Efficient Stream Analysis and its Application to Big Data Processing

Nicolò Rivetti

UNIVERSITE BRETAGNE LOIRE

SAPIENZA UNIVERSITÀ DI ROMA

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Nicoló RIVETTI
DI VAL CERVO

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
Docteur de Sapienza University of Rome
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Date de soutenance 30 septembre 2016

Efficient Stream Analysis and its Application to Big Data Processing

JURY

Président :	**M. Roberto BALDONI**, Professor, Sapienza University of Rome, Italy
Rapporteurs :	**M. Graham CORMODE**, Professor, University of Warwick, United Kingdom **M. Maarten VAN STEEN**, Professor, University of Twente, Netherlands
Examinateurs :	**M. Sofian MAABOUT**, Maître de Conférences, Université de Bordeaux 1, France **M^me^ Clémence MAGNIEN**, Directrice de Recherche, CNRS, Paris, France
Invité :	**M. Peter PIETZUCH**, Associate Professor, Imperial College, London, United Kingdom
Directeurs de thèse :	**M. Achour MOSTÉFAOUI**, Professeur, Université de Nantes, France **M. Leonardo QUERZONI**, Assistant Professor, Sapienza University of Rome, Italy
Co-encadrant de thèse :	**M. Yann BUSNEL**, Maître de Conférences, Crest / Ensai - Inria, Rennes, France

Contents

Chapter 1

Introduction

In 1968, 140 IT companies where solicited to build the ARPANET network. Most of them regarded the project as outlandish and only 12 bid, while today there are 3.4 billions Internet users. The advent of Internet in the 90s, the commercial outbreak of cloud computing in the 2000s and the growing use of connected mobile devices since 2009 pushed the telecommunication and IT world into a new stage of interconnectivity. This phenomenon have given birth to new concepts, such as the Internet of Things (IoT) [14].

This bursting expansion also affected the available data, which has attained in 2009 the bewildering amount of roughly 50 petabytes, mostly user created. The trend has not stopped since, the global internet population has increased from 2.1 billions in 2012 to 3.4 billions in 2016 [37]. For instance, each minute these users do [37] millions of forecast requests on The Weather Channel [90] or millions of likes on Instagram, as well as stream and upload thousands of hours of video through services such as Netflix [74] or YouTube [48]. This tendency is bound to grow with time, 90% of the worldwide data has been created in the last 2 years and it is estimated that in 2018 we will generate 50 terabytes of data per second [92].

Researchers and practitioners understood that the massive amount of available data entailed new opportunities as well as challenges. In particular, common data management techniques where not tailored to match this features and, soon enough, the term *Big Data* emerged to capture all of these characteristics. Big Data is (at minima) defined by three Vs: large Volumes with high Variety and generated with extreme Velocity. This required an evolution of several research fields (*e.g.,* networks, distributed systems and databases) to accommodate these changes and develop new technologies, such as 5G, self-* automation, fog computing and NOSql. On the other hand, it also prompted for means to filter out the noise, as well as to better analyse and extract relevant information (*e.g.,* machine learning, data mining, *etc.*).

Initially, the focus was on volume, *i.e.,* on batch processing. In 2004, Dean and Ghemawat [35] proposed to apply the map-reduce paradigm to batch processing, springing the design of many frameworks (*e.g.,* Hadoop [87]) and languages (*e.g.,* Pig [88]). These solutions provided, at least initially, a satisfactory answer to the need of processing large, non-volatile, amount of data. At the same time, the huge diversity in the sources producing data had increased the need for data integration techniques, such as schema matching or entities resolution. It also boosted the interest in frameworks to better specify and share the information, such as the semantic web, as well as techniques to handle uncertain information.

While the two mentioned aspects are still paramount, in this thesis we focus on the third canonical dimension of Big Data: velocity. It is commonly acknowledged that the results achieved through the new batch processing and data analysis techniques are startling. However most data, and its analysis, has an intrinsic value that decreases over time. In other words, it is critical to move from off-line to on-line analysis. Only recently researchers and practitioners started to address this concern.

The *streaming* approach has at its core that the whole dataset must not/cannot be stored, taking into account that most likely the available memory is not enough. This means that all data is modelled as a sequence of data items, *i.e.,* a potentially infinite stream of tuples. Notice that most of the currently generated data (*e.g.,* RFID tracking, sensors readings, network traffic, news feeds, *etc.*) does fit this model. In addition, also datasets that may be processed through batch processing can be modelled as streams. Nowadays stream analysis is used in, but not restricted to, many context where the amount of data and/or the rate at which it is generated rules out other approaches (*e.g.,* batch processing). As such we have seen the rebound of established computer science fields that entailed such requirements (*e.g.,* data streaming) as well the rise of new fields (*e.g.,* stream processing).

The *data streaming* model [72] provides randomized and/or approximated solutions to compute specific functions over (distributed) stream(s) of items in worst case scenarios (*i.e.,* considering rather strong adversaries tampering with the streams), while striving for small (*i.e.,* sub-linear) resources usage (memory, cpu, network, etc.). The community has also proposed many extensions to the data streaming model, capturing more general scenarios. Among others, there are four classical and basic problems that have been studied for decades in this field: basic counting [34], estimating the size of the stream universe [16], the frequency estimation problem [27,30] and the (distributed) heavy hitters problem [27,30]. Notice that these simple statistical aggregates enable many other (more complex) analysis. As such, these algorithms have a large field of application, spanning from databases to networks, that are closely related to Big Data.

The aim of this thesis is to design novel algorithms in the data streaming model, or extend current solutions, to solve practical issues in Big Data applications. Considering our target, we identified some limitations that we wanted to address in two of the aforementioned problems: the *frequency estimation* problem, *i.e.,* providing an estimation of the number of occurrences for each distinct data item in the stream, and the *distributed heavy hitters* problem, *i.e.,* detecting the items which frequencies are larger than a given threshold.

Considering the former, in general previous solutions take into account the whole stream (*i.e.,* from inception). However in most application there is much more interest on the recent past (*e.g.,* last 24 hours). This requirement is captured by the *sliding window* model [34] and we propose algorithms designed in such model improving with respect to the state of the art. We also define and provide a solution to the item value estimation problem, *i.e.,* a generalization of frequency estimation problem.

Current solutions for the distributed variant of the heavy hitters problem rely on strong assumptions [66, 93, 96, 97]. In this thesis we design a novel algorithm solving the distributed heavy hitters problem and lift these assumptions. We also look into a problem that may not seem related, but whose efficient solution requires the identification of heavy hitters: build a balanced partition of the stream universe considering weights. As shown later, these algorithms can be successfully leveraged to solve major concerns in Big Data applications.

Naturally, we started to apply our algorithms to network monitoring, one of the classical fields of application of data streaming. In particular we are able to monitor computer networks and detect ongoing attacks leveraging our solutions for the distributed heavy hitters problem and for the sliding window frequency estimation problem.

While network monitoring has, by definition, to handle large amount of data flows, we wanted to look into an application tightly coupled with Big Data. *Stream processing* [40] is somehow the offspring of the database and the complex event processing communities. With respect to data streaming, it looks like a complementary and more practical field providing efficient and highly scalable frameworks (stream processing systems) to perform soft real-time generic computation on streams, relying on cloud computing solution. Surprisingly enough, while both data streaming and stream processing share a common community (*i.e.,* database), there are not many applications of data streaming to stream processing. However, we believed that the duality between these two fields is a strong indicator that they can benefit each other.

Among the plethora of possible optimizations in stream processing systems [56], in this thesis we focus on *load balancing* and *load shedding*. Both are well known problems that have been studied for decades in different communities. Load balancing aims to evenly spread the workload on the available resources, while load shedding goal is to prevent the systems from being overwhelmed, dropping some of the input data. In other words, the former strives to maximise the resources usage and the latter guarantees that the resources are enough to handle the workload. In this thesis we apply two novel data streaming algorithms to these two optimizations in stream processing systems.

Contributions Overview

We strongly believe that when designing a new algorithm, even if we are able to argue its usefulness and provide good bounds on its quality and performance, it is still important to validate the motivation and theoretical results with a real world application and extensive experimental evaluation. As such, all of this thesis contributions contain both aspects. Figure 1.1 summarizes the contributions.

Distributed heavy hitters — The current solutions to the detection of distributed heavy hitters are either non space efficient [93], or rely on some strong assumptions [66, 96, 97]. In this thesis we present Distributed Heavy Hitters Estimator (DHHE) [9] [1], an algorithm designed to estimate the frequencies of distributed heavy hitters on the fly with guaranteed error bounds, limited memory and processing requirements. In addition, DHHE does not rely on any assumption on the frequency distribution of the distributed stream. The theoretical analysis of DHHE partially relies on the results presented in [8] [2]. We use DHHE to detect distributed denial of service (DDoS), an infamous problem in network security. The experimental evaluation shows the quality of our algorithm and confirms the theoretical analysis.

Balanced partitioning — In general, systems handling Big Data are distributed and parallel, and a well known problem in this fields is load balancing [98]. Considering streams, we formalise load balancing as building a balanced partition of the stream universe considering weights, *i.e.,* the balanced partitioning problem. In this thesis we

[1] A shorter version has been published at the 18ème Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Algotel 2016)

[2] This thesis does not cover this contribution.

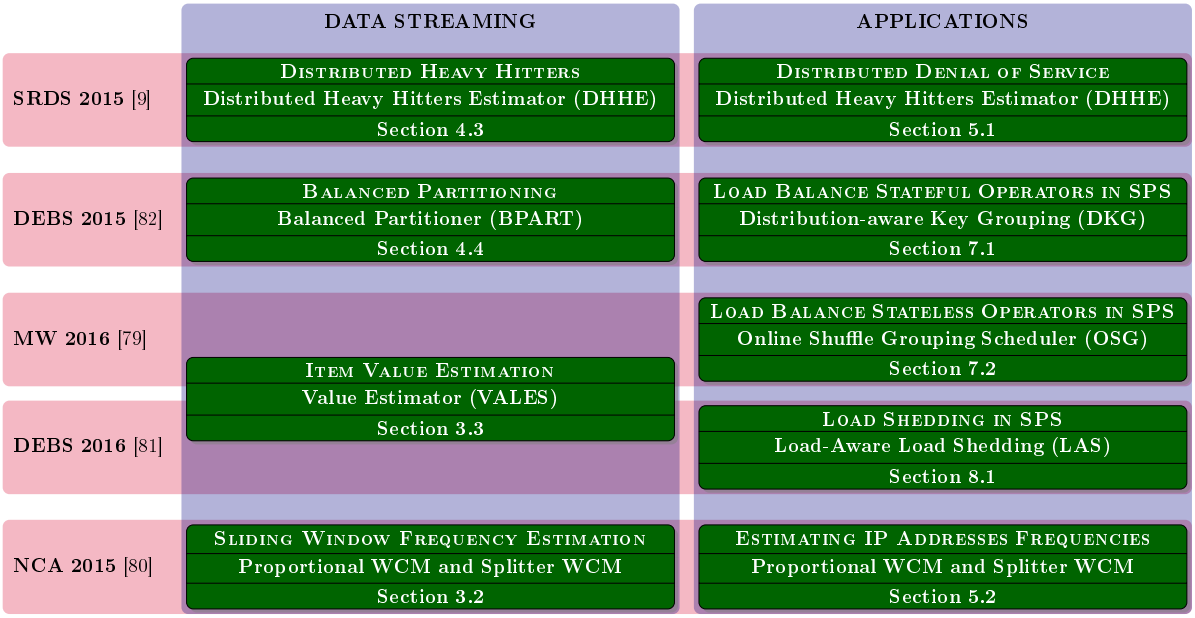


Figure 1.1 – Contributions Diagram.

propose Balanced Partitioner (BPART) [82], an algorithm that closely approximates the optimal partitioning, that strongly relies on the detection of heavy hitters. We leverage BPART to design Distribution-aware Key Grouping (DKG) [82] [3], a load balancing algorithm for stateful parallelized operators in stream processing systems. We prove that BPART/DKG achieve close to optimal balancing and validate the analysis through an experimental evaluation run on a simulator and on a prototype.

Item value estimation — Often an algorithm need to maintain some $\langle \text{key}, \text{value} \rangle$ map, for instance, to collect the results of some heavy computation, or to store some information that is valuable in the future but will not be available there. In many applications, and even more taking into account Big Data, the possible number of distinct keys in the map would be too large to fit in memory and a common solution is to introduce a caching system. Considering a streaming setting, we propose a different approach which maintains an efficient data structure that returns an approximation of the value. We present Value Estimator (VALES) [79, 81], an algorithm solving this problem, that we define as the item value estimation problem (a generalization of the frequency estimation problem). We leverage VALES to design two algorithms: Online Shuffle Grouping Scheduler (OSG) [79], a load balancing algorithm for stateless parallelized operators in stream processing systems, and Load-Aware Load Shedding (LAS) [81], a load shedding algorithm for stream processing systems. These two algorithms are conceived taking into account that the execution times of tasks in stream processing systems are non-uniform, a common assumption leveraged in previous works.

Sliding window frequency estimation — As motivated previously, many applications (*e.g.,* trading, network management, *etc.*) require the frequency distribution of the stream in the recent past. Papapetrou *et al.* [76] provide, at the best of our know-

[3] A shorter version has been published at the 18ème Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Algotel 2016)

ledge, the only solution to the frequency estimation problem in the sliding window model. However the space complexity of their solution is quadratic with respect to the accuracy parameter. We propose two different (on-line) algorithms that approximate the items frequency in a sliding window: Proportional Windowed Count-Min and Splitter Windowed Count-Min [80] [4, 5]. In particular, these algorithm are extensions of the `Count-Min` sketch [30], the data structure underlying DHHE, BPART and VALES. We apply these algorithms to the estimation of IP address frequencies, and the resulting experimental evaluation shows that our solutions provide better accuracy and memory usage than [76].

Layout

The thesis can be split in two parts: Chapters 2 to 4 present our data steaming contributions and their theoretical analysis, while Chapters 5 to 8 show the practical applications and their experimental evaluations.

Chapter 2 introduces the data streaming model. It also formalizes the extensions of the data streaming model that we consider, such as the distributed functional monitoring model and the sliding window model, as well as provide the building blocks that we use throughout this thesis. The next two chapters study the two aforementioned data streaming problems. More in details, Chapter 3 deals with the frequency estimation problem, with the design of the windowed extensions of the `Count-Min`, Proportional Windowed Count-Min and Splitter Windowed Count-Min algorithms, and presents the Value Estimator (VALES) algorithm. Similarly, Chapter 4 presents the Distributed Heavy Hitters Estimator (DHHE) algorithm solving the distributed heavy hitters problem, as well as the Balanced Partitioner (BPART) algorithm.

The following four chapters are the application of these algorithms to more concrete and practical problems. Chapter 5 shows how DHHE can detect distributed denial of service (DDoS) attacks, as well as how Proportional Windowed Count-Min and Splitter Windowed Count-Min can track the frequencies of IP addresses in a network. Chapter 6 formalises the stream processing model. Then, Chapter 7 applies BPART and VALES to load balance parallelized operators, either stateful or stateless, while Chapter 8 leverages VALES to perform load shedding.

Finally, Chapter 9 concludes and provides perspectives.

[4] A shorter version has been published at the 18ème Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Algotel 2016)

[5] This contribution is the recipient of the best student paper award at NCA 2015

Chapter 2

Data Streaming

Data Streaming focuses on estimating metrics over streams, which is an important task in data intensive applications. Many different domains are concerned by such analyses including machine learning, data mining, databases, information retrieval, and network monitoring. In all these applications, it is necessary to quickly and precisely process a huge amount of data.

This can be applied to any other data issued from distributed applications such as social networks or sensor networks. Given our settings, the real time analysis of large streams with small capacities in terms of storage and processing, analysing input data relying on full space algorithms is not feasible. Two main approaches exist to monitor massive data streams in real time: sampling and summaries. The first one consists in regularly sampling the input streams so that only a limited amount of data items is locally kept. This allows to exactly compute functions on these samples, which are expected to be representative. However, the accuracy of this computation, with respect to the stream in its entirety, fully depends on the volume of data that has been sampled and the location of the sampled data. Worse, an adversary may easily take advantage of the sampling policy to hide its attacks among packets that are not sampled, or to prevent "malicious" packets to be correlated. On the other hand, the summary approach consists in scanning on the fly each piece of data of the input stream, and keep locally only compact synopses or sketches containing the most important information about data items. This approach enables to derive some data streams statistics with guaranteed error bounds.

In general, the research done so far has focused on computing functions or statistical measures with ε or (ε, δ)-approximations in poly-logarithmic space over the size and/or the domain size of the stream. These include the computation of the number of distinct data items in a given stream [16,42,62], the frequency moments [5,32], the most frequent data items [30,69,70], and the entropy of the stream [25]. Computing information theoretic measures in the data stream model is challenging essentially because one needs to process a huge amount of data sequentially, on the fly, and by using very little storage with respect to the size of the stream. In addition, the analysis must be robust over time to detect any sudden change in the observed streams (which may be an indicator of some malicious behaviour). The most natural derivation of the data stream model is the sliding window model [34], where we consider only the most recent data items. Another relevant model is the distributed functional monitoring problem [32], which combines features of the streaming model, communication complexity and distributed computing, and thus deals with distributed streams.

2.1 System Models

In the data streaming community, several models have risen in order to match the scenarios where the data streaming approach could be applied. In this section, we discuss and formalise the models that have drawn more attention from the community and that are taken into account in this thesis.

2.1.1 Data Streaming Model

Back in 2005, Muthukrishan [72] published a really extensive overview and survey of the *data streaming* model, that can be formalized as follows.

We consider a stream $\sigma = \langle t_1, \ldots, t_j, \ldots, t_m \rangle$, where each item t is drawn from a universe of size n. More in details, t denotes the value of the item in the set $[n] = \{1, \ldots, n\}$, while the subscript j denotes the position in the stream in the index sequence $[m] = \{1, \ldots, m\}$. Then m is the size (or number of items) of the stream σ. Notice that in general both n and m are large (*i.e.,* network traffic) and unknown.

The stream σ implicitly defines a frequency vector $\mathbf{f} = \langle f_1, \ldots, f_t, \ldots, f_n \rangle$ where f_t is the frequency of item t in the stream σ, *i.e.,* the number of occurrences of item t in σ. We can also define the empirical probability distribution $\mathbf{p} = \langle p_1, \ldots, p_t, \ldots, p_n \rangle$, where $p_t = f_t/m$ is the empirical probability of occurrence of item t in the stream σ. Our aim is to compute some function ϕ on σ. In this model we are allowed only to access the sequence in its given order (no random access) and the function must be computed in a single pass (on-line). The challenge is to achieve this computation in sub-linear space and time with respect to m and n. In other words, the space and time complexities should be at most $o(\min\{m, n\})$ but the goal is $O(\log(m \cdot n))$. However, since reaching the latter bound is quite hard (if not impossible), one deal in general with complexities [1] such as $polylog(m, n)$.

2.1.2 Sliding Window Model

In many applications, there is a greater interest in considering only the "recently observed" items than the whole semi-infinite stream. This gives rise to the *sliding window* model formalised by Datar, Gionis, Indyk and Motwani [34]: items arrive continuously and expire after exactly M steps. The relevant data is then the set of items that arrived in the last M steps, *i.e.,* the *active* window. A step can either be defined as a sample arrival, then the active window would contain exactly M items and the window is said to be count-based. Otherwise, the step is defined as a time tick, either logical or physical, and the window is said to be time-based. More formally, let consider the count-based sliding window and recall that by definition t_j is the item arriving at step j. In other words, the stream σ at step m is the sequence $\langle t_1, \ldots, t_{m-M}, t_{m-M+1}, \ldots, t_m \rangle$. Then, the desired function ϕ has to be evaluated over the stream $\sigma(M) = \langle t_{m-M+1}, \ldots, t_m \rangle$. Notice that if $M = m$, the sliding window model boils down to the classical data streaming model. Two related models may be named as well: the landmark window and jumping window models. In the former, a step (landmark) is chosen and the function is evaluated on the stream following the landmark. In the jumping window model, the sliding window is split into sub-windows and moves forward only when the most recent sub-window is full (*i.e.,* it jumps forward). In general, solving a problem in the sliding window model

[1] In general we have that $\phi(x)$ is $polylog(x) \Leftrightarrow \exists C > 0, \phi(x) = O(\log^C x)$. By abusing the notation, we define $polylog(m, n)$ as $\exists C, C' > 0$ such that $\phi(m, n) = O(\log^C m + \log^{C'} n))$.

is harder than in the classical data streaming model: using little memory (low space complexity) implies some kind of data aggregation, the problem is then how to slide the window removing the updates performed by the items that exited the window and how to introduce new items. For instance, *basic counting*, *i.e.,* count the number of 1 in the stream of bits, is a simple problem whose solution is a prerequisite for efficient maintenance of a variety of more complex statistical aggregates. While this problem is trivial for the data streaming model, it becomes a challenge in the sliding windowed model [34].

2.1.3 Distributed Streaming Model and Functional Monitoring

In large distributed systems, it is most likely critical to gather various aggregates over data spread across a large number of nodes. This can be modelled by a set of nodes, each observing a stream of items. These nodes collaborate to evaluate a given function over the global distributed stream. For instance, current network management tools analyze the input streams of a set of routers to detect malicious sources or to extract user behaviors [7, 43].

The *distributed streaming* model has been formalised by Gibbons and Tirthapura [45]. Consider a set of nodes making observations and cooperating, through a *coordinator*, to compute a function over the union of their observations. More formally, let $\mathcal{I}$ be a set of s nodes $S_1, \ldots, S_i, \ldots, S_s$ that do not communicate directly between them, but only with a coordinator (*i.e.*, a central node or base station). The communication channel can be either one-way or bidirectional. The *distributed* stream σ is the union the sub-streams $\sigma_1, \ldots, \sigma_i, \ldots, \sigma_s$ locally read by the s nodes, *i.e.,* $\sigma = \bigcup_{i=1}^{s} \sigma_i$, where $\sigma_\imath \cup \sigma_\jmath$ is a stream containing all the items of $\sigma_\imath$ and $\sigma_\jmath$ in any order. The generic stream σ_i is the input to the generic node S_i. Then we have $m = \sum_{i=1}^{s} m_i$, where m_i is the size (or number of items) of the i-th sub-stream σ_i of the distributed stream σ. Each sub-stream σ_i implicitly defines a frequency vector $\mathbf{f}_i = \langle f_{1,i}, \ldots, f_{t,i}, \ldots, f_{n,i} \rangle$ where $f_{t,i}$ is the frequency of item t in the sub-stream σ_i, *i.e.,* the number of occurrences of item t in the sub-stream σ_i. Then, we also have $f_t = \sum_{i=1}^{s} f_{t,i}$. We can also define the empirical probability distribution $\mathbf{p}_i = \langle p_{1,i}, \ldots, p_{t,i}, \ldots, p_{n,i} \rangle$, where $p_{t,i} = f_{t,i}/m_i$ is the empirical probability of occurrence of item t in the stream σ_i.

With respect to the classical data streaming model, we want to compute the function $\phi(\sigma) = \phi(\sigma_1 \cup \ldots \cup \sigma_s)$. This model introduces the additional challenge to minimise the communication complexity (in bits), *i.e.,* the amount of data exchanged between the s nodes and the coordinator.

The *distributed functional monitoring* problem [32], has risen only in the 2000's and a survey on this field is provided by Cormode [29] in 2011. With respect to the distributed streaming model, functional monitoring requires a continuous evaluation of $\phi(\sigma) = \phi(\sigma_1 \cup \ldots \cup \sigma_s)$ at each time instant $j \in [m]$. Cormode, Muthukrishnan and Yi [32], bounding ϕ to be monotonic, distinguish two cases: *threshold* monitoring (*i.e.*, raising an alert if $\phi(\sigma)$ exceed a threshold Θ) and *value* monitoring (*i.e.,* approximating $\phi(\sigma)$). They also show that value monitoring can be reduced with a factor $O(1/\varepsilon \log R)$ (where R is the largest value reached by the monitored function) to threshold monitoring. However, this reduction does not hold any more in a more general setting with non-monotonic function (*e.g.,* entropy). Notice that to minimize the communication complexity and maximise the accuracy, the algorithms designed in this model tend to minimise the communication when nothing relevant happens, while reacting as fast as possible in response to outstanding occurrences or patterns.

2.1.4 Adversarial Model

In any of the aforementioned models we can add an adversary actively tampering with the data stream. In general the adversary is omnipotent in the sense that it knows the whole stream σ and it can observe, insert or re-order items in the data stream. Dropping items is considered only in strongly adversarial models. The algorithms are of public knowledge, *i.e.,* security by obscurity is not considered, however the adversary can be characterized with respect to its knowledge of the algorithm evolution:

- The *oblivious* adversary does not get to know the randomized results of the algorithm (*i.e.,* it has not access to the algorithm random values) nor the current algorithm state;
- The *adaptive online* adversary knows all the past decisions of the algorithm (*i.e.,* knows the current algorithm state), however it cannot predict its next randomized move (*i.e.,* it has not access to the algorithm random values);
- The *adaptive offline* adversary is the strongest, since it knows the past and future actions of the algorithm (*i.e.,* it has access to the algorithm random values).

Notice that for a deterministic algorithm, the three adversaries are equivalent. In addition [17], while randomization helps against both the oblivious and adaptive on-line algorithm, a deterministic algorithm fares as well as a randomized one against an adaptive offline adversary. In this thesis we consider at most the oblivious adversary.

In general, a common technique to prove the resilience of the algorithm in the face of the adversary is to provide the effort required to subvert the algorithm (*i.e.*, to break the accuracy guarantees). Notice that in the distributed models, the adversary may also enact strategies to maximising the number of interactions and the amount of data exchanged between the s nodes and the coordinator.

2.2 Building Blocks

In this section, we formalise some of the basic techniques and building blocks used in this thesis.

2.2.1 Sampling

The sampling approach reads once all the items; however, only a few are kept for the computation. As usual, the aim is for a poly-logarithmic storage size with respect to size of the input stream. It is also current that some metadata are associated with the retained items. There are several methods, both deterministic and randomized, to choose which subset of items has to be kept [5, 70, 72].

2.2.2 Approximation and Randomization

To compute ϕ on σ in these models, research so far relies on ε or (ε, δ)-approximations. An algorithm A that evaluates the stream σ in a single pass (on-line) is said to be an (ε, δ)-additive-approximation of the function ϕ on a stream σ if, for any prefix of size m of the input stream σ, the output $\widehat{\phi}$ of A is such that $\mathbb{P}\{\mid \widehat{\phi} - \phi \mid > \varepsilon\psi\} \leq \delta$, where ε, $\delta > 0$ are given as precision parameters and ψ is an arbitrary function (its parameter may be n and/or m). Similarly, an algorithm A that evaluates the stream σ in a single pass (on-line) is said to be an (ε, δ)-approximation of the function ϕ on a stream σ if, for any

prefix of size m of the input stream σ, the output $\widehat{\phi}$ of A is such that $\mathbb{P}\{\mid \widehat{\phi} - \phi \mid > \varepsilon\phi\} \leq \delta$, where ε, $\delta > 0$ are given as precision parameters.

Finally, if $\delta = 0$, independently of the algorithm parametrisation, then A is deterministic and we drop the δ from the approximation notation.

2.2.3 Random Projections and Sketches

Random projections produce a storage size reduction, leveraging pseudo-random vectors and projections. We generate the pseudo-random vector using space-efficient computation over limitedly independent random variables, as in [27, 30].

A relevant class of projections are sketches. A data structure DS is a sketch if there is a space-efficient combining algorithm `COMB` such that, for every two streams σ_1 and σ_2, we have $\texttt{COMB}(DS(\sigma_1), DS(\sigma_2)) = DS(\sigma_1 \cup \sigma_2)$, where $\sigma_1 \cup \sigma_2$ is a stream containing all the items of σ_1 and σ_2 in any order. In order to extend the sketch definition to the sliding window model, we limit the reordering of the items in σ_1 and σ_2 to the last M items (*i.e.,* the active window).

2.2.4 Universal and Independent Hash Functions

A collection $\mathcal{U}$ of hash functions $h_\imath : [N] \to [N']$ is said to be K-independent if for every K distinct items $x_1, \ldots, x_K \in [N]$ and K (not necessarily distinct) hashes $y_1, \ldots, y_K \in [N']$, $\mathbb{P}_{h_\imath \in \mathcal{U}}\{h_\imath(x_1) = y_1 \wedge \ldots \wedge h_\imath(x_K) = y_K\} \leq 1/N'^K$.

While these hash functions are powerful, they are very hard to construct, *i.e.,* it is difficult to build a large set of K-independent hash functions. Let us introduce the 2-universal hash functions, which are a superset of the K-independent hash functions, thus weaker. A collection $\mathcal{H}$ of hash functions $h_\imath : [N] \to [N']$ is said to be 2-universal if for every 2 distinct items $x_1, x_2 \in [N]$, $\mathbb{P}_{h_\imath \in \mathcal{H}}\{h_\imath(x_1) = h_\imath(x_2)\} \leq 1/N'$, which is the probability of collision obtained if the hash function assigns truly random values in $[N']$ to any $x \in [N]$.

Thankfully, Carter and Wegman [24] proposed an efficient algorithm to build large families of hash functions approximating the 2-universal property. Then, algorithms using 2-universal hash functions can be implemented in practice.

Chapter 3

Frequency Estimation

The previous chapter has formalised the data streaming models and the background that is used throughout this thesis. In this chapter we first detail the frequency estimation problem and its related work. In Section 3.2 we present our extensions of the `Count-Min` algorithm in the sliding window model: Proportional Windowed Count-Min and Splitter Windowed Count-Min. Section 3.3 formalises the item value estimation problem, *i.e.,* a generalisation of the frequency estimation problem, and shows the design of the Value Estimator algorithm.

3.1 Related Work

Estimating the frequency moments is one of the first problem to be dealt with in a streaming fashion, namely in the seminal paper from Alon, Matia and Szegdy [5]. As mentioned earlier, a stream σ implicitly defines a frequency vector $\mathbf{f} = \langle f_1, \ldots, f_n \rangle$ where f_t is the frequency of item t in the stream σ. The $\imath$-th frequency moment is defined as $F_\imath = \sum_{j=1}^{n} f_j^\imath = \| f \|_\imath$. F_0 is then the number of distinct items of the stream (assuming that $0^0 = 0$) and F_1 is the length m of the stream. The frequency moments for $F_\imath$ where $\imath \geq 2$ indicate the skew of the data, which is a relevant information in many distributed applications.

The *frequency estimation* problem is closely related, we do not want to provide a single aggregate value to represent the whole stream, but a more detailed view.

Problem 3.1 (Frequency Estimation Problem)**.** *Given a stream $\sigma = \langle t_1, \ldots, t_m \rangle$, provide an estimate $\hat{f}_t$ of f_t for each item $t \in [n]$.*

In other words we want to build an approximation of the frequency distribution of the stream. Estimating the frequency of any piece of information in large-scale distributed data streams became of utmost importance in the last decade (*e.g.,* in the context of network monitoring, Big Data, *etc.*). IP network management, among others, is a paramount field in which the ability to estimate the stream frequency distribution in a single pass (*i.e.,* on-line) and quickly can bring huge benefits. For instance, we could rapidly detect the presence of anomalies or intrusions identifying sudden changes in the communication patterns.

Data streaming model — Considering the data streaming model, Charikar, Chen and Farach-Colon [27] proposed the `Count-Sketch` algorithm in 2002. The underlying data structure is a matrix $\mathfrak{F}$ of size $r \times c$, where $r = \lceil \log 1/\delta \rceil$ and $c = \lceil 3/\varepsilon^2 \rceil$. Each row is

Listing 3.1 – `Count-Min` Sketch Algorithm.

```
1: init (r, c) do
2:     𝔉[1...r, 1...c] ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4: end init
5: function UPDATE(t_j)                   ▷ reads item t_j from the stream σ
6:     for ι = 1 to r do
7:         𝔉[ι, h_ι(t)] ← 𝔉[ι, h_ι(t)] + 1
8:     end for
9: end function
10: function GETFREQ(t)                   ▷ returns f̂_t
11:     return min{𝔉[ι, h_ι(t)] | 1 ≤ ι ≤ r}
12: end function
```

associated with two different 2-universal hash functions $h_{\iota\in[r]} : [n] \to [c]$ and $h'_{\iota\in[r]} : [n] \to \{-1, 1\}$. For each item t_j, it updates each row: $\forall \iota \in [r], \mathfrak{F}[\iota, h_\iota(t)] \leftarrow \mathfrak{F}[\iota, h_\iota(t)] + h'_\iota(t)$. The algorithm returns, as f_t estimation $\hat{f}_t = \text{median}_{\iota\in[r]}(h'_\iota(t)\mathfrak{F}[i, h_\iota(t)])$. The space complexity of this algorithm is $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}(\log m + \log n))$. This estimate satisfies the following accuracy bound: $\mathbb{P}[\mid \hat{f}_t - f_t \mid \geq \varepsilon \parallel \mathbf{f}_{-t} \parallel_2] \leq \delta$, where $\mathbf{f}_{-t}$ represents the frequency vector $\mathbf{f}$ without the scalar f_t (*i.e.,* $|\mathbf{f}_{-t}| = n - 1$).

In 2003, Cormode and Muthukrishnam [30] proposed the `Count-Min` sketch, which is quite similar to the `Count-Sketch`. The bound on the estimation accuracy of the `Count-Min` is weaker than the one provided by the `Count-Sketch`, in particular for skewed distribution. On the other hand, the `Count-Min` shaves a $1/\varepsilon$ factor from the space complexity and halves the hash functions. The `Count-Min` consists of a two dimensional matrix $\mathfrak{F}$ of size $r \times c$, where $r = \lceil \log 1/\delta \rceil$ and $c = \lceil e/\varepsilon \rceil$, where e is the euler's constant. Each row is associated with a different 2-universal hash function $h_{\iota\in[r]} : [n] \to [c]$. For each item t_j, it updates each row: $\forall \iota \in [r], \mathfrak{F}[\iota, h_\iota(t)] \leftarrow \mathfrak{F}[\iota, h_\iota(t)] + 1$. That is, the entry value is the sum of the frequencies of all the items mapped to that entry. Since each row has a different collision pattern, upon request of $\hat{f}_t$ we want to return the entry associated with t minimizing the collisions impact. In other words, the algorithm returns, as f_t estimation, the entry associated with t with the lowest value: $\hat{f}_t = \min_{\iota\in[r]}\{\mathfrak{F}[\iota, h_\iota(t)]\}$. Listing 3.1 presents the global behavior of the `Count-Min` algorithm. The space complexity of this algorithm is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n))$ bits, while the update time complexity is $O(\log 1/\delta)$. Therefore, the following quality bounds hold: $\mathbb{P}[\mid \hat{f}_t - f_t \mid \geq \varepsilon \parallel \mathbf{f}_{-t} \parallel_1] \leq \delta$, while $f_t \leq \hat{f}_t$ is always true.

In [30], the author also provide a more general version of the `Count-Min` algorithm. Considering the *cash register model* [72], each item t_j in the stream carries a value $\omega_{t_j} \geq 0$, *i.e.,* the stream is a sequence of couples: $\sigma = \langle (t_1, \omega_{t_1}), \ldots, (t_m, \omega_{t_m}) \rangle$. Then we can extend the `Count-Min` algorithm (*cf.,* Listing 3.2) adding to the matrix entries associated with item t the value ω_{t_j}. The `Count-Min` sketch returns an estimation of $\sum_{j\in[m]} \omega_{t'_j} \mathbb{1}_{\{t'=t\}}$ for all $t \in [n]$. If ω_{t_j} is fixed for each t (*i.e.,* independent of j), then the `Count-Min` returns the estimation $\omega_t \hat{f}_t$ of $\omega_t f_t$ for all $t \in [n]$.

Sliding window model — Papapetrou *et al.* [76] proposed the only work that, to the best of our knowledge, tackles the frequency estimation problem in the sliding

Listing 3.2 – Generalized `Count-Min` Sketch Algorithm.

```
1: init (r, c) do
2:     𝔉[1...r, 1...c] ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4: end init
5: function UPDATE(t_j, ω_{t_j})          ▷ reads item t_j with value ω_t from the stream σ
6:     for ι = 1 to r do
7:         𝔉[ι, h_ι(t)] ← 𝔉[ι, h_ι(t)] + ω_{t_j}
8:     end for
9: end function
10: function GETESTIMATION(t)                                   ▷ returns f̂_t
11:     return min{𝔉[ι, h_ι(t)] | 1 ≤ ι ≤ r}
12: end function
```

window model. Their proposal, named `ECM-Sketch`, is based on the `wave` data structure presented by Gibbons and Tirthapura [46], providing an $(\varepsilon_{\texttt{wave}})$-approximation for the basic counting problem in the sliding window model. A `wave` is made of $\lceil \log(\varepsilon_{\texttt{wave}} M) \rceil$ levels and the $\imath$-th level is made of $1/\varepsilon_{\texttt{wave}} + 1$ positions of the most recent updates to the `wave` whose position is divisible by $2^\imath$. In addition, a wave stores the current length of the stream and the number of updates performed so far. With some optimization to get rid of redundant information and to reduce computation time, the space complexity is $O(\frac{1}{\varepsilon_{\texttt{wave}}} \log^2(\varepsilon_{\texttt{wave}} M))$ bits, while both update and query time complexities are $O(1)$. Replacing each counter of the `Count-Min` matrix $\mathfrak{F}$ with a `wave`, the authors [76] are able to obtain an $(\varepsilon_{\texttt{wave}}, \varepsilon, \delta)$-additive-approximation with the following quality bound: $\mathbb{P}[\mid \hat{f}_t - f_t \mid \geq (\varepsilon + \varepsilon_{\texttt{wave}} + \varepsilon\, \varepsilon_{\texttt{wave}}) \parallel \mathbf{f}_{-t} \parallel_1] \leq \delta$. The space complexity of the `ECM-Sketch` is $O\left(\frac{1}{\varepsilon\, \varepsilon_{\texttt{wave}}} \log \frac{1}{\delta} \left(\log^2(\varepsilon_{\texttt{wave}} M) + \log n\right)\right)$, while the updated and query time complexities are $O(\log 1/\delta)$.

3.2 Frequency Estimation in the Sliding Window Model

In this section we propose our extensions to the sliding window model of the `Count-Min` sketch [30] presented in Section 3.1. The problem we consider is the sliding window variation of Problem 3.1:

Problem 3.2 (Frequency Estimation Problem in Sliding Window)**.** *Given a stream $\sigma = \langle t_1, \ldots, t_m \rangle$, provide an estimate $\hat{f}_t$ of f_t for each item $t \in [n]$ in the active window $\sigma(M) = \langle t_{m-M+1}, \ldots, t_m \rangle$,* i.e., *taking into account only the M most recent items.*

We propose our approach in two steps: two first naive and straightforward algorithms called Perfect WCM and Simple WCM, followed by two more sophisticated ones, called Proportional WCM and Splitter WCM. In Section 5.2.1 we apply these algorithms to the estimation of the frequencies on IP traffic, comparing their respective performances together with the only comparable solution [76].

3.2.1 Perfect Windowed Count-Min Algorithm

Perfect WCM provides the best accuracy by dropping the complexity space requirements, it trivially stores the whole active window in a queue. When it reads item t_j, it enqueues t_j

Listing 3.3 – Perfect WCM Algorithm.

```
1: init (r, c, M) do
2:     𝔉[1...r, 1...c] ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4:     window an empty queue of items
5: end init
6: function UPDATE(t_j)                   ▷ reads item t_j from the stream σ
7:     for ı = 1 to r do
8:         𝔉[ı, h_ı(t)] ← 𝔉[ı, h_ı(t)] + 1
9:     end for
10:    enqueue t in window
11:    if | window |> M then
12:        t' ← dequeue from window
13:        for ı = 1 to r do
14:            𝔉[ı, h_ı(t')] ← 𝔉[ı, h_ı(t')] − 1
15:        end for
16:    end if
17: end function
18: function GETFREQ(t)                   ▷ returns f̂_t
19:    return min{𝔉[ı, h_ı(t)] | 1 ≤ ı ≤ r}
20: end function
```

and increases all the `Count-Min` matrix entries associated with t. Once the queue reaches size M, it dequeues the expired item $t'_{j'}$ and decreases all the entries associated with t'. The frequency estimation is retrieved as in the classical `Count-Min` (*cf.,* Listing 3.1). Listing 3.3 presents the global behaviour of Perfect WCM.

Theorem 3.3 (Perfect WCM Accuracy)**.** *Perfect WCM is an (ε, δ)-additive-approximation of the frequency estimation problem in the count-based sliding window model where $\mathbb{P}[\mid \hat{f}_t - f_t \mid \geq \varepsilon(M - f_t)] \leq \delta$, and $f_t \leq \hat{f}_t$ is always true.*

Proof. Since the algorithm stores the whole previous window, it knows exactly which item expires in the current step and can decrease the associated counters in the $\mathfrak{F}$ matrix. Then, Perfect WCM provides an estimation with the same error bounds of a `Count-Min` executed on the last M items of the stream. □

Theorem 3.4 (Perfect WCM Space and Time Complexities)**.** *Perfect WCM space complexity is $O(M \log(n))$ bits, while update and query time complexities are $O(\log 1/\delta)$.*

Proof. The algorithm stores M items, then the size of the $\mathfrak{F}$ matrix is negligible and the space complexity is $O(M \log(n))$ bits. An update requires to enqueue and dequeue two items and to manipulate an entry on each row. Thus, the update time complexity is $O(\log 1/\delta)$. A query requires to look up an entry for each row of the $\mathfrak{F}$ matrix, then the query time complexity is $O(\log 1/\delta)$. □

3.2.2 Simple Windowed Count-Min Algorithm

Simple WCM is as straightforward as possible and achieves optimal space complexity with respect to the `Count-Min`. It behaves as the `Count-Min`, except that it resets the $\mathfrak{F}$

Listing 3.4 – Simple WCM Algorithm.

```
1: init (r, c, M) do
2:     𝔉[1...r, 1...c] ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4:     m ← 0
5: end init
6: function UPDATE(t_j)                   ▷ reads item t_j from the stream σ
7:     if m = 0 then
8:         𝔉[1...r, 1...c] ← 0_{r,c}
9:     end if
10:    for ι = 1 to r do
11:        𝔉[ι, h_ι(t)] ← 𝔉[ι, h_ι(t)] + 1
12:    end for
13:    m ← m' + 1  mod M
14: end function
15: function GETFREQ(t)                   ▷ returns f̂_t
16:    return min{𝔉[ι, h_ι(t)] | 1 ≤ ι ≤ r}
17: end function
```

matrix each times it has read M items. Obviously Simple WCM provides a really rough estimation since it simply drops all information about any previous window once a new window starts. Listing 3.4 presents the global behavior of Simple WCM.

Theorem 3.5 (Simple WCM Space and Time Complexities)**.** *Simple WCM space complexity is* $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M+\log n)\right)$ *bits, while update and query time complexities are* $O(\log 1/\delta)$.

Proof. The algorithm uses a counter of size $O(\log M)$ and a matrix of size $r \times c$ ($r = \lceil \log 1/\delta \rceil$ and $c = \lceil e/\varepsilon \rceil$) of counters of size $O(\log M)$. In addition, for each row it stores a hash function. Then the space complexity is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M+\log n))$ bits. An update requires to hash an item, then retrieve and increase an entry for each row, thus the update time complexity is $O(\log 1/\delta)$. We consider the cost of resetting the matrix ($O(\frac{1}{\varepsilon}\log 1/\delta)$) negligible since it is done only once per window. A query requires to hash an item and retrieve an entry for each row: the query time complexity is $O(\log 1/\delta)$. □

3.2.3 Proportional Windowed Count-Min Algorithm

We now present the first extension algorithm, denoted Proportional WCM. The intuition behind this algorithm is as follows. At the end of each window, it stores separately a snapshot $\mathfrak{S}$ of the $\mathfrak{F}$ matrix, which represents what happened during the previous window. Starting from the current $\mathfrak{F}$ state, for each new item, it increments the associated entries and decreases all the $\mathfrak{F}$ matrix entries proportionally to the last snapshot $\mathfrak{S}$. This smooths the impact of resetting the $\mathfrak{F}$ matrix throughout the current window. Listing 3.5 presents the global behavior of Proportional WCM.

More in details, after reading M items, Proportional WCM stores the current $\mathfrak{F}$ matrix in $\mathfrak{S}$ and divides each entry by the window size: $\forall \imath, \jmath \in [r] \times [c],\ \mathfrak{S}[\imath, \jmath] \leftarrow \mathfrak{F}[\imath, \jmath]/M$ (Listing 3.5, Lines 7 to 11). This snapshot represents the average step increment of the $\mathfrak{F}$ matrix during the previous window. When Proportional WCM reads item t_j, it increments the $\mathfrak{F}$ entries associated with t (Listing 3.5, Lines 13 to 15) and subtracts

Listing 3.5 – Proportional WCM Algorithm.

```
1: init (r, c, M) do
2:     𝔉[1...r, 1...c], 𝔖[1...r, 1...c] ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family
4:     m ← 0
5: end init
6: function UPDATE(t_j)                   ▷ reads item t_j from the stream σ
7:     if m = 0 then
8:         for ι = 1 to r and ȷ = 1 to c do
9:             𝔖[ι, ȷ] ← 𝔉[ι, ȷ]/M
10:        end for
11:    end if
12:    for ι = 1 to r and ȷ = 1 to c do
13:        if h_ι(t) = ȷ then
14:            𝔉[ι, ȷ] ← 𝔉[ι, ȷ] + 1
15:        end if
16:        𝔉[ι, ȷ] ← 𝔉[ι, ȷ] − 𝔖[ι, ȷ]
17:    end for
18:    m ← m + 1 mod M
19: end function
20: function GETFREQ(t)                   ▷ returns f̂_t
21:    return round {min{𝔉[ι, h_ι(t)] | 1 ≤ ι ≤ r}}
22: end function
```

$\mathfrak{S}$ from $\mathfrak{F}$: $\forall \imath, \jmath \in [r] \times [c],\ \mathfrak{F}[\imath, \jmath] \leftarrow \mathfrak{F}[\imath, \jmath] - \mathfrak{S}[\imath, \jmath]$ (Listing 3.5, Line 16). Finally, the frequency estimation is retrieved as in the `Count-Min` algorithm.

