

# Load-aware shedding in stream processing systems

Nicoló Rivetti, Yann Busnel, Leonardo Querzoni

▶ **To cite this version:**

Nicoló Rivetti, Yann Busnel, Leonardo Querzoni. Load-aware shedding in stream processing systems. 10th ACM International Conference on Distributed and Event-based Systems (DEBS), Jun 2016, Irvine, United States. pp.61 - 68, 10.1145/2933267.2933311 . hal-01413212

**HAL Id: hal-01413212**
**https://inria.hal.science/hal-01413212**

Submitted on 12 Dec 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# 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 tuple queuing and system trashing. In this paper we propose Load-Aware Shedding (LAS), a novel load shedding solution that drops tuples with the aim of maintaining queuing times below a tunable threshold. Tuple execution durations are estimated at runtime using efficient sketch data structures. We provide a theoretical analysis proving that LAS is an $(\varepsilon, \delta)$-approximation of the optimal online load shedder and show its performance through a practical evaluation based both on simulations and on a running prototype.

## CCS Concepts

**•Software and its engineering → Distributed systems organizing principles; •Theory of computation →** *Online learning algorithms; Sketching and sampling;*

## 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 where data operators (nodes) are interconnected by streams of tuples containing data to be analyzed (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 critical, as input data streams may unpredictably change over time both in rate and content. 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.

Existing load shedding solutions 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 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 and possibly cause 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 goals, 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 and thus build and maintain 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 (i) 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

Accepted to the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)

any assumption on the distribution of tuples, (ii) a theoretical analysis of LAS that points out how it is an $(\epsilon, \delta)$-approximation of the optimal online shedding algorithm and, finally, (iii) 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.

Data injected by source operators 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. We also restrict our model to a topology with an operator $LS$ (*load shedder*) that decides which tuples of its outbound DS $\sigma$ consumed by operator $O$ shall be dropped. Tuples in $\sigma$ 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 $\sigma$. Both $m$ and $n$ are unknown. We denote with $f_t$ the unknown frequency of tuple $t$, *i.e.,* the number of occurrences of $t$ in $\sigma$.

We assume that the execution duration of tuple $t$ on operator $O$, denoted as $w(t)$, depends on the content of $t$. In particular, without loss of generality, we consider a case where $w$ depends on a single, fixed and known attribute value of $t$. The probability distribution of such attribute values, as well as $w$, 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.,* $d = |\mathcal{D}|$. 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 the average queuing latency smaller than a given threshold $\tau$ by dropping as less tuples as possible while the stream unfolds. The quality of the shedder can be evaluated both by comparing the resulting $\overline{Q}$ against $\tau$ and by measuring the number of dropped tuples $d$. More formally, the load shedding problem can be defined as follows[1]:

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

## 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 $t$ in the operator, then we can foresee queuing times and drop all tuples that will cause the queuing latency threshold $\tau$ to be violated. However, the value of $w(t)$ is generally unknown.

LAS builds and maintain 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 LAS 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 processes. 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 LAS design

The operator maintains two `Count Min` [4] 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 **2**-universal hash functions [3]. The latter is a generalized version of the `Count Min` providing $(\varepsilon, \delta)$-additive-approximation of point queries on stream of updates whose value is the tuple execution duration when processed by the instance. The operator will update (Listing 3.1 lines 27-30) both matrices after each tuple execution.

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

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

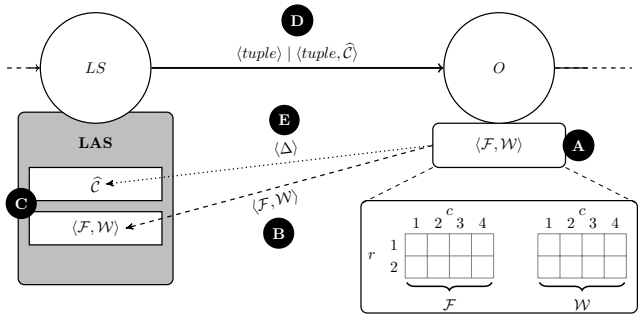



**Figure 1: Load-Aware Shedding design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$).**

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 $\eta$ between the previous snapshot and the current one is smaller than a configurable parameter $\mu$, *i.e.,*

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

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.

