on the operator is doubled. At the beginning of this phase both Straw-Man and LAS perform bad, with queuing latencies that are largely above τ. However, while the phase unfolds LAS quickly updates its data structures and converges toward the given threshold, while Straw-Man diverges as tuples continue to be enqueued on the operator worsening the bottleneck effect. Bringing back the tuple execution durations to the initial values in phase G has little effect on LAS, while the bottleneck created by Straw-Man cannot be recovered as it continues to drop an insufficient number of tuples.

5.3 Prototype

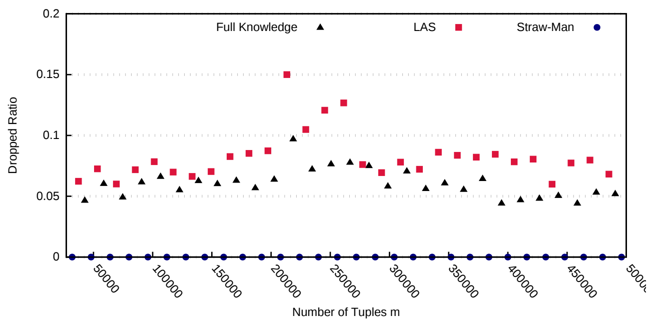
To evaluate the impact of LAS on real applications we implemented it as a bolt within the Apache Storm [12] framework. We have deployed our cluster on Microsoft Azure cloud service, using a Standard Tier A4 VM (4 cores and 7 GB of RAM) for each worker node, each with a single available slot.

The test topology is made of a source (*spout*) and two operators (*bolts*) LS and O. The source generates (reads) the synthetic (real) input stream and emits the tuples consumed by bolt LS. Bolt LS uses either Straw-Man, LAS or Full Knowledge to perform the load shedding on its outbound data stream consumed by bolt O. Finally operator O implements the logic.

Time Series — In this test we ran the simulator using the same synthetic load used for the time series discussed in the previous section. The goal of this test is to show how our simulated tests capture the main characteristic of a real run. Notice, however, that plots in Figure 10 report the average completion latency per tuple instead of the queuing latency. This is due to the difficulties in correctly measuring queuing latencies in Storm. Furthermore, the completion la-



(a) Average completion latency



(b) Dropped tuples d

Figure 11: Prototype use case

tency is, from a practical point of view, a more significant metric as it can be directly perceived on the output. From this standpoint the results, depicted in Figure 10, report the same qualitative behavior already discussed with Figure 9. Two main differences are worth to be discussed: firstly, the behaviors exposed by the shedding solution in response to phase transitions in the input load are in general shifted in time (with respect to the same effects reported in Figure 9) as a consequence of the general overhead induced by the software stack. Secondly, several data points for Straw-Man are missing in phases E and G. This is a consequence of failed tuples that start to appear as soon as the number of enqueued tuples is too large to be managed by Storm. While this may appear as a sort of "implicit" load shedding imposed by Storm, we decided not to consider these tuples in the metric calculation as they have not been dropped as a consequence of a decision taken by the Straw-Man load shedder.

Simple Application with Real Dataset — In this test we pretended to run a simple application on a real dataset: for each tweet of the twitter dataset mentioned in Section 5.1 we want to gather some statistics and decorate the outgoing tuples with some additional information. However the statistics and additional informations differ depending on the class the entities mentioned in each tweet belong. We assumed that this leads to a long execution duration for *media* (*e.g.,* possibly caused by an access to an external DB to gather historical data), an average execution duration for *politicians* and a fast execution duration for *others* (*e.g.,* possibly because these tweets are not decorated). We modeled execution durations with 25 milliseconds, 5 milliseconds and 1 millisecond of busy waiting respectively. Each of the $500,000$ tweets may contain more than one mention, leading to $w_n = 110$ different execution duration values from $w_{min} = 1$ millisecond to $w_{max} = 152$ milliseconds, among which the most frequent (36% of the stream) execution duration is 1 millisecond. The average execution time $\overline{W}$ is equal to 9.7 millisecond, the threshold τ is set to 32 milliseconds and the under-provisioning is set to 0%.