Theorem 3.6 (Proportional WCM Space and Time Complexities)**.** *Proportional WCM space complexity is* $O(\frac{1}{\varepsilon} \log \frac{1}{\delta}(\log M + \log n))$ *bits. Update and query time complexities are* $O(\frac{1}{\varepsilon} \log 1/\delta)$ *and* $O(\log 1/\delta)$.

Proof. The algorithm stores a $\mathfrak{F}$ and a snapshot $\mathfrak{S}$ matrix, as well as a counter of size $O(\log M)$. Then the space complexity is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta}(\log M + \log n))$ bits. An update require to look up all the entries of both the $\mathfrak{F}$ and $\mathfrak{S}$, thus the update time complexity is $O(\frac{1}{\varepsilon} \log 1/\delta)$. A query requires to hash an item and retrieve an entry for each row: the query time complexity is $O(\log 1/\delta)$. □

3.2.4 Splitter Windowed Count-Min Algorithm

Proportional WCM removes the *average* frequency distribution of the previous window from the current window. Consequently, Proportional WCM does not capture sudden changes in the stream distribution. To cope with this flaw, one could track these critical changes through multiple snapshots. However, each row of the $\mathfrak{F}$ matrix is associated with a specific 2-universal hash function, thus changes in the stream distribution do not affect equally each rows.

Therefore, Splitter WCM proposes a finer grained approach analysing the update rate of each entry in the $\mathfrak{F}$ matrix. To record changes in the entry update rate, we add a (fifo) queue of sub-cells to each entry. When Splitter WCM detects a relevant variation

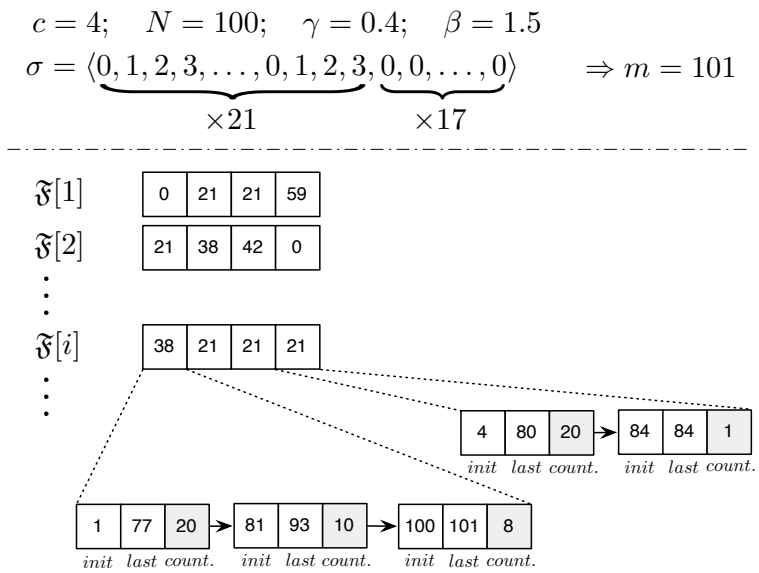


Figure 3.6 – State of the $\mathfrak{F}$ matrix after reading a prefix of $m = 101$ items of σ.

in the entry update rate, it creates and enqueues a new sub-cell. This new sub-cell then tracks the current update rate, while the former one stores the previous rate.

Each sub-cell has a frequency *counter* and 2 timestamps: *init*, that stores the (logical) time where the sub-cell started to be active, and *last*, that tracks the time of the last update. After a short bootstrap, any entry contains at least two sub-cells: the current one that depicts what happened in the very recent history, and a predecessor representing what happened in the past. Listing 3.7 presents the global behaviour of Splitter WCM. Figure 3.6 shows the state of the $\mathfrak{F}$ matrix after reading a prefix of $m = 101$ items of the stream σ, sketched on the top.

Splitter WCM spawns additional sub-cells to capture distribution changes. The decision whether to create a new sub-cell is tuned by two parameters: γ and β, and an error function, ERROR (*cf.,* Listing 3.8). Informally, the function ERROR evaluates the potential amount of information lost by merging two consecutive sub-cells, while β represents the amount of affordable information loss. Performing this check at each item arrival may lead to erratic behaviours. To avoid this, we introduced $\gamma \in]0, 1]$, that sets the minimal length ratio of a sub-cell before taking this sub-cell into account in the decision process.

In more details, when Splitter WCM reads item t_j, it has to phase out the expired data from each sub-cell. Then, for each entry of $\mathfrak{F}$, it retrieves the oldest sub-cell in the queue, denoted *head* (Line 9). If *head* was active precisely M steps ago (Line 10), then it computes the rate at which *head* has been incremented while it was active (Line 11). This value is subtracted from the entry counter v (Line 12) and from *head* counter (Line 13). Having retracted what happened M steps ago, *head* moves forward increasing its *init* timestamp (Line 14). Finally, *head* is removed if it has expired (Lines 15 and 16).

The next part handles the update of the entries associated with item t_j. For each of them (Line 19), Splitter WCM increases the entry counter v (Line 20) and retrieves the current sub-cell, denoted *bottom* (Line 21). **(a)** If *bottom* does not exist, it creates and enqueues a new sub-cell (Line 23). **(b)** If *bottom* has not reached the minimal size to be evaluated (Line 24), *bottom* is updated (Line 25). **(c)** If not, Splitter WCM retrieves the

Listing 3.7 – Splitter Windowed Count-Min Algorithm.

```
1: init (r, c, M, γ, β) do
2:     𝔉[1...r][1...c] ← ⟨empty sub-cells queue, 0⟩_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4:     m ← 0
5: end init
6: function UPDATE(t_j)                    ▷ reads item t_j from the stream σ
7:     for ι = 1 to r and ȷ = 1 to c do
8:         ⟨queue, v⟩ ← 𝔉[ι, ȷ]
9:         head ← head of queue
10:        if ∃head ∧ head.init = m − M then
11:            v' ← head.counter/(head.last − head.init + 1)
12:            v ← v − v'
13:            head.counter. ← head.counter. − v'
14:            head.init ← head.init + 1
15:            if head.init > head.last then
16:                removes head from queue
17:            end if
18:        end if
19:        if h_ι(t) = ȷ then
20:            v ← v + 1
21:            bottom ← bottom of queue
22:            if ∄bottom then
23:                Creates and enqueues a new sub-cell
24:            else if bottom.counter < γM/c then
25:                Updates sub-cell bottom
26:            else
27:                pred ← predecessor of bottom in queue
28:                if ∃pred ∧ ERROR(pred, bottom) ≤ β then
29:                    Merges bottom into pred and renews bottom
30:                else
31:                    Creates and enqueues a new sub-cell
32:                end if
33:            end if
34:        end if
35:        𝔉[ι, ȷ] ← ⟨queue, v⟩
36:    end for
37:    m ← m + 1
38: end function
39: function GETFREQ(t)                    ▷ returns f̂_t
40:    return round{min{𝔉[ι][h_ι(t)].v | 1 ≤ ι ≤ r}}
41: end function
```

Listing 3.8 – Splitter Windowed Count-Min ERROR function.

```
1: function ERROR(pred, bottom)          ▷ Evaluates the amount of information loss
2:     freq_pred ← pred.count/(bottom.init − pred.init)
3:     freq_bottom ← bottom.count/(bottom.init − bottom.init + 1)
4:     if freq_bottom > freq_pred then
5:         return freq_bottom / freq_pred
6:     else
7:         return freq_pred / freq_bottom
8:     end if
9: end function
```

predecessor of *bottom*: *pred* (Line 27). **(c.i)** If *pred* exists and the amount of information lost by merging is lower than the threshold β (Line 28), Splitter WCM merges *bottom* into *pred* and renews *bottom* (Line 29). **(c.ii)** Otherwise it creates and enqueues a new sub-cell (Line 31), *i.e.*, it *splits* the entry.

Lemma 3.7 (Splitter WCM Number of Splits Upper Bound)**.** *Given* $0 < \gamma \leq 1$*, the maximum number of splits* $\overline{\pi}$ *(number of sub-cells spawned to track distribution changes) is* $O(\frac{1}{\varepsilon\gamma}\log\frac{1}{\delta})$.

Proof. A sub-cell is not involved in the decision process of merging or splitting while its counter is lower than $\frac{\gamma M}{c} = \varepsilon\gamma M$. So, no row can own more than $\frac{1}{\varepsilon\gamma}$ splits. Thus, the maximum numbers of splits is $\overline{\pi} = O(\frac{1}{\varepsilon\gamma}\log\frac{1}{\delta})$. □

Theorem 3.8 (Splitter WCM Space and Time Complexities)**.** *Splitter WCM space complexity is* $O(\frac{1}{\gamma\varepsilon}\log\frac{1}{\delta}(\log M + \log n))$ *bits, while update and query time complexities are* $O(\log 1/\delta)$.

Proof. Each entry of the $\mathfrak{F}$ matrix is composed of a counter and a queue of sub-cells made of two timestamps and a counter, all of size $O(\log M)$ bits [1]. Without any split and considering that all entries have bootstrapped, the initial space complexity is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M + \log n))$ bits. Each split costs two timestamps and a counter (size of a sub-cell). Let π be the number of splits, we have $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M + \log n) + \pi\log M)$ bits. Lemma 3.7 establishes the following space complexity bound: $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M + \log n) + \frac{1}{\varepsilon\gamma}\log\frac{1}{\delta}\log M)$ bits.

Each update requires to access each of the $\mathfrak{F}$ matrix entries in order to move the sliding window forward. However, we can achieve the same result by performing this phase-out operation (from Line 10 to Line 18) only on the $\mathfrak{F}$ matrix entries that are accessed by the update and query procedures [2]. Given this optimization, update and query require to lookup one entry by row of the $\mathfrak{F}$ matrix. Then, the query and update time complexities are $O(\log 1/\delta)$. □

Notice that the space complexity can be reduced by removing the entry counter v. However, the query time would increase since this counter must be reconstructed summing all the sub-cell counters.

[1] For the sake of clarity, timestamps are of size $O(\log m)$ bits in the pseudo-code while counters of size $O(\log M)$ bits are sufficient.

[2] For the sake of clarity, the pseudo-code does not implement this optimization.

Table 3.9 – Complexities comparison.

Algorithm	Space (bits)	Update time	Query time
`Count-Min` [30]	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
Perfect WCM	$O(M)$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
Simple WCM	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
Proportional WCM	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log M+\log n))$	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
Splitter WCM	$O(\frac{1}{\gamma\varepsilon}\log\frac{1}{\delta}(\log M+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
`ECM-Sketch` [76]	$O\left(\frac{1}{\varepsilon\ \varepsilon_{\mathtt{wave}}}\log\frac{1}{\delta}\left(\log^2\varepsilon_{\mathtt{wave}}M+\log n\right)\right)$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$

One can argue that sub-cell creations and destructions cause memory allocations and disposals. However, we believe that it is possible to avoid wild memory usage leveraging the sub-cell creation patterns, either through a smart memory allocator or a memory aware data structure.

Finally, Table 3.9 summarizes the space, update and query complexities of the presented algorithms, `Count-Min` and `ECM-Sketch`.

3.2.5 Distributed Windowed Count-Min Algorithm

The $\mathfrak{F}$ matrix is a linear-sketch data structure (*cf.,* Section 2.2.3), which is a suitable property in a distributed context. Each node can run locally the algorithm on its own stream σ_i ($i \in [s]$). To obtain the global matrix the coordinator has to sum [3] all the all the $\mathfrak{F}_i$ matrices ($i \in [s]$), $\mathfrak{F} = \bigoplus_{i\in[s]} \mathfrak{F}_i$. The coordinator would then able to provide the frequency estimation for each item on the global distributed stream $\sigma = \sigma_1 \cup \ldots \cup \sigma_s$.

Taking inspiration from [33], we can define the Distributed Windowed Count-Min (DWCM) algorithm, which sends the $\mathfrak{F}$ matrix to the coordinator each εM items. DWCM can be applied to the four aforementioned algorithms, resulting in a distributed frequency (ε, δ)-additive-approximation in the *sliding windowed functional monitoring* model.

Theorem 3.9 (DWCM Communication Complexity)**.** *DWCM communication complexity is* $O(\frac{s}{\varepsilon^2}\log\frac{1}{\delta}\log M)$ *bits per window.*

Proof. In each window and for each node S_i ($i \in [s]$), DWCM sends the $\mathfrak{F}$ matrix at most $\frac{M}{\varepsilon M} = 1/\varepsilon$ times. Thus the communication complexity is $O(\frac{s}{\varepsilon^2}\log\frac{1}{\delta}\log M)$ bits per window. □

Theorem 3.10 (DWCM Approximation)**.** *DWCM introduces an additive error of at most* $s\varepsilon M$, i.e, *the skew between any entry* $(\imath, \jmath)$ *of the global* $\mathfrak{F}$ *matrix at the coordinator and the sum of the entries* $(\imath, \jmath)$ *of the* $\mathfrak{F}_i$ *matrices* ($i \in [s]$) *on nodes is at most* $s\varepsilon M$.

Proof. Similarly to [33], the coordinator misses for each node S_i ($i \in [s]$) at most the last εM increments. Then, the entries of the global $\mathfrak{F}$ at the coordinator cannot fall behind by more than $s\varepsilon M$ increments with respect to the current sum of the s local matrices $\mathfrak{F}_i$. Thus DWCM introduces at most an additive error of $s\varepsilon M$. □

[3] The $\bigoplus$ operator sums the matrices entry by entry.

3.2.6 Time-based windows

We have presented the algorithms assuming count-based sliding windows, however all of them can be easily extended to time-based sliding windows. Recall that in time-based sliding windows the steps defining the size of the window are time ticks instead of item arrivals.

In each algorithm it is possible to split the update code into the subroutine increasing the $\mathfrak{F}$ matrix and the subroutine phasing out expired data (*i.e.*, decreasing the $\mathfrak{F}$ matrix). Let denote the former as UPDATESAMPLE and the latter as UPDATETICK. The algorithm runs the UPDATESAMPLE subroutine at each item arrival, and the UPDATETICK subroutine at each time tick. Note that time-stamps have to be updated using the current time tick count.

This modification affects the complexities of the algorithms, since M is no longer the number of items, but the number of time ticks. Thus, depending if the number of item arrivals per time tick is greater or lower than 1, the complexities improve or worsen

3.3 Item Values Estimating

In this section we introduce a generalization of the frequency estimation problem, *i.e.,* the *item value estimation* problem, as well as our solution Value Estimator (VALES).

In many streaming applications, such as centrality computation [50] in social networks, we want to known the value of a function $\psi(t)$ for each identifier t read from the stream. Since storing the value of ψ for each distinct item could not be feasible for memory constraints, then $\psi(t)$ is computed again and again, each time t shows up. If the function ψ is steady (or steady for long enough time frames), computing it more than once is a waste of computational resources. To reduce the amount of re-calculation of ψ, a typical approach is to store [4] the most frequent couples $(t, \psi(t))$ in a caching system [38,60,68], *i.e.,* a data structure with a small memory footprint.

Problem statement — The solution we propose takes a different angle: design a sketch returning an approximation $\widehat{\psi}(t_j)$ of $\psi(t_j)$ for all items $t \in [n]$, *i.e.,* not only the ones retained by the caching eviction policy. Let $\psi : [n] \to \mathbb{N}^+$ be an unknown and steady function, in other words its value can be different for each tuple t but does not change over time (*i.e.,* independent from j), then we can set $\psi(t_j) = \omega_t$.

Problem 3.11 (Item Value Estimation)**.** *Given a stream* $\sigma = \langle (t_1, \omega_{t_1}), \cdots, (t_m, \omega_{t_m}) \rangle$ *where* $\omega_t \in \mathbb{N}^+$ *is the value carried by item* t_j (i.e., ω_t *is independent from* j*), provide an estimate* $\widehat{\omega}_t$ *of the value* ω_t *for all items* $t \in [n]$.

Notice that, setting $\omega_t = 1,\ \forall t \in [n]$ reduces the item value estimation problem to frequency estimation. In addition, all the algorithms solving the frequency estimation problem presented previously also solve the item value estimation problem restricted to $\omega_t = C, \forall t \in [n]$, where $C \in \mathbb{N}^+$ is a known constant.

3.3.1 Value Estimator Algorithm

This section describes the details of the Value Estimator (VALES) algorithm, while Section 3.3.2 presents its theoretical analysis.

[4] there are several cache eviction or admission policies.

Listing 3.10 – VALES Algorithm.

```
1: init (r, c) do
2:     𝔉, 𝔚 ← 0_{r,c}
3:     h_1, ..., h_r : [n] → [c]          ▷ r hash functions from a 2-universal family.
4: end init
5: function UPDATE(t, ω_t)
6:     for ι = 1 to r do
7:         𝔉[ι, h_ι(t)] ← 𝔉[ι, h_ι(t)] + 1
8:         𝔚[ι, h_ι(t)] ← 𝔚[ι, h_ι(t)] + ω_t
9:     end for
10: end function
11: function GETESTIMATION(t)                 ▷ returns ω̂_t
12:     ι ← arg min_{ι∈[r]}{𝔉[ι, h_ι(t)]}
13:     return 𝔚[ι, h_ι(t)]/𝔉[ι, h_ι(t)]
14: end function
```

VALES (*cf.,* Listing 3.10) maintains two matrices the first one, denoted as $\mathfrak{F}$, tracks the tuple frequencies $\hat{f}_t$; the second one, denoted as $\mathfrak{W}$, tracks the tuples cumulated carried values $\omega_t \times \hat{f}_t$. Both matrices share the same size $r \times c$, where $r = log\frac{1}{\delta}$ and $c = \frac{e}{\varepsilon}$, and hash functions. The latter is the generalized version of the `Count-Min` presented in Section 3.1, where the entries associated with tuple t are increased with the value of ω_t. VALES updates (Lines 5–10) both matrices for each couple (t_j, ω_t) read from the stream. Ideally, we would like that

$$\exists \imath, \mathfrak{F}[\imath, h_\imath(t)] = f_t \text{ and } \mathfrak{W}[\imath, h_\imath(t)] = \omega_t f_t$$

Then, we would be able to return the exact value of $\omega_t = \mathfrak{W}[\imath, h_\imath(t)]/\mathfrak{F}[\imath, h_\imath(t)]$. However, as for the `Count-Min`, since $\varepsilon \ll [n]$, several items collide in the same entry. To minimize the impact of these collisions on the estimation of ω_t, VALES **(i)** retrieve from $\mathfrak{F}$ the coordinates $(\imath, \jmath = h_\imath(t))$ of the entry associated with t with the smallest value (Line 12); then **(ii)** it returns the ratio $\widehat{\omega}_t = \mathfrak{W}[\imath, h_\imath(t)]/\mathfrak{F}[\imath, h_\imath(t)]$ (Line 13).

Theorem 3.12 (VALES Time complexity)**.** *For each tuple read from the input stream and for each query, the time complexity of VALES is* $\mathcal{O}(\log 1/\delta)$.

Proof. By Listing 3.10, for each item read from the input stream, the algorithm increments an entry per row of both the $\mathfrak{F}$ and $\mathfrak{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting update time complexity is $O(\log 1/\delta)$. Similarly, each query requires to read an entry per row of both the $\mathfrak{F}$ and $\mathfrak{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting query time complexity is $O(\log 1/\delta)$. □

Theorem 3.13 (VALES Space Complexity)**.** *The space complexity of VALES is* $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n))$ *bits.*

Proof. VALES 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 VALES is $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$ bits. □

3.3.2 Theoretical Analysis

VALES uses two matrices, $\mathfrak{F}$ and $\mathfrak{W}$, to estimate value ω_t of each tuple t of σ. Let start considering a single row $\imath \in [r]$ of the two matrices and denote with C_t and with V_t the values of the entries associated to item t in this row, *i.e.*, $C_t = \mathfrak{F}[\imath, h_\imath(t)]$ and $V_t = \mathfrak{W}[\imath, h_\imath(t)]$. From the `Count-Min` algorithm, and for any $t \in [n]$, we have for a given 2-universal hash function h,

$$C_t = \sum_{u=1}^{n} f_u \mathbb{1}_{\{h(u)=h(t)\}} = f_t + \sum_{u=1, u\neq t}^{n} f_u \mathbb{1}_{\{h(u)=h(t)\}}.$$

and

$$V_t = f_t\omega_t + \sum_{u=1, u\neq t}^{n} f_u\omega_u \mathbb{1}_{\{h(u)=h(t)\}},$$

Let us denote by $\min_\omega$ and $\max_\omega$ the respectively minimum and maximum values of $\omega_t, \forall t \in \sigma$. We have trivially

$$\frac{V_t}{C_t} \leq \frac{f_t \max_\omega + \sum_{u=1,u\neq t}^{n} f_u \max_\omega \mathbb{1}_{\{h(u)=h(t)\}}}{f_t + \sum_{u=1,u\neq t}^{n} f_u \mathbb{1}_{\{h(u)=h(t)\}}},$$

and thus

$$\min_\omega \leq \frac{V_t}{C_t} \leq \max_\omega.$$

This first bound guarantees that, at least, the estimation $\widehat{\omega_t}$ returned by VALES belong to the interval of $[\min_\omega, \max_\omega]$.

For any $\imath = 0, \ldots, n-1$, we denote by $U_\imath(t)$ the set whose elements are the subsets of $[n] \setminus \{t\}$ whose size is equal to $\imath$, that is

$$U_\imath(t) = \{J \subseteq [n] \setminus \{t\} \mid |J| = \imath\}.$$

We have $U_0(t) = \{\emptyset\}$.

For any $t \in [n]$, $\imath = 0, \ldots, n-1$ and $J \in U_\imath(t)$, we introduce the event $B(t, \imath, J)$ defined by

$$B(t, \imath, J) = \{h(u) = h(t),\ \forall u \in J \text{ and } h(u) \neq h(t), \forall u \in [n] \setminus (J \cup \{t\})\}$$

From the independence of the $h(u)$, we have

$$\mathbb{P}[B(t, \imath, J)] = \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath}.$$

Let us consider the ratio V_t/C_t. For any $\imath = 0, \ldots, n$, we define

$$R_\imath(t) = \left\{ \frac{f_t w_t + \sum_{u\in J} f_u w_u}{f_t + \sum_{u\in J} f_u},\ J \in U_\imath(t) \right\}.$$

We have $R_0(t) = \{w_t\}$. We introduce the set $R(t)$ defined by

$$R(t) = \bigcup_{\imath=0}^{n-1} R_\imath(t).$$

Thus with probability 1, we have $V_t/C_t \in R(t)$.

Theorem 3.14 (VALES Expected Value). *Let C_t and V_t be the values of the entries associated with item t in row $\imath$,* i.e., *$C_t = \mathfrak{F}[\imath, h_\imath(t)]$ and $V_t = \mathfrak{W}[\imath, h_\imath(t)]$, then*

$$\mathbb{E}[V_t/C_t] = \frac{\sum_{u=1}^{n} w_u - w_t}{n-1} - \frac{c(\sum_{u=1}^{n} w_u - nw_t)}{n(n-1)} \left(1 - \left(1 - \frac{1}{c}\right)^n\right).$$

Proof. Let $x \in R(t)$. We have

$$\begin{aligned}
\mathbb{P}[V_t/C_t = x] &= \sum_{\imath=0}^{n-1} \sum_{J \in U_\imath(t)} \mathbb{P}[V_t/C_t = x \mid B(t,\imath,J)]\mathbb{P}[B(t,\imath,J)] \\
&= \sum_{\imath=0}^{n-1} \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath} \sum_{J \in U_\imath(t)} \mathbb{P}[V_t/C_t = x \mid B(t,\imath,J)] \\
&= \sum_{\imath=0}^{n-1} \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath} \sum_{J \in U_\imath(t)} \mathbb{1}_{\{x = X(t,J)\}}.
\end{aligned}$$

where $X(t,J)$ is the fraction:

$$X(t,J) = \frac{f_t w_t + \sum_{u \in J} f_u w_u}{f_t + \sum_{u \in J} f_u}.$$

Note that we have $\sum_{x \in R(t)} \mathbb{1}_{\{x = X(t,J)\}} = 1$, thus

$$\begin{aligned}
\mathbb{E}[V_t/C_t] &= \sum_{\imath=0}^{n-1} \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath} \sum_{J \in U_\imath(t)} \sum_{x \in R(t)} x \, \mathbb{1}_{\{x = X(t,J)\}} \\
&= \sum_{\imath=0}^{n-1} \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath} \sum_{J \in U_\imath(t)} \frac{f_t w_t + \sum_{u \in J} f_u w_u}{f_t + \sum_{u \in J} f_u}.
\end{aligned}$$

□

Applying the Markov's inequality to Theorem 3.14, we have that VALES returns an (ε, δ)-additive-approximation of ω_t.

Finally, let us assume that all the f_t are equal, that is for each t, we have $f_t = m/n$. Simulations (*cf.,* Sections 7.2.3 and 8.3) tend to show that the worst cases scenario of input streams are exhibited when all the items show the same number of occurrences in the input stream. We get

$$\mathbb{P}[V_t/C_t = x] = \sum_{\imath=0}^{n-1} \left(\frac{1}{c}\right)^\imath \left(1 - \frac{1}{c}\right)^{n-1-\imath} \sum_{J \in U_\imath(t)} \mathbb{1}_{\{x = (w_t + \sum_{u \in J} w_u)/(\imath+1)\}}.$$

Notice that this result does not depend on m.

Chapter Summary

This chapter discussed the frequency estimation problem and presented two algorithms, Proportional WCM and Splitter WCM, solving the frequency estimation problem in the sliding window. It has also presented the value estimation problem, which generalises the frequency estimation problem, and our proposed solution VALES.

The next chapter deals with a tightly related problem, the heavy hitters problem.

Chapter 4

Heavy Hitters

In this chapter we look at the heavy hitters problem. After discussing the related work, in Section 4.2 we provide a mean case analysis for the `Space Saving`, a well known algorithm that deterministically solves this problem. Section 4.3 introduces Distributed Heavy Hitters Estimator, an (ε, δ)-approximation for the distributed heavy hitters problem in a more general model than the one considered in previous works. Finally, Section 4.4 defines the balanced partition problem, *i.e.,* build a partition $\mathcal{K}$ of the streams item universe $[n]$ with exactly k parts, where the parts are balanced. Section 4.4 also presents the Balanced Partitioner algorithm that, reusing the results shown in Section 4.2, builds a $(1+\theta)$-optimal balanced k-partition.

Real-time analysis of network traffic [39] is one among the application fields of data stream techniques. A specific problem that has claimed a lot of attention in the community focuses on the tracking of frequent packets and/or IP addresses in the traffic flowing through a router. For instance, this information can be used for accounting, performance tuning or detecting malicious behaviour. The abstraction that is used in general to capture, among others, the notion of frequent packets and/or IP addresses is *heavy hitters*.

An item t of a stream is called heavy hitter if the empirical probability p_t with which item t appears in the stream satisfies $p_t \geq \theta$ for some given threshold $0 < \theta \leq 1$. In other words, a heavy hitter is an item t which frequency f_t satisfies $f_t \geq \theta m$. In the following we call *sparse items* the items of the stream that are not heavy hitters. We denote with $\mathcal{HH} \subseteq [n]$ and $\mathcal{SI} \subseteq [n]$ the set of heavy hitters and of sparse items. There are many different definitions and related problems to heavy hitters, such as approximate counts and the θ-frequent problem. However, there are two main definitions for the heavy hitters problem: simply identifying the heavy hitters, or also provide an estimation of the their frequencies. Obviously, the second is a harder definition of the problem, and is the one that we consider in this thesis.

Notice that there is a tight relation between the heavy hitters and the frequency estimation problems since one can be reduced to the other, but with some limitations. Knowing the frequency of all the items let you also identify the items satisfying the heavy hitters problem. On the other hand, one could argue that the heavy hitters are the most representative elements of the stream. Thus, a possible solution to the frequency estimation problem would be to return the estimated frequencies for the heavy hitters and 0 for the sparse items.

Moreover, the heavy hitters problem can be reduced to the top-k (*i.e.,* finding the k most frequent items) problem. Given the heavy hitters threshold θ, once can compute

the maximum number of possible heavy hitters in the stream (*i.e.,* when the distribution is uniform). Thus, setting $k = \lceil 1/\theta \rceil$, an algorithm solving the top-k problem (and providing their frequency estimation) also solve the heavy hitters problem. Notice that the converse does not hold: without any additional information on the stream distribution, one cannot derive the correct value of θ given the value of k. Finally, one can reduce the top-k problem to the frequency estimation problem.

4.1 Related Work

Data streaming model — Misra and Gries [70] provide an elegant deterministic sampling algorithm for this problem. The `MG` algorithm keeps a associative array MG of $\langle item, counter \rangle$ pairs. For each item t, if the associated counter exists in MG, then it is increased. Otherwise, if there is room, a new pair is added to MG. When $\mid MG \mid \geq \lceil \frac{1}{\varepsilon} \rceil - 1$, where $\varepsilon \leq \theta$ for each new item (*i.e.*, not yet in MG), all counters are decreased. If a counter reaches 0, the pair is removed from MG. At all times there are no more than $\lceil \frac{1}{\varepsilon} \rceil - 1$ pairs in MG, and using a slightly clever data structure, we achieve a $O(\frac{1}{\varepsilon}(\log m + \log n)$ space complexity and $O(1)$ for time complexity. The heavy hitters are the items contained in MG, and their under-estimation is the value of the associated counter which satisfies the following quality bounds: $f_t - \varepsilon m \leq \hat{f}_t \leq f_t$ for any $0 < \varepsilon \leq \theta \leq 1$.

`Sticky Sampling` and `Lossy Counting` are two well known algorithms proposed by Manku and Motwani [67]. However their space complexities, respectively $O(\frac{1}{\varepsilon} \log \frac{1}{\theta\delta}$ $(\log n + \log m))$ bits and $O(\frac{1}{\varepsilon} \log \varepsilon m$ $(\log n + \log m))$ bits have been superseded by another counter-based solution: the `Space Saving` algorithm presented by Metwally *et al.* [69]. `Space Saving` has two parameters: θ and ε, such that $0 < \varepsilon < \theta \leq 1$. Similarly to `MG` [70], it maintains an associative array SS of $\langle item, counter, error \rangle$ triples which maximum size is set to $\lceil 1/\varepsilon \rceil$. The heavy hitters are all items in SS where $counter - error \geq \theta m$ (*cf.,* Listing 4.1), and their frequency estimation is the value of $counter$. The over-estimation made by the algorithm on the estimated frequency $\hat{f}_t$ of heavy hitter t verifies $f_t + \varepsilon m \geq \hat{f}_t \geq f_t$ for any $0 < \varepsilon < \theta \leq 1$.

Windowed model — Golab *et al.* [47] propose a deterministic algorithm providing the heavy hitters with sliding windows with a space complexity of $O(n \log 1/\varepsilon + M\varepsilon/\theta$ $(\log 1/\varepsilon + \log M))$ bits. The following year, Arasu and Gurmeet Singh [12] provide a solution for ε-approximate frequency counts (similar to heavy hitters) with jumping windows. They use a data structure $C_{M,\varepsilon}$, called sliding window sketch, made of $\lceil \log \frac{4}{\varepsilon} \rceil$ levels each made of contiguous blocks (*i.e.,* sub-windows). The higher level has a single block, and, going down, the number of block doubles at each level. Each block uses the Misra-Gries [70] algorithm to track frequent items. Blocks are said to be active if all its sample belong to the current window. The estimation provided by higher-level blocks is more accurate than the estimation from lower-level blocks. The estimation over the active window is the sum of the estimations from the non-overlapping active blocks, starting from the highest level. The sliding window sketch has a space complexity of $O(\frac{1}{\varepsilon} \log^2 \frac{1}{\varepsilon})$ bits. They also apply the sliding window sketch to ε-approximating quantiles and provide a randomized version of both sliding window algorithms. Homen and Carvaldo [58], building on top of their previous contribution [57], extend the `Space Saving` [69] algorithm to the sliding window mode. The basic principle is to enrich the entries of the SS data structure with histograms tracking the counts in sub-windows of

Listing 4.1 – `Space Saving` algorithm.

```
1: init (θ, ε) do
2:     SS: an associative array of ⟨item, counter, error⟩ triples
3:     min ← 0
4:     m ← 0
5: end init
6: function UPDATE(t_j)                          ▷ reads item t_j from the stream σ
7:     m = m + 1
8:     if ⟨t, counter, error⟩ ∈ SS then
9:         SS ← SS ∪ ⟨t, counter + 1, error⟩
10:    else if |SS| < 1/ε then
11:        SS ← SS ∪ ⟨t, 1, 0⟩
12:    else
13:        let ⟨v, counter, error⟩ ∈ SS be such that counter = min
14:        SS ← SS \ {⟨v, counter, error⟩}
15:        SS ← SS ∪ ⟨t, counter + 1, counter⟩
16:    end if
17:    min ← smallest value of counter in SS
18: end function
19: function GETHEAVYHITTERS()          ▷ returns detected heavy hitters as a list of ⟨t, f̂_t⟩
20:    for all ⟨t, counter, error⟩ ∈ SS do
21:        if counter − error ≥ ⌊θm⌋ then
22:            ret ← ret ∪ ⟨t, counter⟩
23:        end if
24:    end for
25:    return ret
26: end function
```

fixed sizes.

Distributed model — The detection of heavy hitters originating from multiple streams (*i.e., distributed heavy hitters* problem) has first been studied by Manjhi *et al.* [66]. In their paper, the authors propose a solution for detecting recent distributed heavy hitters by relying on a multi-level structure where each node must guarantee a degree of precision that depends on its level in the hierarchical structure. This structure helps to minimise the communication between nodes and the central coordinator. On the other hand, they make the assumption that distributed heavy hitters are locally detectable at each node. In [96], the authors do not assume any more that nodes can locally detect distributed heavy hitters, however they suppose that there exists a gap between sparse items frequencies and heavy hitters ones. That is, items that appear at least θm times from the inception of the stream are heavy hitters, while there are no items whose frequency belongs to an interval $(a\theta m, \theta m)$, with $0 < a < 1$. Based on this assumption, the authors can accurately detect the presence of distributed heavy hitters, however they do not provide their frequency estimation. In [97], the authors propose a solution that identifies items whose aggregated frequency over the distributed streams exceeds some given threshold, irrespective of the size of the input streams. Thus, their motivation is to detect heavy hitters only during a fixed time interval (note that the same definition has been adopted by [96]). The authors in [97] suppose that each node locally knows the exact frequency of all the items it will receive, prior to the algorithm execution. This model is commonly called the "distributed bag model". Finally, Yi and Zhang

in [93] propose a solution in the spirit of the distributed functional monitoring model to minimise the communication cost between the nodes and the coordinator. However each node maintains at any time the exact frequency of each received item in its stream, *i.e.,* their solution is not space efficient.

4.2 Space Saving Mean Case

In the data streaming field, most of the research assumes that the input stream is manipulated by an adversary. Such worst case scenario rarely occurs in many real-world applications and do not capture the notion of average case analysis [51]. This is in particular the case for the `Space Saving` algorithm [69] where the accuracy bounds provided by the authors are for a worst case scenario. In order to provided a more fitting analysis for applications where we want to use the `Space Saving` algorithm in a non worst case scenario, we derive in Theorem 4.1 the expected error made by the `Space Saving` algorithm.

Theorem 4.1 (`Space Saving` Mean Error)**.**
For any $0 < \varepsilon < \theta$, the expected error made by `Space Saving` algorithm to estimate the frequency of any heavy hitter t is bounded by $1 - \varepsilon$, that is,

$$0 \leq \mathbb{E}[\hat{f}_t] - f_t \leq 1 - \varepsilon,$$

where $\hat{f}_t$ represents the estimated frequency of item t.

Proof. We denote by $\mathbf{p} = (p_1, \ldots, p_n)$ the probability distribution of the occurrence of the items $1, \ldots, n$ in the input stream. For $j = 1, \ldots, m$, we denote by Y_j the value of the item at position j in the stream. These random variables are supposed to be independent and identically distributed with probability distribution $\mathbf{p}$. We thus have $\mathbb{P}[Y_j = t] = p_t$ and $f_t = mp_t$. Items in the stream are successively selected at each discrete time $j \geq 1$, and are inserted in the `Space Saving` buffer (if not already present) according to the `Space Saving` algorithm (*cf.,* Listing 4.1). In the following $SS(j)$ denote the content of `Space Saving` at discrete time j. We necessarily have $|\mathcal{HH}| \leq 1/\theta < 1/\varepsilon$ (where $\mathcal{HH}$ is the set of heavy hitters). It has been proven in [69] that at time m all the items $t \in \mathcal{HH}$ are present in $SS(m)$. It follows that for every $t \in \mathcal{HH}$ there exists a random time $Z_t \leq m$ such that item t is inserted for the last time in SS at time Z_t (*i.e.,* item t was not present in the `Space Saving` buffer at time $Z_t - 1$ and thus replaces another item), and is never removed from SS afterwards. More precisely, we have

$$Z_t = \inf\{j \geq 1 \mid t \in SS(a) \text{ for every } a = j, \ldots, m\}.$$

For every $t \in SS(j)$, we denote by $V_t(j)$ the counter value of item t at discrete time j. For every $t \in \mathcal{HH}$, by definition of Z_t item t is not in the buffer at time $Z_t - 1$, thus we have,

$$V_t(Z_t) = \min\{V_v(Z_t - 1),\ v \in SS(Z_t - 1)\} + 1,$$

and again by definition of Z_t, we have, for every $j = Z_t + 1, \ldots, m$,

$$V_t(j) = V_t(j-1) + \mathbb{1}_{\{Y_j = t\}}.$$

Putting together these results, we get

$$V_t(m) = \min\{V_v(Z_t - 1),\ v \in SS(Z_t - 1)\} + 1 + N_t(Z_t + 1, m),$$

where $N_t(a, a') = \sum_{j=a}^{a'} \mathbb{1}_{\{Y_j=t\}}$ is the number of occurrences of item t in the stream between discrete instants a and a'. It is easy to see that $N_t(a, a')$ has a binomial distribution with parameters $a' - a + 1$ and p_t. This relation can also be written as

$$0 \leq V_t(m) - N_t(1, m) = \min\{V_v(Z_t - 1),\ v \in SS(Z_t - 1)\} + 1 - N_t(1, Z_t).$$

Note that this also implies that $N_t(1, Z_t) \leq \min\{V_v(Z_t - 1),\ v \in SS(Z_t - 1)\} + 1$.

Since for every $j \geq 1$, we have $\min\{V_t(j),\ t \in SS(j)\} \leq \varepsilon j$, we obtain, taking the expectations,

$$0 \leq \mathbb{E}\{V_t(m)\} - mp_t \leq \mathbb{E}[Z_t - 1]\varepsilon + 1 - \mathbb{E}[Z_t]p_t.$$

This leads to

$$0 \leq \mathbb{E}[V_t(m)] - mp_t \leq 1 - \varepsilon + \mathbb{E}[Z_t](\varepsilon - p_t).$$

By the `Space Saving` algorithm, $V_t(m)$ represents the estimated frequency of any heavy hitter t of the input stream, since $\varepsilon - p_t \leq 0$ we finally get

$$0 \leq \mathbb{E}[V_t(m)] - mp_t \leq 1 - \varepsilon.$$

□

This analysis assumes sampling with replacement. The same analysis applies to sampling without replacement (hypergeometric distribution) because this distribution and the binomial distribution have the same mean.

Using the Markov inequality and the fact that $p_t \geq \varepsilon$, we get for all $\varepsilon > 0$,

$$\mathbb{P}\left[\frac{V_t(m) - mp_t}{mp_t} \geq \varepsilon\right] \leq \frac{1/\varepsilon - 1}{mp_t} \leq \frac{1 - \varepsilon}{\varepsilon^2 m}.$$

Note that the last relation shows in particular that the relative error $\dfrac{V_t(m) - mp_t}{mp_t}$ converges in probability (and thus in law) to 0 when m tends to ∞, *i.e.,* for all $\varepsilon > 0$, we have

$$\lim_{m \longrightarrow \infty} \mathbb{P}\left[\frac{V_t(m) - mp_t}{mp_t} \geq \varepsilon\right] = 0.$$

4.3 Distributed Heavy Hitters

Identifying distributed heavy hitters has applications in several areas. In a Content Delivery Network (CDN) [66] with several nodes (*e.g.,* Akamai [4]), the caching policy can be improved knowing the frequently accessed contents. Considering system monitoring, detecting that some Dynamically Linked Libraries (DLL) [97] are being modified on a large number of hosts through the organisation may help tracking down an unknown malware.

Problem Statement — The distributed heavy problem has first been formalised by Estan and Varghese [39], and then adapted to different contexts. Here we extend this problem to the general distributed functional monitoring model [31]. Informally, the distributed heavy hitters problem lies, for the coordinator, in quickly identifying any item t whose aggregated number of occurrences over the union of the distributed streams approximately exceeds some given fraction of the total size of all the streams since their inception. Such an item t is called a *distributed heavy hitter*.

Problem 4.2 (Distributed Heavy Hitters). *For any given threshold $\theta \in (0,1]$, approximation parameter $\varepsilon \in [0,1]$, probability of failure $\delta \leq 1/2$, and distributed stream $\sigma = \sigma_1 \cup \ldots \cup \sigma_s$ the distributed heavy hitters problem consists for the coordinator in*

- *outputting item t if $f_t \geq \theta m$, and*
- *never outputting item t if $f_t < (1-\varepsilon)\theta m$.*

with $f_t = \sum_{i=1}^{s} f_{t,i}$ and $m = \sum_{i=1}^{s} m_i$, where m_i and $f_{t,i}$ are respectively the size of sub-stream σ_i and the frequency of item t in σ_i.

4.3.1 Distributed Heavy Hitters Estimator Algorithm

We now propose the Distributed Heavy Hitters Estimator algorithm (DHHE) that solves the distributed heavy hitters problem without making any assumption on the probability of occurrence of items in σ.

With respect to previous work (*cf.,* Section 4.1), we do not assume that the distributed heavy hitters are locally detectable. Furthermore, our solution provides an estimation of the heavy hitters frequencies and is space efficient.

DHHE requires the knowledge of the probability $p_{t,i}$, *i.e.,* the empirical probability of occurrence of item t in the sub-stream σ_i, for all items $t \in [n]$ and for each sub-stream $\sigma_{i\in[s]}$. However this information is in general unknown. To overcome this problem, DHHE runs at each node $S_{i\in[s]}$ an instance of the `Count-Min` algorithm, which provides an estimation $\hat{f}_t$ of the frequency f_t of each item t in the local stream σ_i. Given this estimation, we are then able to compute an estimation $\widehat{p}_{t,i} = \hat{f}_t/m_i$ of $p_{t,i}$.

The `Count-Min` algorithm strongly relies on 2-universal hash functions, to guarantee an (ε,δ)-approximation of item frequencies. If all the s nodes use the same set of hash functions in $\mathcal{H}$, one may argue that the adversary could launch a dictionary attack to easily tamper with these estimations. In our solution, each of the s nodes locally and independently chooses at random its hash functions from $\mathcal{H}$. We show in Section 4.3.2 that this does not prevent the coordinator from (ε,δ)-approximating the detection of distributed heavy hitters.

The pseudo-code of the algorithms run by each of the s nodes and by the coordinator are respectively provided in Listing 4.2, 4.4 and 4.5. The algorithm works as follows. Each node $S_i \in \mathcal{I}$ reads on the fly and sequentially its input stream σ_i. For each received item t_j, node S_i updates its `Count-Min` instance $CountMin$, and maintains the current size m_i of its input stream, which is the number of items received so far in σ_i. If $\widehat{p}_{t,i} \geq \theta$ then t is a potential candidate for being a heavy hitter and thus S_i keeps track of item t if not already done (*cf.,* Lines 14–35 of Listing 4.2). This is achieved by maintaining g buffers $\mathcal{G}_{1,i},\ldots,\mathcal{G}_{g,i}$, where $g = \lfloor \log_2 s \rfloor + 1$ (*cf.,* Figure 4.3) on each node S_i. The interval $[\theta,1]$ is split into g sub-intervals in a geometric way. Each buffer $\mathcal{G}_{\kappa,i}$, whose size is denoted by ν_κ, only contains items t whose estimated probability $\widehat{p}_{t,i}$ matches the κ-th interval. That is, buffer $\mathcal{G}_{\kappa,i}$, with $1 \leq \kappa \leq g-1$, contains items t whose estimated probability $\widehat{p}_{t,i}$ verifies $\theta + (1-\theta)/2^{\kappa} < \widehat{p}_{t,i} \leq \theta + (1-\theta)/2^{\kappa-1}$, and $\mathcal{G}_g$ contains items t whose estimated probability $\widehat{p}_{t,i}$ satisfies $\theta \leq \widehat{p}_{t,i} \leq \theta + (1-\theta)/2^{g-1}$. The size ν_κ of each of buffer $\mathcal{G}_{\kappa,i}$, with $1 \leq \kappa \leq g$, is set to $\lceil n_\kappa \times \rho \rceil$, where n_κ is the maximum number of items that could fit the κ-th buffer, and $0 < \rho \leq 1$. Since $\mathcal{G}_{\kappa,i}$, for $1 \leq \kappa \leq g-1$, contains items whose estimated probability $\widehat{p}_{t,i}$ verifies $\theta + (1-\theta)/2^{\kappa} < \widehat{p}_{t,i} \leq \theta + (1-\theta)/2^{\kappa-1}$, we set n_κ to $\lfloor 1/(\theta + (1-\theta)/2^{\kappa}) \rfloor$ and $n_g = \lfloor 1/\theta \rfloor$. Note that for $\rho = 1$, that is for all κ, $\nu_\kappa = n_\kappa$, if some buffer say $\mathcal{G}_{\kappa,i}$ is filled with ν_κ items then it means that all the items t in the stream are uniformly distributed with $\widehat{p}_{t,i} = \theta + (1-\theta)/2^{\kappa}$ (and $\widehat{p}_{t,i} = \theta$ for $\kappa = g$), and thus none of the other buffers can receive any single item. In practice ρ is

Listing 4.2 – DHHE algorithm at any node $S_i \in \mathcal{I}$.