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).

Listing 3.1: Operator

```
1: init do
2:     F ← 0_{r,c}                          ▷ zero matrices of size r × c
3:     W ← 0_{r,c}
4:     S ← 0_{r,c}
5:     r hash functions h_1, ..., h_r : [n] → [c] from a 2-universal
   family.
6:     m ← 0
7:     state ← START
8: end init
9: function UPDATE(tuple : t, execution time : l, request : Ĉ)
10:    m ← m + 1
11:    if Ĉ not null then
12:        Δ ← C − Ĉ
13:        send ⟨Δ⟩ to LS
14:    end if
15:    if state = START ∧ m mod N = 0 then        ▷ Figure 2.A
16:        update S
17:        state ← STABILIZING
18:    else if state = STABILIZING ∧ m mod N = 0 then
19:        if η ≤ μ (Eq. 1) then                  ▷ Figure 2.C
20:            send ⟨F, W⟩ to LS
21:            state ← START
22:            reset F and W to 0_{r,c}
23:        else                                   ▷ Figure 2.B
24:            update S
25:        end if
26:    end if
27:    for i = 1 to r do
28:        F[i, h_i(t)] ← F[i, h_i(t)] + 1
29:        W[i, h_i(t)] ← W[i, h_i(t)] + l
30:    end for
31: end function
```

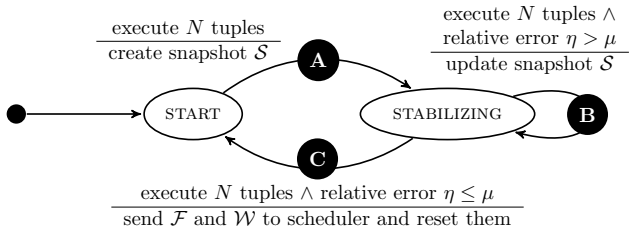



**Figure 2: Operator finite state machine.**

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 $\tau$. Then, CHECK tries to add $\hat{q}$ to the current average queuing latency (Listing 3.2 lines 31). If the result is larger than $\tau$ (**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.

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

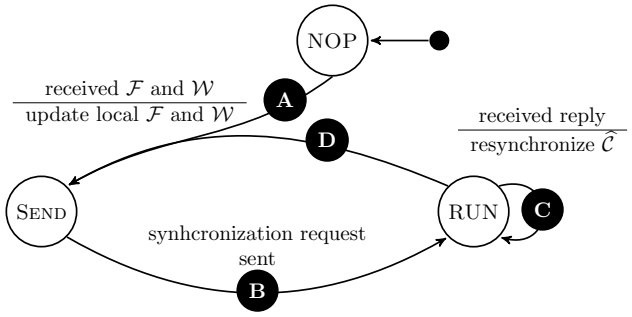



**Figure 3: Load shedder *LS* finite state machine.**

the estimated tuple execution duration $\hat{w}(t)$, increased by a factor $1+\varepsilon$ to mitigate potential under-estimations[2], 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* $\mathcal{O}(\log 1/\delta)$.

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.*

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.*

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 $N = 1024$). Proofs for the previous theorems are available in [8].

# 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. Here we provide 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 $t$). For the sake of clarity, and to abide to space limits, all the proofs are available in a companion paper [8].

We suppose that tuples cannot be preempted, i.e. they must be processed in an uninterrupted fashion on the available operator instance. Given our system model, here 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 $\tau$, $\forall t \in \sigma$. 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.