Figure 11 reports the average completion latency (left) and dropped ratio λ (right) as the stream unfolds. As the plots show, LAS provides completion latencies that are extremely close to Full Knowledge, dropping a similar amount of tuples. Conversely, Straw-Man completion latencies are at least one order of magnitude larger. This is a consequence of the fact that in the given setting Straw-Man does not drop tuples, while Full Knowledge and LAS drop on average a steady amount of tuples ranging from 5% to 10% of the stream. These results confirm the effectiveness of LAS in keeping a close control on queuing latencies (and thus provide more predictable performance) at the cost of dropping a fraction of the input load.

6. RELATED WORK

Aurora [1] is the first stream processing system where shedding has been proposed as a technique to deal with bursty input traffic. Aurora employs two different kinds of shedding, the first and better detailed being random tuple dropping at specific places in the application topology.

A large number of works has proposed solutions aimed at reducing the impact of load shedding on the quality of the system output. These solutions falls under the name of *semantic* load shedding, as drop policies are linked to the significance of each tuple with respect to the computation results. Tatbul et al. first introduced in [11] the idea of semantic load shedding. Het et al. in [5] specialized the problem to the case of complex event processing. Babcock et al. in [2] provided an approach tailored to aggregation queries. Finally, Tatbul et al. in [10] ported the concept of semantic load shedding in the realm of DSPS. All the previous works are based on a same goal, *i.e.,* to reduce the impact of load shedding on the semantics of the queries deployed in the stream processing system, while avoiding overloads. We believe that avoiding an excessive degradation in the performance of the DSPS and in the semantics of the deployed query output are two orthogonal facets of the load shedding problem. In our work we did not consider the latter and focused on the former. The integration of the two approaches is left for future work.

To the best of our knowledge, all these works assume that each tuple induces the same load in the system, independently from their content.

A different approach has been proposed in [9], with a system that build summaries of dropped tuples to later produce approximate evaluations of queries. The idea is that such approximate results may provide users with useful information about the contribution of dropped tuples.

A classical control theory approach based on a closed control loop with feedback has been considered in [7, 13, 14]. In all these works the focus is on the design of the loop controller, while data is shed using a simple random selection strategy. In all these cases the goal is to reactively feed the stream processing engine system with a bounded tuple rate, without proactively considering how much load these tuples will generate.

7. CONCLUSIONS

In this paper we introduced Load-Aware Shedding (LAS), a novel solution for load shedding in DSPS. LAS exploits a characteristics of many stream-based applications, *i.e.,* the fact that load on operators depends both on the input rate and on the content of tuples, to smartly drop tuples and avoid the appearance of performance bottlenecks. In particular, LAS leverages sketch data structures to efficiently collect at runtime information on the operator load characteristics and then use this information to implement a load shedding policy aimed at maintaining the average queuing latencies close to a given threshold. Through a theoretical analysis, we proved that LAS is an (ϵ, δ)-approximation of the optimal algorithm. Furthermore, we extensively tested LAS both in a simulated setting, studying its sensitivity to changes of several characteristics of the input load, and with a prototype implementation integrated within the Apache Storm DSPS. Our tests confirm that by taking into account the specific load imposed by each tuples, LAS can provide performance that closely approach a given target, while dropping a limited number of tuples.

8. REFERENCES

[1] 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 International Journal on Very Large Data Bases (VLDB Journal)*, 12(2):120–139, 2003.

[2] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, pages 350–361. IEEE, 2004.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18, 1979.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55, 2005.

[5] Y. He, S. Barman, and J. F. Naughton. On load shedding in complex event processing. In *Proceedings of the 17th International Conference on Database Theory (ICDT '14)*, pages 213–224. OpenProceedings.org, 2014.

[6] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak. Cloud-based data stream processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*, pages 238–245. ACM, 2014.

[7] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload management in data stream processing systems with latency guarantees. In *7th IEEE International Workshop on Feedback Computing (Feedback Computing'12)*, 2012.

[8] REDACTED. Proactive online scheduling for shuffle grouping in distributed stream processing systems. Technical report, REDACTED. Available at https://goo.gl/4FUzUx, 2015.

[9] 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 (ICDE '05)*, pages 155–156. IEEE, 2005.

[10] 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.

[11] 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 (VLDB '03)*, pages 309–320. VLDB Endowment, 2003.

[12] The Apache Software Foundation. Apache Storm. http://storm.apache.org.

[13] 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 (VLDB '06)*, pages 787–798. VLDB Endowment, 2006.

[14] Y. Zhang, C. Huang, and C. Huang. A novel adaptive load shedding scheme for data stream processing. In *Future Generation Communication and Networking (FGCN '07)*, pages 378–384. IEEE, 2007.