1: **init** $(\theta,\ \rho,\ g,\ r,\ c)$ **do**
2: **for all** $\kappa \in \{1, g-1\}$ **do**
3: $\mathcal{G}_{\kappa,i} \leftarrow \emptyset$; $\nu_\kappa \leftarrow \lceil \rho \lfloor 1/(\theta + (1-\theta)/2^\kappa) \rfloor \rceil$; $\tau_\kappa \leftarrow (\text{not-active}, 0)$
4: **end for**
5: $\mathcal{G}_g \leftarrow \emptyset$; $\nu_g \leftarrow \lceil \rho \lfloor 1/\theta \rfloor \rceil$; $\tau_g \leftarrow (\text{not-active}, 0)$
6: $m_i \leftarrow 0$; $Freq_i \leftarrow \emptyset$; $\mathcal{J}_i \leftarrow \emptyset$
7: $CountMin \leftarrow$ `Count-Min` with parameters r and c.
8: **end init**
9: **function** UPDATE(t_j) ▷ reads item t_j from the stream σ
10: $CountMin$.UPDATE(t)
11: $\hat{f}_t \leftarrow CountMin$.GETFREQ($t$)
12: $m_i \leftarrow m_i + 1$
13: $\widehat{p}_{t,i} \leftarrow \hat{f}_t / m_i$
14: **if** $\widehat{p}_{t,i} \geq \theta$ **then**
15: **if** $\widehat{p}_{t,i} \leq \theta + (1-\theta)/2^{g-1}$ **then**
16: $\kappa \leftarrow g$
17: **else**
18: $\kappa \leftarrow \min\{a \in \{1, g-1\} \mid \theta + (1-\theta)/2^a < \widehat{p}_{t,i}\}$
19: **end if**
20: **if** $t \notin \mathcal{G}_{\kappa,i}$ **then**
21: **if** $\mathcal{G}_{\kappa,i} = \emptyset$ **then**
22: $\tau_\kappa \leftarrow (\text{active}, 0)$
23: **end if**
24: $\mathcal{G}_{\kappa,i} \leftarrow \mathcal{G}_{\kappa,i} \cup \{t\}$
25: **if** $|\mathcal{G}_{\kappa,i}| = \nu_\kappa$ **then**
26: **for all** $t \in \mathcal{G}_{\kappa,i}$ **do**
27: $Freq_i \leftarrow Freq_i \cup \{\langle t, \hat{f}_t \rangle\}$
28: **end for**
29: send message $\langle warn, m_i, Freq_i \rangle$ to coordinator
30: $\mathcal{G}_{\kappa,i} \leftarrow \emptyset$
31: $Freq_i \leftarrow \emptyset$
32: $\tau_\kappa \leftarrow \langle \text{not-active}, 0 \rangle$
33: **end if**
34: **end if**
35: **end if**
36: **for all** active timer τ_κ, $k \in \{1, g\}$ **do**
37: $\tau_\kappa \leftarrow (\text{active}, \tau_\kappa + 1)$
38: **if** $\tau_\kappa > H_{\lfloor 1/\theta \rfloor}/\theta$ **and** $\mathcal{G}_{\kappa,i} \neq \emptyset$ **then**
39: **for all** $t \in \mathcal{G}_{\kappa,i}$ **do**
40: $Freq_i \leftarrow Freq_i \cup \{\langle t, \hat{f}_t \rangle\}$
41: **end for**
42: send message $\langle warn, m_i, Freq_i \rangle$ to coordinator
43: $\mathcal{G}_{\kappa,i} \leftarrow \emptyset$
44: $Freq_i \leftarrow \emptyset$
45: $\tau_\kappa \leftarrow \langle \text{not-active}, 0 \rangle$
46: **end if**
47: **end for**
48: **end function**

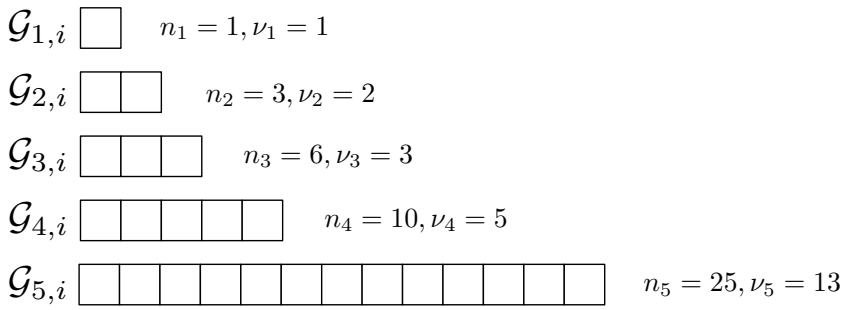


Figure 4.3 – Data structure on node $S_i \in \mathcal{I}$, with $\theta = 0.04$, $\rho = 0.5$ and $s = 20$.

set to a value less than or equal to 1/2. When some buffer $\mathcal{G}_{\kappa,i}$ is full (*i.e*, $\mathcal{G}_{\kappa,i}$ contains ν_κ items), node S_i sends all the items of $\mathcal{G}_{\kappa,i}$, together with their frequency counts and the stream size m_i to the coordinator (*cf.,* Lines 25–32 of Listing 4.2), and empties $\mathcal{G}_{\kappa,i}$. Upon receipt of such a buffer, the coordinator queries the other $s-1$ nodes to get the frequency of each of the items sent by node S_i, if any (*cf.,* Lines 6–11 of Listing 4.5).

Combining all these pieces of information, the coordinator checks whether any of these items is a distributed heavy hitter or not (*cf.,* Lines 14–18 of Listing 4.5).

Finally, since the distribution of the items in σ_i is unknown, one cannot guarantee that ν_κ distinct items with a probability that matches $\mathcal{G}_{\kappa,i}$ exist in the stream σ_i. Actually, even no item can appear in the stream with those probabilities. Thus to guarantee that all the potential heavy hitters in $\mathcal{G}_{\kappa,i}$ are at some point sent to the coordinator, a timer τ_κ is set upon receipt of the first item in $\mathcal{G}_{\kappa,i}$, and is incremented each time node S_i reads an item from the input stream σ_i. Timeout τ_κ of $\mathcal{G}_{\kappa,i}$ is set to $H_{\lfloor 1/\theta \rfloor}/\theta$, where $H_\imath$ is the $\imath$-th harmonic number defined by $H_0 = 0$ and $H_\imath = 1 + 1/2 + \cdots + 1/\imath$. Lemma 4.3 shows the derivation of this timeout. Each time the coordinator detects a distributed heavy hitter t, it informs all the s nodes that t is such (this amounts to returning GI in Line 20 of Listing 4.5 to both the application and the s nodes). Nodes locally enqueue t (in FIFO order) in a specific buffer, denoted by $\mathcal{J}_i$, whose size is equal to $\lceil \rho/\theta \rceil$. Hence, when a node S_i detects that t is a potential heavy hitter, if t belongs to $\mathcal{J}_i$ node S_i does not notify the coordinator. Note that as the oldest detected heavy hitters are progressively dequeued, then $\mathcal{J}_i$ contains at any time the last $\lceil \rho/\theta \rceil$ heavy hitters. The rationale of this feedback is to prevent nodes from repeatedly informing the coordinator about already detected distributed heavy hitters.

4.3.2 Theoretical Analysis

In this section we provide an upper bound of the number of bits communicated between the s nodes and the coordinator (*cf.,* Theorem 4.4), and show that DHHE is an (ε, δ)-approximation of the distributed heavy hitters problem, for any $\varepsilon \in (0, 1)$ and probability of failure $\delta \leq 1/2$, *i.e.,* solves Problem 4.2.

For the sake of clarity, we first perform the analysis on FK_{DHHE}, which is the Full Knowledge version of DHHE, *i.e.,* FK_{DHHE} knows the exact frequencies f_t of each item t, and thus also the exact probabilities of occurrence p_t. Next we show that DHHE, which instead relies on the estimation provided by the `Count-Min`, is an (ε, δ)-approximation of FK_{DHHE}.

Listing 4.4 – DHHE algorithm at any node $S_i \in \mathcal{I}$ (contd.)

upon receive $\langle request, ID \rangle$ from coordinator **do**
  for all $t \in ID$ **do**
    $Freq_i \leftarrow Freq_i \cup \{\langle t, CountMin.\text{GETFREQ}(t) \rangle\}$
    if $\exists \kappa \in [g], t \in \mathcal{G}_{\kappa,i}$ **then**
      $\mathcal{G}_{\kappa,i} \leftarrow \mathcal{G}_{\kappa,i} \setminus \{t\}$
    end if
  end for
  send message $\langle reply, m_i, Freq_i \rangle$ to C
end upon
upon receive $\langle detection, GI \rangle$ from coordinator **do**
  enqueue GI in $\mathcal{J}_i$
end upon

Listing 4.5 – DHHE algorithm at the coordinator.

upon receive $\langle warning, m_i, Freq_i \rangle$ from node S_i **do**
  $m \leftarrow m_i$
  $GI \leftarrow Freq_i$
  $ID \leftarrow$ identifiers in $Freq_i$
  for all $S_\jmath \in \mathcal{I} \setminus \{S_i\}$ **do**
    send message $\langle request, ID \rangle$ to S_i
  end for
  for all received message $\langle reply, m_\jmath, Freq_\jmath \rangle$ from $S_\jmath$ **do**
    $m \leftarrow m + m_\jmath$
    for all $\langle v, \hat{f}_{v,\jmath} \rangle \in Freq_\jmath \wedge \langle v, \hat{f}_v \rangle \in GI$ **do**
      $GI \leftarrow GI \cup \{\langle v, \hat{f}_v + \hat{f}_{v,\jmath} \rangle\}$
    end for
  end for
  for all $(v, \hat{f}_v) \in GI$ **do**
    if $\hat{f}_v < \theta m$ **then**
      $GI \leftarrow GI \setminus \{\langle v, \hat{f}_v \rangle\}$
    end if
  end for
  if $GI \neq \emptyset$ **then**
    return GI
    send message $\langle detection, GI \rangle$ to each $S_\imath \in \mathcal{S}$
  end if
end upon

FK$_{DHHE}$ Communication Complexity

To Maximise the communication between the s nodes and the coordinator the adversary has to manipulate the s input streams so that the time needed to locally fill the buffers is minimised, and thus the frequency at which nodes communicate with the coordinator is maximised. Theorem 4.4 shows that this is achieved if, for all $S_i \in \mathcal{I}$, all the heavy hitters appear with the same probability in σ_s.

Communication between the s nodes and coordinator is triggered each time a buffer becomes full or upon timeout. Indeed, we have to determine the number of items, in expectation, that must be received at any node $S_i \in \mathcal{I}$ to locally fill buffer $\mathcal{G}_\kappa$, $1 \leq \kappa \leq g$. To analyse the communication cost, we study a generalisation of the coupon collector problem [10,11,41]. The number $T_{a,n}(\mathbf{p})$ is the number of items that have to be read from a stream with a probability distribution $\mathbf{p}$ in order to collect a different items belonging to $[n]$.

In the following, we denote by $U_{a,n}$ all the sub-sets of items of $[n]$ whose size is exactly equal to a, that is $U_{a,n} = \{V \subseteq [n] \mid |V| = a\}$. Note that we have $U_{0,n} = \{\emptyset\}$. For every $V \in U_{a,n}$, we define $P_V = \sum_{t \in V} p_t$, with $P_\emptyset = 0$.

For each $\kappa = 1, \ldots, g-1$, we denote with V_κ the set of all the items t in σ_i whose probability of occurrence $p_{t,i}$ verifies $\theta + (1-\theta)/2^\kappa < p_{t,i} \leq \theta + (1-\theta)/2^{\kappa-1}$, *i.e.,* the items that can be stored in $\mathcal{G}_\kappa$. We can than define $v_k = |V_\kappa|$ and we have $v_k \leq n_k$ (recall from Section 4.3.1, that n_κ represents the maximal number of items whose probability of occurrence is equal to the lower bound of the range, that is $n_\kappa = \lfloor 1/(\theta + (1-\theta)/2^\kappa) \rfloor$). For $\kappa = g$, we denote by V_g the set of all the items t in σ_i whose probability of occurrence $p_{t,i}$ verifies $\theta < p_{t,i} \leq \theta + (1-\theta)/2^{g-1}$. We set $v_g = |V_g|$, and we have $v_g \leq n_g$ with $n_g = \lfloor 1/\theta \rfloor$. We also define $\mathbf{p}_{\kappa,i} = (\mathbf{p}_i)_{t \in V_\kappa}$. Hence, $U_{a,v_\kappa} = \{V \subseteq V_\kappa \mid |V| = a\}$, and for every $V \in U_{a,v_\kappa}$, $P_V = \sum_{t \in V} p_{t,i}$.

Then, for each node $S_i \in \mathcal{I}$, the expected number of items that need to be received from a stream σ_i with a probability distribution $\mathbf{p}_i$ to fill buffer Γ_κ is given by $\mathbb{E}[T_{\nu_\kappa,V_\kappa}(\mathbf{p}_i)]$.

Timer Dimensioning — The lenght of the timer $\tau_{\kappa \in [g]}$ is constrained by two opposite needs. A short timer yields a better detection latency but sends the buffer more often. On the other end, a long timer reduces the communication but may delay the detection of a distributed heavy hitter. The basic idea is to set a length that matches the expected time to fill the buffer, independently of the frequency distribution. In [9] we prove the following lemma:

Lemma 4.3 (Coupon Collector Expected Value)**.** *Let For every $n \geq 1$, $a = 1, \ldots, n$, and $0 < \theta \leq p_t$ for $t \in [n]$, we have*

$$\mathbb{E}[T_{a,n}(\mathbf{p})] \leq H_{\lfloor 1/\theta \rfloor}/\theta$$

Thus in accordance with Lemma 4.3, we set for every $\kappa = 1, \ldots, g$, $\tau_\kappa = H_{\lfloor 1/\theta \rfloor}/\theta$.

Theorem 4.4 (FK$_{DHHE}$ Communication Cost Upper Bound)**.** *The FK$_{DHHE}$ exchanges in average no more than*

$$2ms(\log m + \log n) \sum_{\kappa=1}^{g} \frac{\nu_\kappa}{\sum_{i=0}^{\tau-1} \mathbb{P}[T_{\nu_\kappa,n_\kappa}(\mathbf{u}_\kappa) > i]} \text{ bits.} \tag{4.1}$$

where $\mathbf{u}_\kappa$ is the uniform distribution with n items, i.e., $\forall t \in V_\kappa, u_{t,\kappa} = 1/\nu_\kappa$

Proof. From Listings 4.2, 4.4 and 4.5, node $S_i \in \mathcal{I}$ triggers a transmission with the coordinator each time one of its buffers is full or upon timeout. Thus, assuming that S_i maintains a single buffer $\mathcal{G}_{\kappa,i}$, then for m_i large, the expected number of times node S_i sends the content of $\mathcal{G}_{\kappa,i}$ to the coordinator, denoted $C_{\kappa,i}$, verifies

$$C_{\kappa,i} \leq \frac{m_i}{\mathbb{E}[\min(T_{\nu_\kappa,v_\kappa}(\mathbf{p}_{\kappa,i})),\tau]} = \frac{m_i}{\sum_{\imath=0}^{\tau-1}\mathbb{P}[T_{\nu_\kappa,v_\kappa}(\mathbf{p}_{\kappa,i}) > \imath]},$$

where $\mathbb{P}[T_{\nu_\kappa,v_\kappa}(\mathbf{p}_{\kappa,i}) > a] =$

$$\sum_{\imath=0}^{\nu_\kappa-1}(-1)^{\nu_\kappa-1-\imath}\binom{v_\kappa-\imath-1}{v_\kappa-\nu_\kappa}\sum_{V\in S_{\imath,v_\kappa}}(p_0+P_V)^a.$$

It has been observed (Theorem 3 in [10]) that for vectors $\mathbf{p}_{\kappa,i} = (p_{t,i})_{t\in V_\kappa}$ and with [1] $p_{0,\kappa} = 1 - P_{V_\kappa}$ and $0 < \theta \leq p_{t,i}$,

$$\mathbb{P}[T_{\nu_\kappa,v_\kappa}(\mathbf{u}_\kappa) > a] \leq \mathbb{P}[T_{\nu_\kappa,v_\kappa}(\mathbf{p}_{\kappa,i}) > a].$$

Thus, we have

$$C_{\kappa,i} \leq \frac{m_i}{\sum_{\imath=0}^{\tau-1}\mathbb{P}[T_{\nu_\kappa,v_\kappa}(\mathbf{p}_{\kappa,i}) > \imath]} \leq \frac{m_i}{\sum_{\imath=0}^{\tau-1}\mathbb{P}[T_{\nu_\kappa,n_\kappa}(\mathbf{u}_\kappa) > \imath]}.$$

Finally, we need to determine the number of bits sent each time buffer $\mathcal{G}_{\kappa,i}$ is full or upon timeout. From Listing 4.2, the sending of message *warning* requires $\log m_i + \nu_\kappa(\log n + \log m_i)$ bits to be transmitted to the coordinator (where $\log m_i$ represents the number of bits needed to code item frequencies, and $\log n$ the one needed to code item identifiers). By Listing 4.5, this triggers a round trip communication between the coordinator and the remaining $s-1$ nodes to collect the frequencies of all potential heavy hitters of $\mathcal{G}_{\kappa,i}$ (Line 56 of Listing 4.4 and Line 6 of 4.5). This generates respectively $s-1$ messages of $\nu_\kappa \log n$ bits from the coordinator and a message of $\log m_{i'} + \nu_\kappa(\log m_{i'} + \log n)$ bits from each node $S_{i'} \in \mathcal{I} \neq S_i$, where $m_{i'}$ is the size of $\sigma_{i'}$. As the sum of all the local streams is equal to m, the number of bits sent because $\mathcal{G}_{\kappa,i}$ is full is less than $2s\nu_\kappa(\log m + \log n)$. Thus, the fact that $C_{\kappa,i}$ must be computed for all the buffers at every node $S_i \in \mathcal{I}$, allows us to complete the proof of the lemma.

Note that when $\theta \to 0$, which is the case in the distributed heavy hitters problem, we have $\tau \to \infty$, and the denominator in Relation 4.1 tends to $n_\kappa(H_{n_\kappa} - H_{n_\kappa-\nu_\kappa})$ [8]. □

$\mathbf{FK}_{DHHE}$ Correctness

We investigate whether malicious nodes can prevent the detection of a heavy hitter. This amounts to showing that, for any given threshold $\theta \in (0,1]$ and approximation parameter $\varepsilon \in (0,1]$, the adversary cannot (by finely distributing the occurrences of item t among the s streams) compel the coordinator to return any items t such that $f_t < (1-\varepsilon)\theta m$ and cannot prevent the coordinator from returning all items t such that $f_t \geq \theta m$. We now prove that the FK_{DHHE} algorithm guarantee such properties.

[1] Notice that P_{V_κ} may be smaller than 1 and then $\mathbf{p}_{\kappa,i}$ is not a probability distribution. To overcome this issue we introduce the item 0 with probability of occurrence $p_{0,\kappa}$. Item 0 can be drawn but does not count towards the number of distinct items collected

Theorem 4.5 (FK$_{DHHE}$ Correctness)**.** *The FK$_{DHHE}$ algorithm deterministically and exactly solves the distributed heavy hitters problem (*i.e*, $\delta = 0$ and $\varepsilon = 0$).*

Proof. The proof consists in showing that the algorithm does not output any items t such that $f_t < \theta m$ (*i.e.*, no false positive) and returns all items t such that $f_t \geq \theta m$ (*i.e.*, no false negative).

Let us first focus on false positives. Suppose by contradiction that the coordinator returns at time j some item t such that $f_t < \theta m$. Then, by Listing 4.5, this means that the coordinator has received from some node $S_i \in \mathcal{I}$ a message *warning* for which $t \in Freq_i$. By Listing 4.2, this can only happens if node S_i has identified t as a potential heavy hitter, that is, $p_{t,i} \geq \theta$ (see Line 14 of Listing 4.2). Now, upon receipt of $Freq_i$, the coordinator collects from the other $s-1$ nodes the frequency of each item $t' \in Freq_i$, and in particular t frequency, as well as the current size of their input stream (*cf.,* Lines 6–11 of Listing 4.5). By Lines 15 and 16, the coordinator removes from GI all the items whose frequency is less than θm, and in particular item t. Thus the coordinator cannot return t at time j.

A false negative means that the coordinator does not return a true global heavy hitter. Suppose that there exists t such that $f_t \geq \theta m$ and t is not returned. Thus there exists at least one stream σ_i such that $p_{t,i} \geq \theta$. By Lines 15–24 of Listing 4.2, t is inserted in the buffer $\mathcal{G}_\kappa$ that matches its occurrence probability $p_{t,i}$ (if not already present). By Lines 29 and 42, t is sent to the coordinator after at most $H_{\lfloor 1/\theta \rfloor}/\theta$ reading. By applying an argument similar to the above case, we get a contradiction with the assumption of the case. □

To summarize, Theorem 4.5 has shown that FK$_{DHHE}$ accurately tracks distributed heavy hitters, even if they are hidden in distributed streams. In addition, we have provided with Theorem 4.4 an upper bound on the communication cost between the s nodes and the coordinator. This bound is reached when each buffer is fed with items that all occur with the same probability.

DHHE Approximation

The following lemma shows that using the `Count-Min` algorithm to estimate $\widehat{p}_{t,i} = \hat{f}_t/m$, DHHE is a (ε, δ)-approximation of FK$_{DHHE}$, for any $\varepsilon \in (0,1)$ and probability of failure $\delta \leq 1/2$. This is achieved by sharpening the bounds obtained in [30] and by relying on Theorem 4.5. In [8] we prove the following lemma:

Lemma 4.6 (Refined `Count-Min` Accuracy Bounds)**.** *For a stream σ, for any $\varepsilon \in (0,1)$ and probability of failure $\delta \leq 1/2$, the `Count-Min` algorithm with $r = \lceil \log(1/\delta) \rceil$ rows and $c = \lceil 2(1-\theta)/\varepsilon\theta \rceil$ columns guarantees that*

$$\forall t \in \sigma, \begin{cases} \mathbb{P}\left[\hat{f}_t - f_t \geq \varepsilon f_t\right] \leq \delta & \text{if } p_t \geq \theta \\ \mathbb{P}\left[\hat{f}_t - f \geq \varepsilon\,(m-f)\right] \leq \delta & \text{otherwise.} \end{cases}$$

Theorem 4.7 (DHHE Approximation)**.** *The DHHE algorithm uses $O((\log n + \log m)$ $\log(1/\delta)$ $(1/\theta - 1)/\varepsilon +$ $(\log s \log n)/\theta)$ bits of space on each node to (ε, δ)-approximate tracking of distributed heavy hitters in arbitrarily distributed input streams.*

Proof. By Lemma 4.6 applied to any stream σ_i of length m_i, with $S_i \in \mathcal{I}$, the `Count-Min` algorithm returns an (ε, δ)-approximation of $f_{t,i}$ for all $t \in \sigma_i$. Thus, using $((1/\varepsilon\theta) \log(1/\delta) (\log n + \log m_s))$ bits of space, DHHE (ε, δ)-approximates the tracking of potential distributed heavy hitters at each node S_i. We now demonstrate that the coordinator correctly (ε, δ)-approximates the frequency of each item that has been locally detected as potential distributed heavy hitter at any node S_i. We have,

$$\begin{aligned}\mathbb{P}\left[\sum_{S_i \in \mathcal{I}} \hat{f}_{t,i} - f_t \geq \varepsilon(m - f_t)\right] &= \mathbb{P}\left[\sum_{S_i \in \mathcal{I}} \left(\hat{f}_{t,i} - f_{t,i}\right) \geq \varepsilon(m - f_t)\right] \\ &= \mathbb{P}\left[\sum_{S_i \in \mathcal{I}} \min_{u \in [r]} X_{t,i}^{(u)} \geq \varepsilon(m - f_t)\right],\end{aligned}$$

where $X_{t,i}^{(u)}$ denote the random variable that measures, given $u \in [r]$, the excess of a counter $\mathfrak{F}[u][h_{u,i}(t)]$ on a specific node S_i. We clearly have $\min_{u \in [r]} \sum_{S_i \in \mathcal{I}} X_{t,i}^{(u)} \geq \sum_{S_i \in \mathcal{I}} \min_{u \in \{1,\ldots,r\}} X_{t,i}^{(u)}$. Then, by the mutual independence of the r estimators, and since $\mathbb{E}[X_{t,i}^{(u)}] \leq \frac{m_i - f_{t,i}}{r}$ [30], we have

$$\begin{aligned}&\mathbb{P}\left[\sum_{S_i \in \mathcal{I}} \hat{f}_{t,i} - f_t \geq \varepsilon(m - f_t)\right] \\ &\leq \prod_{u=1}^{r} \mathbb{P}\left[\sum_{S_i \in \mathcal{I}} X_{t,i}^{(u)} < \varepsilon(m - f_t)\right] \leq \prod_{u=1}^{r} \frac{\mathbb{E}\left[\sum_{S_i \in \mathcal{I}} X_{t,i}^{(u)}\right]}{\varepsilon(m - f_t)} \\ &= \prod_{u=1}^{r} \frac{\sum_{S_i \in \mathcal{I}} (m_s - f_{t,i})}{c\varepsilon(m - f_t)} = \prod_{u=1}^{r} \frac{m - f_t}{c\varepsilon(m - f_t)} \leq \frac{1}{2^r} \leq \delta.\end{aligned}$$

Similarly, by applying the same argument as above and the one of Lemma 4.6, we get that $\mathbb{P}\left\{\sum_{S_i \in \mathcal{I}} \hat{f}_{t,i} - f_t \geq \varepsilon f_t\right\} \leq \delta$ if $f_t \geq \theta m$ in the global stream σ. Hence, DHHE (ε, δ)-approximate FK_{DHHE}. By Theorem 4.5, FK_{DHHE} implements the detection of distributed heavy hitters while being robust to any biased distributed input streams, which completes the second part of the proof. Finally, by Lemma 4.6, on each node, $O((\log n + \log m_i) \log(1/\delta)(1/\theta - 1)/\varepsilon)$ bits of space are required to (ε, δ)-approximate item frequencies. Moreover, the space required to locally track potential heavy hitters is the sum of the size used by each buffer $\mathcal{G}_{\kappa,i}$, that is $\sum_{\kappa=1}^{g} \nu_\kappa \log n$ bits. By construction $\nu_\kappa \leq 1/\theta$. Thus, an upper bound of the total space used is equal to $O(\frac{1}{\theta} \log s \log n)$. □

4.4 Balanced Partitioning

Load balancing is a problem that arises in many fields, such as scheduling tasks on multiprocessors, flow routing in *software defined networks* (SDN), or the partitioning of large graphs being loaded in a distributed graph-based computation framework [28]. In general the problem can be described as follows: let $[n]$ be the universe of items $t \in [n]$, build a partition $\mathcal{K}$ of k parts, each containing roughly the same number of items. This problem can trivially be solved through hash functions (*e.g.,* consistent hashing [63] or 2-universal

hash functions), which can build a fast, deterministic and compact mapping from $[n]$ to $[k]$.

Instead, if we take into account that each item may have a weight ω_t (non negative integer), then the balancing should be performed with respect to the cumulated weight of the items in a part. Notice that hash functions are usually designed to uniformly spread values from their domain to their codomain; if ω_t values have a skewed distribution, the mapping induced by the hash function may not be balanced.

A solution could lie in defining an explicit one-to-one mapping between $[n]$ and $[k]$ that could take into account the skewed value distribution and thus uniformly spread the weight; this solution is considered impractical as it requires to have precise knowledge on the input value distribution (usually not known *a priori*) and imposes a memory footprint that is proportional to the number of possible values in the input domain to store the mapping (commonly a huge number if you consider, as an example, the domains of length-constrained strings or floating-point numbers).

In this section we propose to translate this problem in the data streaming model, and tackle it with its techniques.

Problem Statement — We consider a single node S (*i.e.,* $s = 1$) receiving an input stream $\sigma = \langle t_1, \cdots, t_m \rangle$ with $t_j \in [n]$. Let $\mathcal{K}$ be a partition of the item universe $[n]$, *i.e.,* $\emptyset \notin \mathcal{K}$, $\bigcup_{p \in [k]} O_p = [n]$ and $O_p, O_{p'} \in \mathcal{K} \implies O_p \cap O_{p'} = \emptyset$. We can define the weight W_p of each part $O_p \in \mathcal{K}$ as the sum of the cumulated weights of the items belonging to the part, *i.e.,* $W_p = \sum_{t \in O_p} \omega_t f_t$. However, the weight ω_t an item can be modelled as if, for each occurrence of item t, the item would appear ω_t times and not just once. In other words, we can redefine the frequency of item t in the stream as $f_t = \omega_t \times f'_t$ (where f'_t is the original frequency in the stream).

Thus, we can redefine the weight W_p of each part $O_p \in \mathcal{K}$ as the sum of the frequencies of the items belonging to the part, *i.e.,* $W_p = \sum_{t \in O_p} f_t$. Ideally, to be balanced, the weights of the parts must satisfy the following equation:

$$\forall O_p, O_{p'} \in \mathcal{K}, W_p = W_{p'}. \tag{4.2}$$

However, whether Equation 4.2 can be satisfied or not depends on frequency vector $\mathbf{f}$ (*e.g.,* if $m \mod k \neq 0$). Then, to embrace all possible configurations, we define the following problem:

Problem 4.8 (Balanced Partitioning)**.** *Given a stream $\sigma = \langle t_1, \cdots, t_m \rangle$ with $t_j \in [n]$, build a partition $\mathcal{K}$ of $[n]$ with exactly k parts ($|\mathcal{K}| = k$) such that the weights $W_{p \in [k]} = \sum_{t \in O_p} f_t$ of the parts $O_p \in \mathcal{K}$ minimize $\max_{p \in [k]} W_p$.*

Notice that, in all configurations where Equation 4.2 can be satisfied, minimizing $\max_{p \in [k]} W_p$ or Equation 4.2 is equivalent.

4.4.1 Balanced Partitioner Algorithm

In this section we present our solution, Balanced Partitioner (BPART), consisting of a two-phase algorithm: **(i)** in the *learn* phase the algorithm becomes aware of the stream distribution; **(ii)** in the following *build* phase, it constructs a global mapping function taking into account the previously gathered knowledge of the stream.

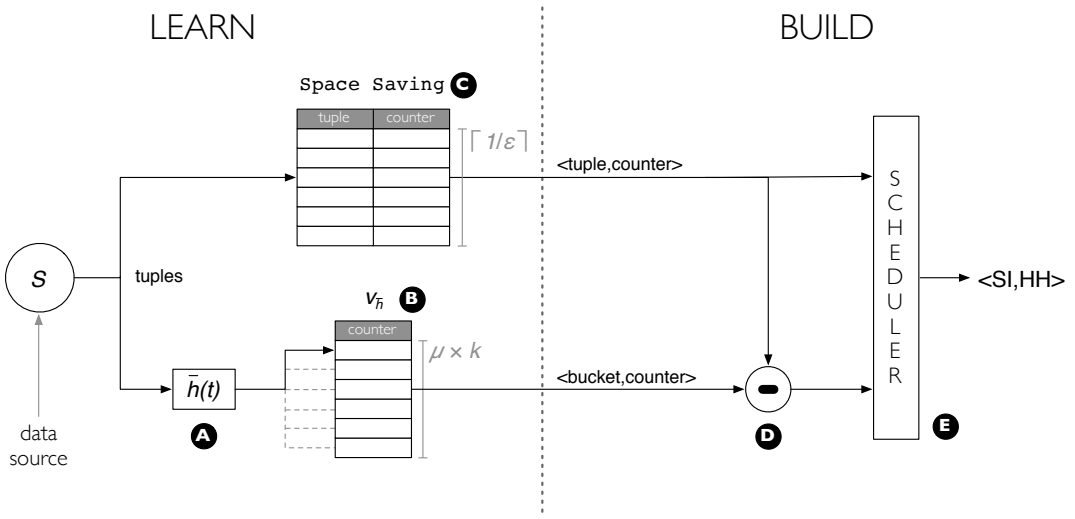


Figure 4.6 – BPART architecture and working phases.

Greedy Multi Processor Scheduling (GMPS) — A classical problem in the load balancing literature is to schedule independent tasks on identical machines minimizing the makespan, *i.e.,* the *multiprocessor scheduling* problem. We adapt this problem to our setting: we have **(i)** a set of jobs $\mathcal{B}$ and **(ii)** a set of instances $\mathcal{K}$. Each job $b \in \mathcal{B}$ has an processing time $w(b)$, and each instance $O_p \in \mathcal{K}$ has an total processing time W_p. The total processing time of instance O_p is equal to the sum of the processing times of the jobs assigned by the scheduling to instance O_p. The goal is to associate each job $b \in \mathcal{B}$ to an instance $O_p \in \mathcal{K}$, minimizing the maximum total processing time on the instances: $\max_{p=1,\dots,k} W_p$. To solve efficiently this problem, known to be NP-hard, a classic solution is the Least Processing Time First (LPTF) approximation algorithm. The algorithm (referred as GMPS algorithm in the following) assigns the job $b \in \mathcal{B}$ with the largest processing time $w(b)$ to the instance $O_p \in \mathcal{K}$ with the lowest load W_p, then removes b from $\mathcal{B}$ and repeats until $\mathcal{B}$ is empty. It is proven [49] that this algorithm provides a $(\frac{4}{3} - \frac{1}{3k})$-approximation of the optimal mapping.

As previously mentioned, the challenge in building the partitiong $\mathcal{K}$ is mainly due to the skewness in the input stream distribution (and/or in the values of the weights ω_t). For instance, let u and v be the stream heavy hitters, most likely the partition should assign them to different parts. In addition, if $f_u > m/k$, the partition should isolate u in a part. However, a randomly chosen hash function will almost certainly assign other (sparse) items with u. Even worse, the hash function may end up putting u and v in the same part.

To cope with these issues, BPART becomes aware of the stream distribution through a learning phase. It is then able to build a partition that avoids pathological configurations and that achieves close to optimal balancing. Listing 4.7 show the pseudo code for both phases. BPART (Figure 4.6.A) chooses a hash function $\overline{h} : [n] \to [k\mu]$ randomly from a 2-universal hash functions family, where μ is a user defined parameter (Line 2). Increasing the co-domain size reduces the collision probability, thus enhances the odds of getting a good assignment. Having more buckets (elements of $\overline{h}$ co-domain) than parts, we can assign buckets to parts minimizing the unbalancing. More in details, BPART feeds to

Listing 4.7 – BPART Algorithm.

```
1: init (θ, ε, k, μ) do
2:     h̄ : [n] → [kμ] : a randomly chosen 2-universal hash functions.
3:     v_h̄ array of size μ × k.
4:     SpaceSaving ← Space Saving algorithm instance with parameters ε and θ.
5:     HH associative array mapping heavy hitters to parts.
6:     SI associative array mapping h̄ buckets to parts.
7: end init
8: function LEARN(t : item)
9:     v_h̄[ h̄(t) ] ← v_h̄[ h̄(t) ] + 1
10:    SpaceSaving.UPDATE(t)
11: end function
12: function BUILD()
13:    SS ← SpaceSaving.GETHEAVYHITTERS()
14:    for all ⟨v, f̂_v⟩ ∈ SS do
15:        v_h̄[ h̄(v) ] ← v_h̄[ h̄(v) ] − f̂_v
16:    end for
17:    ⟨ SI, HH ⟩ ← GMPS( v_h̄, SS )
18: end function
19: function GETPARTITION(t : item)
20:    if t ∈ HH then
21:        return HH[ t ]
22:    else
23:        return SI [ h̄(t) ]
24:    end if
25: end function
```

the previously presented GMPS algorithm the buckets of $\overline{h}$ as the set of jobs $\mathcal{B}$ and the number of parts/instances k, *i.e.,* $|\mathcal{K}| = k$. The "execution time" of bucket/job b fed to the GMPS algorithm is the sum of the frequencies of the item belonging to the bucket, *i.e.,* $w(b) = \sum_{t\in[n]} f_t \mathbb{1}_{\overline{h}(t)=b}$

The frequencies of $\overline{h}$'s buckets are computed in the learning phase as follows. BPART keeps an array $\mathbf{v}_{\overline{h}}$ of size $k\mu$ (Line 3). When receiving item t, BPART increments the entry associated through $\overline{h}$ with t (Line 9, Figure 4.6.B). In other words $\mathbf{v}_{\overline{h}}$ represents how $\overline{h}$ maps the stream σ items to its own buckets.

While this mechanism improves the balancing, it still does not deterministically guarantee that heavy hitters are correctly handled. If the buckets have in average the same load, the GMPS algorithm should be able to produce a good mapping. To approximate this ideal configuration we have to remove all the heavy hitters. As such, BPART uses the `Space Saving` algorithm (Line 4) to detect heavy hitters and manage them ad-hoc. To match the required detection and estimation precisions, the `Space Saving` algorithm monitors $1/\varepsilon$ distinct keys, where $\varepsilon < \theta$. Recall that θ is the relative frequency of heavy hitters and is a user defined parameter. In the learning phase, BPART updates the `Space Saving` algorithm instance with the values of t (Line 10, Figure 4.6 point C). At the end of the learning phase, the `Space Saving` algorithm stores the most frequent values of t and their estimated frequencies. Then (Figure 4.6.D), BPART removes the frequency count of each heavy hitter from $\mathbf{v}_{\overline{h}}$ (Line 15). Finally (Figure 4.6.E), it feeds to the

GMPS algorithm the $\mathbf{v}_{\overline{h}}$ array and the heavy hitters from the `Space Saving` algorithm, with the frequencies of both, as the set of jobs $\mathcal{B}$. The GMPS algorithm returns an approximation of the optimal mapping from $\overline{h}$ buckets and detected heavy hitters to instances (Line 17).

Theorem 4.9 (BPART Time Complexity)**.**
BPART time complexity is $O(\log 1/\varepsilon)$ *per update in the learning phase,* $O((k\mu + 1/\varepsilon) \log(k\mu+1/\varepsilon))$ *to build the global mapping function, and* $O(1)$ *to return the part associated with an item.*

Proof. By Listing 4.7, the learning phase performs a hash of an integer value, increments an array entry and updates the `Space Saving`. Leveraging the implementation proposed in [69] with a hash table and a min-heap, both of size $O(1/\varepsilon)$, the worst case `Space Saving` update time complexity is $O(\log 1/\varepsilon)$. Notice however that the mean time complexity is $O(1)$.

By Listing 4.7, building the global mapping requires **(i)** to build the sorted set of jobs $\mathcal{B}$ with the heavy hitters in `Space Saving` and the buckets of $\overline{h}$, **(ii)** run the GMPS algorithm on $\mathcal{B}$ and **(iii)** populate the HH and SI data structures. Implementing $\mathcal{B}$ as a binary search tree, the first phase can be achieved in $O\left((k\mu + 1/\varepsilon) \log\left(k\mu + 1/\varepsilon\right)\right)$ steps. Implementing the set $\mathcal{K}$ as a max-heap, GMPS runs in $O\left((k\mu + 1/\theta) \log k\right)$. Finally, the construction of both HH and SI requires $O\left(k\mu + 1/\theta\right)$ steps.

By Listing 4.7, implementing HH as a hash table and SI as an array, the part retrieval time complexity is $O(1)$ □

Theorem 4.10 (BPART Space Complexity)**.**
BPART space complexity is $O\left((k\mu + 1/\varepsilon) \log m + \log n\right)$ *bits in the learning phase and to build the partition* $\mathcal{K}$*. To store the partition* $\mathcal{K}$*, BPART requires* $O\left((k\mu + 1/\theta) \log n\right)$ *bits.*

Proof. By Listing 4.7, the learning phase stores a 2-universal hash function $\overline{h}$ using $O(\log n + \log(k\mu))$ bits, an array of $O(k\mu \log m)$ bits and, through the `Space Saving` instance, a hash table and a min-heap both of $O(1/\varepsilon \log m)$ bits. Building the partition $\mathcal{K}$ adds a binary search tree and a heap of $O\left((k\mu + 1/\varepsilon) \log m\right)$ bits. Finally, the partition $\mathcal{K}$ (Listing 4.7) is made of the 2-universal hash function $\overline{h}$, the hash table HH of $O(1/\theta \log n)$ bits and the array SI of $O(k\mu \log k)$ bits. □

4.4.2 Theoretical Analysis

In this section we analyse the quality of the balancing of the partitioning built by BPART, considering that the data set follows a Zipfian distribution (a frequent case in many application scenarios [18, 64]) as well as that the order of the items in the stream is random (*i.e.,* no adversary).

We characterize the mean error made by our algorithm with respect to the *optimal* mapping according to the *percentage of imbalance* metric [77], which is defined as follows:

$$\lambda(\mathbf{W}) = \left(\frac{\max_{p \in [k]} \left(W_p\right)}{\overline{W}} - 1 \right) \times 100 \tag{4.3}$$

where W_p is the weight on part O_p, $\overline{W}$ is the mean weight over all k parts and $\mathbf{W}$ is the vector of size k of the parts weights.

Theorem 4.11 (BPART Approximation)**.** *BPART provides an $(1+\theta)$-optimal mapping for key grouping using $O(k\mu \log m + \log n)$ bits of memory in the learning phase and $O(k\mu \log n)$ bits of memory to store the global mapping function, where $\mu \geq 1/\theta \geq k$*

Proof. The accuracy of the estimated frequency of heavy hitters is given by Theorem 4.1. We now estimate the mean value of each counter in $\mathbf{v}_{\overline{h}}$, after having removed from these counters the weights of the heavy hitters (*cf.,* Line 15 of Listing 4.7). The value of every counter $\mathbf{v}_{\overline{h}}[\imath]$, $1 \leq \imath \leq \mu k$, is equal to the sum of the exact frequencies of all the received items t such that $\overline{h}(t) = \imath$, that is, the sum of the frequencies of all the tuples that share the same hashed value. We denote by X_i the random variable that measures the value of $\mathbf{v}_{\overline{h}}[\imath]$. We have

$$X_\imath = \sum_{t=1}^{n} f_t \mathbb{1}_{\{\overline{h}(t)=\imath\}} - \sum_{t \in \mathcal{HH}} \hat{f}_t \mathbb{1}_{\{\overline{h}(t)=\imath\}},$$

where, as defined above, $\hat{f}_t$ represents the frequency of heavy hitter $t \in \mathcal{HH}$ as estimated by the `Space Saving` algorithm.

By the 2-universality property of the family from which hash function $\overline{h}$ is drawn, we have $\mathbb{P}[\overline{h}(v) = \overline{h}(u)] \leq 1/(k\mu)$. Thus, by linearity of the expectation,

$$\mathbb{E}[X_\imath] = \sum_{t=1}^{n} \mathbb{E}\left[f_t \mathbb{1}_{\{\overline{h}(t)=\imath\}}\right] - \sum_{t \in \mathcal{HH}} \mathbb{E}\left[\hat{f}_t \mathbb{1}_{\{\overline{h}(t)=\imath\}}\right] \leq \frac{m - \sum_{t \in \mathcal{HH}} \mathbb{E}\left[\hat{f}_t\right]}{k\mu}.$$

Let m_{SI} denote the sum of the frequencies of all sparse items. From Theorem 4.1, we get

$$\mathbb{E}[X_\imath] \leq \frac{m - \sum_{t \in \mathcal{HH}} f_t}{k\mu} \leq \frac{m_{SI}}{k\mu}.$$

By definition $\mu \geq 1/\varepsilon$ and $\varepsilon \leq \theta$, thus $k\mu \geq 1/\theta$, that is in average, there is at most one heavy hitter in each counter of $\mathbf{v}_{\overline{h}}$. Thus, once the frequency of heavy hitters has been removed from the counters, the mean error of each counter is upper bounded by 0 and lower bounded by $1 - \varepsilon$ (from Theorem 4.1). We now estimate the percentage of imbalance λ (Relation 4.3) of the mapping provided by BPART with respect to the optimal one. Let $\mathbf{W}^{BPART}$ denote the weight vector induced by the mapping provided by BPART and $\mathbf{W}^{OPT}$ denote the optimal one. The error Y introduced by BPART is given by

$$Y = \lambda(\mathbf{W}^{BPART}) - \lambda(\mathbf{W}^{OPT}) = \frac{\max_{p \in [k]} W_p^{BPART} - \max_{p \in [k]} W_p^{OPT}}{\overline{W}}.$$

The analysis of the expected value of Y is split into three cases. Recall that any item whose probability of occurrence is greater than or equal to θ is a heavy hitter, and conservatively, we set $\theta \leq 1/k$.

Case 1 $\exists t \in \mathcal{HH}$ such that $f_t > \overline{W} = m/k$. Let v denote the item with the largest frequency. We have $\lambda(\mathbf{W}^{OPT}) = f_v k/m$ since the optimal solution cannot do better than scheduling item v alone to a single part. We also have $\lambda(\mathbf{W}^{BPART}) = f_v k/m$ since from above $\hat{f}_v$ ε-approximates f_j with probability close to 1 (by construction, $\varepsilon \gg 1/m$), and thus GMPS also isolates item v to a single part. Thus the error Y introduced by BPART in Case 1 is null.