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

Listing 3.2: Load shedder

```
1: init do
2:     Ĉ ← 0
3:     ⟨F, W⟩ ← ⟨0_{r,c}, 0_{r,c}⟩      ▷ zero matrices pair of size r × c
4:     Same hash functions h_1 ... h_r of the operator
5:     state ← NOP
6: end init
7: upon ⟨F', W'⟩ do                          ▷ Figure 3.A and 3.D
8:     state ← SEND
9:     ⟨F, W⟩ ← ⟨F', W'⟩
10: end upon
11: upon ⟨Δ⟩ do                              ▷ Figure 3.C
12:    Ĉ ← Ĉ + Δ
13: end upon
14: function SHED(tuple : t)
15:    if state = NOP then
16:        return false
17:    end if
18:    q̂ ← Ĉ− elapsed time from first tuple
19:    if CHECK(q̂) then
20:        return true
21:    end if
22:    i ← arg min_{i∈[r]}{F[i, h_i(t)]}
23:    Ĉ ← Ĉ + (W[i, h_i(t)]/F[i, h_i(t)]) × (1 + ε)
24:    if state = SEND then                  ▷ Figure 3.B
25:        piggy back Ĉ to operator on t
26:        state ← RUN
27:    end if
28:    return false
29: end function
30: function CHECK(q)
31:    if Q/ℓ > τ then
32:        return true
33:    end if
34:    Q ← Q + q
35:    ℓ ← ℓ + 1
36:    return false
37: end function
```

Let OPT be the online algorithm that provides the optimal solution to Problem 2.1. We denote with $\mathcal{D}^{\sigma}_{OPT}$ (resp. $d^{\sigma}_{OPT}$) the set of dropped tuple indices (resp. the number of dropped tuples) produced by the OPT algorithm fed by stream $\sigma$ (*cf.,* Section 2). We also denote with $d^{\sigma}_{\text{LAS}}$ the number of dropped tuples produced by LAS introduced in Section 3.2 fed with the same stream $\sigma$.

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

Now we remove the previous assumptions and analyze the approximation made on execution duration $w(t)$ for each tuple $t$. 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 [4] 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

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

Let assume that all the frequencies are equal[3], 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

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

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 $(\varepsilon, \delta)$-approximate $w(t)$ for any $t \in [n]$. Then, according to Theorem 4.1, LAS is an $(\varepsilon, \delta)$-optimal algorithm for load shedding, as defined in Problem 2.1, over all possible data streams $\sigma$.

# 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 and with a prototype of LAS integrated within Apache Storm. Due to space constraints, the exhaustive presentation of these experiments are available in the companion paper [8].

## 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 $[w_{min}, w_{max}]$. 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.

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

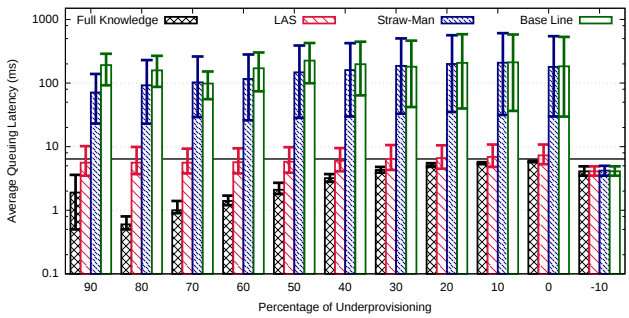



**Figure 4: Average queuing latency $\overline{Q}$ varying the amount of under-provisioning.**

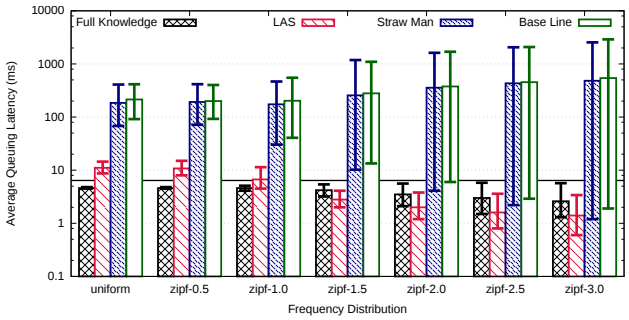
Let $\overline{W}$ be the average execution duration of the stream tuples, then the stream maximum theoretical input throughput sustainable by the setup is equal to $1/\overline{W}$. When fed with an input throughput smaller than $1/\overline{W}$ the system will be over-provisioned (*i.e.,* possible underutilization of computing resources). Conversely, an input throughput larger than $1/\overline{W}$ will result in an under-provisioned system. We refer to the ratio between the maximum theoretical input throughput and the actual input throughput as the percentage of under-provisioning 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 hash functions used by LAS. This means that each single experiment reports the mean outcome of $5,000$ independent runs.