Case 2 $\forall t \in \mathcal{HH}$, we have $\theta m \leq f_t \leq \overline{W}$. By construction of GMPS (that maps each tuple into the less loaded part in the decreasing frequency order), and by the fact that the value of each counter of $\mathbf{v}_{\overline{h}}$ is equal to $m_{SI}/(k\mu)$, we have that $\arg\max_{p\in[k]} W_p^{BPART}$ corresponds to the last part chosen by GMPS. This statement can be trivially proven by contradiction (if the last mapping does not correspond to the final largest loaded one, it must exist another part that has been chosen previously by GMPS without being the smallest loaded one). Let $\mathbf{W}'$ be the load vector before this last mapping. We have in average

$$\sum_{p=1}^{k} W'_p = \sum_{p=1}^{k} \mathbf{W}_p^{BPART} - \frac{m_{SI}}{k\mu}.$$

Thus, one can easily check that $\min_{p\in[k]} W'_p$ cannot exceed $(m_{SI} - m_{SI}/(k\mu))/k = (k\mu - 1)m_{SI}/k^2\mu$, leading to

$$\max_{p\in[k]} W_p^{BPART} - \overline{W} = \left(\min_{p\in[k]} W'_p + \frac{m_{SI}}{k\mu}\right) - \frac{m}{k} \leq \frac{(\mu+1)m_{SI}}{k\mu} \tag{4.4}$$

Moreover, by assumption, the distribution of the items in the stream follows a Zipfian distribution with an unknown parameter α. This means that, for any $t \in [n]$, $p_t = 1/(t^\alpha H_{n,\alpha})$ where $H_{n,\alpha}$ is the n-th generalized harmonic number, that is $H_{n,\alpha} = \sum_{t=1}^{n} 1/t^\alpha$ (we assume without loss of generality that items are ranked according to their decreasing probability). We have

$$m_{SI} = m - \sum_{t\in\mathcal{HH}} f_t = m\left(1 - \frac{H_{|\mathcal{HH}|,\alpha}}{H_{n,\alpha}}\right). \tag{4.5}$$

Recall that $|\mathcal{HH}| = |\{t \in [n] \mid p_t \geq \theta\}|$. Thus, we get

$$\frac{1}{(|\mathcal{HH}|+1)^\alpha H_{n,\alpha}} < \theta \quad \text{that is} \quad |\mathcal{HH}| > (\theta H_{n,\alpha})^{-1/\alpha} - 1.$$

From Relations 4.4 and 4.5 we have

$$\max_{p\in[k]} W_p^{BPART} - \overline{W} \leq \frac{(\mu+1)\left(H_{n,\alpha} - H_{|\mathcal{HH}|,\alpha}\right)m}{k\mu H_{n,\alpha}} \leq \frac{(\mu+1)m}{k\mu}.$$

Given the fact that $\max_{p\in[k]} W_p^{OPT} \geq \overline{W} = m/k$ by definition, the expected error introduced by BPART is given by

$$\mathbb{E}[Y_2] \leq \frac{\max_{p\in[k]} W_p^{BPART} - \overline{W}}{\overline{W}} \leq \frac{\mu+1}{\mu}.$$

Case 3 $\forall t$ in the input stream of length m, we have $f_t < \theta m$. In that case, there are no heavy hitters, and thus $m_{SI} = m$. This case is handled similarly as Case 2 by considering that the largest potential item has a frequency equal to $\theta m - 1$. Thus $\min_{p\in[k]} W'_p$ cannot exceed $(m - (m/k\mu + \theta m - 1)/k = (k(m\mu + \theta m\mu - \mu) - m)/(k^2\mu)$, leading to

$$\max_{p\in[k]} W_p^{BPART} - \overline{W} = \left(\min_{p\in[k]} W'_p + \frac{m}{k\mu} + \theta m - 1\right) - \frac{m}{k} \leq \frac{(\theta\mu k + 1)m}{k\mu}.$$

By applying the same argument as Case 2, the expected error introduced by BPART is given by

$$\mathbb{E}[Y_3] \leq \frac{\max_{p\in[k]} W_p^{BPART} - \overline{W}}{\overline{W}} \leq \frac{1+\mu k\theta}{\mu}.$$

Combining the three cases, we deduce an upper bound of the expected error introduced by BPART, which is given by

$$\mathbb{E}[Y] \leq \max\{\mathbb{E}[Y_1], \mathbb{E}[Y_2], \mathbb{E}[Y_3]\} \leq \max\{0, \frac{1+\mu}{\mu}, \frac{1+\mu k\theta}{\mu}\}.$$

By assumption, we have $k\theta \leq 1$ and $1/\mu \leq \theta$. Thus

$$\mathbb{E}[Y] \leq 1+\theta \tag{4.6}$$

□

Chapter Summary

This chapter discussed the heavy hitters problem and presented the DHHE algorithm, solving the distributed heavy hitters problem without any assumptions on the stream frequency distribution. It has also presented a mean case analysis for the `Space Saving` algorithm. Building on this results, it shown how BPART builds a partition $\mathcal{K}$ of the streams item universe $[n]$ with exactly k parts (*i.e.*, $|\mathcal{K}| = k$) and where the weight W_p of the parts $O_p \in \mathcal{K}$ ($p \in [k]$) are balanced.

The following chapters show the application to practical issues and experimental evaluation of the algorithms presented so far, starting with network monitoring.

Chapter 5

Network Monitoring

Network monitoring analyses computer networks to provide statistics on the traffic flowing through it. This data can be used to improve the network itself, identify troubles such as down services, track user behaviours or detect attacks. By its own nature, networks are a primary field of application for data streaming.

In this section we apply three of the previously presented algorithms to network monitoring. More in details, in Section 5.1 we use Distributed Heavy Hitters Estimator (DHHE, *cf.,* Section 4.3) to track down a particular class of Distributed Denial of Service (DDoS) attacks. Then, in Section 5.2, we investigate the performance of Proportional WCM and Splitter WCM (*cf.,* Section 3.2) to track the frequency distribution of IP traffic.

5.1 DDoS Detection

A Denial of Service (DoS) attack tries to take down an Internet resource (*e.g.*, e-commerce websites) by flooding this resource with more requests than it is capable of handling. A Distributed Denial of Service (DDoS) attack is a DoS attack triggered by many machines that have been infected by a malicious software, increasing the number of sources and thus also the attack strength. A common approach to detect and to mitigate DDoS attacks is to monitor network traffic through routers and to look for highly frequent signatures that might suggest ongoing attacks. However, a recent strategy followed by the attackers is to hide their massive flows of requests distributing them in a multitude of routes, so that locally, the malicious sub-flows do not appear highly frequent, while globally they represent a significant percentage θ of the network traffic [66]. The term "global iceberg" has been introduced to describe such an attack as only a very small part of the latter can be observed from each single router [96]. A natural solution to track and detect global icebergs is to rely on multiple routers that locally scan their network traffic, and regularly provide monitoring information to a server in charge of collecting and aggregating all the monitored information. To be applicable, two issues must be solved. First, routers must be capable of monitoring a very large number of flows to discover the presence of potential global icebergs and this must be done on the fly to have some chance to detect icebergs soon enough. Secondly, the frequency of the communications between routers and the server must be low enough to prevent the server from being overloaded by iterative exchanges with all the routers; however reducing the

frequency of these exchanges must not jeopardise the detection latency of global icebergs and must not introduce false negatives (that is, the non-detection of global icebergs).

Notice that these requirements fit the distributed functional monitoring model (*cf.,* Section 2.1.3), while the detection of DDoS/global icebergs maps to the distributed heavy hitters problem (*cf.,* Section 4.3) with the additional requirement of being timely, *i.e.,* minimize the time between the appearance of an iceberg in the stream and its detection. Given the steady state setting (*i.e.,* the stream distribution does not change over time) this boils down to detect the distributed heavy hitters reading the least amount of items. Then, we can apply the DHHE algorithm presented in Section 4.3.1 to detect DDoS/global icebergs.

Implementation details — The pseudo code presented in Listings 4.2, 4.4 and 4.5 imply that for each *warning* message sent by any of the s nodes, the coordinator has to initiate a request/reply protocol with the remaining $s-1$ nodes. This is obviously a bottleneck and the coordinator can easily fall behind. To circumvent this issue and improve, at least in practice, the network usage of DHHE, the prototype introduces a batching mechanism. More in details, while a request/reply round-trip is ongoing, all the *warning* messages received in the while by the coordinator are stored. Once the current request/reply round-trip has ended, all the $Freq_i$ buffers carried by the waiting *warning* messages are coalesced into a single ID set and a single request/reply round-trip is triggered towards all the s nodes.

5.1.1 Experimental Evaluation

Setup — All these experiments have been achieved on a testbed of $s = 20$ single-board computers (Raspberry Pi Model B) and two servers connected through a Gigabit Ethernet network. Single-board computers are small computers with limited memory and storage capacities (100 Mbps, 250MB RAM and 700MHz CPU). Each Raspberry Pi hosts a node, one of the two servers hosts the coordinator, while the other one generates all the s streams. Real routers (as for example, Cisco or Juniper type M or T, 10 Gbps throughput, from 768MB to 4GB of memory, Pentium CPU 1GHz) would definitively handle the code run by the small Raspberries with a much larger input rate.

Several values of θ have been considered, namely, $\theta \in [0.005; 0.1]$, with logarithmic steps, as well as different values of ρ, *i.e.*, $\rho \in [0.005; 1.0]$. For clarity reasons we show only the results of the experiments with the two extremum values of ρ, *i.e*, $\rho = 0.005$ (each buffer $\mathcal{G}_{\kappa,i}$ contains a single item, which amounts for the nodes to directly send each potential global iceberg to the coordinator), and $\rho = 1$ (all the buffer $\mathcal{G}_{\kappa,i}$ have their maximal size n_κ, *cf.,* Section 4.3.1). In the following both cases are respectively referred to as *no buffer* and *with buffers*.

The probability of failure δ and the error ε of DHHE have been respectively set to $\delta = 0.1$ and $\varepsilon = 0.1$. The space complexity bounds (*cf.,* Lemma 4.6) yield a memory usage of at most 7.5% for each node with respect to the naive solution (*i.e.,* maintain a counter for each received distinct item). Furthermore, we show that using as little as 1.43% of the space of the naive algorithm is sufficient (see [1] Table 5.1). This is achieved by reducing the number of columns c in the `Count-Min` sketch from $c = \lceil 2(1-\theta)/(0.1\theta) \rceil$ (*cf.,* Lemma 4.6) to $c = \lceil e/\theta \rceil$.

[1] For simplicity reason, we present memory usage with classical 32-bit coding. This is an overestimation of the requirement as only $\log n$ and/or $\log m$ are sufficient.

Table 5.1 – Memory usage with buffers, and counters of 32 bytes.

θ	`Count-Min`	$\sum_{\kappa=1}^{5} \lvert\mathcal{G}_{\kappa,i}\rvert$	$\lvert\mathcal{J}_i\rvert$	Space Memory Gain
0.1	3.48 kB	0.77 kB	0.32 kB	98.57%
0.005	69.59 kB	7.20 kB	6.40 kB	74.00%

We have fed the single-board computers with both real-world datasets and with synthetic traces. This allows to capture phenomenons that may be difficult to obtain from real-world traces, and thus allows to check the robustness of DHHE.

Synthetic traces — Synthetic streams have been generated using Zipfian distributions with $\alpha \in \{0.5, 1.0, 2.0, 3.0\}$, denoted respectively by Zipf-0.5, Zipf-1, Zipf-2, and Zipf-3. Each stream is made of $m_i = 100,000$ items (*i.e.*, the global stream is made of $m = 2,000,000$ items) picking them from an universe $[n]$ whose size is equal to $n = 10,000$. Each node receives around $4,000$ items per second. Each run has been executed a hundred times, and we provide the mean over the repeated runs.

Real dataset — We have retrieved from the CAIDA repository [19,20] a real world trace registered during a DDoS attack. We refer to this dataset as CAIDA TRACE. This trace contains most of the IP packet header, however we consider only the destination IP addresses, *i.e.,* the stream items are the destination IP addresses. The total raw data size over the 20 nodes is $m = 2 \times 10^8$. There are in total $n = 825,695$ unique destination addresses in this dataset (*i.e.,* distinct items). This represents a 15-minute trace of traffic, monitored on OC192 Internet backbone links. Note that this dataset is definitely larger than the synthetic ones, drastically increasing the number of collisions in the `Count-Min`. This dataset has been split into 20 streams, each one sent to a different node of our testbed.

Metrics — The following metrics are evaluated:

- Precision, $\frac{\text{true positives}}{\text{true positives + false positives}}$, and recall, $\frac{\text{true positives}}{\text{true positives + false negatives}}$;
- The number of false positives;
- The frequency estimation of global icebergs;
- The communication cost measured as the ratio between the number of bits/-messages exchanged by DHHE and the size in bits/messages of the global input stream;
- The detection latency measured as the ratio between the number of items read by the nodes before the coordinator has detected all global icebergs and the size of the global input stream.

In the subsequent plots, the points are linked together by lines to improve the readability.

Simulations Results

Precision and recall — Figure 5.3 shows the precision and recall of our solution. The precision is the number of generated global icebergs divided by the number of generated global icebergs and of DHHE false positives. The recall is the number of generated global icebergs divided by the number of generated global icebergs and of DHHE false

Table 5.2 – Frequency gaps between global icebergs and sparse item.

θ	Zipf-0.5	Zipf-1	Zipf-2	Zipf-3
0.1	N/A	5.1×10^{-2}	8.4×10^{-2}	7.3×10^{-2}
0.005	1.5×10^{-3}	2.4×10^{-4}	8.0×10^{-4}	2.8×10^{-3}

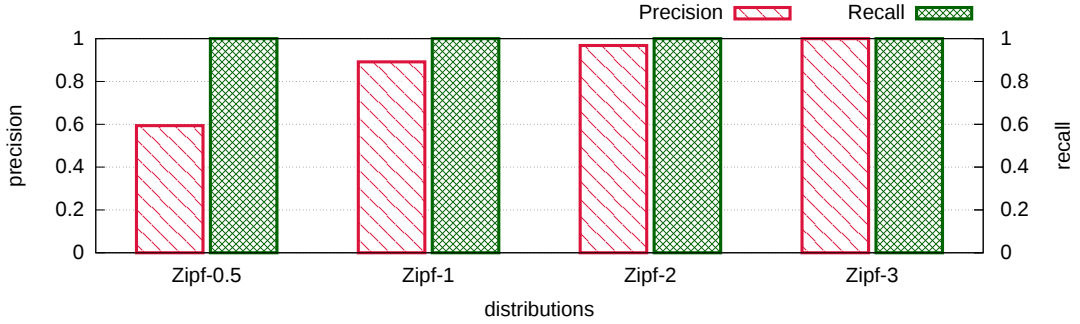


Figure 5.3 – Precision and recall as a function of the input distributions for $\theta = 0.005$.

negatives. The main result is that recall is always equal to 1 whatever the input streams features. This is a very important property of our solution as it shows that all the global icebergs are perfectly detected (there are no false negatives), even if global icebergs are well hidden in all the distributed streams (*i.e.*, $\theta = 0.005$). Now, this figure shows that precision is also very high when input streams follow Zipfian distributions with $\alpha \geq 1$ while for $\alpha = 0.5$ it decreases. This is easily explained by the fact that with slightly skewed distributions (case with $\alpha \leq 0.5$), there are very few global icebergs (one or two), and the gap between those global icebergs and the most frequent sparse items (the ones with a relative frequency close but less than θ) is very small. Table 5.2 shows the frequency gap between the least frequent global iceberg and the most frequent sparse item with $\theta \in \{0.005, 0.1\}$. Thus even a small over-approximation of `Count-Min` can wrongly tag them as global icebergs. In addition, detecting two items as global icebergs when there is a single one drops the precision to 0.5. Notice that both the precision and the recall are independent from the size of the local buffers $\mathcal{G}_{\kappa,i}$.

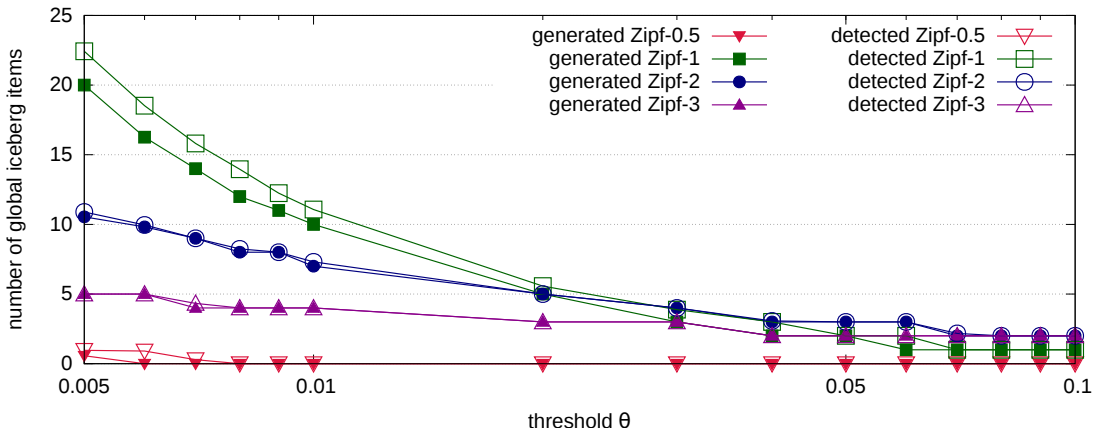


Figure 5.4 – Number of detected and generated global icebergs as a function of θ.

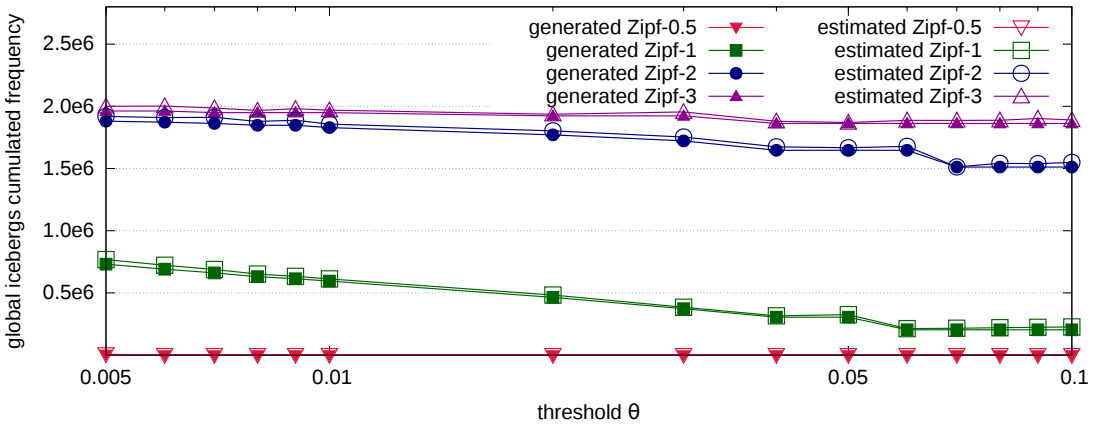


Figure 5.5 – Total global icebergs frequency (estimated and generated) as a function of θ.

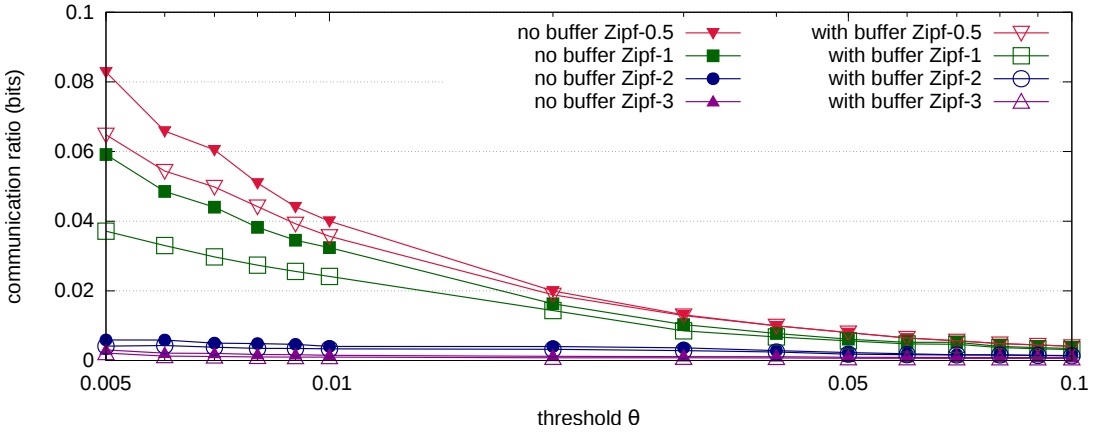


Figure 5.6 – Communication ratio in bits as a function of θ.

Figure 5.4 shows more details on the precision of DHHE by showing the number of global icebergs that should be detected (referred to as *generated*) and the number of items that have been effectively detected as global icebergs by DHHE (referred to as *detected*) as a function of θ. By construction of the `Count-Min` algorithm, item frequencies are over-estimated, thus the difference between the number of detected and effectively generated global icebergs corresponds to the number of false positives. Notice that the number of false positives may slightly increase with the size of the system due to the over-approximation of the sketch algorithm at each node. However, the absence of false negatives is guaranteed. For $\theta = 0.005$, Figure 5.4 shows the results presented in Figure 5.3. When θ increases, the precision of our solution drastically augments whatever the form of the input distributions.

Frequency estimation — Figure 5.5 compares the total number of occurrences of all the global icebergs as estimated by DHHE with the total number of occurrences effectively generated. The overestimation of frequent items (*i.e.*, those whose relative frequency exceeds θ) is negligible as expected by Lemma 4.6.

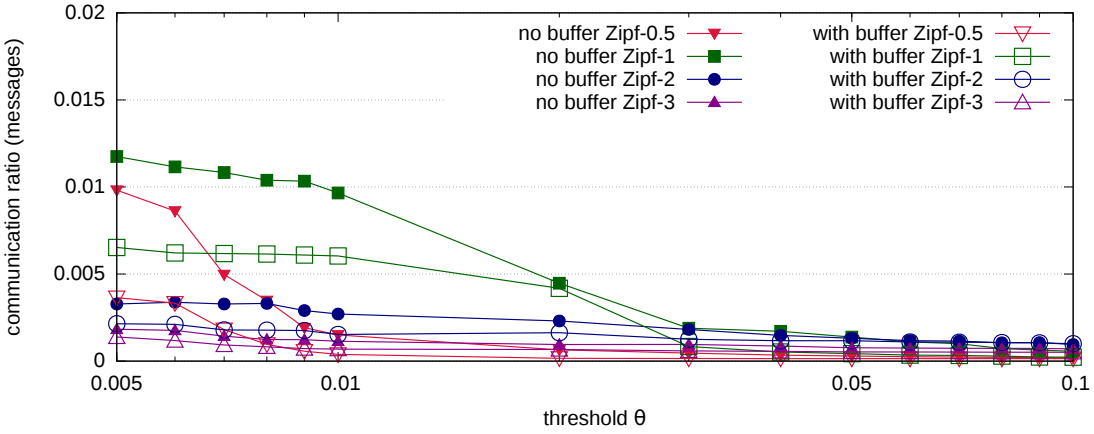


Figure 5.7 – Communication ratio in messages as a function of θ.

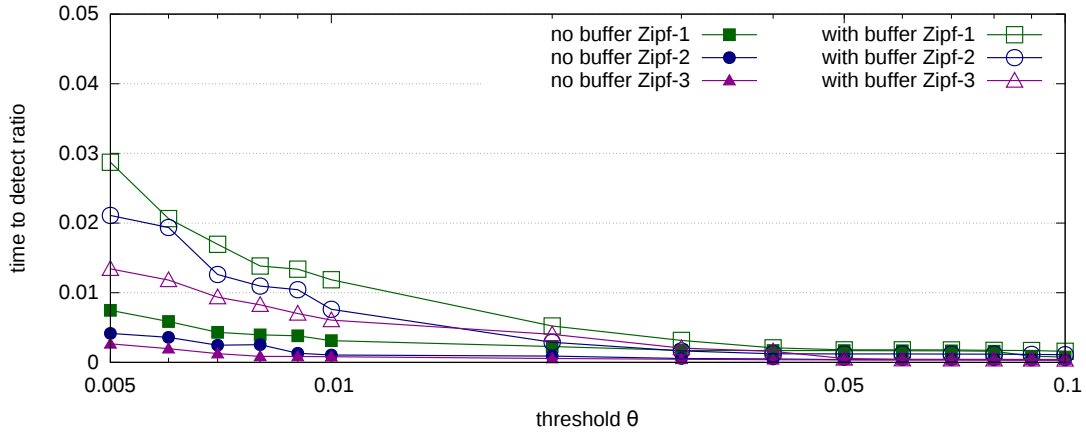


Figure 5.8 – Time to detect ratio as a function of θ.

Communication cost — Figure 5.6 shows the ratio between the number of bits exchanged by DHHE and the number of bits received in the input streams as a function of θ. The primary remark is the negligible communication overhead induced by the algorithm to accurately detect global icebergs: in the worst case strictly less than 8.5% of the size of all the distributed streams is exchanged by the nodes and the coordinator. This holds even for slightly skewed distributions, which by Theorem 4.4 are the distributions that lead to the communication upper bound. Notice the impact of buffers $\mathcal{G}_{\kappa,i}$ on the communication overhead for these slightly skewed distributions, for small values of θ. Notice that in terms of messages (*cf.*, Figure 5.7), the ratio is even lower (1.2%) since each message exchanged with the coordinator may carry multiple items.

Detection latency — Figure 5.8 shows the latency of our solution to detect global icebergs. The detection latency is the ratio between the number of items read by DHHE to detect all the global icebergs and the size of the stream m. Remarkably enough, DHHE needs to read (in the worst case) less than 3% of the global stream to detect all the global icebergs (when the streams are randomly ordered). We can also see the impact of buffers $\mathcal{G}_{\kappa,i}$ on the detection latency: setting the size of the $\mathcal{G}_{\kappa,i}$ buffer to the maximum increases the detection latency by a factor 4 while it decreases the communication overhead by a

factor 2 (see Zipf-1 with $\theta = 0.005$ in Figure 5.6). There must exist some optimal value of ρ that should minimise the detection latency and communication overhead. The study of this optimum is an open question.

DDoS dataset — Table 5.9 summarizes the results with the CAIDA TRACE dataset. The DDoS target is the unique global iceberg, with a probability of occurrence equals to 0.15 ($\theta = 0.1$), as any other items occur with a probability lower than 5×10^{-3}. As illustrated in Table 5.9, the DDoS target is correctly detected (no false positives or negatives). The bits communication ratio (considering only the destination address as payload) is at most equal to 3.6×10^{-4}. These results are mainly due to the large gap between probabilities of occurrence of the unique global iceberg and the other items, which occurs in classical DDoS attack.

Table 5.9 – Results with the DDoS trace (CAIDA TRACE) ($\theta = 0.1$).

Stream Size (m)	2×10^8
Distinct Items (n)	825,695
Global Icebergs	1
Detected Global Icebergs	1
False Positives	0
False Negatives	0
Global Iceberg Frequency	3.2×10^6
Estimated Global Iceberg Frequency	3.4×10^6
Communication Ratio (bits) without buffer	3.6×10^{-4}
Communication Ratio (bits) with buffers	3.2×10^{-4}

5.2 Estimating the frequency of IP addresses

Network traffic does not deal only with the detection of malicious behaviour. Knowing the frequency distribution of the traffic flowing through a network can help the network engineers in many other maintenance tasks. In this section we apply the windowed versions of the `Count-Min` algorithm presented in Section 3.2 to this problem and compare their performances

5.2.1 Experimental Evaluation

This section provides the performance evaluation of our algorithms. We have conducted a series of experiments on different types of streams and parameter settings.

Setup — We consider 5 different algorithms:

- The Simple WCM algorithm, also referred as Simple, we consider it as a lower bounds for all the algorithms performances;
- The Perfect WCM algorithm, also referred as Perfect, we consider it as an upper bound for all the algorithms performances;
- The Proportional WCM algorithm, also referred as Proportional, our first and simpler solution;

- The Splitter WCM algorithm, also referred as Splitter, our second and more complex solution;
- The `ECM-Sketch` algorithm, the `wave` based algorithm proposed by Papapetrou *et al.* [76].

The `wave`-based [46] version of `ECM-Sketch` that we have implemented replaces each counter of the $\mathfrak{F}$ matrix with a `wave` data structure. Each `wave` is a set of lists, the number and the size of such lists is set by the parameter $\varepsilon_{\texttt{wave}}$. Then, setting $\varepsilon_{\texttt{wave}} = \varepsilon$, the `wave`-based `ECM-Sketch` space complexity is $O\left(1/\varepsilon^2 \log(1/\delta)\left(\log^2 \varepsilon M + \log n\right)\right)$ bits.

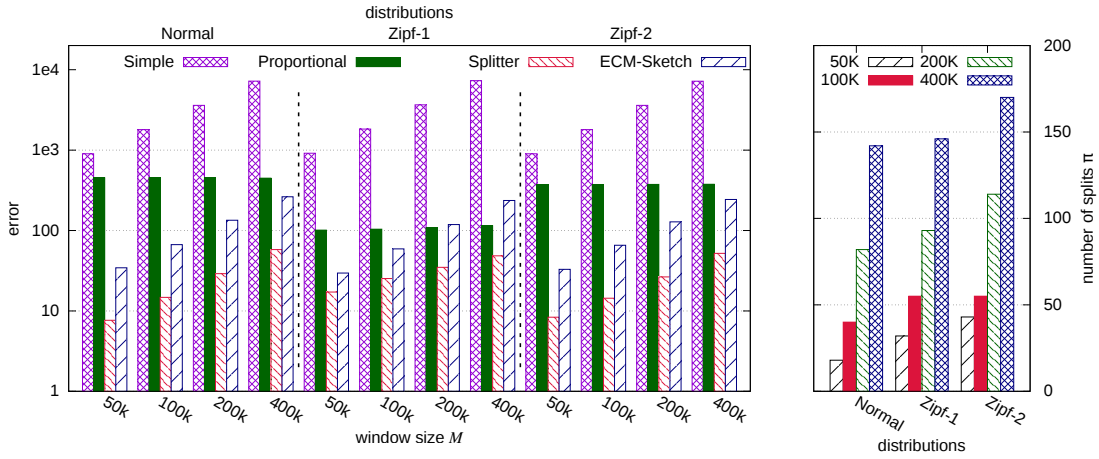
The `Count-Min` uses two parameters: δ that sets the number of rows r, and ε, which tunes the number of columns c. In all simulations, we have set $\varepsilon = 0.1$, meaning $c = \lceil \frac{e}{0.1} \rceil = 28$ columns. Most of the time, the `Count-Min` matrix has several rows. However, analysing results using multiple rows requires taking into account the interaction between the hash functions. If not specified, for the sake of clarity, we present the results for a single row ($\delta = 0.5$).

Moreover, recall that Splitter WCM has two additional parameters: β and γ. We provide the results for $\beta = 1.5$ and $\gamma = 0.05$. Given these parameters, we have an upper bound of at most $\overline{\pi} = 560$ spawned sub-cells (*cf.* Lemma 3.7). With the parameters stated so far and the provided memory usage upper bounds, `ECM-Sketch` uses at least twice the memory required by Splitter WCM. Notice however that the upper bound of $\bar{\pi} = 560$ spawned sub-cells is never reached in any test. According to our experiments, `ECM-Sketch` uses at least 4.5 times the memory required by Splitter WCM in this evaluation.

To verify the robustness of our algorithms, we have fed them with synthetic traces and real-world datasets. The latter give a representation of some existing monitoring applications, while synthetic traces allow to capture phenomena that may be difficult to obtain otherwise.

Synthetic traces — We evaluate the performance by generating families of synthetic streams, following four distributions: Uniform, Normal, Zipf-1 and Zipf-2 (*i.e.,* Zipfian distributions with $\alpha = 1.0$ and $\alpha = 2.0$). In order to simulate changes in the frequency distribution over time, our stream generator shifts the distribution right by $2r$ positions each period with a size of $10,000$ items. If not specified otherwise, the window size is $M = 50,000$ and synthetic streams are of length $m = 3M$ (*i.e.* $m = 150,000$) with $n = 1,000$ distinct items. Note that we restrict the stream to 3 windows since the behavior of the algorithms in the following windows does not change, as each algorithm relies only on the latest past window. We skip the first window where all algorithms are trivially perfect. Each run has been executed a hundred times, and we provide the mean over the repeated runs.

Real dataset — We use the CAIDA TRACE [19, 20] dataset, *i.e.,* the same of the previous experimental evaluation (*cf.,* Section 5.1.1), restricted to the first $400,000$ items (*i.e.*, when the DDoS attack begins). The stream is composed by $n = 4.9 \times 10^4$ distinct items. The item representing the DDoS target has an empirical probability equal to 0.09, while the second most frequent item has an empirical probability of 0.004.



(a) Average estimation error (b) Average splits number

Figure 5.10 – Results for different window sizes M.

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

- The mean absolute error of the frequency estimation of all n items returned by the algorithms with respect to Perfect WCM, that is

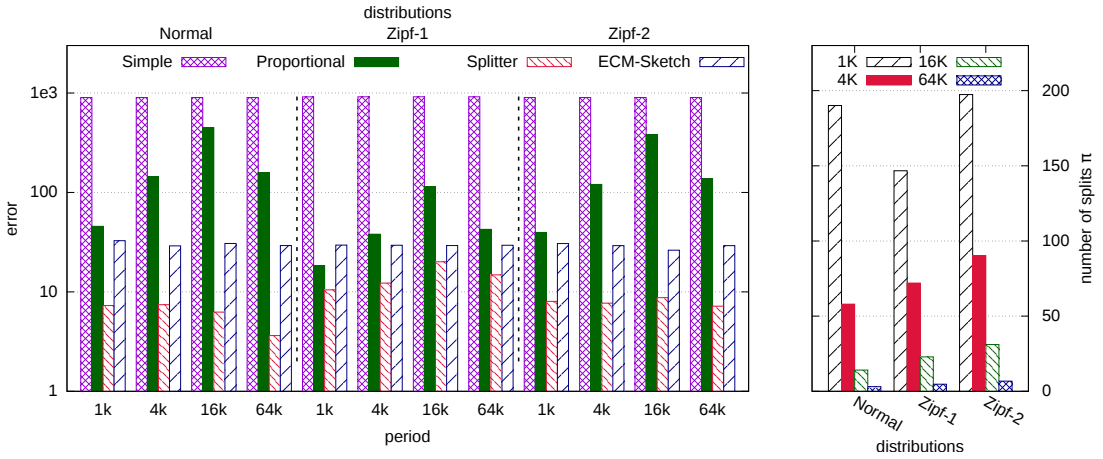
$$\frac{1}{n}\left(\sum_{t\in[n]}\left|\hat{f}_t^{\,\textsc{TestedAlgorithm}} - \hat{f}_t^{\,\text{Perfect WCM}}\right|\right),$$

  we refer to this metric as *error*;
- The exact number of splits π, *i.e.,* the additional space used by Splitter WCM with respect to the vanilla `Count-Min` due to the merge and split mechanisms;
- The frequency estimation.

Simulations Results

Window sizes — Figure 5.10a presents the estimation error of the Simple WCM, Proportional WCM, Splitter WCM and `ECM-Sketch` algorithms considering the Normal, Zipf-1 and Zipf-2 distributions, with $M = 50,000$ (and *a fortiori* $m = 150,000$), $M = 100,000$ (with $m = 300,000$), $M = 200,000$ (with $m = 600,000$) and $M = 400,000$ (with $m = 1,200,000$). Note that the y-axis (*error*) is in logarithmic scale and error values are averaged over the whole stream. Simple WCM is always the worst (with an error equals to $3,395$ in average), followed by Proportional WCM (451 in average), `ECM-Sketch` (262 in average) and Splitter WCM (57 in average). In average, Splitter WCM error is 4 times smaller than `ECM-Sketch`, with 4 times less memory requirement. The error estimation of Simple WCM, Proportional WCM, `ECM-Sketch` and Splitter WCM increases in average respectively with a factor 2.0, 1.1, 1.9 and 1.7 for each 2-fold increase of M.

Figure 5.10b gives the number of splits spawned by Splitter WCM in average to keep up with the distribution changes. The number of splits grows in average with a factor 1.7 for each each 2-fold increase of M. In fact, as γ is fixed, the minimal size of each sub-cell grows with M, and so does the error.



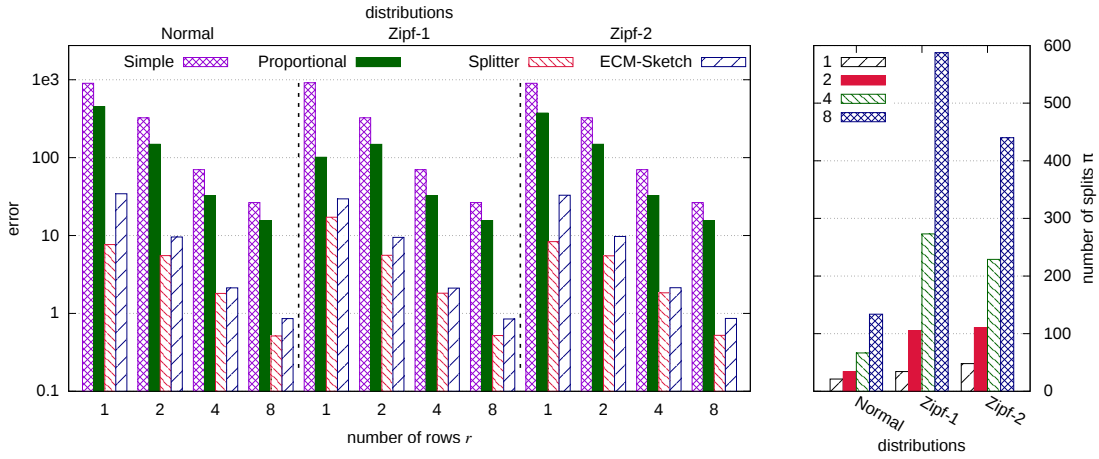
(a) Average estimation error (b) Average splits number

Figure 5.11 – Results for different periods sizes.

Periods — Recall that the distribution is shifted after each period. Figure 5.11 shows the estimation error and the number of splits with different period sizes: $\{1,000; 4,000; 16,000; 64,000\}$. Again, Splitter WCM (20 at most) is always better than `ECM-Sketch` (26 at best) achieving roughly a 4 fold improvement. Simple WCM is always the worst (more than 900), followed by Proportional WCM (roughly 140 in average). In more details, Proportional WCM grows from $1,000$ to $16,000$ items, because slower shifts cast the error on less items, resulting in a larger mean absolute error. However, for $64,000$ we have less than a shift per window, meaning that some windows have a non-changing distribution and Proportional WCM is almost perfect. In general Splitter WCM estimation error is not heavily affected by decreasing the period size since it keeps up by spawning more sub-cells. For a size of $64,000$ items we have at most 7 splits, while for a size of $1,000$ items we have in average 166 splits. Each 4-fold decrease in the period size increases the number of splits by $3.4\times$ in average.

Rows — The `Count-Min` algorithm uses a hash-function for each row mapping items to entries. Using multiple rows produces different collisions patterns, decreasing the probability of failure, and thus increasing the overall accuracy. Figure 5.12 presents the estimation error and splits for $r = 1$ (*i.e.,* $\delta = 0.5$), $r = 2$ ($\delta = 0.25$), $r = 4$ ($\delta = 0.0625$) and $r = 8$ rows ($\delta = 0.004$). Increasing the number of rows do enhance the accuracy of the algorithms. However, the ordering among the algorithms does not change: Simple WCM, Proportional WCM, `ECM-Sketch` and Splitter WCM achieve respectively 331; 126; 11 and 4 in average. For each distribution shift, $4r$ items change their occurrence probability, then (without collisions) $4r^2$ entries may change their update rate, *i.e.,* there can be $4r^2$ potential splits per shift. Thankfully, experiments illustrate that the number of splits growth is not quadratic: in average it increases by $2.4\times$ for each 4-fold increase of r.

Time series — Figure 5.13 presents the estimation error evolution as the stream unfolds. The stream distribution changes each $60,000$ items in the following order: Uniform, Normal, Uniform, Zipf-1, Uniform, Zipf-2, Uniform. In addition the period is set to $15,000$ items. The streams is of length $m = 400,000$. Note that, in order to avoid



(a) Average estimation error (b) Average splits number

Figure 5.12 – Results for different number of rows r.

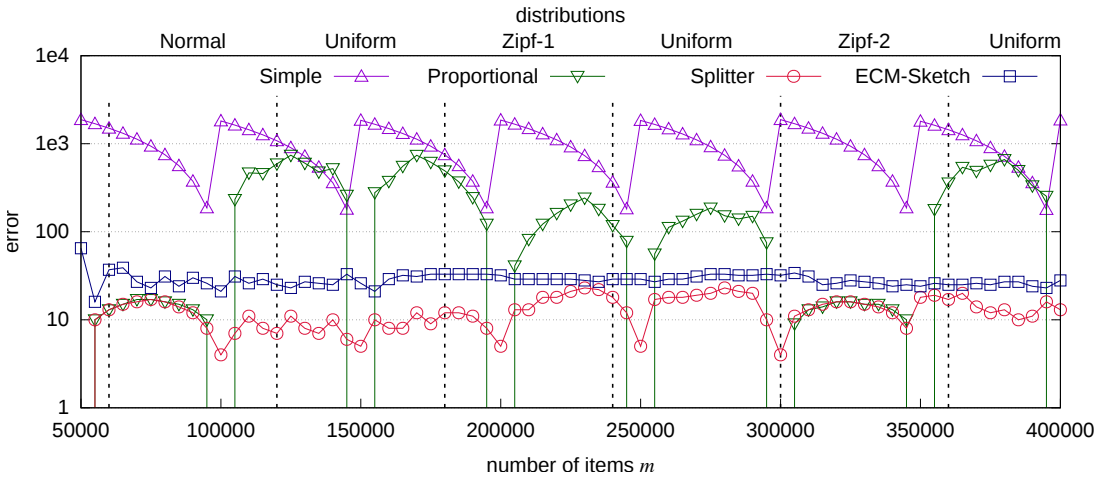


Figure 5.13 – Estimation error with the time series.

side effect, the distribution shift and swap periods are not synchronised with the window size ($M = 50,000$).

Splitter WCM error does not exceed 23 (and is equal to 13 in average). `ECM-Sketch` maximum error is 65 (29 in average), as Proportional WCM goes up to 740 (207 in average) and Simple WCM reaches 1,877 (1,035 in average). Since at the beginning of each window Simple WCM resets its `Count-Min` matrix, there is a periodic behavior: the error burst when a window starts and shrinks towards the end. In the 1-st window period (0 to 50,000) and in the 6-th windows (250,000 to 300,000) the distribution does not change over time (shifting Uniform has no effect). This means that Splitter WCM does not capture more information than Proportional WCM, thus they provide the same estimations in the 2-nd and the 7-th windows (respectively between 50,000 and 100,000 samples then between 300,000 and 350,000 samples).

Figure 5.14 presents the value of f_1 and its estimations over time. The differences among the algorithms shown in the figure would not be visible if we also plotted the results for Simple WCM, for sake of clarity we decided to omit it. The plain line represents the

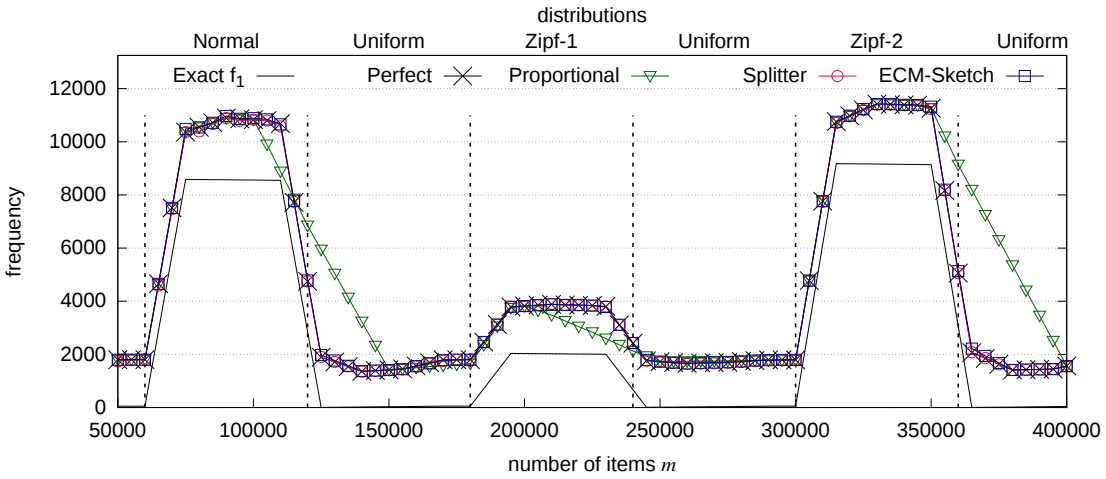


Figure 5.14 – Frquency estimation of item 1 with the time series.

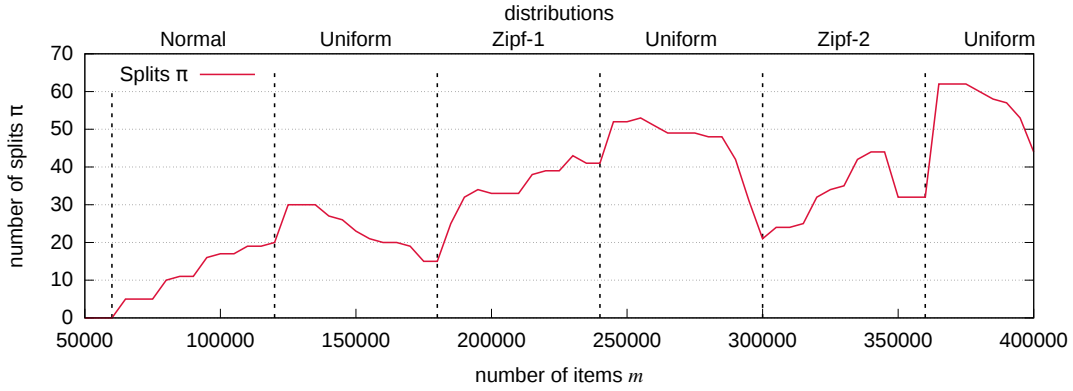
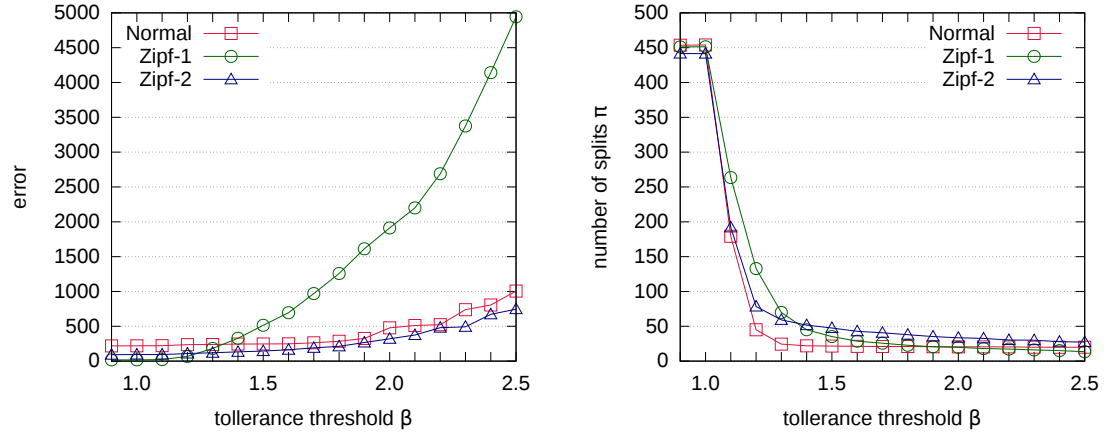


Figure 5.15 – Number splits π with the time series.