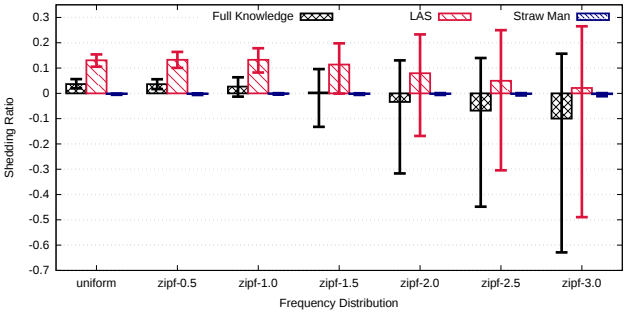
As constraints on the queuing latency we considered $\texttt{AVG}(\tau)$, thus requiring that the total average queuing latency does not exceeds $\tau$ milliseconds: $\forall i \in [m] \setminus D, \overline{Q}(i) \leq \tau$. In all our experiments we set $\tau = 6.4$.

The LAS operator window size parameter $N$, the tolerance parameter $\mu$ and the number of rows of the $\mathcal{F}$ and $\mathcal{W}$ matrices $\delta$ 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




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




(b) Shedding ratio $\lambda$

**Figure 5: LAS performance varying the frequency probability distributions.**

equal to 0.065.

**Tested Algorithms** —We compare LAS performance against three other algorithms: (i) a Base Line algorithm that takes as input the percentage of under-provisioning and drops at random an equivalent fraction of tuples from the stream, (ii) a Straw-Man algorithm that uses the same shedding strategy of LAS considering the average execution duration $\overline{W}$ as the estimated execution duration $\hat{w}(t)$ for each tuple $t$ and (iii) a Full Knowledge algorithm that uses the same shedding strategy of LAS feeding 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 shedding ratio $\lambda = (d^{alg} - d^{\text{Base Line}})/d^{\text{Base Line}}$ representing the ratio of tuples dropped by algorithm $alg$ with respect to Base Line, the average queuing latency $\overline{Q} = \sum_{i \in [m] \setminus \mathcal{D}} q(i)/(m - d)$ and 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

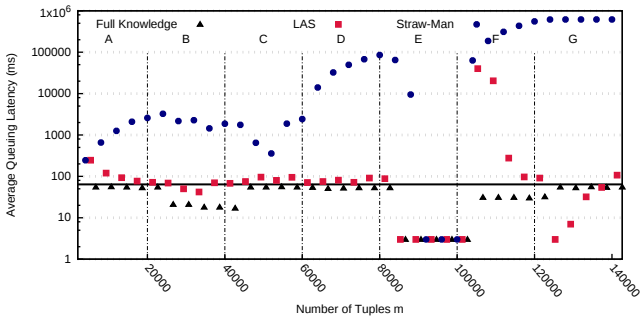
In this section we analyze through simulations the sensitivity of LAS while varying several characteristics of the input load. The simulator faithfully simulates the execution of LAS and the other algorithms and simulates the execution of each tuple $t$ on $O$ doing busy waiting for $w(t)$ milliseconds. Further results, not included in this paper, are available in [8].

**Input Throughput** — Figure 4 shows the average queuing latency $\overline{Q}$ as a function of the percentage of under-provisioning ranging from 90% to -10% (*i.e.,* the system is 10% over-provisioned 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. The resulting average queuing latency is strongly linked to which tuples are dropped: Base Line and Straw-Man, in particular, shed the same amount of tuples, LAS slightly more and Full Knowledge is in the middle. This result corroborates our initial claim that dropping tuples on the basis of the load they impose allows to design more effective load shedding strategies.