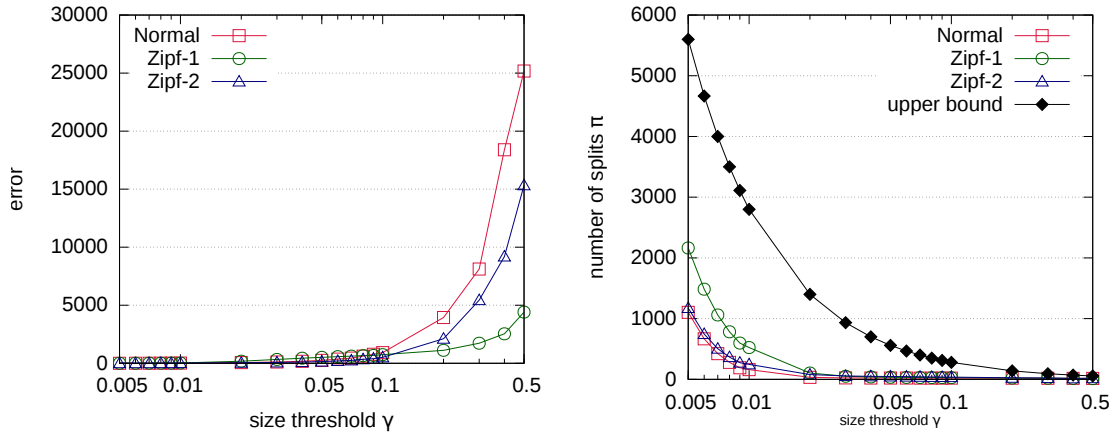
exact value of f_1 according to time, which also reflects the distribution changes. The plots for Perfect WCM, Splitter WCM and `ECM-Sketch` are overlapping (exes, circles and squares). Except for the error introduced by the `Count-Min` approximation, they all follow the f_1 shape precisely. However, even that is not clearly visible on Figure 5.14, notice that `ECM-Sketch` has always a slightly larger error than Splitter WCM. On the other hand, we can observe that item 1 probability of occurrence changes significantly in the following intervals: $[60k, 75k]$, $[180k, 195k]$ and $[300k, 315k]$. Proportional WCM fails to follow the f_1 trend in the windows following those intervals, namely the 3-rd, 5-th and 8-th, since it is unable to correctly assess the previous window distribution.

Finally, Figure 5.15 presents the number of splits π according to time. There are in average 33 and at most 63 splits (while the theoretical upper bound $\overline{\pi}$ is 560 according to Lemma 3.7). Interestingly enough, splits decrease when the distribution does not change (in the Uniform intervals for instance). That means that, as expected, some sub-cells expire and no new sub-cells are created. In other words, Splitter WCM correctly detects that no changes occur. Conversely, when a distribution shifts or swaps, there is a steep growth, *i.e.,* the change is detected. This pattern is clearly visible in the 2-nd window.



(a) Average estimation error (b) Average number of splits

Figure 5.16 – Performance comparison with $\gamma = 0.05$.



(a) Average estimation error (b) Average number of splits

Figure 5.17 – Performance comparison with $\beta = 1.5$.

Splitter parameters — Figure 5.16 presents the estimation error and the number of splits with several values of $\beta \in \{0.9, 2.5\}$ and a fixed $\gamma = 0.05$. As expected, the estimation error grows with β. Zipf-1 goes from 18 ($\beta = 0.9$) to 4,944 ($\beta = 2.5$), while the other distributions in average go from 110 ($\beta = 0.9$) to 684 ($\beta = 2.5$). Conversely, increasing β decreases the number of splits. Since ERROR cannot return a value lower than 1.0, going from 1.0 to 0.9 has almost no effect with at most 454 splits, which represents roughly 19% less than the theoretical upper bound. From $\beta = 1.0$ to 1.3, the average falls down to 51, reaching 20 at $\beta = 2.5$. There is an obvious trade-off around $\beta = 1.5$ that should represents a nice parameter choice for a given user.

Figure 5.17 presents the estimation error and the number of splits according to the parameter $\gamma \in \{0.005, 0.5\}$, with a fixed $\beta = 1.5$. Note that the x-axis (γ) is logarithmic. As for β, the estimation error increases with γ: the average starts at 4 (with $\gamma = 0.005$), reaches 610 at $\gamma = 0.1$ and grows up at 12,198 (for $\gamma = 0.5$). Conversely, increasing γ decreases the number of splits: the average starts at 1,659 ($\gamma = 0.005$), reaches 77 at $\gamma = 0.02$ and ends up at 14 ($\gamma = 0.5$). In order to illustrate the accuracy of our splitting

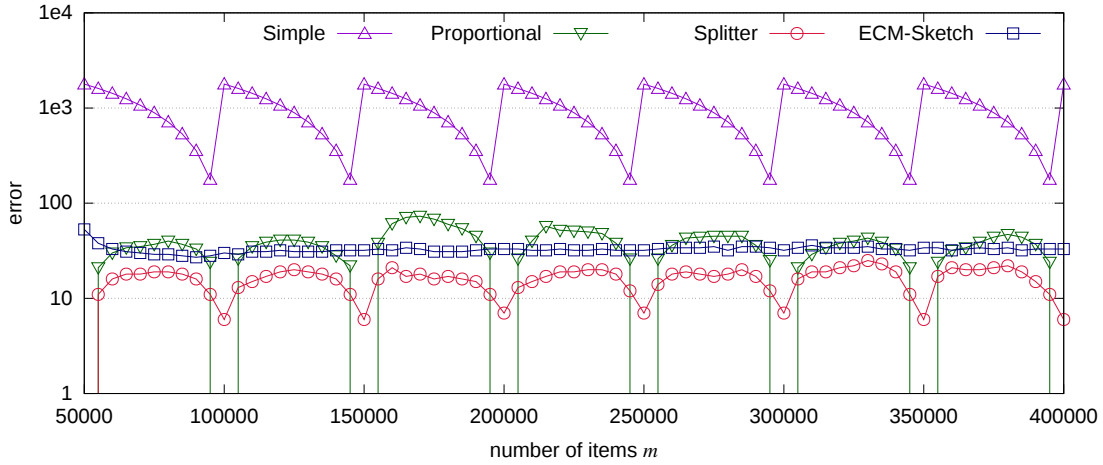


Figure 5.18 – Estimation error with the DDoS trace (CAIDA TRACE)

heuristic, 5.17b shows also the theoretical upper bound $\overline{\pi}$. There seems to be a sweet spot around $\gamma = 0.05$.

To summarize, the trend in all the last four plots (and the results for different values of r and period sizes) hints to the existence of some optimal value of β and γ that should minimise the error and the splits. This optimal value seems to either be independent from the stream distribution or computed based on the recent behavior of the algorithm and some constraints provided by the user. Seeking for a extensive analysis of this optimum represents a challenging open question.

DDoS dataset — As illustrated in the global iceberg problem (*cf.,* Section 5.1), tracking locally the most frequent items in distributed data streams is not sufficient to detect Distributed Denial of Service (DDoS). As such, one should be able to estimate the frequency of any item. Figure 5.18 presents the estimation error evolution over time for the CAIDA TRACE dataset. In order to avoid drowning the estimation error in the high number of items, we have restricted the computation to the most frequent 7500 items, which cover 75% of the stream[2]. Figure 5.18 illustrates some trends similar to the previous test, however the estimation provided by Proportional WCM, `ECM-Sketch` and Splitter WCM are quite close since the stream changes much less over time. Simple WCM does not make less error than 178 (that is 1,002 in average), while Proportional WCM, `ECM-Sketch` and Splitter WCM do not exceed respectively 73 (34 in average), 53 (33 in average) and 25 (16 in average). On the other hand, for Splitter WCM, there are at most 154 splits with an average of 105 splits.

[2]The remaining items have a frequency proportion lower than 2×10^{-5}.

Chapter Summary

In this chapter, DHHE, Proportional WCM and Splitter WCM, have been successfully applied to practical problems, *i.e.,* detecting Distributed Denial of Service (DDoS) attacks and tracking the frequency distribution of IP traffic. The experimental evaluation has confirmed the theoretical claims on DHHE (*cf.,* Section 4.3). It has also shown the quality of both Proportional WCM and Splitter WCM (Section 3.2), and in particular how the latter outperforms state of the art solutions.

The next chapter introduces the stream processing model. In the following we apply VALES and BPART to design three algorithms optimizing stream processing systems.

Chapter 6

Optimizations in Stream Processing Systems

Stream processing systems [40] (SPS) are today gaining momentum as a tool to perform analytics on continuous data streams. These applications have become ubiquitous due to increased automation in telecommunications, health-care, transportation, retail, science, security, emergency response, and finance. As a result, various research communities have independently developed programming models for streaming.

A stream processing application is commonly modelled as a direct acyclic graph where data operators, represented by nodes, are interconnected by streams of tuples containing data to be analysed, the directed edges. Each stream is a conceptually infinite sequence of data items (similarly as in data streaming) and each operator conceptually has its own thread of control. Since operators run concurrently, stream graphs inherently expose parallelism. A common way to achieve scalability is to parallelize each data operator using multiple instances, each of which handles a subset of the tuples conveyed by the operator's ingoing stream.

Notice that stream processing systems (SPS) are not restrained to the academic world; their ability to produce analysis results with sub-second latencies, coupled with their scalability, also makes them the preferred choice for many big data companies. Typical use cases include financial trading and monitoring of manufacturing equipment or logistics data. These scenarios require a high throughput and a low end to end latency from the system, despite possible fluctuations in the workload.

In the last decade, a large number of different academic prototypes as well as commercial products have been built to fulfill these requirements. In general, stream processing systems (SPS) can be divided into three generations [55]:

First generation stream processing systems have been built as stand-alone prototypes or as extensions of existing database engines. They were developed with a specific use case in mind and are very limited regarding the supported operator types as well as available functionalities. Representatives of this generation include Telegraph [26] and Aurora [2].

Second generation systems extended the ideas of data stream processing with advanced features such as fault tolerance, adaptive query processing, as well as an enhanced operator expressiveness. Important examples of this class are Borealis [1] and System S [59].

Third generation system design is strongly driven by the trend towards cloud computing, which requires the data stream processing engines to be highly scalable

and robust towards faults. Well-known systems of this generation include Apache S4 [75], Apache Spark [94] and Apache Storm [89].

The communities that have focused the most on streaming optimizations are digital signal processing, operating systems and networks, complex event processing and databases. In their survey, Hirzel *et al.* [56] propose a catalogue encompassing 11 stream processing optimizations developed by these communities. In this thesis we focus on two of these optimizations, namely *load balancing* and *load shedding*.

6.1 Related Work

Load balancing in distributed computing is a well known problem that has been extensively studied since the 80s [21,98]. It has received new attention in the last decade with the introduction of peer-to-peer systems. Distributed stream processing systems have been designed, since the beginning, by taking into account the fact that load balancing is a critical issue to attain the best performance. In their survey, Hirzel *et al.* [56] identify two ways to perform load balancing in stream processing systems: either when placing the operators on the available machines [22, 23] or when assigning load to parallized operator instances.

Scalability is usually attained at the deployment phase where each data operator can be parallelized using multiple instances, each of which handles a subset of the tuples conveyed by the operator's ingoing stream. Balancing the load among the instances of a parallel operator is important as it yields to better resource utilization and thus larger throughputs and reduced tuple processing latencies. How tuples pertaining to a single stream are partitioned among the set of parallel instances of a target operator strongly depends on the application logic implemented by the operator itself. The strategy used to route tuples in a stream toward available instances of the receiving operator is embodied in a so-called *grouping* function. Two main approaches are commonly adopted: either the assignment is based on the specific value of data contained in the tuple (*key* or *field grouping*), or tuples are randomly assigned to instances (*random* or *shuffle grouping*).

As load balancing strives to fully use the available resources, an orthogonal problem is to provide the system with enough resources to run with good performances. Correctly provisioning computing resources for SPS 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 SPS is not economically sensible, thus system designers are today moving towards approaches based on elastic scalability [55, 83], 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. Load shedding is an orthogonal technique designed to handle short-term fluctuations as well as to prevent the failure of the systems in the face of excessive resource requirements.

Load balancing parallelized operators — When the target operator is *stateful*, its state must be maintained continuously synchronized among its instances, with possibly severe performance degradation at runtime; a well-known workaround to this problem

consists in partitioning the operator state and let each instance work on the subset of the input stream containing all and only the tuples affecting its state partition. In this case key grouping is the preferred choice as the stream partitioning can be performed to correctly assign all and only the tuples containing specific data values to a same operator instance, greatly simplifying the work of developing parallelizable stateful operators.

The downside of using key grouping is that it may induce noticeable imbalances in the load experienced by the target operator whenever the data value distribution is skewed, a common case for many application scenarios. This is usually true with key grouping implementations based on hash functions: part of the data contained in each tuple is hashed and the result is mapped, for example using modulo, to a target instance. However, hash functions are usually designed to uniformly spread values from their domain to available instances in their co-domain; if different values appear with skewed frequency distribution in the input stream, instances receiving tuples containing the most frequent values will incur the largest load.

In the last few years there has been new interest on improving load balancing with *key grouping* [44, 73]. Based on the intuition that the skew in the load distribution among parallelized key grouped operators is due to the skew in the key value distribution, Gedik [44] proposes to use the `Lossy-Counting` algorithm to keep track of the heavy hitters that are then explicitly mapped to target sub-streams. On the other hand, sparse items are mapped using a consistent hash function. Nasir *et al.* [73] target the same problem and propose to apply the *power of two choices* approach to provide better load balancing. Their solution, namely `Partial Key Grouping` (`PKG`), provides increased performance by mapping each key to two distinct sub-streams and forwarding each tuple to the less loaded of the two sub-streams associated with the tuple itself; this is roughly equivalent to working on a modified input stream where each value is split into two keys, each with half the original load. However, the algorithm proposed in [73] cannot be applied in general, but it is limited to operators that allow a *reduce* phase to reconcile the split state for each key.

On the other hand, for *stateless* operators, *i.e.,* data operators whose output is only function of the current tuple in input, the parallelization is straightforward. The grouping function is free to assign the next tuple in the input stream to any available instance of the receiving operator (contrarily to statefull operators, where tuple assignment is constrained). Such grouping functions are often called shuffle grouping and represent a fundamental element of a large number of stream processing applications. Shuffle grouping implementations are designed to balance as much as possible the load on the receiving operator instances as this increases the system efficiency in available resource usage. Notable implementations [89] leverage a simple Round-Robin scheduling strategy guaranteeing that each operator instance will receive the same number of input tuples. This approach is effective as long as the time taken by each operator instance to process a single tuple (tuple *execution time*) is the same for all tuples. In this case, all parallel instances of the same operator will experience over time, on average, the same load.

Considering *shuffle grouping*, Sharaf *et al.* [84] propose a comprehensive solution to schedule multiple continuous queries minimizing the response time. Arapaci *et al.* [13] as well as Amini *et al.* [6], among other contributions, provide solutions to maximize the system efficiency when the execution times of the operator instances are non-uniform, either because the hardware is heterogeneous or due to the fact that each instance carries out different computations.

Load Shedding — Bursty input load represents a problem for SPS 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 time). 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 (*e.g.,* elastic scaling), rather than being an alternative.

Aurora [2] 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 to drop random tuple at specific places in the application topology. A large number of works have 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 [85] the idea of semantic load shedding. Het et al. in [54] specialized the problem to the case of complex event processing. Babcock et al. in [15] provided an approach tailored to aggregation queries. Finally, Tatbul et al. in [86] ported the concept of semantic load shedding in the realm of SPS. All these works have the 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. A different approach has been proposed in [78], 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 [61, 91, 95]. 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.

At the best of our knowledge, there is no prior work directly addressing neither load balancing with key/shuffle grouping nor load shedding on non-uniform operator instances considering that the tuples execution time depends on content of the tuple themselves. In general, previous solutions rely either on an a priori known cost model of the tuples or on the assumption that all the tuple have the same execution time.

6.2 System Model

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

Data injected by the source is encapsulated in units called tuples and each stream is a sequence of tuples. Without loss of generality, here we assume that each tuple 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 with a single non negative integer value. Thus tuples in a data-stream can be represented as items in a stream $\sigma = \langle t_1, \ldots, t_j, \ldots t_m \rangle$.

Let consider two operators, S and O, connected through a directed edge from S to O, *i.e.,* a stream $\sigma_{S \to O}$. Both operators can be parallelized, creating s independent instances $S_1, \ldots, S_s \in \mathcal{I}$ of S and k independent instances $O_1, \ldots, O_k \in \mathcal{K}$ of O, as well as by partitioning the stream $\sigma_{S \to O}$ into $s \times k$ sub-streams $\sigma_{S_1 \to O_1}, \ldots, \sigma_{S_s \to O_k}$. In other words, each operator S_i ($i \in [s]$) has k outbound sub-streams $\sigma_{S_i \to O_1}, \ldots, \sigma_{S_i \to O_k}$ and each operator O_p ($p \in [k]$) has s inbound sub-streams $\sigma_{S_1 \to O_p}, \ldots, \sigma_{S_s \to O_p}$. We denote the union of the outbound sub-stream of operator instance S_i with $\sigma_{S_i} = \cup_{p=1}^{k} \sigma_{S_i \to O_p}$. Similarly, we denote the union of the inbound sub-streams of operator instance O_p with $\sigma_{O_p} = \cup_{i=1}^{s} \sigma_{S_i \to O_p}$.

Each operator instance has a FIFO input queue where tuples are buffered while the instance is busy processing previous tuples.

We define as $w_p(t_j)$ the *execution time* of tuple t_j on the operator instance O_p, *i.e.,* the amount of time needed by the operator instance O_p to perform the computation on tuple t_j (queuing time excluded). We can then define the average execution time $\overline{W}$ as:

$$\overline{W} = \frac{W}{m} = \sum_{p=1}^{k} \sum_{t_j \in \sigma_{O_p}} \frac{w_p(t_j)}{m}$$

where W is the total execution time.

We define as $q(t_j)$ the *queuing time* of tuple t_j, *i.e.,* the amount of time tuple t_j spends in the buffering queue of an instance of operator O. Notice that this value depends on the position (*i.e.,* the subscript j) of tuple t_j and not on its value t. We can then define the average queueing time $\overline{Q}$ as:

$$\overline{Q} = \frac{Q}{m} = \sum_{p=1}^{k} \sum_{t_j \in \sigma_{O_p}} \frac{q(t_j)}{m}$$

where Q is the total queuing time.

We define as $\ell(t_j)$ the *completion time* of tuple t_j, *i.e.,* the time elapsed for tuple t_j from being injected in the stream σ by the operator S until the end of its execution by operator O. Notice that this value depends on both the position (*i.e.,* the subscript j) of tuple t_j and on its value t. We can then define the average queueing time $\overline{L}$ as:

$$\overline{L} = \frac{L}{m} = \sum_{p=1}^{k} \sum_{t_j \in \sigma_{O_p}} \frac{\ell(t_j)}{m}$$

where $L(m)$ is the total completion time.

With some abuse of terms, in the rest of this chapter we use as synonyms the sub-streams and the parallel instances of the target operator to which such sub-streams are inbound.

Chapter Summary

This chapter had introduced the stream processing model and discussed load balancing and load shedding related work.

The next chapter applies BPART and VALES to load balance key- and shuffle-grouped parallelized operators in stream processing systems.

Chapter 7

Load Balancing Parallelized Operators in Stream Processing Systems

As mentioned in the previous chapter, depending on the type of operators, stateful or stateless, two different grouping policies can be applied: key or shuffle grouping. Section 7.1 presents Distribution-aware Key Grouping (DKG), a direct application of BPART (*cf.,* Section 4.4) to load balance stateful parallelized operators. Section 7.2.1 shows Online Shuffle Grouping Scheduler (OSG), built on top of VALES (*cf.,* Section 3.3), to load balance stateless parallelized operators. For both algorithms we provide an experimental evaluation using a simulator and a prototype.

7.1 Load Balancing Stateful Parallelized Operators

When the k instances of operator O are stateful, the grouping applied to the tuple in $\sigma_{S \to O}$ must keep the operator state consistent. The standard solution is key grouping, *i.e.,* define a key x for each tuple $t_j \in \sigma_{S \to O}$, and then partition the stream $\sigma_{S \to O}$ assigning all the tuples with the same key x to the same operator instance O_p. The key is in general defined through a user defined function $\text{KEY}(t_j)$ returning the value of the key for tuple t_j. For the sake of clarity, and without loss of generality, in the following we consider that the value t of the tuples is the value of the key, *i.e.,* $\text{KEY}(t_j) = t_j$. In other words, given a key t, key grouping places all tuples of the stream $\sigma_{S \to O}$ containing the same value t in the same sub-stream $\sigma_{S \to O_p}$. The problem we target is how to perform the partitioning such that all sub-streams $\sigma_{S \to O_p}$ are load balanced, *i.e.,* all sub-streams contain, on average, the same number of tuples per time unit.

In general, the SPSs implementation of key grouping is based on an hash function h, the index of the target sub-stream for a tuple t_j is given by $h(t_j) \mod k$. This solution is simple and highly scalable as the definition of h is the only information that needs to be known by the operator that partitions the stream. On the other hand, it can produce skewed load on the parallel instances of the target operator O, especially when the grouping key is characterized by a skewed value distribution (like a Zipfian distribution), a frequent case in many application scenarios [18, 64]. A second possible option is to use an explicit mapping between the possible values of the grouping key and the set of k available sub-streams; if the distribution of values is known a-priori,

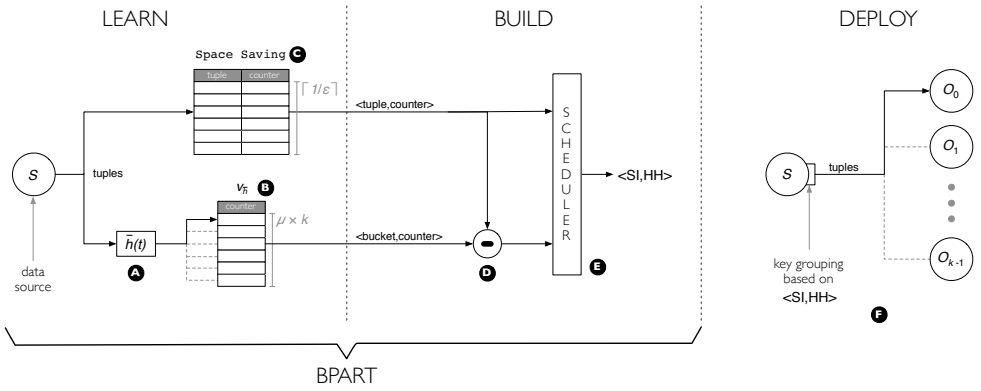


Figure 7.1 – DKG architecture and working phases.

it is theoretically possible to build an explicit mapping that produces the optimal load balancing on the target operator; however, this approach is rarely considered as **(i)** the key value distribution is in general unknown, and **(ii)** the map would easily grow to unmanageable sizes, even for typical value domains (*e.g.,* strings).

Problem Statement We assume that the values of the grouping key $\text{KEY}(t_j) = t_j$ are characterized by a Zipfian distribution with unknown moments (*i.e.,* the value for parameter α is not known) and that each sequence of tuples extracted from $\sigma_{S \to O}$ has the same statistical characteristics of the whole stream (*i.e.,* no adversary in the model). Finally, we assume that the execution time per tuple is uniform (*i.e.,* $w_p(t_j) = C$) and that the execution time induced by managing a sub-stream on each operator instance O_p is proportional to the number of tuples per time unit transported by that sub-stream, *i.e.,* $W_p = C \times m_p$, where C is an unknown constant. Then, the problem we want to solve is the following:

Problem 7.1 (Load Balance Key Grouped Operators)**.** *Build a partition of the stream $\sigma_{S \to O}$ in k sub-streams $\sigma_{S \to O_p}$ ($p \in [k]$), minimizing the sizes $m_{p \in [k]}$ of the sub-streams and such that $\forall t \in [n],\ t \in \sigma_{O_p} \wedge t \in \sigma_{O_{p'}} \implies p = p'$.*

Notice that this problem can be easily reduced to Problem 4.8. If the cost is superlinear (*e.g.,* quadratic) then our solution still works, however results from the theoretical and experimental analysis provided in the following section may not hold anymore. Finally, if the cost for sub-stream is constant, then the problem becomes trivial.

7.1.1 Distribution-aware Key Grouping Algorithm

In this section, we propose a new key grouping technique called Distribution-aware Key Grouping (DKG) targeted toward applications working on input streams characterized by a skewed value distribution. DKG (*cf.,* Listing 7.2) is a straightforward application of the Balanced Partitioner (BPART) algorithm presented in Section 4.4: on operator S (*cf.,* Figure 7.1), DKG instantiate BPART to **(i)** learn the distribution of the output stream $\sigma_{S \to O}$, **(ii)** build a close to optimal mapping and **(iii)** use this mapping to route the tuples of $\sigma_{S \to O}$ to the parallel instances of O.

Briefly recalling the solution principles, DKG monitors the incoming stream to identify the heavy hitters and estimate their frequency (*cf.,* Figure 7.1.C), and thus the load

Listing 7.2 – DKG algorithm.

```
1: init (θ, ε, k, μ) do
2:     bpart ← BPART algorithm instance with parameters θ, ε, k and μ.
3: end init
4: function LEARN(t_j)
5:     bpart.LEARN(t)
6: end function
7: function BUILD
8:     bpart.BUILD()
9: end function
10: function GETINSTANCE(t_j)
11:     return bpart.GETPART(t)
12: end function
```

they will impose on their target instance. At the same time it maps the sparse items, with a standard hash function (*cf.,* Figure 7.1.A), to a fixed set of buckets and tracks their frequency with a vector (*cf.,* Figure 7.1.B). After this initial training phase, whose length can be tuned, DKG removes the frequencies of the heavy hitters from the frequencies of the buckets of sparse items (*cf.,* Figure 7.1.D). The final mapping is obtained by running a greedy multiprocessor scheduling algorithm (*cf.,* Figure 7.1.E) that takes as input the heavy hitters with their estimated frequencies and the buckets with their sizes and outputs a one-to-one mapping of these elements to the available target instances. The final result is a mapping that is fine-grained for heavy hitters, that must be carefully placed on the available target instances to avoid imbalance, and coarse-grained for the sparse items whose impact on load is significant only when they are considered in large batches. At the deployment phase (*cf.,* Figure 7.1.F), DKG can leverage this mapping to route the tuples of $\sigma_{S \to O}$ to the parallel instances of O.

7.1.2 Experimental Evaluation

In this section we first evaluate DKG through a simulator. Then we look at the performances of a prototype implementing DKG as a custom grouping function in Apache Storm.

Setup — To compare the performance of DKG, we considered four algorithms:

- Full-Knowledge DKG (FK_{DKG}) is a variant of DKG with complete a priori information on the distribution of the stream portion used as validation set and with $\theta = 0$. From this point of view we consider FK_{DKG} as an upper bound for DKG performance.
- Single Instance Allocation (SIA) is an algorithm that statically allocates all the tuples on a single instance, thus producing the worst possible balancing. From this point of view we consider SIA as a lower bound for DKG performance.
- Universal provides a base line with respect to a family of hash functions known to sport nice properties. It returns $h(t)$, where $h : [n] \to [k]$ is chosen at random from a family of 2-universal hash functions.
- Apache Storm Key Grouping (ASKG) is the standard key grouping implementation in Apache Storm.

For the implementation of DKG, $\bar{h}$ was built using the same parameters of h, except for the co-domain size. To comply with the `Space Saving` requirements (*i.e.*, $\theta > \varepsilon$), in all tests we set $\varepsilon = \theta/2$. In general θ and μ are set to arbitrary values, in other words the bound $1/k \geq \theta > 1/\mu$ stated in Section 4.4.2 does not hold. In all tests we assume a linear cost for managing incoming tuples on operators. We also performed partial tests considering a quadratic cost; their results confirm the general findings discussed in this section, with different absolute values for the measured metrics.

Synthetic traces — In our tests we considered generated streams of integer values (items) representing the values of the tuples. The streams are made of $100,000$ tuples containing a value chosen among $n = 10,000$ distinct items. Taking from the machine learning field's methodology each stream was divided in two parts: a first part of $m = 80,000$ tuples was used as training set in the learning phase, while the last $20,000$ tuples of the stream (validation set) where used to perform the evaluation. Synthetic streams have been generated using Zipfian distributions with different values of α, Normal distributions with different values of mean and variance, as well as the Uniform distribution. For the sake of brevity, we omit the results for the Normal and Uniform distributions as they did not show any unexpected behaviour. As such, we restrict the results showed in the following to the Zipfian distributions with $\alpha \in \{1.0, 2.0, 3.0\}$, denoted respectively as Zipf-1, Zipf-2 and Zipf-3.

In order to have multiple different streams for each run, we generate randomly $10,000$ n-permutations of the set $\{1, \ldots, 100 \times n\}$. In other words we built $10,000$ injective random mappings of the distribution's universe $\{1, \ldots, n\}$ into a universe 100 times larger. As such, the $10,000$ distinct streams **(i)** do not share the same tuple values, **(ii)** the probability distribution of an item is not related to the natural ordering of the item values and **(iii)** there are random gaps in the natural ordering of the item values.

This dataset generation procedure was used to average the performance of hash functions employed in the tested algorithms and to avoid that a casual match between the tuple values from the stream and the used hash function always deliver very good or very bad performance that would drive the results in an undesirable manner (as this match cannot be forecasted or forced by the developer). In most of the plots presented in the following, the worst performance figures among all the runs are reported as well.

Real dataset — As a real dataset we considered data provided by the DEBS 2013 Grand Challenge [36]. The dataset contains a sequence of readings from sensors attached to football players in a match, whose values represent positions on the play field. We refer to this data set as DEBS TRACE.

Metrics — To characterize the unevenness of the operator instances loads, we take into account two well known [77] load balance metrics:

- The standard deviation SD, measuring if the load of the instances tends to be closer (small values of SD) or farther (large values of SD) from the mean. It's the average number of tuples that each instance handles in excess or deficiency with respect to the mean.
- The percentage of imbalance λ, measuring the performance lost due to imbalanced load or, dually, the performance that could be reclaimed by perfectly balancing the load. λ has been already defined in Equation 4.3.

In addition, we also consider:

Table 7.3 – Linear regression coefficients for the plots in Figure 7.4.

Algorithm	a	b
FK_{DKG}	60.7	−100
Universal Mean	60.7	−60.7
Universal Worst Case	87.3	−73.0
DKG Mean	60.7	−100
DKG Worst Case	62	−100
SIA	100	−100

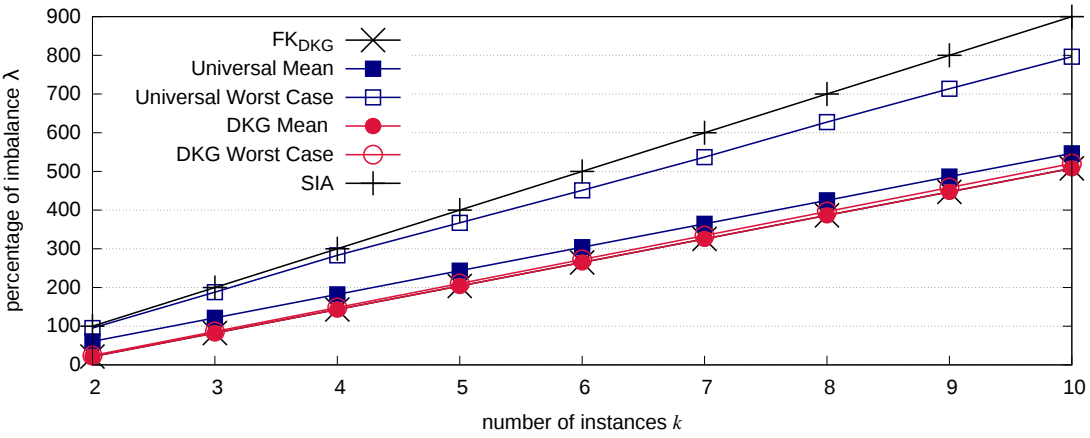


Figure 7.4 – Imbalance λ as a function of k with Zipf-2.

- The cpu load measured in Hz.
- The throughput measured in tuples processed per second.

Simulation Results

Notice that in most of the subsequent plots, points representing measured values have been linked by lines. This has been done with the sole purpose of improving readability: the lines do not represent actual measured values.

Imbalance λ with respect to the number of instances k — Figure 7.4 shows the imbalance as a function of k for DKG, Universal, FK_{DKG} and SIA, with Zipf-2 ($\theta = 0.1$ and $\mu = 2$). For both DKG and Universal, two curves are shown that report the mean and the worst case values respectively; while the mean curves report the average performance for each of the two algorithms, the worst case curves report the worst performance we observed among all the runs.

Looking at SIA's curve we get a better grasp of the meaning of λ, with $k = 10$, SIA sports 900% of imbalance: this means that it "wastes" 900% of the power provided by a single resource (*i.e.,* 9 instances are not used) which is exactly what SIA's implementation does. With Zipf-2, the empirical probability of the most frequent item is 0.60, as such it is not possible to achieve 0% imbalance with more than 1 instance ($k > 1$). This observation, with the definition of imbalance (*cf.,* Relation 4.3), justifies the monotonically increasing behavior of all curves.

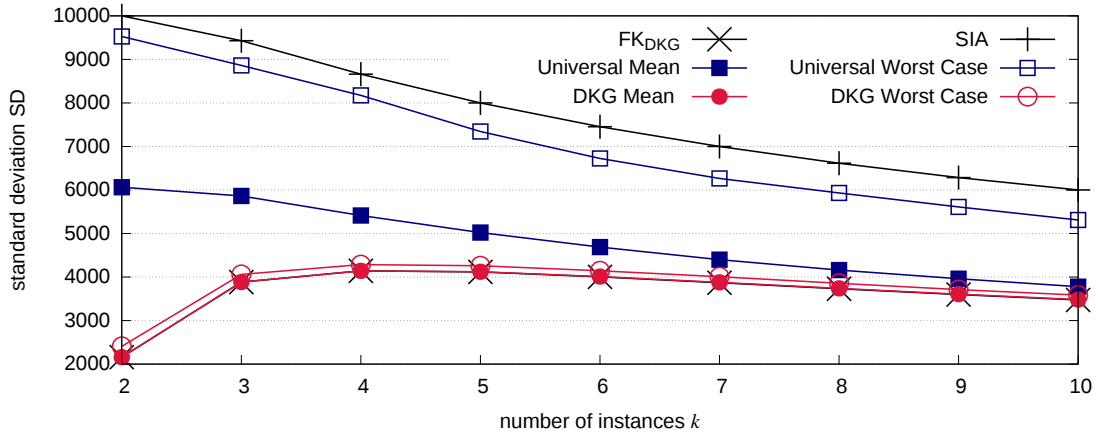


Figure 7.5 – Standard deviation SD as a function of k with Zipf-2.

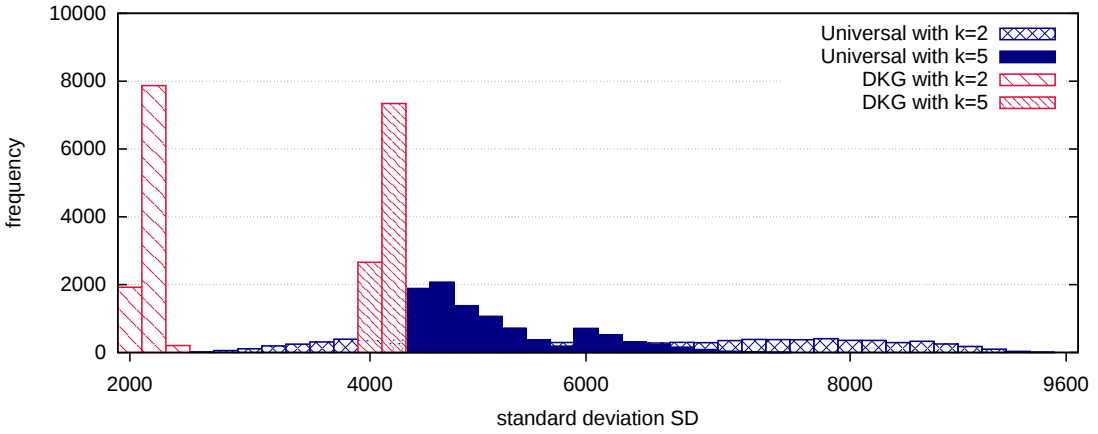


Figure 7.6 – Standard deviation SD distribution for Zipf-2.

The take away message from this figure is that in the mean case of DKG matches FK$_{DKG}$, and that its worst case is still pretty close. This means that DKG is not susceptible to unlucky choices made by the hash functions. Furthermore, there is a noteworthy gap between DKG and Universal mean values, while Universal worst case is closer to SIA than to FK$_{DKG}$. In other words DKG provides in any case a close to optimal balancing, whereas using a vanilla hash function can even approach the worst possible balancing in unlucky cases. Table 7.3 shows the values of the linear regression coefficients of the plotted results: Universal worst case has indeed a value of a closer to SIA than FK$_{DKG}$ while DKG worst case has a value of a close to FK$_{DKG}$; besides, Universal mean has not the same b of FK$_{DKG}$, while DKG mean matches FK$_{DKG}$. Notice that the results shown for DKG mean fall into the first case of the proof of Theorem 4.11 (*cf.,* Section 4.4.2), validating the claim that, with $\theta \leq \frac{1}{k}$ and with at least one heavy hitter with a frequency larger than $\overline{L}$, DKG is optimal. We also run tests for a uniform distribution or Zipfian with $\alpha < 1$; however, in these settings our solution still provides better balancing than Universal. While the absolute gain is less sizeable, it proves that it is safe (and in general worthwhile) to deploy our solution with any degree of skewness, in particular given the little memory and computational overhead (*cf.,* Section 4.4.1).

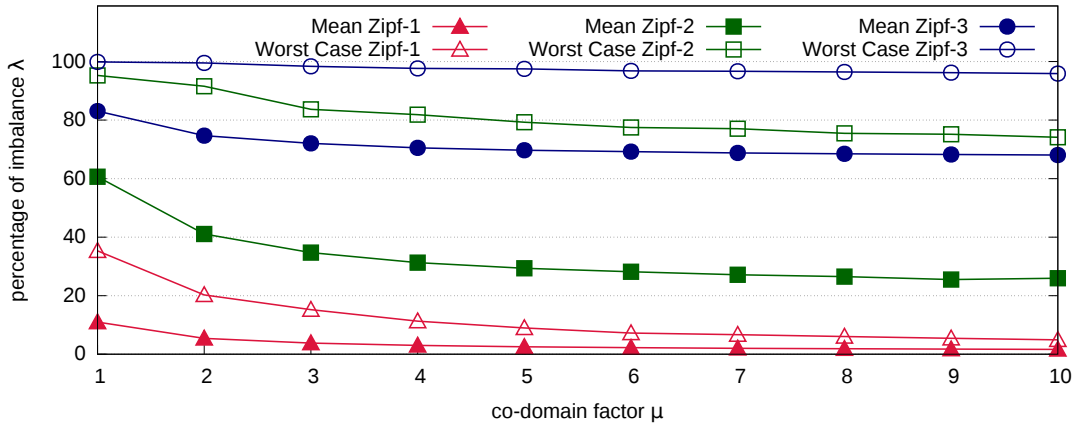


Figure 7.7 – DKG imbalance as a function of μ.

Standard deviation SD with respect to k — Figure 7.5 shows the standard deviation for the same configuration of Figure 7.4. The main difference with respect to Figure 7.4 is in the trend of the plots. SD is normalized by k and represents the average number of unbalanced items per instance. As such, SD for SIA decreases with increasing values of k, as the number of unbalanced items is fixed. On the other hand, SD for FK_{DKG} grows from $k = 2$ to $k = 4$, and then decreases. This means that the number of unbalanced items grows more than k for $k < 4$ (*i.e.,* $k = 4$ is a more difficult configuration for load balancing than $k = 2$), and grows less than k for $k \geq 4$. However, SD for SIA decreases faster than for FK_{DKG}, thus increasing k shrinks the gap between SIA and FK_{DKG}, as well as reducing DKG's gain.

Figure 7.6 shows the distribution of SD values achieved by DKG and Universal for $k \in \{2, 5\}$ in the same runs of Figure 7.5. Each bar represents how many times a SD value has been measured among the $10,000$ runs. Values are rounded to the nearest multiple of 200. We can clearly see that Universal does no better than the worst result from DKG. In addition, DKG boasts an extremely small variance, *i.e.,* the values are concentrated in a small interval of SD for both values of k. Conversely, Universal has a large variance, for instance it spans from 2800 to 9600 items for $k = 2$.

Impact of μ — Figure 7.7 shows DKG mean and worst case imbalance λ as a function of μ for Zipf-1, Zipf-2 and Zipf-3 ($\theta = 1.0$ and $k = 2$). Notice that with $\theta = 1.0$, this plot isolates the effect of μ.

Increasing μ from 1 to 2 significantly decreases the mean imbalance, while for larger values of μ the gain is less evident. Instead, the worst case imbalance is only slightly affected by μ. A larger co-domain size allows $\bar{h}$ to spread the items more evenly on more buckets and gives more freedom to the GMPS algorithm. Thus, increasing μ reduces the chances to incur in pathological configurations, but cannot fully rule them out. This mechanism grants DKG the ability, as stated previously, to provide better balancing than Universal even with non skewed distributions.

Impact of θ — Figure 7.8 shows DKG mean and worst case imbalance λ as a function of θ for Zipf-1, Zipf-2 and Zipf-3 ($\mu = 1.0$ and $k = 2$). Notice that with $\mu = 1.0$, this plot isolates the effect of θ.

Once θ reaches the empirical probability of the most frequent heavy hitter in the stream (0.8 for Zipf-3, 0.6 for Zipf-2 and 0.1 for Zipf-1), both the mean and worst case

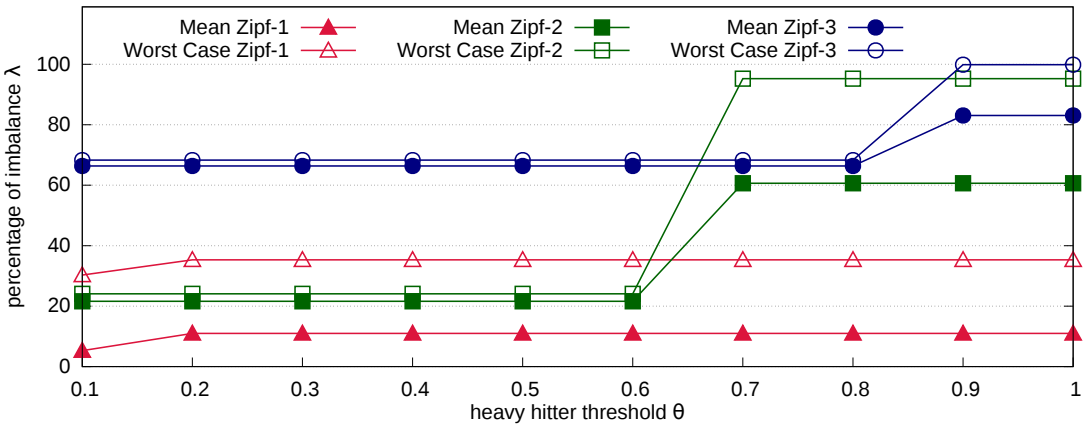


Figure 7.8 – DKG imbalance as a function of θ.

values drop. Further decrements of θ do not improve significantly the performance. For Zipf-3 and Zipf-2, the values of both the mean and worst case are very close, proving that the mechanism used to separately handle the heavy hitters is able to rule out pathological configurations. Conversely, most of the unbalancing for Zipf-1 comes from a bad mapping induced by $\bar{h}$ on the sparse items. In other words, with $\mu = 1.0$ the GMPS algorithm does not have enough freedom to improve the sparse item mapping.

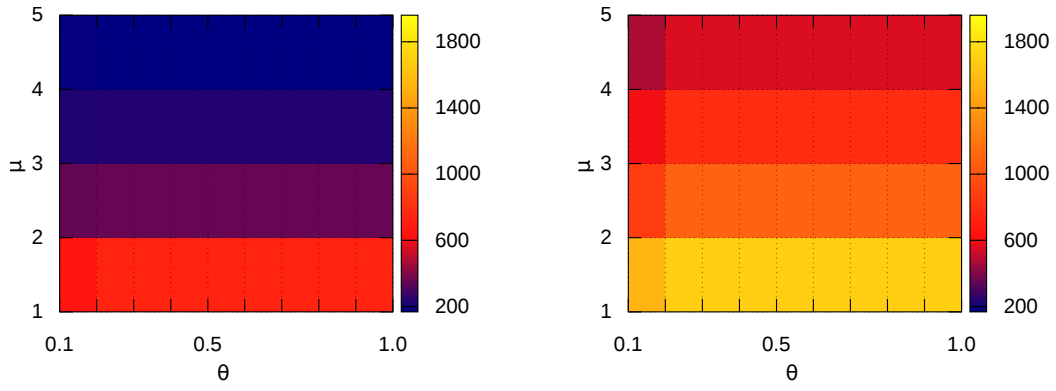
We do not show FK_{DKG} in Figure 7.8, however notice that for both Zipf-3 and Zipf-2 with $\theta = 0.8$ and $\theta = 0.6$ respectively, DKG mean imbalance is equal to FK_{DKG} imbalance. With Zipf-3 and Zipf-2, the theoretical analysis (*cf.,* Section 4.4.2) guarantees optimality with $1/\mu < \theta \leq 1/k = 0.5$. In other words the user can either leverage the theoretical results to be on the safe side with any value of α, or, given some a priori knowledge on the stream (*i.e.,* a lower bound on the value of α), use different values of θ, ε and μ to reduce resources usage.

θ and μ trade-off — The heat-maps in Figure 7.9 show DKG mean and worst case standard deviations SD as a function of both θ and μ for Zipf-1, Zipf-2 and Zipf-3 ($k = 5$). Notice that the heat-maps for different distributions do not share the color-scale. In all figures darker is better.

Figures 7.9a and 7.9b confirm that, as for imbalance, with non- or lightly-skewed distributions, μ drives the performance, while θ has a negligible impact. Figure 7.9c, 7.9d, 7.9c and 7.9d confirm that, for skewed distributions, as soon as θ matches the empirical probability of the most frequent heavy hitters, there is not much to gain by further increasing μ or decreasing θ.

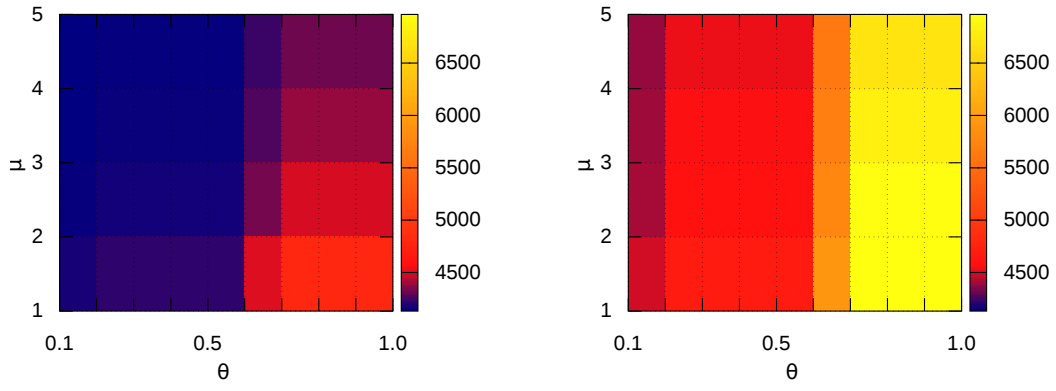
Comparison with respect to [44] — Gedik proposed a solution that is close to our from several perspectives. However, differently from our solution, sparse items are mapped using a consistent hash function, while we map them in buckets that are later scheduled, depending on their load, to sub-streams.

Figure 7.10 shows the imbalance λ as a function of α of both the mean and worst cases for DKG and DKG WOSIM. The latter is a modified version of DKG where sparse items are mapped to sub-streams directly using the $\overline{h}$ function whose co-domain has been set to k. In other words the scheduling algorithm does not take into account the mapping of sparse items, which is the main difference with respect to the solution proposed by



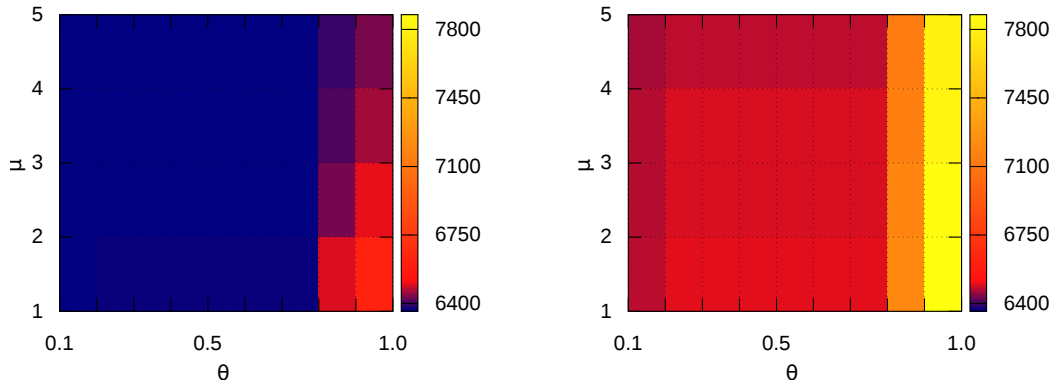
(a) Mean Zipf-1

(b) Max Zipf-1



(c) Mean Zipf-2

(d) Max Zipf-2



(e) Mean Zipf-3

(f) Max Zipf-3

Figure 7.9 – DKG standard deviation SD as a function of both θ and μ for Zipf-1, Zipf-2 and Zipf-3.

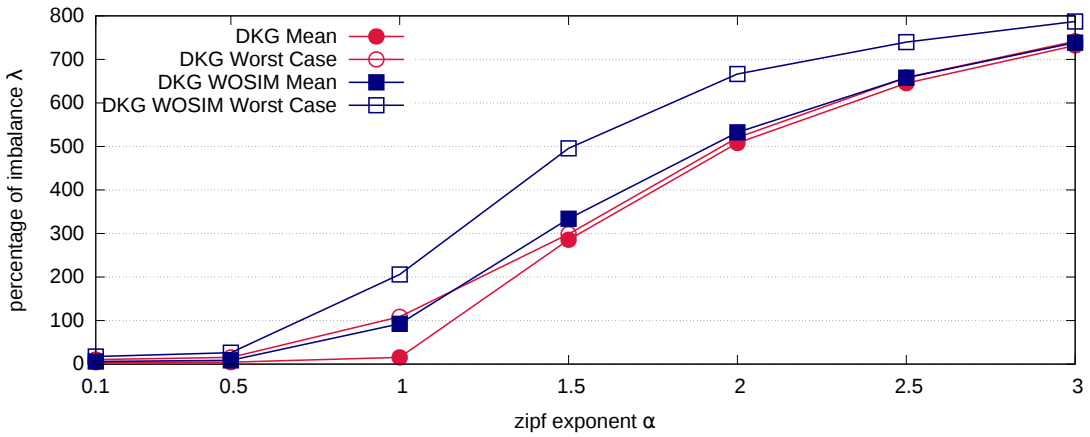


Figure 7.10 – Comparing DKG and [44]: imbalance λ as a function of α.

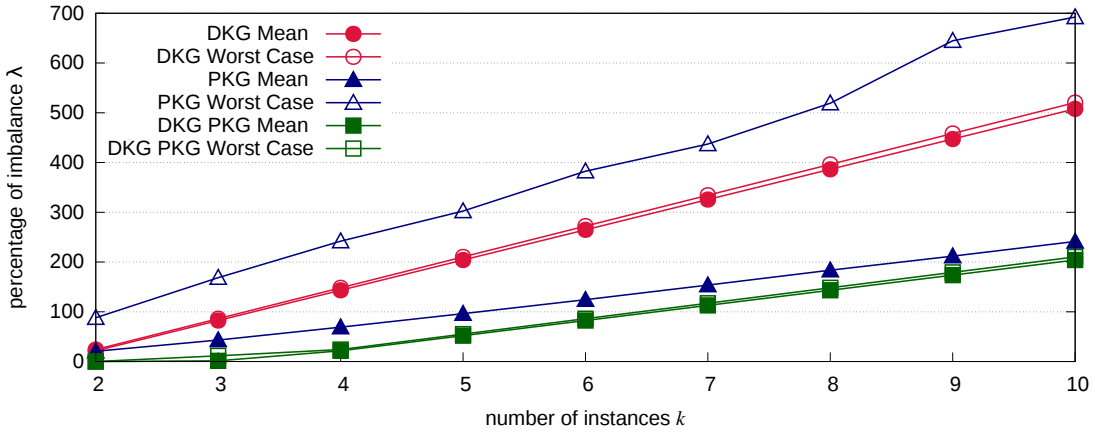


Figure 7.11 – Comparing DKG and `PKG` [73]: imbalance λ as a function of k.

Gedik [44]. As the curves show, DKG always outperforms the WOSIM version for all tested values of α. One further important difference between this work and [44] is that the latter proposes a set of heuristics for key mapping that take into account the cost of operator migration, making its solution tailored to systems that must adapt at runtime to changing workload distributions.

Comparison with respect to [73] — Nasir *et al.* applied the *power of two choices* approach to key grouping, achieving impressive load balancing. Differently from our solution, `PKG` [73] cannot be applied in general, but it is limited to operators that allow a *reduce* phase to reconcile the split state for each key. Interestingly, our solution can work in conjunction with the one proposed in [73] to provide even better performance.

Figure 7.11 shows both the mean and worst case imbalance λ as a function of k for DKG, `PKG`, and DKG PKG. The latter is a modified version of DKG where we plug into it the `PKG` logic. Each heavy hitter is fed to the scheduling algorithm as two distinct items with half its original frequency. As such, the GMPS algorithm provides two different mappings for each heavy hitter. When hashing a heavy hitter, DKG PKG returns the less loaded instance between the two associated with the heavy hitter. Notice that sparse items are still associated with a single instance. `PKG` is the implementation provided

by the authors of [73]. The curves show that combining both solutions it is possible to obtain even better performance. Interestingly, both DKG and DKG PKG worst case performance are better than PKG worst case performance, stressing again how our solution is able to provide stable performance irrespective of exogenous factors.

Prototype Results

To evaluate the impact of DKG on real applications we implemented it [1] as a custom grouping function within the Apache Storm [89] framework. More in details, DKG implements the CUSTOMSTREAMGROUPING interface offered by the Storm API, defining two methods: PREPARE() and CHOOSETASKS(t). The former is a setup method, while the latter returns the replica(s) identifier(s) associated with tuple t. In DKG, the class constructor and the PREPARE() method implement the INIT() pseudo-code (*cf.,* Listing 7.2). In particular they take θ, ε, μ, k, the size of the learning set m, and the function KEY as parameters. KEY is a user defined function that returns a positive integer value representing the grouping key(s) value(s) of tuple t. In all our tests we set $\theta = 0.1$, $\varepsilon = 0.05$ and $\mu = 2$, a set of sensible values that proved to be meaningful in our testbed. The CHOOSETASKS(t) method implements the rest of the pseudo-code (*cf.,* Listing 7.2): it **(i)** uses the first m tuples to learn (LEARN(t)), then **(ii)**, once it has read m tuples, stops learning and builds (BUILD()) the global mapping function and **(iii)**, finally, returns GETINSTANCE(t) for any following tuple t.

The use case for our tests is a partial implementation of the third query from the DEBS 2013 Grand Challenge [36]: splitting a play field in four grids, each with a different granularity. The goal is to compute how long each of the monitored players is in each grid cell, taking into account four different time windows.

The test topology is made of a source (*spout* in Storm jargon) and an operator (*bolt*) with k instances (*tasks*). To avoid I/O to be a bottleneck for our tests, the source store the whole sensors reading data file in memory. For each reading, it emits 4 tuples (one for each granularity level) towards the operator instances. The grouping key is the tuple cell identifier (*i.e.,* row, column and granularity level). We take into account the second half of the match, which is made up of roughly 2.5×10^7 readings, generating close to 10^8 tuples. The training set is the first half of the trace, while the renaming half is the validation set.

We deployed the topology on a server with a 8-cores Intel Xeon CPUs with HyperThreading clocked at at 2.00GHz and with 32GB of RAM. The source thread (*spout's task*) and the k operator instances threads (*bolts' tasks*) are spawned each on a different JVM instance. We performed tests for $k \in \{1, \ldots, 10\}$ with both the default key grouping implementation (ASKG) and the DKG prototype.

Imbalance λ with respect to the number of instances k — Figure 7.12 shows the imbalance as a function of k for FK_{DKG}, Universal/ASKG and DKG with the DEBS TRACE. For Universal/ASKG and DKG we show both the simulated results and the outcome from the prototype (*i.e.,* the imbalance of the number of tuples received by the operator instances). We can notice that, for both DKG and Universal/ASKG, results from the tested topology closely match those provided by the simulations. DKG sticks very close to FK_{DKG} for all k, and, except for $k \in \{7, 9\}$, largely outperforms Univer-

[1] The implementation's code is available at the following repository: `http://github.com/rivetti/dkg_storm`

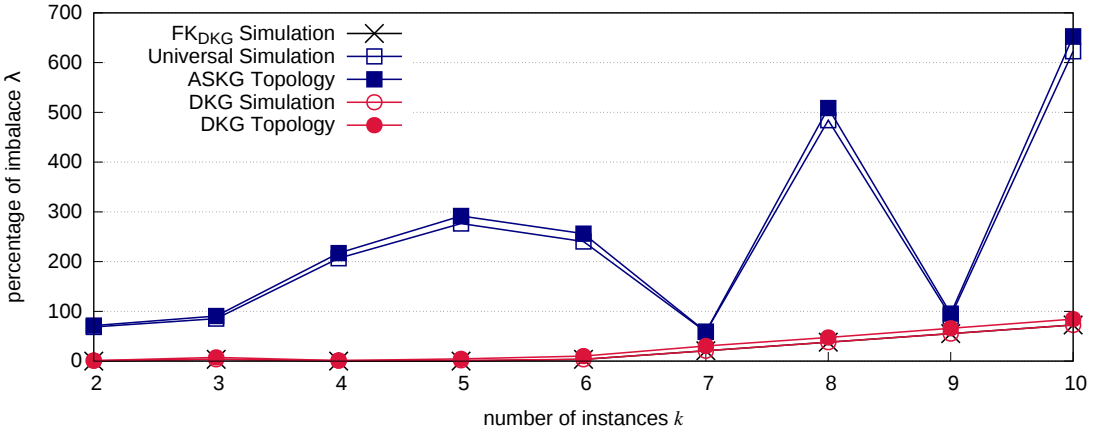


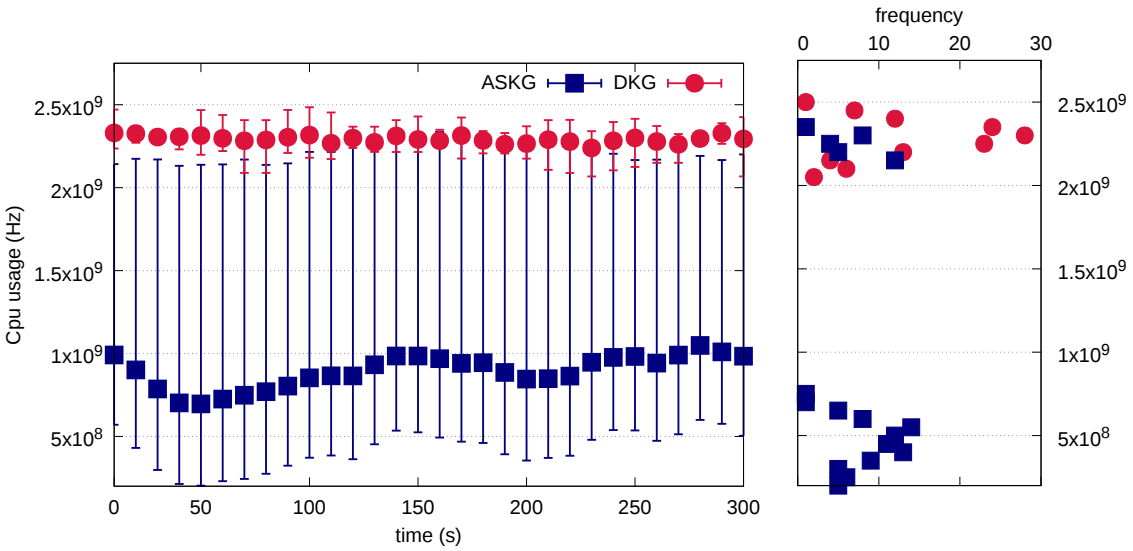
Figure 7.12 – Imbalance λ for the DEBS TRACE as a function of k.

sal/ASKG. Furthermore, we can clearly see the hardly predictable behavior faced when using a vanilla hash function. For $k \in \{2, \ldots, 5\}$, FK_{DKG} achieves 0% imbalance. However the most frequent item of the source outgoing stream has an empirical probability of roughly 1/5. It is then impossible to achieve 0% imbalance for $k \geq 6$ and FK_{DKG}, as well as DKG, imbalance grows with k.

Cpu usage and throughput — Figure 7.13a shows the cpu usage (Hz) over time (300 seconds of execution) for DKG and ASKG with the DEBS TRACE in the test topology with $k = 4$ instances. The cpu usage was measured as the average number of hertz consumed by each instance every 10 seconds. Plotted values are the mean, maximum and minimum cpu usage on the 4 instances.

The mean cpu usage for DKG is close to the maximum cpu usage for ASKG, while the mean cpu usage for ASKG is much smaller. This was expected as there is a large resource under-utilisation with ASKG. Figure 7.13b shows the cpu usage's distribution for the same time-slice. In other words the data point x-axis value represents how many times an instance has reached this cpu usage (Hz). Notice that the values are rounded to the nearest multiple of 5×10^7. This plot confirms that the cpu usage for DKG's replicas is concentrated between 2×10^9 and 2.5×10^9 Hz, *i.e.,* all instances are well balanced with DKG. On the other hand ASKG does hit this interval, but most of the data points are close to 5×10^8 Hz. The key grouping provided by DKG loads evenly all available instances. Conversely, with ASKG some instance (in particular 3) are underused, leaving most of the load on fewer instances (in particular 1).

This improvement in resource usage translates directly into a larger throughput and reduced execution time, as clearly shown by Figure 7.14; in particular, in our experiments, DKG delivered 2× the throughput of ASKG for $k \in \{4, 5, 6, 8, 10\}$.



(a) Timeline (b) Distribution

Figure 7.13 – Cpu usage (Hz) for 300s of execution.

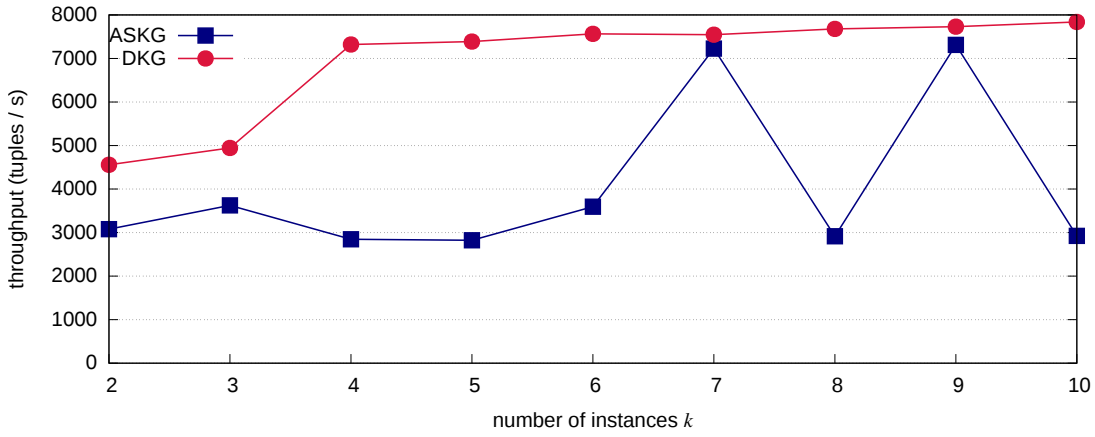


Figure 7.14 – Throughput (tuples/s) for the DEBS TRACE as a function of k.

7.2 Load Balancing Stateless Parallelized Operators

Load balancing shuffle grouped operators is in general considered a trivial problem that can be solved through simple policies such as Round-Robin. However, this relies on the assumption that all the tuples have the same execution time, which does not hold for many practical use cases. The tuple execution time, 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 time changes from tuple to tuple. As a practical example consider an operator that works on a stream of input tweets and that enriches them with historical data extracted from a database, where this historical data is added only to tweets that contain specific hashtags: only tuples that get enriched require an access to the database, an operation that typically introduces non negligible latencies at execution time. In this case shuffle grouping implemented with Round-Robin may produce imbalance between the operator instances, and this typically causes an increase in the time needed for a tuple to be completely processed by the application (tuple *completion time*) as some tuple may end-up being queued on some overloaded operator instances, while other instances are available for immediate processing.

On the basis of the previous observation the tuple scheduling strategies for shuffle grouping must be re-thought: tuples must be scheduled with the aim of balancing the overall processing time on operators in order to reduce the average tuple execution time. However, tuple processing times are not known in the scheduling phase.

Problem Statement The execution time $w_p(t)$ is modelled as an unknown function[2] of the content of tuple t and that may be different for each operator instance (*i.e.,* we do not assume that the operator instances are uniform). We simplify the model assuming that $w_p(t)$ depends on a single fixed and known attribute value of t. The probability distribution of such attribute values, as well as w_p are unknown 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. Abusing the notation, we may omit in w_p the operator instance identifier subscript.

The general goal we target in this work is to minimize the average tuple completion time $\overline{L}$. Such metric is fundamentally driven by three factors: (**i**) tuple execution times at operator instances, (**ii**) network latencies and (**iii**) queuing delays. More in detail, we aim at reducing queuing delays at parallel operator instances that receive input tuples through shuffle grouping.

Problem 7.2 (Load Balance Shuffle Grouped Operators)**.** *Build a schedule of all the tuples of the stream $\sigma_{S\to O}$ to k sub-streams $\sigma_{S\to O_p}$ ($p \in [k]$), minimizing the average tuple completion time $\overline{L} = \frac{1}{m}\sum_{t_j \in \sigma_{S\to O}} \ell(t_j)$.*

Typical implementation of shuffle grouping are based on Round-Robin scheduling. However, this tuple to sub-streams assignment strategy may introduce additional queuing delays when the execution time of input tuples is not similar. For instance, let a_0, b_1, a_2 be a stream with an inter tuple arrival delay of 1s, where a and b are tuples with the respective execution time: $w(a) = w(a) = 10$s and $w(b) = 1$s. Scheduling this stream with Round-Robin on $k = 2$ operator instances would assign a_0 and a_2 to instance O_1

[2] In the experimental evaluation we relax the model by taking into account the execution time variance

and b_1 to instance O_2, with a cumulated completion time equal to $\ell(a_0)+\ell(b_1)+\ell(a_2) =$ 29s, where $\ell(a_0) = 10$s, $\ell(b_1) = 1$s and $\ell(a_2) = (8 + 10)$s, and $\overline{L} = 9.66$s. Note the wasted queuing delay of 8s for tuple a_2. A better schedule would be to assign a_0 to instance 1, while b_1 and a_2 to instance 2, giving a cumulated completion time equals to $10 + 1 + 10 = 21$s (*i.e.,* no queuing delay), and $\overline{L} = 7$s.

7.2.1 Online Shuffle Grouping Scheduler Algorithm

Online Shuffle Grouping Scheduler is a shuffle grouping implementation based on a simple, yet effective idea: if we assume to know the execution time $w_p(t)$ of each tuple t on any of the operator instances, we can schedule the execution of incoming tuples on such instances with the aim of minimizing the average per tuple completion time at the operator instances. Still, the value of $w_p(t)$ is generally unknown. A common solution to this problem is to build a cost model for the tuple execution time and then use it to proactively schedule incoming 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.

To overcome all these issues, OSG takes decisions based on the estimation $\widehat{W}_p$ of the total execution time assigned to instance O_p, that is $W_p = \sum_{t \in \sigma_{O_p}} w_p(t)$. In order to do so, OSG computes an estimation $\widehat{w}_p(t)$ of the execution time $w_p(t)$ of each tuple t on each operator instance O_p. Then, OSG can also compute the sum of the estimated execution times of the tuples assigned to an instance O_p, *i.e.,* $\widehat{W}_p = \sum_{t \in \sigma_{O_p}} \widehat{w}_p(t)$, which in turn is the estimation of W_p. A greedy scheduling algorithm is then fed with estimations for all the available operator instances.

To implement this approach, each operator instance builds a sketch (*i.e.,* a memory efficient data structure) that tracks the execution time of the tuples it processes. When a change in the stream or instance(s) characteristics affects the tuples execution times on some instances, the concerned instance(s) forward an updated sketch to the scheduler which will then be able to (again) correctly estimate the tuples execution times. This solution does not require any *a priori* knowledge on the stream composition or the system, and is designed to continuously adapt to changes in the input distribution or on the instances load characteristics. In addition, this solution is *proactive*, namely its goal is to avoid unbalance through scheduling, rather than detecting the unbalance and then attempting to correct it. A *reactive* solution can hardly be applied to this problem, in fact it would schedule input tuples on the basis of a previous, possibly stale, load state of the operator instances. In addition, reactive scheduling typically imposes a periodic overhead even if the load distribution imposed by input tuples does not change over time.

Greedy Online Multi Processor Scheduling (GOMPS) A classical problem in the load balancing literature is to schedule independent tasks on identical machines minimizing the makespan, *i.e.,* the *Multiprocessor Scheduling* problem. We adapt this problem to our setting, *i.e.,* to schedule *online* independent tuples on non-uniform operator instances in order to minimize the average per tuple completion time $\overline{L}$. Online scheduling means that the scheduler does not know in advance the sequence of tasks it has to schedule. The GOMPS algorithm assigns the currently submitted tuples to the less loaded available operator instance. In Section 7.2.2 we prove that this algorithm closely approximates an optimal omniscient scheduling algorithm, that is an algorithm

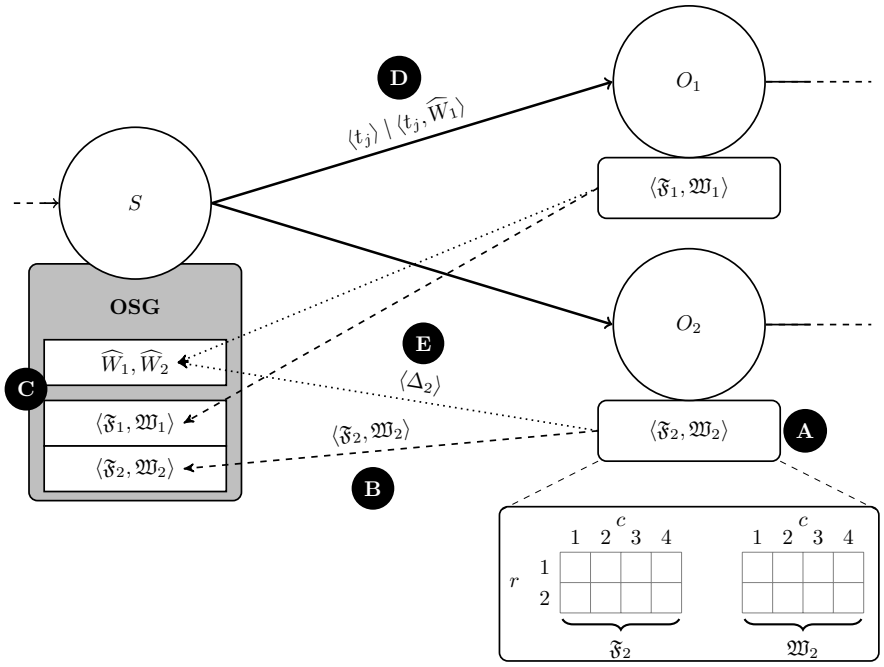


Figure 7.15 – OSG design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$) and $k = 2$.

that knows in advance all the tuples it will receive. Notice that this is a variant of the join-shortest-queue (JSQ) policy [52, 71], where we measure the queue length as the time needed to execute all the buffered tuples, instead of the number of buffered tuples.

OSG design

The sketch used by OSG to estimate the tuple execution time is a straightforward extension of VALES (*cf.,* Section 3.3.1). Each operator instance O_p maintains two matrices (Figure 7.15.A): the first one, denoted by $\mathfrak{F}_{O_p}$, tracks the tuple frequencies f_{t,O_p}; the second, denoted by $\mathfrak{W}_{O_p}$, tracks the tuples cumulated execution times $w_{O_p}(t) \times f_{t,O_p}$. Both matrices have the same size $r \times c$, where $r = log\frac{1}{\delta}$ and $c = \frac{e}{\varepsilon}$, and hash functions. The operator instance updates them (Listing 7.16) after each tuple execution.

The operator instances are modelled as a finite state machine (Figure 7.17) with two states: START and STABILIZING. The START state lasts until instance O_p has executed M^{OSG} tuples, where M^{OSG} is a user defined window size parameter. The transition to the STABILIZING state (Figure 7.17.A) triggers the creation of a new snapshot $\mathfrak{S}_p$. A snapshot is a matrix of size $r \times c$ where $\forall \imath \in [r], \jmath \in [c] : \mathfrak{S}_p[\imath, \jmath] = \mathfrak{W}_p[\imath, \jmath]/\mathfrak{F}_p[\imath, \jmath]$. We say that the $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices are stable when the relative error η_p between the previous snapshot and the current one is smaller than $\overline{\eta}$, that is if

$$\eta_p = \frac{\sum_{\imath=1}^{r} \sum_{\jmath=1}^{c} \left| \mathfrak{S}_p[\imath, \jmath] - \frac{\mathfrak{W}_p[\imath, \jmath]}{\mathfrak{F}_p[\imath, \jmath])} \right|}{\sum_{\imath=1}^{r} \sum_{\jmath=1}^{c} \mathfrak{S}_p[\imath, \jmath]} \leq \overline{\eta} \tag{7.1}$$

is satisfied. Then, each time instance O_p has executed M^{OSG} tuples, it checks whether Equation 7.1 is satisfied. **(i)** If not, then $\mathfrak{S}_p$ is updated (Figure 7.17.B). **(ii)** Otherwise the operator instance sends the $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices to the scheduler (Figure 7.15.B), resets them and moves back to the START state (Figure 7.17.C).

Listing 7.16 – OSG algorithm on operator instance O_p.

```
1: init (r, c) do
2:     F_p, W_p ← 0_{r,c}
3:     r hash functions h_1, ..., h_r : [n] → [c] from a 2-universal family.
4: end init
5: function UPDATE(t_j, w(t_j))
6:     for ι = 1 to r do
7:         F_p[ι, h_ι(t)] ← F_p[ι, h_ι(t)] + 1
8:         W_p[ι, h_ι(t)] ← W_p[ι, h_ι(t)] + w(t_j)
9:     end for
10: end function
```

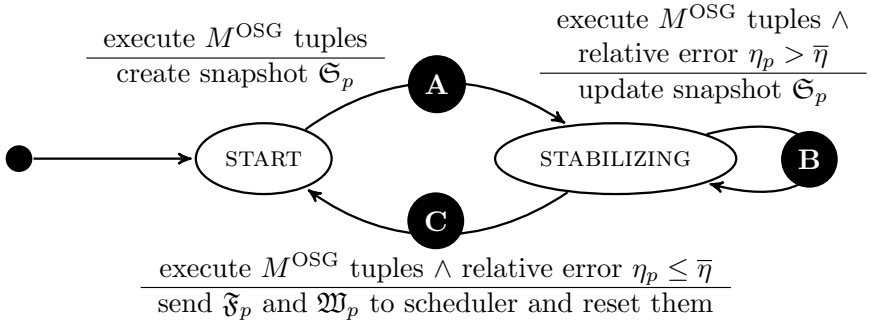


Figure 7.17 – OSG operator instance finite state machine.

There is a delay between any change in the stream or operator instances characteristics and when the scheduler receives the updated $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices from the affected operator instance(s). This introduces a skew in the cumulated execution times estimated by the scheduler. In order to compensate for this skew, we introduce a synchronization mechanism that springs whenever the scheduler receives a new pair of matrices from any operator instance. Notice also that there is an initial transient phase in which the scheduler has not yet received any information from operator instances. This means that in this first phase it has no information on the tuples execution times and is forced to use the Round-Robin policy. This mechanism is thus also needed to initialize the estimated cumulated execution times when the Round-Robin phase ends.

The scheduler (Figure 7.15.C) maintains the estimated cumulated execution time a scalar vector $\widehat{W}_p$ for each instance ($p \in [k]$), and the set of pairs of matrices $\{\langle \mathfrak{F}_p, \mathfrak{W}_p \rangle\}$, initially empty. It is modelled as a finite state machine (Figure 7.19) with four states: ROUND ROBIN, SEND ALL, WAIT ALL and RUN.

The ROUND ROBIN state is the initial state in which scheduling is performed with the Round-Robin policy. In this state, the scheduler collects the $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices sent by the operator instances (Figure 7.19.A). After receiving the two matrices from each instance (Figure 7.19.B), the scheduler is able to estimate the execution time for each submitted tuple and moves into the SEND ALL state. When in SEND ALL state, the scheduler sends the synchronization requests towards the k instances. To reduce overhead, requests are piggy backed (Figure 7.15.D) with outgoing tuples applying the Round-Robin policy for the next k tuples: the i-th tuple is assigned to operator instance $i \mod k$. On the other hand, the estimated cumulated execution $\widehat{W}_p$ is updated with the tuple estimated execution time using the UPDATE $\widehat{W}$ function (Listing 7.18). When

Listing 7.18 – OSG scheduler on operator S.

```
1: init do(r, c, k)
2:     A set of k counters Ŵ_p
3:     A set of ⟨𝔉_p, 𝔚_p⟩ matrices pairs
4:     h_1, ..., h_r : [n] → [c], the same r hash functions of the operator instances
5: end init
6: function SUBMIT(t_j)
7:     return arg min_{O_p∈[k]}{Ŵ_p}
8: end function
9: function UPDATEŴ(t_j, operator : O_p)
10:    ι ← arg min_{ι∈[r]}{𝔉_p[ι, h_ι(t)]}
11:    Ŵ_p ← Ŵ_p + (𝔚_p[ι, h_ι(t)]/𝔉_p[ι, h_ι(t)])
12: end function
```

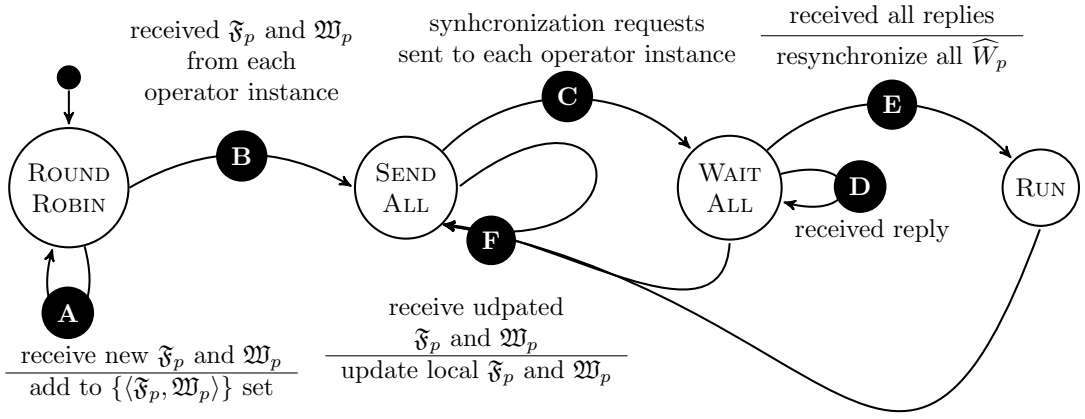


Figure 7.19 – OSG scheduler finite state machine.

all the requests have been sent (Figure 7.19.C), the scheduler moves into the WAIT ALL state. This state collects the synchronization replies from the operator instances (Figure 7.19.D). Operator instance O_p reply (Figure 7.15.E) contains the difference Δ_p between the instance cumulated execution time W_p and the scheduler estimation $\widehat{W}_p$.

In the WAIT ALL state, scheduling is performed as in the RUN state, using both the SUBMIT and the UPDATE $\widehat{W}$ functions (Listing 7.18). When all the replies for the current epoch have been collected, synchronization is performed and the scheduler moves in the RUN state (Figure 7.19.E). In the RUN state, the scheduler assigns the input tuple applying the GREEDY ONLINE MULTI PROCESSOR SCHEDULING algorithm, *i.e.,* assigns the tuple to the operator instance with the least estimated cumulated execution time (SUBMIT function, Listing 7.18). Then it increments the target instance estimated cumulated execution time with the estimated tuple execution time (UPDATE $\widehat{W}$ function, Listing 7.18). Finally, in any state except ROUND ROBIN, receiving an updated pair of matrices $\mathfrak{F}_p$ and $\mathfrak{W}_p$ moves the scheduler into the SEND ALL state (Figure 7.19.F).

Theorem 7.3 (OSG Time Complexity)**.**
For each tuple read from the input stream, the time complexity of OSG for each instance is $O(\log(1/\delta))$*. For each tuple submitted to the scheduler, OSG time complexity is* $O(k + \log(1/\delta))$*.*

Proof. By Listing 7.16, for each tuple read from the input stream, the algorithm increments an entry per row of both the $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices. Since each has $\log(1/\delta)$ rows, the resulting update time complexity is $O(\log(1/\delta))$. By Listing 7.18, for each submitted tuple, the scheduler has to retrieve the index with the smallest value in the vector $\widehat{W}$ of size k, and to retrieve the estimated execution time for the submitted tuple. This operation requires to read entry per row of both the $\mathfrak{F}_p$ and $\mathfrak{W}_p$ matrices. Since each has $\log(1/\delta)$ rows, the resulting update time complexity is $O(k + \log(1/\delta))$. □

Theorem 7.4 (OSG Space Complexity)**.**
The space complexity of OSG for the operator instances is $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$*, while the space complexity for the scheduler is* $O\left(\frac{k}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$*.*

Proof. Each operator instance stores two matrices, each one requiring $\frac{1}{\varepsilon}\log(1/\delta)\log m$ bits. In addition, it also stores a hash function whose domain size is n. Then the space complexity of OSG on each operator instance is $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$. The scheduler stores the same matrices, one for each instance, as well as k counters. Then the space complexity of OSG on the scheduler is $O\left(\frac{k}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n)\right)$. □

Theorem 7.5 (OSG Communication Complexity)**.** *The communication complexity of OSG is of* $O(\frac{m}{M^{OSG}})$ *messages and* $O\left(\frac{m}{M^{OSG}}\frac{1}{\varepsilon}\log\frac{1}{\delta}\log m\right)$ *bits.*

Proof. After executing M tuples, an operator instance may send the $\mathfrak{F}_p, \mathfrak{W}_p$ matrices to the scheduler. This generates a communication cost of $O(\frac{m}{M^{OSG}})$ messages and $O\left(\frac{m}{M^{OSG}}\frac{1}{\varepsilon}\log\frac{1}{\delta}\log m\right)$ bits. When the scheduler receives these matrices, the synchronization mechanism springs and triggers a round trip communication (half of which is piggy backed by the tuples) with each instance. This introduces an additional communication cost of $O(\frac{m}{M^{OSG}})$ messages and $O(\frac{m}{M^{OSG}}\log m)$ bits. □

7.2.2 Theoretical Analysis

We first analyse FK_{OSG}, a variant of OSG where we assume that the processing time $w_p(t)$ is known for each tuple t. We suppose that tuples cannot be preempted, that is tuples must be processed in an uninterrupted fashion on the operator instance it has been scheduled on. Finally, given our system model, the problem of minimizing the average completion time $\overline{L}$ can be reduced to the following problem (in terms of *makespan*):

Problem 7.6 (Minimum Makespan)**.** *Given* k *identical operator instances, and a sequence of tuples* $\sigma = \langle t_1, \ldots, t_m \rangle$ *that arrive online from the input stream. Find an online scheduling algorithm that minimizes the makespan of the schedule produced by the online algorithm when fed with* σ*.*

Let OPT be the schedule algorithm that minimizes the makespan over all possible sequences σ, and C_{OPT} denote the makespan of the schedule produced by the OPT algorithm fed by sequence σ. Notice that finding C_{OPT} is an NP-hard problem. We show that GOMPS builds a schedule that is within some factor of the lower bound of the quality of the optimal scheduling algorithm OPT. Let us denote by W_{GOMPS} the makespan of the schedule produced by the greedy algorithm fed with σ.

Theorem 7.7 (GOMPS Approximation)**.** *For any σ, we have $C_{\text{GOMPS}} \leq (2-1/k)C_{OPT}$.*

Proof. Let O_p be the instance on which the last tuple t is executed. By construction of the algorithm, when tuple t starts its execution on instance O_p, all the other instances are busy, otherwise t would have been executed on another instance. Thus when tuple t starts its execution on instance O_p, each of the k instances must have been allocated a load at least equivalent to $(\sum_{j=1}^{m} w(t'_j) - w(t))/k$. Thus we have,

$$\begin{aligned} C_{\text{GOMPS}} - w(t) &\leq \frac{\sum_{j=1}^{m} w(t'_j) - w(t)}{k} \\ C_{\text{GOMPS}} &\leq \frac{\sum_{j=1}^{m} w(t'_j)}{k} + w(t)(1 - \frac{1}{k}) \end{aligned} \tag{7.2}$$

Now, it is easy to see that

$$C_{OPT} \geq \frac{\sum_{j=1}^{m} w(t'_j)}{k}, \tag{7.3}$$

otherwise the total load processed by all the operator instances in the schedule produced by the OPT algorithm would be strictly less than $\sum_{j=1}^{m} w(t'_j)$, leading to a contradiction. We also trivially have

$$C_{OPT} \geq \max_j w(t'_j). \tag{7.4}$$

Thus combining relations (7.2), (7.3), and (7.4), we have

$$\begin{aligned} C_{\text{GOMPS}} &\leq C_{OPT} + C_{OPT}\left(1 - \frac{1}{k}\right) \\ &= \left(2 - \frac{1}{k}\right) C_{OPT} \end{aligned} \tag{7.5}$$

that concludes the proof. □

This lower bound is tight, that is, there are sequences of tuples for which the GOMPS algorithm produces a schedule whose completion time is exactly equal to $(2-1/k)$ times the completion time of the optimal scheduling algorithm [53].

Consider the example of $k(k-1)$ tuples with all the same processing time equal to $w(t) = a/k$ and one tuple with a processing time equal to $w(t') = a$. Suppose that the $k(k-1)$ tuples are scheduled first and then the longest one. Then the greedy algorithm exhibits a makespan equal to $a(k-1)/k + a = a(2-1/k)$ while the OPT scheduling has a makespan equal to a.

OSG feeds GOMPS with the estimated execution times $\widehat{w}_p(t)$ provided by VALES, then OSG (ε, δ)-approximates FK_{OSG}.

7.2.3 Experimental Evaluation

In this section we evaluate the performance obtained by using OSG to perform shuffle grouping. We first describe the general setting used to run the tests and then discuss the results obtained through simulations and with a prototype of OSG targeting Apache Storm.

Setup — We present the results for 4 algorithms:

- The Round-Robin algorithm is the implementation of the classical Round-Robin policy. We consider Round-Robin as the base-line in the simulations.

- The ASSG algorithm is the standard Apache Storm shuffle grouping implementation. We consider ASSG as the base-line in the use-cases.
- The OSG algorithm is our solution.
- The FK_{OSG} algorithm, is a variant of OSG where the estimation is exact, *i.e.,* the scheduling algorithm is fed with the exact execution times for each tuple. We consider FK_{OSG} as an upper bound bounds of OSG performance.

OSG parameters are the operator window size M^{OSG}, the tolerance parameter $\overline{\eta}$, and the parameters of the matrices $\mathfrak{F}$ and $\mathfrak{W}$: ε and δ. These are respectively set to $M^{OSG} = 1,024$, $\overline{\eta}$, $\varepsilon = 0.05$ (*i.e.,* $c = 54$ columns) and $\delta = 0.1$ (*i.e.,* $r = 4$ rows).

Synthetic traces — For synthetic datasets we generate streams of integer values (items) representing the values of the tuple attribute driving the execution time when processed on an operator instance. We consider streams of $m = 100,000$ tuples, each containing a value chosen among $n = 4,096$ distinct items. Synthetic streams have been generated using the Uniform distribution 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 n_w as the number of distinct execution time values that the tuples can have. These n_w values are selected at *constant* distance in the interval $[\min_w, \max_w]$. We have also run tests generating the execution time values in the interval $[\min_w, \max_w]$ with geometric steps without noticing unpredictable differences with respect to the results reported in this section.

Unless otherwise specified, the frequency distribution is Zipf-1.0 and the stream parameters are set to $n_w = 64$, $\min_w = 1$ ms and $\max_w = 64$ ms, this means that the execution times are picked in the set $\{1, 2, \cdots, 64\}$. If not stated otherwise, the operator instances are uniform (*i.e.,* a tuple has the same execution time on any instance) and there are $k = 5$ instances. Let $\overline{W}$ be the average execution time of the stream tuples, then the stream maximum theoretical input throughput sustainable by the setup is equal to $k/\overline{W}$. When fed with an input throughput smaller than $k/\overline{W}$ the system is over-provisioned (*i.e.,* possible underutilization of computing resources). Conversely, an input throughput larger than $k/\overline{W}$ results in an undersized system. We refer to the ratio between the maximum theoretical input throughput and the actual input throughput as the percentage of over-provisioning that, unless otherwise stated, was set to 100%.

In order to generate 100 different streams, we randomize the association between the n_w execution time values and the n distinct items: for each of the n_w execution time values we pick *uniformly* at random n/n_w different values in $[n]$ associated to that execution time value. This means that the 100 different streams we use in our tests do not share the same association between execution time and item as well as the association between frequency and execution time (thus each stream has also a different average execution time $\overline{W}$). We have also build these associations using other distributions, namely geometric and binomial, without noticing unpredictable differences with respect to the results reported in this section. Finally, we have run each stream using 50 different seeds for the hash function generation, yielding a total of 5,000 executions.

Real datasets — For the two use case we provide in this experimental evaluation, we use two different datasets: MENTION and REACH. The former (MENTION) is a dataset containing a stream of preprocessed tweets related to Italian politicians crawled during the 2014 European elections. Among other information, the tweets are enriched with a field *mention* containing the *entities* (*i.e.,* Twitter users) mentioned in the tweet. We

consider the first 500,000 tweets, mentioning roughly $n = 35,000$ distinct entities and where the most frequent entity ("Beppe Grillo") has an empirical probability of occurrence equal to 0.065.
The second dataset (REACH) is generated using the LDBC Social Network Benchmark [65]. Using the default parameters of this benchmark, we obtained a followers graph of 9,960 nodes and 183,005 edges (the maximum out degree was 734) as well as a stream of 2,114,269 tweets where the most frequent author has an empirical probability of occurrence equal to 0.0038.

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

- the average per tuple completion time $\overline{L}^{alg}$ (simply *average completion time* in the following), where alg is the algorithm used for scheduling.
- the average per tuple completion time speed up Λ_L^{alg} (simply *speed up* in the following) achieved by OSG or FK_{OSG} with respect to Round-Robin.
- the throughput of the system expressed as tuples processed per second.

Recall that $\ell^{alg}(t_j)$ is the completion time of the j-th tuple of the stream when using the scheduling algorithm alg. To take into account any overhead introduced by the tested algorithm, we redefine the completion time $\ell^{alg}(t_j)$ as the time it takes from the injection of the j-th tuple at the source until it has been processed by the operator. Then we can define the average completion time $\overline{L}^{alg}$ and speed up Λ_L^{alg} as follows:

$$\overline{L}^{alg} = \frac{\sum_{j=1}^{m} \ell^{alg}(t_j)}{m} \text{ and } \Lambda_L^{alg} = \frac{\sum_{j=1}^{m} \ell^{\text{Round-Robin}}(t_j)}{\sum_{t=1}^{m} \ell^{\text{alg}}(t_j)}$$

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

Simulation Results

Here we report the results obtained by simulating the behavior of the considered algorithm on an ad-hoc multi-threaded simulator. The simulator streams the input dataset through the scheduler that enqueues it on the available target instances. Each target instance dequeues as soon as possible the first element in its queue and simulates processing through busy-waiting for the corresponding execution time. With this implementation the simulated processing times are characterized by only a slight variance mainly due to the precision in estimating the busy-waiting time, which depends on the operating system clock precision and scheduling.