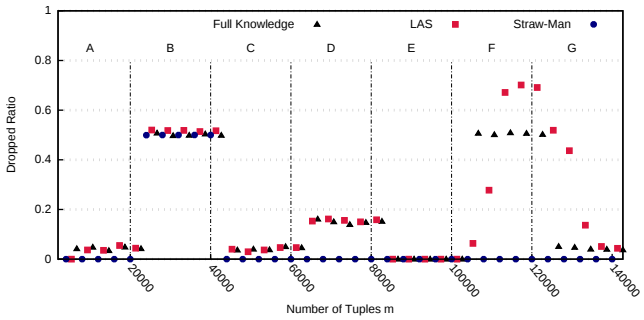
**Frequency Probability Distributions** — Figure 5 shows the average queuing latency $\overline{Q}$ (left) and shedding ratio $\lambda$ (right) as a function of the input frequency distribution. As Figure 5a shows Straw-Man and Base Line perform invariably bad with any distribution. The span between 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 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 5b) that decreases, on average, as the value of $\alpha$ for the distribution increases.

**Time Series** — Figure 6 shows the average queuing latency $\overline{Q}$ (left) and dropped ratio $\alpha$ (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 $\alpha$ 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% under-provisioning);
**phase B** : the input throughput is increased to induce 50% of under-provisioning;
**phase C** : same as phase A;




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




(b) Dropped ratio $\alpha$

**Figure 6: Simulator time-series.**

**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 over-provisionig;

**phase F** : the input throughput is increased back to 0% under-provisioning 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.

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 $\tau$. 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 over-provisioned, 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 under-provisioning, 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 $\tau$. 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 [11] 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 reads from the dataset 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.

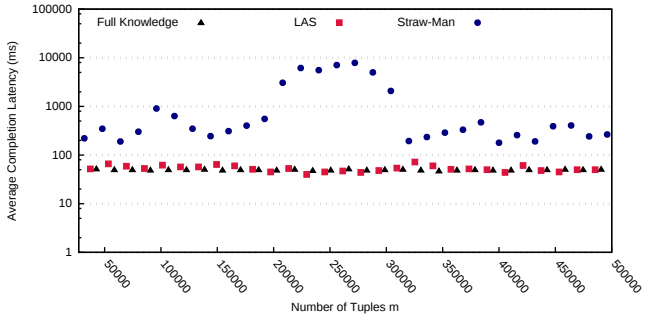



**Figure 7: Average completion latency in the prototype use case.**

**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 $\tau$ is set to 32 milliseconds and the under-provisioning is set to 0%.

Figure 7 reports the average completion latency as the stream unfolds. As the plot show, LAS provides completion latencies that are extremely close to Full Knowledge, while 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. A large number of works has proposed solutions aimed at reducing the impact of load shedding on the quality of the system output (*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 [10] 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 [9] ported the concept of semantic load shedding in the realm of DSPS. 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. 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 [7], 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 [6, 12, 13]. 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 is based on the observation that load on operators depends both on the input rate and on the content of tuplesLAS 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 $(\epsilon, \delta)$-approximation of the optimal algorithm. Furthermore, tests conducted both on a simulated environment and on a prototype implementation confirm that by taking into account the specific load imposed by each tuple, LAS can provide performance that closely approach a given target, while dropping a limited number of tuples.

# 8. ACKNOWLEDGMENT

This work has been partially founded by the French ANR project SocioPlug (ANR-13-INFR-0003) and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

# 9. 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*, 12(2), 2003.

[2] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 20th International Conference on Data Engineering*. 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 *Proc. of the 17th International Conference on Database Theory*. OpenProc..org, 2014.

[6] 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*, 2012.

[7] F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in TelegraphCQ. In *Proc. of the 21st International Conference on Data Engineering*. IEEE, 2005.

[8] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Load-aware shedding in stream processing systems. Technical report, Inria - Available at https://hal.inria.fr/hal-01311970, 2016.

[9] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proc. of the 33rd international conference on Very large data bases*, 2007.

[10] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th International Conference on Very Large Data Bases*. VLDB Endowment, 2003.

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

[12] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proc. of the 32nd International Conference on Very Large Data Bases*, 2006.

[13] Y. Zhang, C. Huang, and C. Huang. A novel adaptive load shedding scheme for data stream processing. In *Future Generation Communication and Networking*, 2007.