Frequency probability distribution — Figure 7.20 shows the average completion time $\overline{L}^{alg}$ for OSG, Round-Robin and FK_{OSG} with different frequency probability distributions. Increasing the skewness of the distribution reduces the number of distinct tuples that, with high probability, will be fed for scheduling, thus simplifying the scheduling process. This is why all algorithms perform better with highly skewed distributions. On the other hand, uniform or lightly skewed (*i.e.,* Zipf-0.5) distributions seem to be worst cases, in particular for OSG and Round-Robin. With all distributions the Full Knowledge algorithm outperforms OSG which, in turn, always provide better performance than Round-Robin. However, for uniform or lightly skewed distributions (*i.e.,* Zipf-0.5), the gain introduced by OSG is limited (in average 4%). Starting with Zipf-1.0 the gain

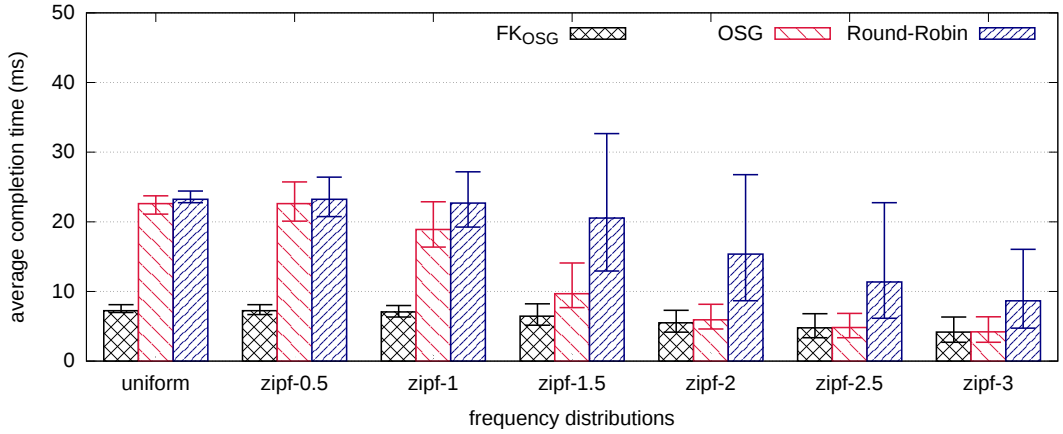


Figure 7.20 – Average per tuple completion time L^{alg} with different frequency probability distributions.

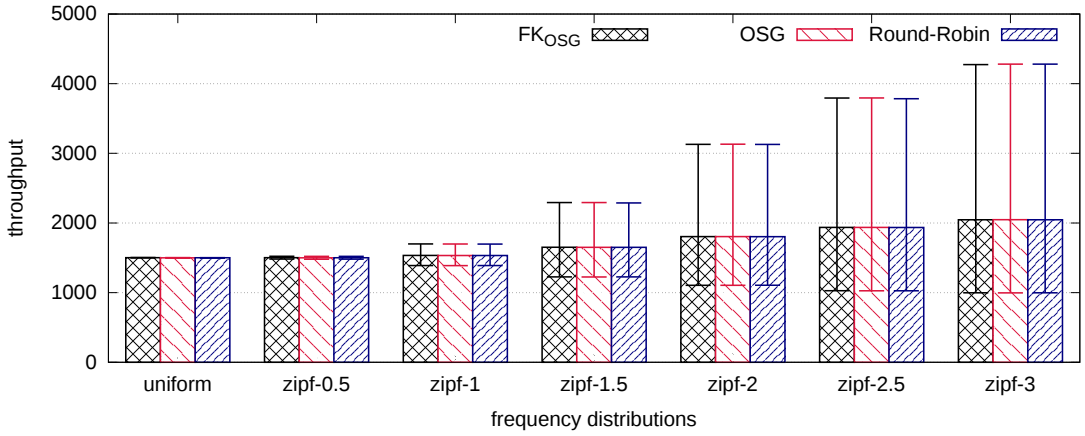


Figure 7.21 – Throughput with different frequency probability distributions.

is much more sizeable (20%) and with Zipf-1.5 we have that the maximum average completion time of OSG is almost smaller than the minimum average completion time of Round-Robin. Finally, with Zipf-2, OSG matches the performance of FK_{OSG}. This behavior for OSG stems from the ability of its sketch data structures (VALES, see Section 3.3) to capture more useful information for skewed input distributions.

Figure 7.21 shows the throughput for the same configurations. Clearly, all the algorithm achieve the same throughput. Each scheduling achieves different average completion times $\overline{L}^{alg}$ since they affect the queuing time experienced by the tuples. However the throughput results shows that for all three algorithm the total load imposed on each operator instance is balanced. Since this is a quite consistent behavior for all simulation, we omitted the remaining throughput plots.

Input throughput — Figure 7.22 shows the speed up Λ_L^{OSG} as a function of the percentage of over-provisioning. When the system is strongly undersized (95% to 98%), queuing delays increase sharply, reducing the advantages offered by OSG. Conversely, when the system is oversized (109% to 115%), queuing delays tend to 0, which in turns also reduces the improvement brought by our algorithm. However, in a correctly sized

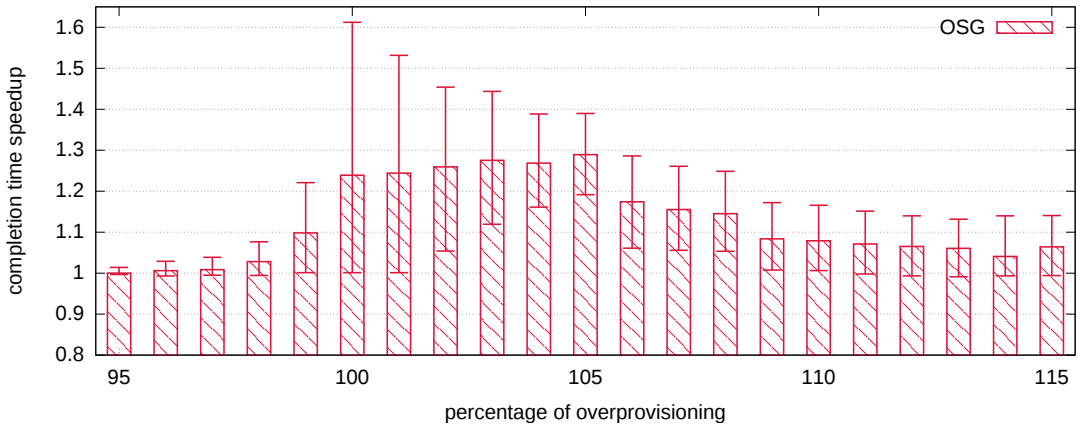


Figure 7.22 – Speed up Λ_L^{OSG} as a function of the percentage of over-provisioning.

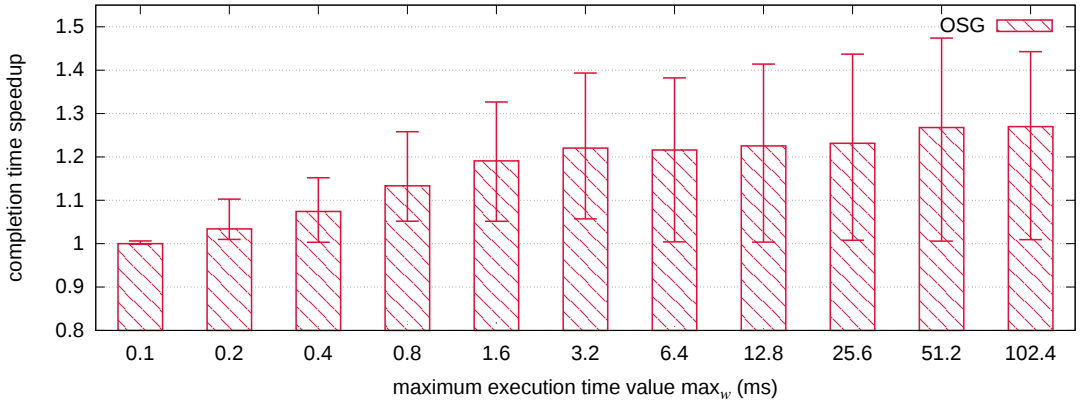


Figure 7.23 – Speed up Λ_L^{OSG} as a function of the maximum execution time value $\max_w$.

system (*i.e.*, from 100% to 108%), our algorithm introduces a noticeable speed up Λ_L^{OSG}, in average at least 1.14 with a peak of 1.29 at 105%. Finally, even when the system is largely oversized (115%), we still provide an average speed up of 1.06. We omitted FK_{OSG} to improve the readability of the plot. The general trend is the same of OSG, achieving the same speed up when the system is undersized. When the systems is oversized, FK_{OSG} hits a peak of 3.6 at 101% and provides a speed up of 1.57 at 115%. In all configurations, all algorithm achieve roughly the same output throughput. In particular it is maximum when the system is undersized (95% to 99%), and decreases accordingly with the percentage of over-provisioning when the system is oversized (100% to 115%).

Maximum execution time value $\max_w$ — Figure 7.23 shows the speed up Λ_L^{OSG} as a function of the maximum execution time $\max_w$. With $\max_w = 0.1$ms, all the tuples have the same execution time $w(t) = 0.1$ms, thus all algorithm achieve the same result. Increasing the value of $\max_w$ increments the gap between the 64 possible execution times, allowing more room to improve the scheduling, then OSG speed up Λ_L^{OSG} grows for $\max_w \geq 0.2$. However notice that OSG seems to hit an asymptote at 1.25 for $\max_w \geq 102.4$. We omitted FK_{OSG} to improve the readability of the plot. The general trend is the same of OSG, however starting with $\max_w \geq 0.4$ FK_{OSG} achieves a larger

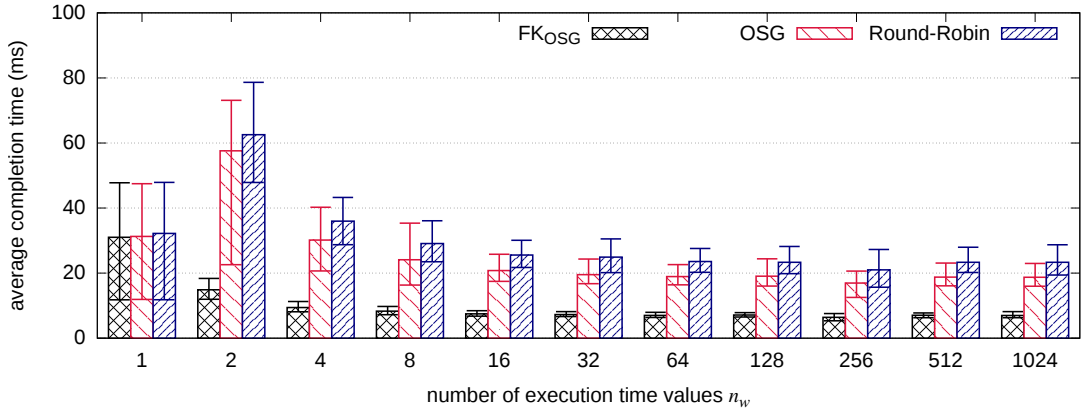


Figure 7.24 – Average per tuple completion time $\overline{L}^{alg}$ as a function of the number of execution time values n_w.

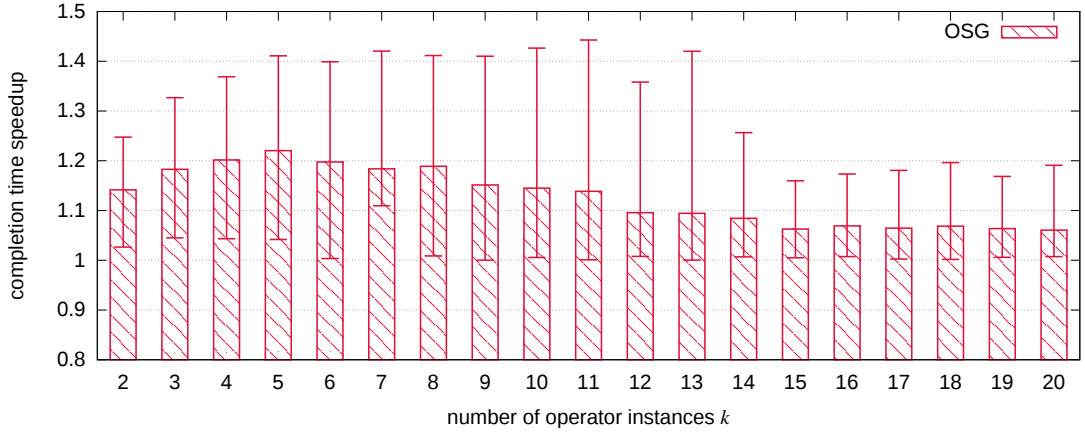


Figure 7.25 – Speed up Λ_L^{OSG} as a function of the number of operator instances k.

speed up and hits an asymptote at 3.4 for $\max_w \geq 102.4$.

Number of execution time values n_w — Figure 7.24 shows the average completion time $\overline{L}^{alg}$ for OSG, Round-Robin and FK_{OSG} as a function of the number of execution time values n_w. We can notice that for growing values of n_w both the average completion time values and variance decrease, with only slight changes for $n_w \geq 16$. Recall that n_w is the number of completion time values in the interval $[\min_w, \max_w]$ that we assign to the n distinct attribute values. For instance, with $n_w = 2$, all the tuples have a completion time equal to either 0.1 or 6.4 ms. Then, assigning either of the two values to the most frequent item strongly affects the average completion time. Increasing n_w reduces the impact that each single execution time has on the average completion time, leading to more stable results. The gain between the maximum, mean and maximum average completion times of OSG and Round-Robin (in average 19%) is mostly unaffected by the value of n_w.

Number of operator instances k — Figure 7.25 shows the speed up Λ_L^{OSG} as a function of the number of parallel operator instances k. From $k = 2$ to $k = 5$ the

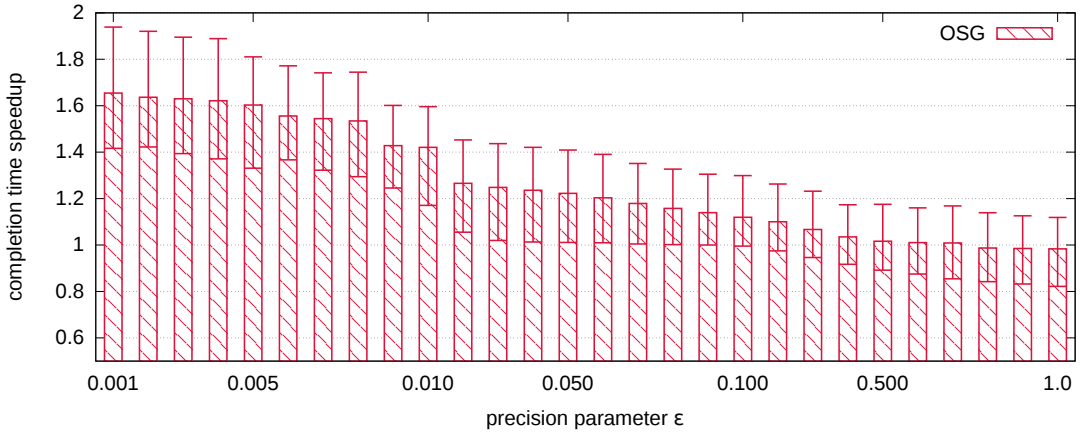


Figure 7.26 – Speed up Λ_L^{OSG} as a function of the precision parameter ε (*i.e.,* number of columns $c = e/\varepsilon$).

speed up Λ_L^{OSG} grows, starting with an average value of 1.14 and reaching an average peak of 1.23. Then the speed up Λ_L^{OSG} decreases to 1.09 ($k = 12$) and reaches what seems to be an asymptote at 1.06 ($k = 20$). In other words, for moderate values of k (*i.e.,* $k \leq 12$), OSG introduces a sizeable improvement in the average completion latency with respect to Round-Robin. On the other hand, for large value of k (*i.e.,* $k > 12$), the impact of OSG is mitigated. As the number of available instances increases, Round-Robin is able to better balance the load, thus limiting OSG effectiveness. We omitted FK_{OSG} to improve the readability of the plot. The general trend is the same of OSG, hitting a peak of 4.0 at $k = 11$ and the decreasing toward an asymptote at 2.7 for $k = 20$.

Precision parameter ε — Figure 7.26 shows the speed up Λ_L^{OSG} as a function of the precision parameter ε value that controls the number of columns in the $\mathfrak{F}$ and $\mathfrak{W}$ matrices. With smaller values of ε, OSG is more precise but also uses more memory, *i.e.,* for $\varepsilon = 1.0$ there is a single entry per row, while for $\varepsilon = 0.001$ there are 2781 entries per row. As expected, decreasing ε improves OSG performance: in average a 10 time decrease in ε (thus a 10 time increase in memory) results in a 25% increase in the speed up. Large values of ε do not provide good performance; however, starting with $\varepsilon \leq 0.09$ the minimum average completion time speed up is always larger than 1.

Time Series — Figure 7.27 shows the completion time as the stream unfolds (the x axis is the number of tuples read from the stream) for both OSG and Round-Robin for a single execution. Each point on the plot is the maximum, mean and minimum completion time over the previous $2,000$ tuples. The plot for Round-Robin has been artificially shifted by $1,000$ tuples to improve readability. In this test the stream is of size $m = 150,000$ split into two periods: the tuple execution times for operator instances 1, 2, 3, 4 and 5 are multiplied by 1.05, 1.025, 1.0, 0.975 and 0.95 respectively for the first $75,000$ tuples, and for the remaining $75,000$ tuples by 0.90, 0.95, 1.0, 1.05 and 1.10 respectively. This setup mimics an abrupt change in the load characteristic of target operator instances (possibly due to exogenous factors).

With the prototype we could not achieve the same measurement precision as with the simulator. Then, to be able to compare the simulator and prototype time series, we increased by a factor of 10 the execution times.

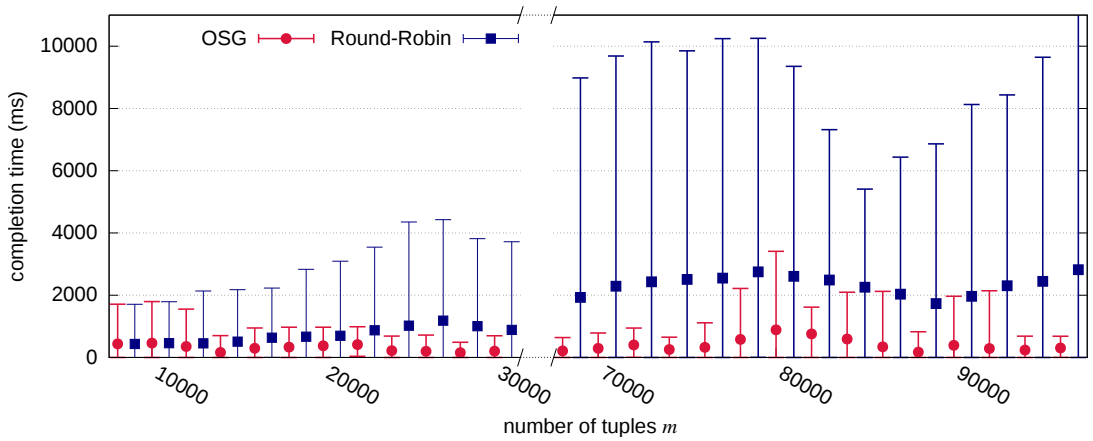


Figure 7.27 – Simulator per tuple completion time time-series.

In the leftmost part of the plot, we can see OSG and Round-Robin provide the same exact results up to $m = 10,690$, where OSG starts to diverge by decreasing the completion time as well as the completion time variance. This behaviour is the result of OSG moving into the RUN state at $m = 10,690$. After this point it starts to schedule using the $\mathfrak{F}$ and $\mathfrak{W}$ matrices and the GOMPS algorithm, improving its performance with respect to Round-Robin, also reducing the completion time variance.

At $m = 75,000$ we inject the load characteristic change described above. Immediately OSG performance degrades as the content of $\mathfrak{F}$ and $\mathfrak{W}$ matrices is outdated. At $m = 84,912$ the scheduler receives the updated $\mathfrak{F}$ and $\mathfrak{W}$ matrices and recovers. This demonstrates OSG ability to adapt at runtime with respect to changes in the load distributions.

Prototype Results

To evaluate the impact of OSG on real applications we implemented it as a custom grouping function within the Apache Storm [89] 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. This choice helped us ruling out from tests possible side-effects caused by co-sharing of CPU cores among processes that are not part of the Storm framework. The prototype was first tested with the synthetic dataset and application, as for the previous simulations, and then on two realistic stream processing applications.

Time Series — In this first test the topology was made of a source (*spout*) and operators (*bolts*) S and O. The source generates (reads) the synthetic (real) input stream. Bolt S uses either OSG or the Apache Storm standard shuffle grouping implementation (ASSG) to route the tuples toward the k instances (*tasks*) of bolt O.

Figure 7.28 provides results for the prototype with the same settings of the test whose results were reported in Figure 7.27. We can notice the same general behavior both in the simulator and in the prototype. In the right part of the plot OSG diverges from ASSG at $m = 20,978$ and decreases the completion time as well as the completion time variance. On the right part of the plot, after $m = 75,000$ OSG performance degrade due to the change in the load distributions. Finally, at $m = 82,311$ the scheduler receives the

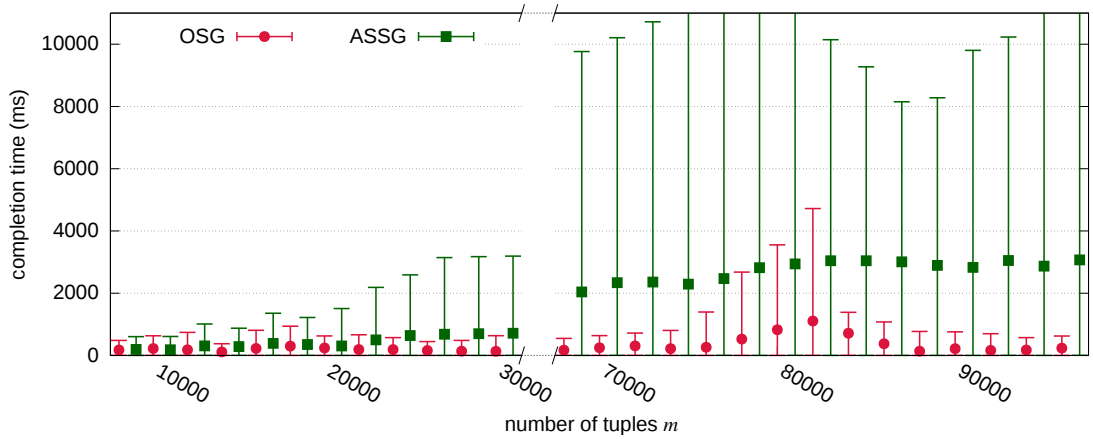


Figure 7.28 – Prototype per tuple completion time time-series.

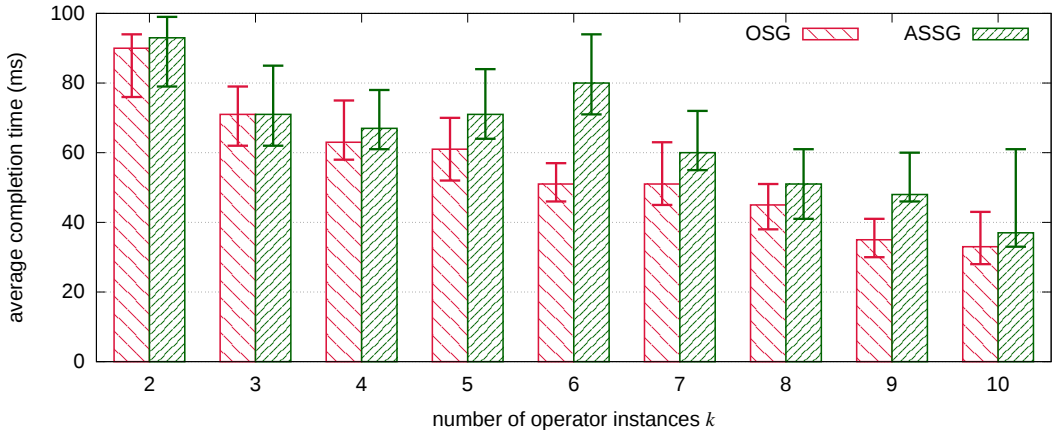


Figure 7.29 – Prototype average per tuple completion time $\overline{L}^{alg}$ as a function of the number of operators k in use-case mentions.

updated $\mathfrak{F}$ and $\mathfrak{W}$ matrices and starts to recover. Notice also that $1,600$ tuples timed out (and where not recovered as the topology was configured disabling Storm fault tolerance mechanisms) during the execution with ASSG. This clearly shows how the shuffle grouping scheduling policy can have a large impact on the system performances.

Use-case mentions — In this test we run a simple application using the MENTION dataset as input: for each tweet of the stream we want to extract the mentions it contains and accordingly decorate the outgoing tuple with additional information on them. In particular, for each mention carried by a tweet the bolt performs a sets of queries (based on the mention itself) on an external database. The larger is the number of mentions contained in the tweet, the more processing time it takes for the bolt to complete the queries and decorate the outgoing tuple. The mention execution times belong to the interval $[0.1, 23]$ ms, each mention adds in average 1 ms to the processing time to execute the corresponding query. The number of mentions is only limited by the number of users registered on Twitter and summed up to a total of $35,000$ unique identities in the MENTION dataset. Globally, the tuple execution times belong to the interval $[1.8, 36]$ ms, while the average per tuple execution time is 7 ms.

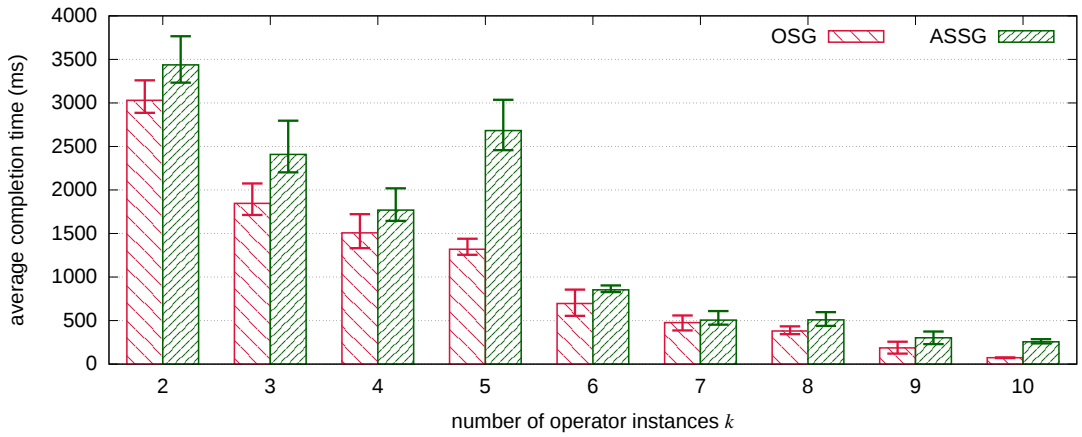


Figure 7.30 – Prototype average per tuple completion time $\overline{L}^{alg}$ as a function of the number of operators k in use-case reach.

Figure 7.29 shows the mean, maximum and minimum average completion time L^{alg} for both OSG and ASSG as a function of the number of instances k over 10 executions. For all values of k, OSG provides lower or equal average completion times than ASSG, with a mean, minimum and maximum speed up Λ_L^{OSG} of 1.27, 1.0 and 2.16. For $k = 5$ and $k = 6$, we can notice an unanticipated behavior of ASSG: adding one more instance increases the completion times. On the other hand, OSG average completion time always decreases with growing values of k. Notice also that, to provide this improvement, OSG exchanged only a few hundred additional messages against a stream of size $m = 500,000$.

Use-case reach — In this test we want to compute the *reach* of twitted terms using the REACH dataset as input. The reach of a term is the total number of estimated unique Twitter users to which were delivered tweets about the search term. Usually, this metric is calculated through a periodic batch process using the followers graph, where edges are enriched with re-tweet probabilities. We propose instead to compute this value in a streaming fashion, for each tweet, restricting the computation to a depth of 3 in the followers graph of 9,960 nodes. Globally, the tuple execution times belong to the interval $[0.01, 70]$ ms, the most frequent tuple execution time is in average 65 ms, while the average per tuple execution time is 20 ms.

Figure 7.30 shows the mean, maximum and minimum average completion time $\overline{L}^{alg}$ for both OSG and ASSG as a function of the number of instances k over 10 executions. Except for the unanticipated spike of ASSG for $k = 5$, the completion latency decreases as k increases. For all k, OSG has a smaller mean average completion latency than ASSG. In addition, for most values of k, the maximum average completion latency of OSG is smaller or equal to the minimum average completion latency of ASSG. Finally, the average speed up Λ_L^{OSG} of OSG with respect to ASSG is at least 1.05, at most 3.4 and in average 1.5. To achieve these results, OSG exchanges only a few thousand additional messages, against a stream size of $m = 2,114,269$.

7.3 Parallelized load balancers

For the sake of clarity, so far we have considered a topology with a single operator S. To handle stages where operator S is itself parallelized with s instances $S_1, \ldots, S_s$, we have to extends our algorithm as follows:

Distribution-aware Key Grouping Each instance of S should run a modified version of DKG and collaborate to build the global mapping. More in details, the `Space Saving` instance should be replaced by an algorithm solving the distributed heavy hitters (*e.g.,* DHHE presented in Section 4.3) in the s sub-streams σ_{S_i} ($i \in [s]$). We also need to introduce a coordinator that, at the end of the learning phase, retrieves the frequency estimation of the distributed heavy hitters and of the buckets of sparse items. The coordinator is then able to build a close to optimal mapping through the GMPS algorithm, as in the non-parallelized version.

Online Shuffle Grouping Scheduler Parallelizing OSG is less straightforward: decisions are taken continuously and on-line, thus the decision process cannot be moved on a coordinator as for DKG. The solution we propose is to run independent instances of OSG on each of the s instances of operator S, *i.e.,* as Round-Robin is parallelized. Ideally, since each instance of OSG strives to spread evenly its outbound load, the overall load induced on each instance of operator O should be even. Alas, it is easy to see that in the worst case GOMPS (and thus OSG) performance degrades linearly with the parallelization degree. Indeed, in absence of any additional communication between the parallel schedulers, each of them cannot do better than providing a schedule in isolation with respect to the other schedulers. On the other hand, the same degradation applies to Round-Robin scheduling, consequently our approach still provides better results and can handle a moderate parallelization degree.

Notice that both load balancing solutions work on a hop-by-hop basis. Under the realistic assumption that there is no cross stage dependency among tasks with respect to scheduling, we can apply DKG and OSG on any key-grouped and, respectively, shuffle grouped stage. Through these two algorithms, we can then load balance any sequence of key- and shuffle-grouped stage of the topology.

Chapter Summary

This chapter has introduced the design of DKG, based on BPART (*cf.,* Section 4.4), and of OSG, based on VALES (*cf.,* Section 3.3). Both DKG and OSG have been extensively tested with both simulation and a running prototype, confirming the theoretical analysis of the algorithms. In particular, both yielded a sizeable improvement with respect to previous solutions in load balancing key- and shuffle-grouped parallelized operators in stream processing systems. Finally, this chapter also introduced the notion of non-uniform tuple execution time.

The next chapter presents the design of a load shedding algorithm based on VALES. Similarly to OSG, this algorithm exploit the non-uniformity of the tuple execution time.

Chapter 8

Load Shedding in Stream Processing Systems

In this chapter we tackle the load shedding problem taking into account that, as argued previously (*cf.,* Section 7.2), the execution time of the tuple may not be uniform for many practical use cases. 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. Load-Aware Load Shedding (LAS) is a novel solution for load shedding in SPS that gets rid of the aforementioned assumptions and provides efficient shedding aimed at matching given queuing time targets, while dropping as few tuples as possible. To reach this goal LAS leverages the VALES algorithm (*cf.,* Section 3.3) 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 most previous work (*cf.,* Section 6.1), the solution takes into account a priority model to decide which tuples can be dropped. However, we believe that the keeping the system performances stable and minimising the error in the topology output are orthogonal problems.

Problem Statement With respect to the system model presented in Section 6.2, we introduce the set of dropped tuples $\mathcal{D} \subseteq [m]$ in a stream of length m, *i.e.,* dropped tuples are thus represented in $\mathcal{D}$ by their indices in $[m]$ of the stream σ. Moreover, let $d \leq m$ be the number of dropped tuples in a stream of length m, *i.e.,* $d = |\mathcal{D}|$. We also set the queuing time for dropped tuple to 0, *i.e.,* $\forall j \in \mathcal{D}, q(t_j) = 0$. Then we can define the average queuing time as: $\overline{Q}(\imath) = \frac{1}{\imath - d} \sum_{j \in [\imath] \setminus \mathcal{D}} q(t_j)$ for all $\imath \in [m]$.

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

Problem 8.1 (Load Shedding)**.** *Given a data stream $\sigma = \langle t_1, \ldots, t_m \rangle$, find the smallest set $\mathcal{D}$ such that*

$$\forall \imath \in [m], \overline{Q}(\imath) = \frac{1}{\imath - d} \sum_{j \in [\imath] \setminus \mathcal{D}} q(t_j) \leq \overline{Q}^{\max}.$$

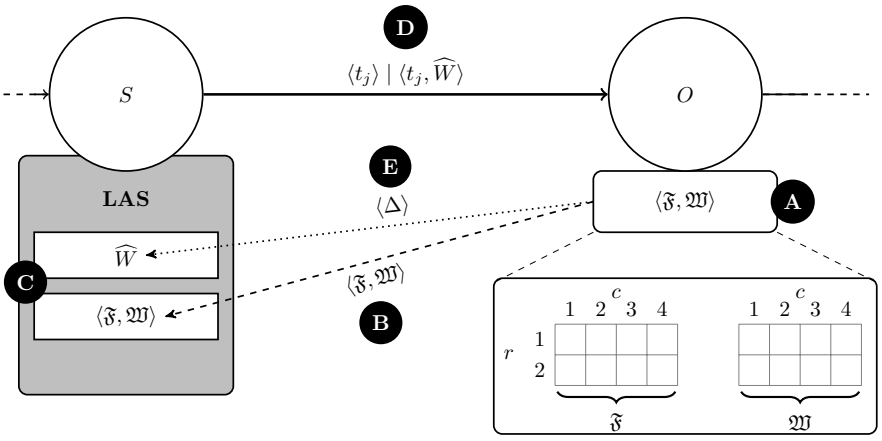


Figure 8.1 – LAS design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$).

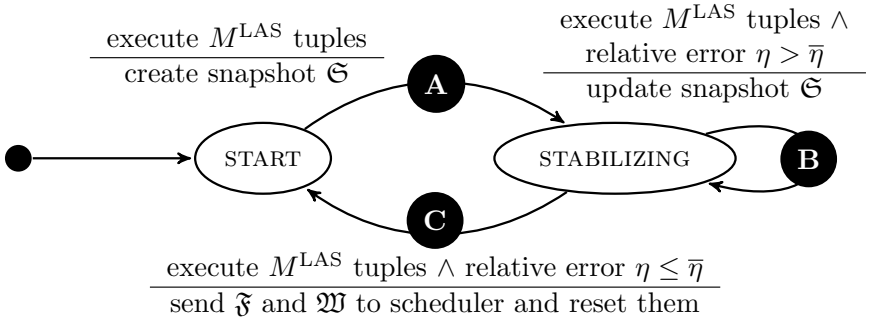


Figure 8.2 – LAS operator finite state machine.

8.1 Load-Aware Load Shedding Algorithm

Similarly to OSG (*cf.,* Section 7.2.1), Load-Aware Load Shedding (LAS) is based on a simple, yet effective, idea: if we assume to know the execution time $w(t)$ 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 time threshold $\overline{Q}^{\max}$ to be violated. However, the value of $w(t)$ is generally unknown.

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

To enable this approach, LAS estimates the tuple execution time extending VALES (*cf.,* Section 3.3.1) as shown in Section 7.2.1).

The operator maintains two matrices (Figure 8.1.A): the first one, denoted as $\mathfrak{F}$, tracks the tuple frequencies f_t, the second one, denoted as $\mathfrak{W}$, tracks the tuples cumulated execution times $w(t) \times f_t$. Both matrices share the same size $r \times c$, where $r = log\frac{1}{\delta}$ and $c = \frac{e}{\varepsilon}$, and hash functions.

Listing 8.3 – LAS algorithm on operator O

```
1: init (r, c, M^LAS) do
2:     𝔉, 𝔚, 𝔖 ← 0_{r,c}
3:     r hash functions h_1, ..., h_r : [n] → [c] from a 2-universal family.
4:     m ← 0
5:     state ← START
6: end init
7: function UPDATE(t_j, w(t_j), request : Ŵ)
8:     m ← m + 1
9:     if Ŵ not null then
10:        Δ ← W − Ŵ
11:        send ⟨Δ⟩ to S
12:    end if
13:    if state = START ∧ m mod M^LAS = 0 then                ▷ Figure 8.2.A
14:        update 𝔖
15:        state ← STABILIZING
16:    else if state = STABILIZING ∧ m mod M^LAS = 0 then
17:        if η ≤ η̄ (Eq. 8.1) then                            ▷ Figure 8.2.C
18:            send ⟨𝔉, 𝔚⟩ to S
19:            state ← START
20:            reset 𝔉 and 𝔚 to 0_{r,c}
21:        else                                               ▷ Figure 8.2.B
22:            update 𝔖
23:        end if
24:    end if
25:    for ι = 1 to r do
26:        𝔉[ι, h_ι(t)] ← 𝔉[ι, h_ι(t)] + 1
27:        𝔚[ι, h_ι(t)] ← 𝔚[ι, h_ι(t)] + w(t_j)
28:    end for
29: end function
```

The operator is modeled as a finite state machine (Figure 8.2) with two states: START and STABILIZING. The START state lasts as long as the operator has executed M^{LAS} tuples, where M^{LAS} is a user defined window size parameter. The transition to the STABILIZING state (Figure 8.2.A) triggers the creation of a new snapshot $\mathfrak{S}$. A snapshot is a matrix of size $r \times c$ where $\forall \imath \in [r], \jmath \in [c] : \mathfrak{S}[\imath, \jmath] = \mathfrak{W}[\imath, \jmath]/\mathfrak{F}[\imath, \jmath]$ (Listing 8.3, Lines 13–16). We say that the $\mathfrak{F}$ and $\mathfrak{W}$ matrices are stable when the relative error η between the previous snapshot and the current one is smaller than a configurable parameter $\overline{\eta}$, *i.e.*,

$$\eta = \frac{\sum_{\forall \imath,\jmath} |\mathfrak{S}[\imath, \jmath] - \frac{\mathfrak{W}[\imath,\jmath]}{\mathfrak{F}[\imath,\jmath])}|}{\sum_{\forall \imath,\jmath} \mathfrak{S}[\imath, \jmath]} \leq \overline{\eta} \tag{8.1}$$

is satisfied. Then, each time the operator has executed M^{LAS} tuples (Listing 8.3, Lines 16–24), it checks whether Equation 8.1 is satisfied. **(i)** In the negative case $\mathfrak{S}$ is updated (Figure 8.2.B). **(ii)** In the positive case the operator sends the $\mathfrak{F}$ and $\mathfrak{W}$ matrices to the load shedder (Figure 8.1.B), resets their content and moves back to the START state (Figure 8.2.C).

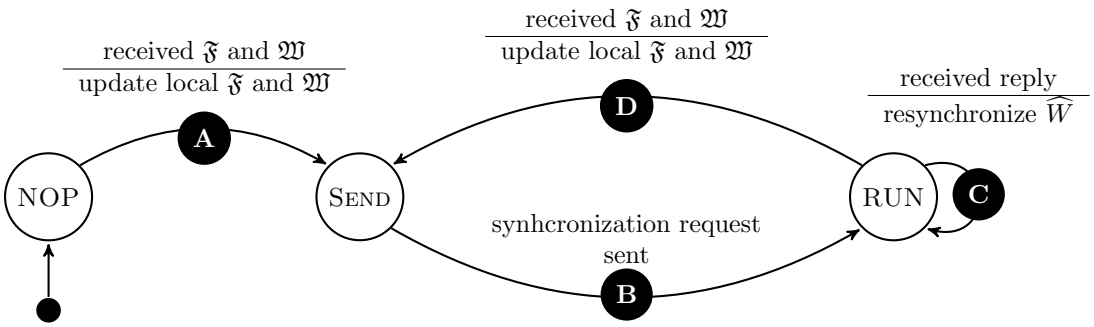


Figure 8.4 – LAS shedder finite state machine.

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

The load shedder (Figure 8.1.C) maintains the estimated cumulated execution time of the operator $\widehat{W}$ and a pairs of initially empty matrices $\langle \mathfrak{F}, \mathfrak{W} \rangle$. The loadshedder is modeled as a finite state machine (Figure 8.4) with three states: NOP, SEND and RUN. The load shedder executes the code reported in Listing 8.5. In particular, every time a new tuple t arrives at the load shedder, the function SHED is executed. The load shedder starts in the NOP state where no action is performed (Listing 8.5, 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 the load shedder receives the first pair $\langle \mathfrak{F}, \mathfrak{W} \rangle$ of matrices (Figure 8.4.A), it moves into the SEND state and updates its local pair of matrices (Listing 8.5, Lines 7–10). While being in the SEND state, the load shedder sends to O the current cumulated execution time estimation $\widehat{W}$ (Figure 8.1.D) piggy backing it with the first tuple t that is not dropped (Listing 8.5, Lines 24–27) and moves in the RUN state (Figure 8.4.B). This information is used to synchronize the load shedder with operator O and remove the skew between O's cumulated execution time W and the estimation $\widehat{W}$ at the load shedder. Operator O replies to this request (Figure 8.1.E) with the difference $\Delta = W - \widehat{W}$ (Listing 8.3, Lines 9–12). When the load shedder receives the synchronization reply (Figure 8.4.C) it updates its estimation $\widehat{W} + \Delta$ (Listing 8.5, Lines 11–13).

In the RUN state, the load shedder computes, for each tuple t, the estimated queuing time $\hat{q}(i)$ as the difference between the operator estimated execution time $\widehat{W}$ and the time elapsed from the emission of the first tuple (Listing 8.5, Line 18). It then verifies if the estimated queuing time for t satisfies the CHECK method (Listing 8.5, Lines 19–21).

This method encapsulates the logic to decide if a desired condition on queuing latencies is violated or not. As stated in Problem 8.1, we aim at maintaining the average queuing time below a threshold $\overline{Q}^{\max}$. Then, CHECK tries to add $\hat{q}$ to the current average queuing time (Listing 8.5, Lines 31). If the result is larger than $\overline{Q}^{\max}$ **(i)**, it simply returns *true*; otherwise **(ii)**, it updates its local value for the average queuing time and returns *false* (Listing 8.5, Lines 34–36). Note that different goals, based on the queuing time, can be defined and encapsulated within CHECK, *e.g.,* maintain the final average queuing time below $\overline{Q}^{\max}$, or maintain the average queuing time calculated on a sliding window below $\overline{Q}^{\max}$.

Listing 8.5 – LAS shedder on operator S

```
1: init (r, c, Q̄^max) do
2:     Ŵ ← 0
3:     ⟨𝔉, 𝔚⟩ ← ⟨0_{r,c}, 0_{r,c}⟩
4:     h_1, ..., h_r : [n] → [c], the same r hash functions of the operator instances
5:     state ← NOP
6: end init
7: upon ⟨𝔉′, 𝔚′⟩ do                                   ▷ Figure 8.4.A and 8.4.D
8:     state ← SEND
9:     ⟨𝔉, 𝔚⟩ ← ⟨𝔉′, 𝔚′⟩
10: end upon
11: upon receive ⟨Δ⟩ from operator do                  ▷ Figure 8.4.C
12:    Ŵ ← Ŵ + Δ
13: end upon
14: function SHED(t_j)
15:    if state = NOP then
16:        return false
17:    end if
18:    q̂(t_j) ← Ŵ − elapsed time from first tuple
19:    if CHECK(q̂(t_j)) then
20:        return true
21:    end if
22:    ι ← arg min_{ι∈[r]}{𝔉[ι, h_ι(t)]}
23:    Ŵ ← Ŵ + (𝔚[ι, h_ι(t)]/𝔉[ι, h_ι(t)]) × (1 + ε)
24:    if state = SEND then                          ▷ Figure 8.4.B
25:        piggy back Ŵ to operator on t
26:        state ← RUN
27:    end if
28:    return false
29: end function
30: function CHECK(q̂)
31:    if (Q̂ + q̂)/m > Q̄^max then
32:        return true
33:    end if
34:    Q̂ ← Q̂ + q̂
35:    m ← m + 1
36:    return false
37: end function
```

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 time $\widehat{W}$ is updated with the estimated tuple execution time $\hat{w}(t)$, increased by a factor $1+\varepsilon$ to mitigate potential under-estimations (Listing 8.5, Lines 22–23), and the load shedder returns *false* (Listing 8.5, Line 28), *i.e.,* the tuple must not be dropped. The correction factor derives from the fact that $\widehat{w}(t)$ is a (ε, δ)-approximation of $w(t)$ as shown in Section 3.3.2. Finally, if the load shedder receives a new pair $\langle \mathfrak{F}, \mathfrak{W} \rangle$ of matrices (Figure 8.4.D), it again update its local pair of matrices and move to the SEND state (Listing 8.5, Lines 7–10).

Theorem 8.2 (LAS Time Complexity)**.**
For each tuple read from the input stream, the time complexity of LAS for the operator and the load shedder is $O(\log 1/\delta)$.

Proof. By Listing 8.3, for each tuple read from the input stream, the algorithm increments an entry per row of both the $\mathfrak{F}$ and $\mathfrak{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting update time complexity is $O(\log 1/\delta)$. By Listing 8.5, for each submitted tuple, the scheduler has to retrieve the estimated execution time for the submitted tuple. This operation requires to read entry per row of both the $\mathfrak{F}$ and $\mathfrak{W}$ matrices. Since each has $\log 1/\delta$ rows, the resulting query time complexity is $O(\log 1/\delta)$. □

Theorem 8.3 (LAS Space Complexity)**.**
The space complexity of LAS for the operator and load shedder is $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$ *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 $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$ 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 $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$ bits. □

Theorem 8.4 (LAS Communication Complexity)**.** *The communication complexity of LAS is of* $O\left(\frac{m}{M}\right)$ *messages and* $O\left(\frac{m}{M^{LAS}}\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$ *bits.*

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

Parallel load shedders For the sake of clarity, so far we have considered a topology with a single operator S. To handle stages where operator S is itself parallelized with s instances $S_1,\ldots,S_s$, we have to extends LAS as follows.

Similarly to OSG, LAS takes decisions continuously and on-line, thus the decision process cannot be moved on a coordinator. However, while OSG handles the whole outbound sub-stream σ_{S_i}, LAS works on a single sub-stream $\sigma_{S_i\to O_p}$. Then LAS can easily be parallelized running $s\times k$ isolated instances, each handling a single sub-stream from instance S_i to instance O_p.

8.2 Theoretical Analysis

This section provides the analysis of the quality of the shedding performed by LAS, studying the correctness and optimality of the shedding algorithm. We assume that the execution time of each tuple $w(t)$ is known.

We first analyse FK_{LAS}, a variant of LAS where we assume that the processing time $w_p(t)$ is known for each tuple t. We suppose that tuples cannot be preempted, that is they must be processed in an uninterrupted fashion on the available operator instance. Finally, given our system model, we consider the problem of minimizing d, the number of dropped tuples, while guaranteeing that the average queuing latency $\overline{Q}$ is upper-bounded by $\overline{Q}^{\max}$, $\forall t_j\in\sigma$. The solution must work on-line, thus the decision of enqueueing or dropping a tuple has to be made based only on the tuples received so far in the stream.

Let OPT be the online algorithm that provides the optimal solution to Problem 8.1. We denote with $\mathcal{D}_{OPT}$ (respectively d_{OPT}) the set of dropped tuple indices (respectively

the number of dropped tuples) produced by the OPT algorithm fed by stream σ. We also denote with $d_{\mathrm{FK}_{LAS}}$ the number of dropped tuples produced by FK_{LAS} when fed with the same stream σ.

Theorem 8.5 (FK_{LAS} Correctness and Optimality)**.** *For any σ, we have $d_{\mathrm{FK}_{LAS}} = d_{OPT}$ and $\forall t_j \in \sigma, \overline{Q}_{\mathrm{FK}_{LAS}}(j) \leq \overline{Q}^{\max}$.*

Proof. Given a stream σ, consider the sets of indices of tuples dropped by respectively OPT and FK_{LAS}, namely $\mathcal{D}_{OPT}$ and $\mathcal{D}_{\mathrm{FK}_{LAS}}$. Below, we prove by contradiction that $d_{\mathrm{FK}_{LAS}} = d_{OPT}$.

Assume that $d_{\mathrm{FK}_{LAS}} > d_{OPT}$. Without loss of generality, we denote $\imath_1, \ldots, \imath_{d_{\mathrm{FK}_{LAS}}}$ the ordered indices in $\mathcal{D}_{\mathrm{FK}_{LAS}}$, and $\jmath_1, \ldots, \jmath_{d_{OPT}}$ the ordered indices in $\mathcal{D}_{OPT}$. Let us define a as the largest natural integer such that $\forall x \leq a, \imath_x = \jmath_x$ (*i.e.*, $\imath_1 = \jmath_1, \ldots, \imath_a = \jmath_a$). Thus, we have $\imath_{a+1} \neq \jmath_{a+1}$.

- Assume that $\imath_{a+1} < \jmath_{a+1}$. Then, according to Listing 8.5, the $\imath_{a+1}$-th tuple of σ has been dropped by FK_{LAS} as the method CHECK returned *true*. Thus, as $\imath_{a+1} \notin \mathcal{D}_{OPT}$, the OPT run has enqueued this tuple violating the constraint $\overline{Q}^{\max}$. But this is in contradiction with the definition of OPT.
- Assume now that $\imath_{a+1} > \jmath_{a+1}$. The fact that FK_{LAS} does not drop the $\jmath_{a+1}$ tuple means that CHECK returns *false*, thus that tuple does not violate the constraint on $\overline{Q}^{\max}$. However, as OPT is optimal, it may drop some tuples for which CHECK returns *false*, to drop less tuples overall. Therefore, if it drops the $\jmath_{a+1}$ 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 on-line algorithm, and the contradiction follows.

Then, we have that $\imath_{a+1} = \jmath_{a+1}$. By induction, we iterate this reasoning for all the remaining indices from $a+1$ to d_{OPT}. We then obtain that $\mathcal{D}_{OPT} \subseteq \mathcal{D}_{\mathrm{FK}_{LAS}}$.

As by assumption $d_{OPT} < d_{\mathrm{FK}_{LAS}}$, we have that $\exists j \in \mathcal{D}_{\mathrm{FK}_{LAS}} \setminus \mathcal{D}_{OPT}$ such that t_j has been dropped by FK_{LAS}. This means that, with the same tuple index prefix shared by OPT and FK_{LAS}, the method CHECK returned *true* when evaluated on t_j, and OPT would violate the condition on $\overline{Q}^{\max}$ by enqueuing it. That leads to a contradiction. Then, $\mathcal{D}_{\mathrm{FK}_{LAS}} \setminus \mathcal{D}_{OPT} = \emptyset$, and $d_{OPT} = d_{\mathrm{FK}_{LAS}}$.

Furthermore, by construction, FK_{LAS} never enqueues a tuple that violates the condition on $\overline{Q}^{\max}$ because CHECK would return *true*. Consequently, $\forall t_j \in \sigma, \overline{Q}_{\mathrm{FK}_{LAS}}(j) \leq \overline{Q}^{\max}$. □

LAS computes $\overline{Q}$ with the estimated execution times $\widehat{w}(t)$ provided by VALES, then LAS (ε, δ)-approximates FK_{LAS}.

8.3 Experimental Evaluation

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

Setup — We compare LAS performance against three other algorithms:

- The Base Line algorithm takes as input the percentage of under-provisioning and drops at random an equivalent fraction of tuples from the stream;
- The Straw-Man algorithm uses the same shedding strategy of LAS, however it uses the average execution time $\overline{W}$ as the estimated execution time $\hat{w}(t)$ for each tuple t;
- The FK_{LAS} algorithm uses the same shedding strategy of LAS, however it feeds it with the exact execution time $w(t)$ for each tuple t.

The LAS operator window size parameter M^{LAS}, the tolerance parameter μ and the number of rows of the $\mathfrak{F}$ and $\mathfrak{W}$ matrices δ are respectively set to $M^{LAS} = 1,024$, $\mu = 0.05$ and $\delta = 0.1$ (*i.e.,* $r = 4$ rows). By default, the LAS precision parameter (*i.e.,*, the number of columns of the $\mathfrak{F}$ and $\mathfrak{W}$ matrices) is set to $\varepsilon = 0.05$ (*i.e.,* $c = 54$ columns), however in one test we evaluate LAS performance using several values: $\varepsilon \in [0.001, 1.0]$.

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

- ABS($\overline{q}$) requires that the queuing latency for each tuple is at most $\overline{q}$ ms: $\forall j \in [m] \setminus \mathcal{D}, q(t_j) \leq \overline{q}$;
- AVG($\overline{Q}^{\max}$) requires that the total average queuing latency does not exceeds $\overline{Q}^{\max}$ ms: $\forall j \in [m] \setminus \mathcal{D}, \overline{Q}(j) \leq \overline{Q}^{\max}$.

While not being a realistic requirement, the straightforwardness of the ABS($\overline{Q}^{\max}$) 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.

Synthetic traces — Similarly to the evaluation of OSG (*cf.,* Section 7.2.3), synthetic datasets are built as streams of integer values (items) representing the values of the tuple attribute driving the execution time 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 n_w as the number of distinct execution time values that the tuples can have. These n_w values are selected at *constant* distance in the interval $[\min_w, \max_w]$. We run experiments with $n_w \in \{1, 2, \cdots, 64\}$, however, for sake of concision, we only report results for $n_w = 64$, and with $\max_w \in \{0.1, 0.2, \ldots, 51.2\}$ ms. Tests performed with different values for n_w 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 $n_w = 64$, $\min_w = 0.1$ ms and $\max_w = 6.4$ ms; this means that the $n_w = 64$ execution times are picked in the set $\{0.1, 0.2, \cdots, 6.4\}$ ms.

Let $\overline{W}$ be the average execution time 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 is over-provisioned (*i.e.,* possible underutilization of computing resources). Conversely, an input throughput larger than $1/\overline{W}$ results 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, is set to 25%. In order to generate 100 different streams, we randomize the association between the n_w execution time values and the n distinct items: for each of the n_w execution time values we pick *uniformly* at random n/n_w different values in $[n]$ associated to that execution time va-

lue. This means that the 100 different streams we use in our tests do not share the same association between execution time and item as well as the association between frequency and execution time (thus each stream has also a different average execution time $\overline{W}$). 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. Whenever applicable we provide the maximum, mean and minimum figures over the $5,000$ runs.

Real dataset — We used the MENTION dataset (*cf.,* Section 7.2.3). It contains a stream of preprocessed tweets related to the 2014 European elections. Among other information, the tweets are enriched with a field *mention* listing 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 equal to 0.065.

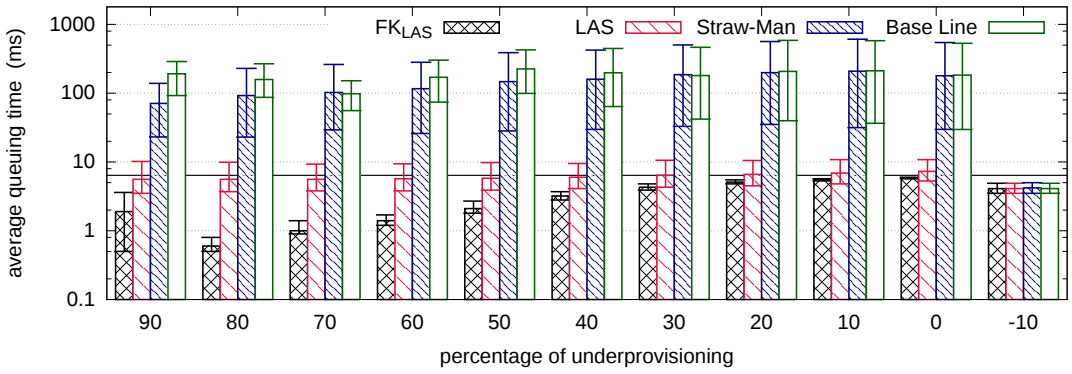
Metrics — The evaluation metrics we provide, when applicable, are:

- the dropped ratio $D = d/m$;
- the ratio of tuples dropped by algorithm alg with respect to Base Line $\Lambda_d = (d^{\text{alg}} - d^{\text{Base Line}})/d^{\text{Base Line}}$, in the following we refer this metric as shedding speed up;
- the average queuing latency $\overline{Q} = \sum_{j \in [m] \setminus \mathcal{D}} q(t_j)/(m-d)$;
- the average completion latency $\overline{L}$, *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.

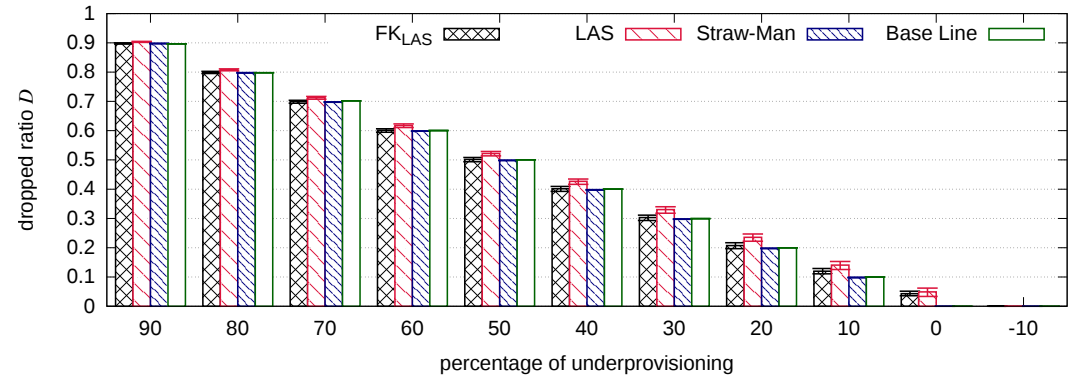
Simulation Results

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

Input Throughput — Figure 8.6 shows the average queuing time $\overline{Q}$ (top) and dropped ratio D (bottom) 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 time. On the other hand, LAS average queuing time is quite close to the required value of $\overline{Q}^{\max} = 6.4$ ms, even if this threshold is violated in some of the tests. Finally, FK_{LAS} always abide to the constraint and is even able to produce a much lower average queuing time while dropping no more tuples that the competing solutions. Comparing the two plots we can clearly see that the resulting average queuing time 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 FK_{LAS} is in the middle. This result shows dropping tuples on the basis of the load they impose allows to design more effective load shedding strategies.



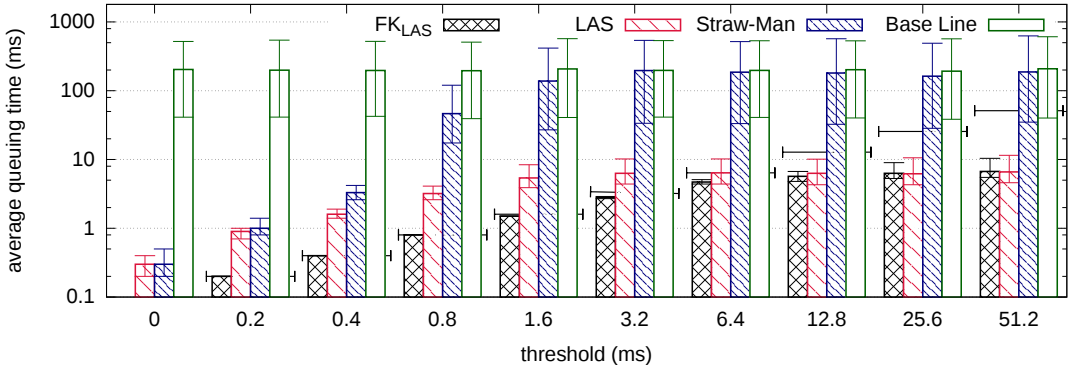
(a) Average queuing time $\overline{Q}$



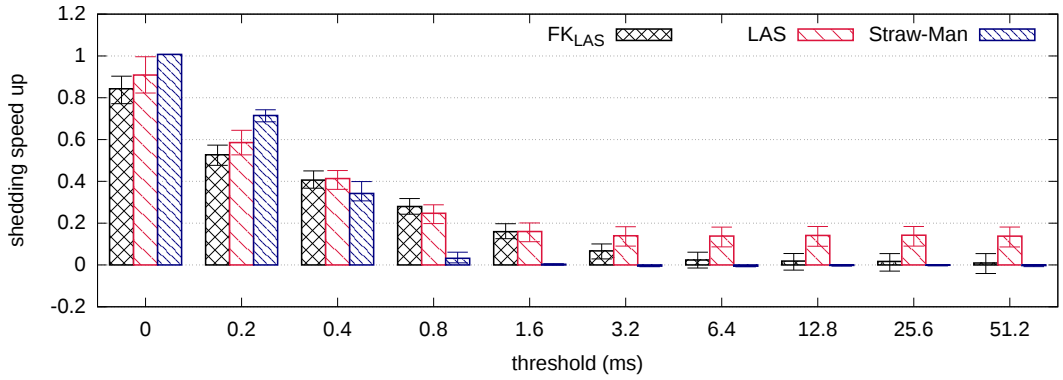
(b) Dropped ratio D

Figure 8.6 – LAS performance varying the amount of underprovisioning.

Threshold $\overline{Q}^{\max}$ — Figure 8.7 shows the average queuing time $\overline{Q}$ (top) and shedding speed up Λ_d (bottom) as a function of the $\overline{Q}^{\max}$ threshold. Notice that with $\overline{Q}^{\max} = 0$, we do not allow any queuing, while with $\overline{Q}^{\max} = 6.4$ we allow at least a queuing time equal to the maximum execution time $\max_w$. In other words, we believe that with $\overline{Q}^{\max} < 6.4$ the constraint is strongly conservative, thus representing a difficult scenario for any load shedding solution. Since Base Line does not take into account the time constraint $\overline{Q}^{\max}$ it always drops the same amount of tuples and achieves a constant average queueing time. For this reason Figure 8.7b reports the shedding speed up Λ_d achieved by FK_{LAS}, LAS and Straw-Man with respect to Base Line. The horizontal segments in Figure 8.7a represent the distinct values for $\overline{Q}^{\max}$. As the graph shows FK_{LAS} always perfectly approaches the queuing time threshold, but for $\overline{Q}^{\max} = 12.8$ 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 8.7b. However, as $\overline{Q}^{\max}$ becomes larger (*i.e.*, $\overline{Q}^{\max} \geq 0.8$) Straw-Man average queuing time 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 time that for large values of $\overline{Q}^{\max}$ approaches the one provided by FK_{LAS}. While delivering these performance LAS drops a slightly larger amount of tuples with respect to FK_{LAS}, to account for the approximation in calculating tuple execution times.

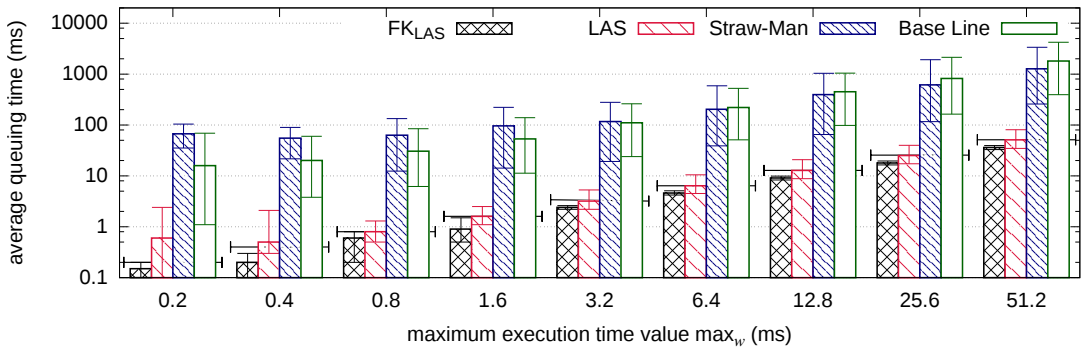


(a) Average queuing time $\overline{Q}$

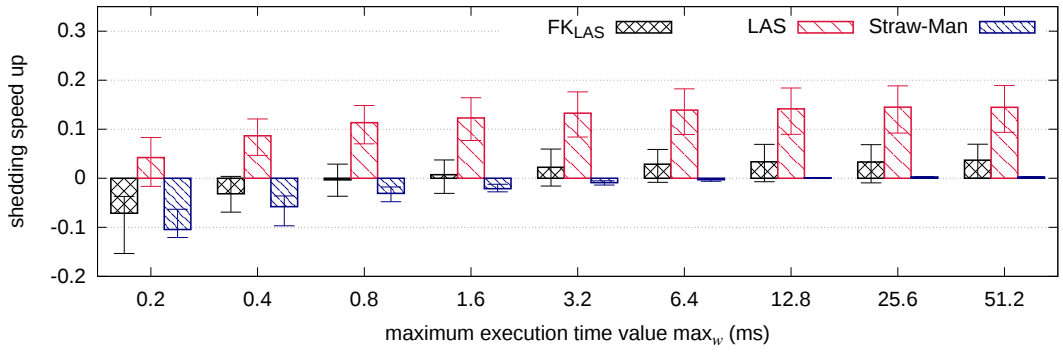


(b) Shedding speed up Λ_d

Figure 8.7 – LAS performance varying the threshold $\overline{Q}^{\max}$.

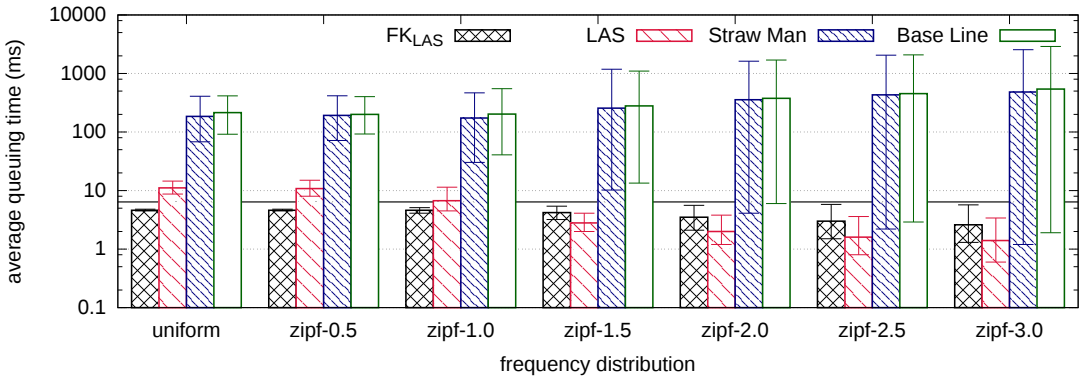


(a) Average queuing time $\overline{Q}$

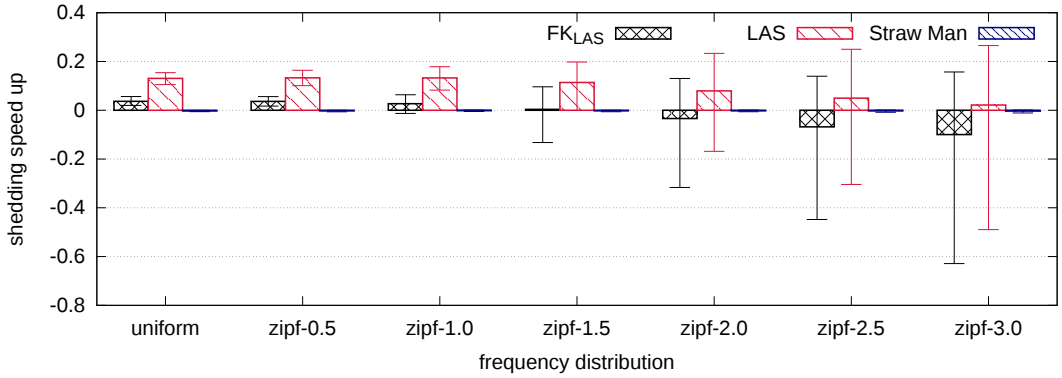


(b) Shedding speed up Λ_d

Figure 8.8 – LAS performance varying the maximum execution time value $\max_w$.



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

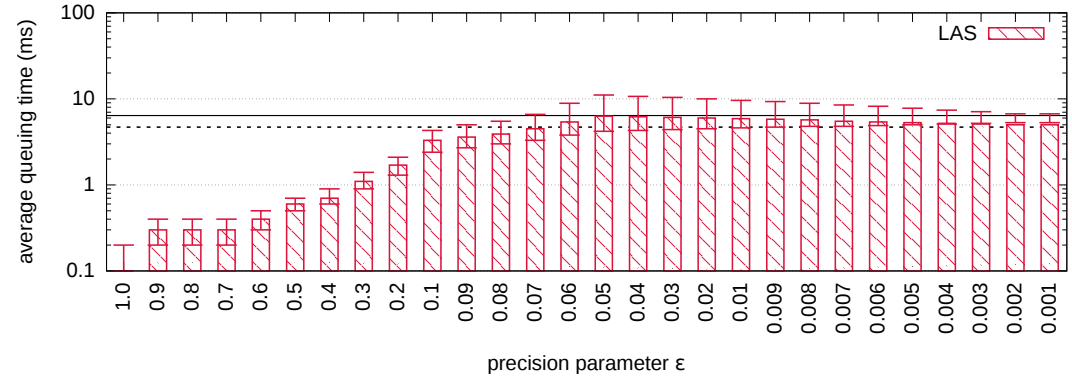


(b) Shedding speed up Λ_d

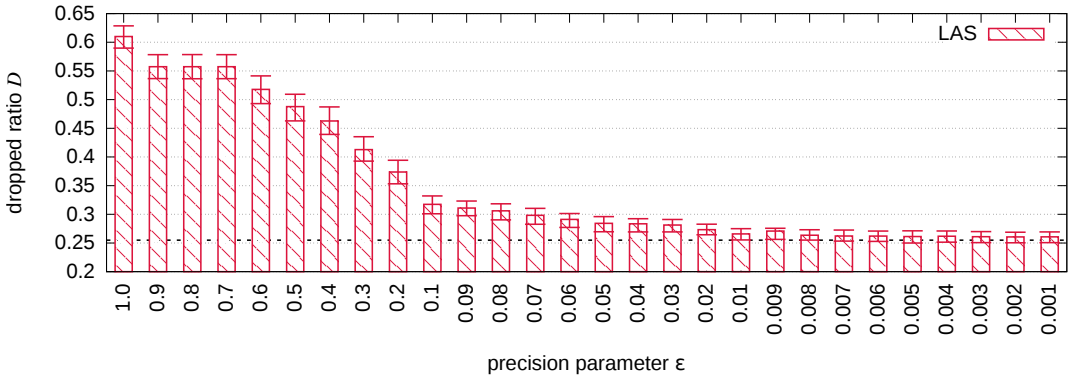
Figure 8.9 – LAS performance varying the frequency probability distributions.

Maximum execution time value $\max_w$ — Figure 8.8 shows the average queuing time $\overline{Q}$ (top) and shedding speed up Λ_d (bottom) as a function of the maximum execution time value $\max_w$. Notice that in this test we varied the value for $\overline{Q}^{\max}$ setting it equal to $\max_w$. Accordingly, Figure 8.8a shows horizontal lines that mark the different thresholds $\overline{Q}^{\max}$. As the two graphs show, the behavior for LAS is rather consistent while varying $\max_w$; 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 speed up that, as shown in Figure 8.8b is always positive.

Frequency probability distributions — Figure 8.9 shows the average queuing time $\overline{Q}$ (top) and shedding speed up Λ_d (bottom) as a function of the input frequency distribution. As Figure 8.9a 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 times perform at their best on strongly skewed distribution, rather than on uniform ones. This result is confirmed by looking at the shedding speed up (Figure 8.9b) that decreases, on average, as the value of α for the distribution increases.



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

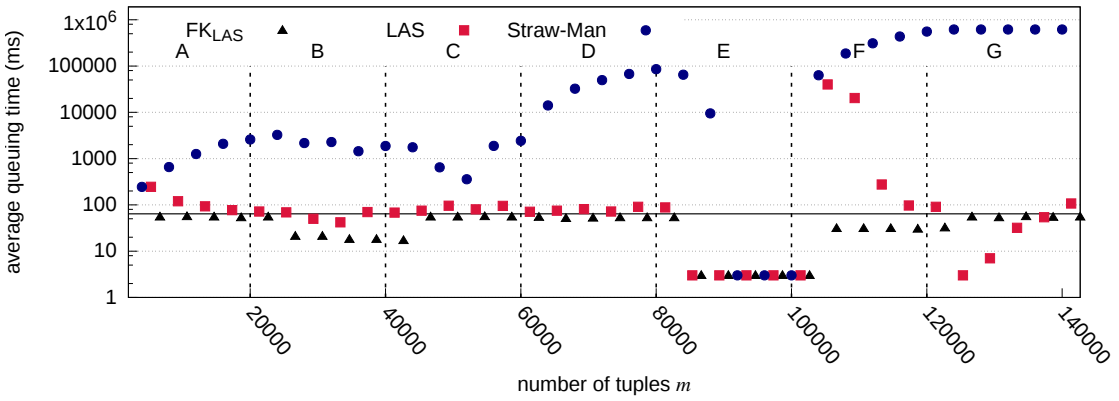


(b) Dropped ratio D

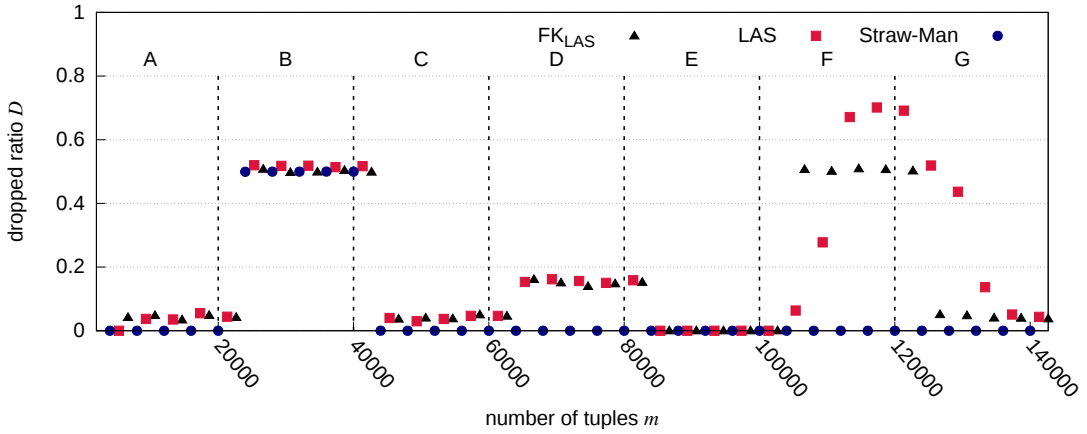
Figure 8.10 – LAS performance varying the precision parameter ε.

Precision parameter ε — Figure 8.10 shows the average queuing time $\overline{Q}$ (top) and dropped ratio D (bottom) 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 $1+\varepsilon$ (*cf.*, Listing 8.5, Line 23) in this case is so large that it pushes LAS to largely overestimate the execution time 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 $\overline{Q}^{\max} = 6.4$ while dropping a smaller number of tuples. The dotted lines in both graphs represent the performance of FK_{LAS} and are provided as a reference.

Time Series — Figure 8.11 shows the average queuing time $\overline{Q}$ (top) and dropped ratio D (bottom) 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 time $\overline{Q}$ or the dropped ratio D computed on the previous $4,000$ tuples. Notice that the points for Straw-Man, LAS and FK_{LAS} related to a same value of the x-axis are artificially shifted to improve readability. In this test we set $\overline{Q}^{\max} = 64$ ms. 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



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



(b) Dropped ratio D

Figure 8.11 – Simulator time-series.

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 $w(t) = \max_w$, inducing an abrupt change in the tuple values frequency distribution and in the average execution time $\overline{W}$;

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 time $w(t)$ for each tuple, simulating a change in the operator resource availability;

phase G same as phase A.

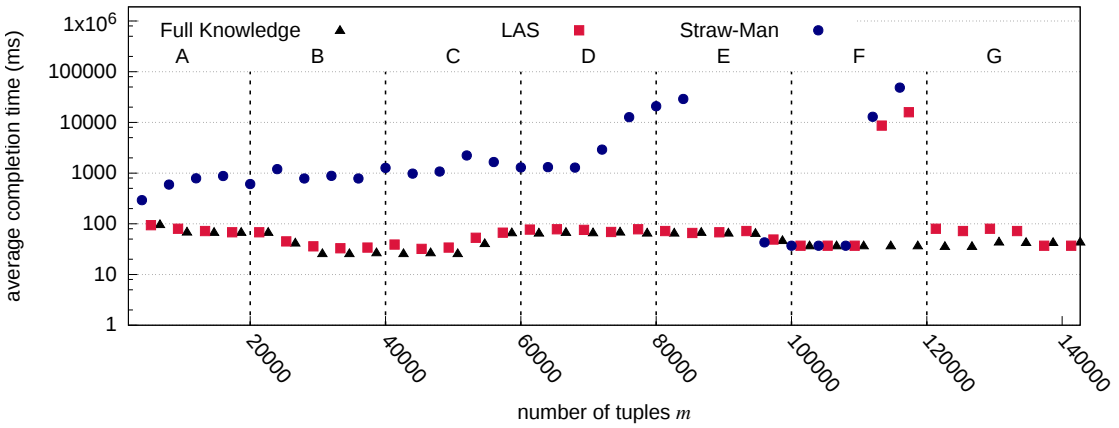
As the graphs show, during phase A the queuing latencies of LAS and Straw-Man diverge: while LAS quickly approaches the performance provided by FK_{LAS}, Straw-Man average queuing latencies quickly grow. In the same timespan, both FK_{LAS} and LAS

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 FK_{LAS} and LAS compensate this change by starting to drop more tuples, but still maintaining the average queuing time close to the desired threshold $\overline{Q}^{\max}$. 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 $\overline{Q}^{\max}$. 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 times 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.

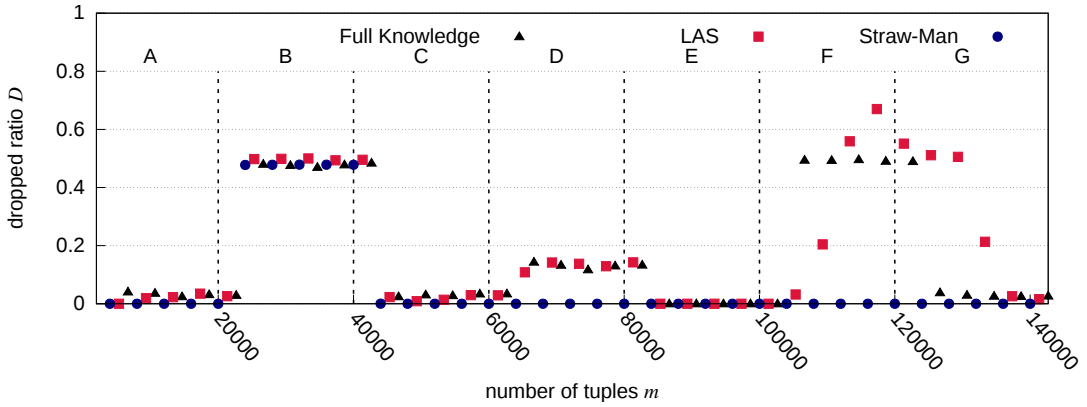
Prototype Results

To evaluate the impact of LAS on real applications we implemented it as a bolt within the Apache Storm [89] 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 FK_{LAS} 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 8.12 report the average completion time per tuple instead of the queuing time. This is due to the difficulties in correctly measuring queuing latencies in Storm. Furthermore, the completion time 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 8.12, report the same qualitative behavior already discussed with Figure 8.11. 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 8.11) 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.



(a) Average completion time $\overline{L}$

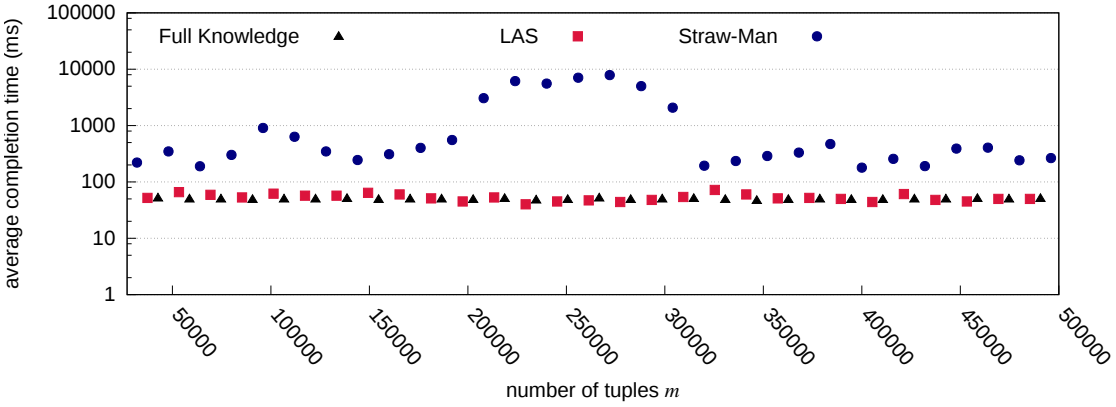


(b) Dropped ratio D

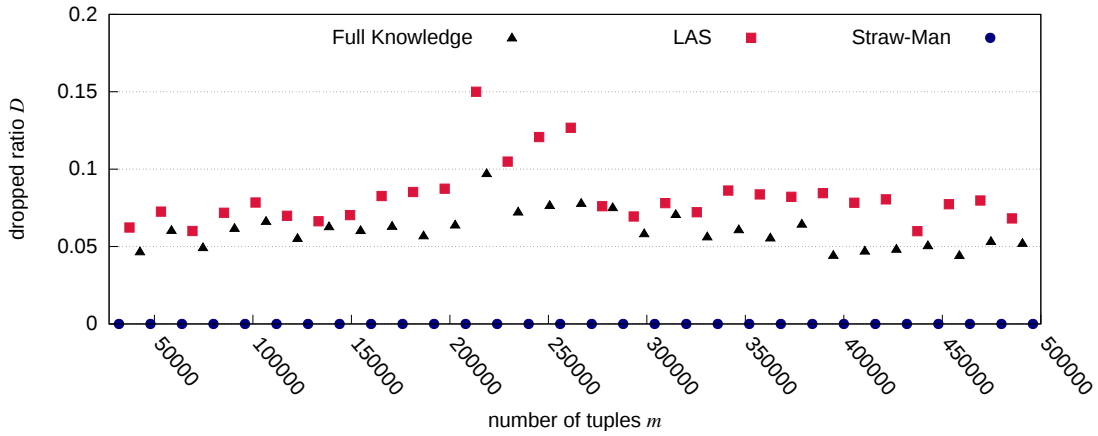
Figure 8.12 – Prototype time-series.

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 (MENTION), 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 time for *media* (*e.g.,* possibly caused by an access to an external DB to gather historical data), an average execution time for *politicians* and a fast execution time for *others* (*e.g.,* possibly because these tweets are not decorated). We modeled execution times with 25 ms, 5 ms and 1 millisecond of busy waiting respectively. Each of the 500,000 tweets may contain more than one mention, leading to $n_w = 110$ different execution time values from $\min_w = 1$ millisecond to $\max_w = 152$ ms, among which the most frequent (36% of the stream) execution time is 1 millisecond. The average execution time $\overline{W}$ is equal to 9.7 millisecond, the threshold $\overline{Q}^{\max}$ is set to 32 ms and the under-provisioning is set to 0%.

Figure 8.13 reports the average completion time (top) and dropped ratio Λ_d (bottom) as the stream unfolds. As the plots show, LAS provides completion latencies that are extremely close to FK_{LAS}, dropping a similar amount of tuples. Conversely, Straw-Man completion latencies are at least one order of magnitude larger. This is a consequence of



(a) Average completion time $\overline{L}$



(b) Dropped ratio D

Figure 8.13 – Prototype mention use-case.

the fact that in the given setting Straw-Man does not drop tuples, while FK_{LAS} 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.

Chapter Summary

This chapter has shown the design of LAS, based on VALES (*cf.*, Section 3.3), a load shedding algorithm taking into account that the execution time of tuples is non-uniform. The load shedding quality has been assessed through experimental evaluation run with both a simulator and a prototype targeting Apache Storm.

The next chapter concludes this thesis.

Chapter 9

Conclusion

This chapter first summarizes the contributions covered in this thesis [1] and then provides some perspectives.

9.1 Summary

The aim of this thesis is to solve practical issues related to Big Data leveraging the data streaming model. The focus has been placed on two classical problems that have many applications and have been extensively studied in this model: the frequency estimation problem and the heavy hitters problem. This thesis lifts some of the limitations in the current solutions and shows how they can be handy in tackling major issues in Big Data applications.

Section 4.3 presents Distributed Heavy Hitters Estimator (DHHE), an algorithm that deterministically detects heavy hitters finely spread in massive and physically distributed data streams, in a more challenging model than the one considered in previous works. Moreover, DHHE also (ε, δ)-approximate the size of each heavy hitter. Recently, Agarwal *et al.* [3], have shown that `MG` [70] and `Space Saving` [69] are mergeable summaries, which possibly entails new and more efficient solutions to solve the distributed heavy hitters problem.

Section 5.1 applies DHHE to the detection of Distributed Denial of Service (DDoS). A thorough experimental evaluation, run on a cluster of single-board computers, illustrates the enjoyable properties of our solution in terms of precision, recall, communication overhead and detection latency.

In Section 4.4 we design Balanced Partitioner (BPART), a novel algorithm to build a partition $\mathcal{K}$ of the streams item universe $[n]$ with exactly k parts (*i.e.,* $|\mathcal{K}| = k$) and where the weight W_p of the parts $O_p \in \mathcal{K}$ ($p \in [k]$) are balanced. We prove that BPART is able to build a $(1 + \theta)$-optimal balanced k-partition when applied on skewed input streams characterized by a Zipfian distribution, *i.e.,* the items weights have a Zipfian distribution of unknown exponent α.

Section 7.1.1 applies BPART to design Distribution-aware Key Grouping (DKG), a novel key grouping solution to load balance parallelized stateful operators in stream processing systems. Through both a simulator and a prototype targeting Apache Storm, we

[1] The summary does not follow the order in which the contributions have been presented.

performed an experimental evaluation showing that DKG outperforms standard approaches to key grouping (*i.e.,* modulo and universal hash functions) and how this positively impacts a real stream processing applications.

Section 3.3 shows the design of Value Estimator (VALES), a sketch data structure providing an (ε, δ)-approximation of the value ω_t carried by tuples.

The first application of VALES is the Online Shuffle Grouping Scheduler (OSG) algorithm (Section 7.2.1), a novel shuffle grouping algorithm to load balance parallelized stateless operators in stream processing systems. OSG aims at reducing the average tuple completion time by scheduling tuples on operator instances on the basis of their estimated execution time. OSG makes use of VALES to keep track of tuples execution time on the operator instances in a compact and scalable way. This information is then fed to a greedy scheduling algorithm to assign incoming load. We first prove that FK_{OSG} algorithm is a $(2 - 1/k)$-approximation of the optimal off-line scheduler, and then that OSG (ε, δ)-approximates FK_{OSG}. Furthermore, we extensively tested OSG performance through both a simulator and with a prototype implementation integrated within the Apache Storm framework. The results showed how OSG provides important speedups in tuple completion time when the workload is characterized by skewed distributions. Further research is needed to explore how much the load model affect performance. For instance, it would be interesting to include other metrics in the load model, *e.g.,* network latencies, to check how much these may improve the overall performance.

The second application of VALES is the Load-Aware Load Shedding (LAS) algorithm (Section 8.1), a novel solution for load shedding in stream processing systems. LAS exploits the differences in the tuples execution time to smartly drop tuples and avoid the appearance of performance bottlenecks. As OSG, LAS leverages VALES 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 time below a given threshold. Through a theoretical analysis, we proved that LAS is an (ε, δ)-approximation of the optimal algorithm FK_{LAS}. Furthermore, we extensively tested LAS both in a simulated setting and with a prototype implementation integrated within Apache Storm. Our tests 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.

With respect to the frequency estimation problem, in Section 3.2 we have presented two (ε, δ)-additive-approximations in the sliding windowed: Proportional Windowed Count-Min and Splitter Windowed Count-Min. They have a space complexity of respectively $O(\frac{1}{\varepsilon} \log \frac{1}{\delta} (\log M + \log n))$ and $O(\frac{1}{\gamma\varepsilon} \log \frac{1}{\delta} (\log M + \log n))$ bits, while their update and query time complexities are both $O(\log \frac{1}{\delta})$. In Section 5.2, we have applied these algorithms to the estimation of IP address frequencies in a network. The resulting performance evaluation measures their respective efficiency and also compares them to the only similar related work [76]. In particular, Splitter WCM reaches better estimation with respect to the state of the art proposal and uses 4 times less memory.

9.2 Perspectives

The contributions of this thesis definitely show that the data streaming model allows not only to design algorithms with sound theoretical roots, but that these algorithms can also have a strong impact on existing Big Data practical applications.

Sliding window model In Section 5.2 we have motivated the need to design data streaming algorithms in the sliding window model, since in many applications it is preferable to take decisions on what has occurred recently, not on what has happened ages ago. In particular, through the sliding window model, we can seamlessly capture changes in the stream or system characteristics. However, for simplicity, in all our other contributions, we have either assumed a steady state scenario (*i.e.,* VALES, DHHE, BPART and DKG) or used jumping windows (*i.e.,* OSG and LAS).

Considering DHHE, a naive solution would be to deploy several instances in sequence, as in a jumping window. However, considering the targeted application, *i.e.,* detecting distributed denial of service, an adversary could leverage this pattern to avoid being detected. A more refined solution is to extend DHHE to the sliding window model. Based on the results presented in Section 5.2, this may be as straightforward as swapping the `Count-Min` instances used at each node with either the Proportional WCM or Splitter WCM (depending on the context).

The same reasoning can be applied to VALES, thus moving both OSG and LAS from jumping windows to sliding windows. Considering BPART, the migration is slightly more complex as we need to replace two data structures: the `Space Saving` and the vector tracking the frequencies of the sparse items $\mathbf{v}_{\overline{h}}$. Notice that the vector $\mathbf{v}_{\overline{h}}$ is nothing more than a single row of a `Count-Min`, thus it can be replaced with one its sliding window extensions. Similarly, the `Space Saving` instance can be replaced by its sliding window version [58] (*cf.,* Section 4.1). Notice however that the sliding window version of BPART cannot be applied to DKG without taking into account that changing the partitioning incurs in a migration cost. This means that to extends DKG to the sliding window model, one must also introduce a mechanism to decide whether the new balancing is worth with respect to the migration cost.

Finally, while for some applications a sliding window is clearly the better choice, in some scenario this may not the true. For instance, if the application does not consider adversaries, *e.g.,* as in stream processing systems (DKG, OSG and LAS), a jumping window may fare as well as a sliding window, while being easier to handle.

Non uniform tuple execution time Considering stream processing systems, in Sections 7.2.1 and 8.1 we showed that, taking into account that the execution time of the tuples depends on the content of the tuples themselves yields sizeable improvements in load balancing shuffle grouped operators and load shedding. This notion can be directly applied to load balance key grouped operators (*cf.,* Section 7.1). More in details, moving the BPART data structures of the parallelized version of DKG (*cf.,* Section 7.3) from the instances of operator S to the instances of operator O, would allow to track the cumulated execution time for both heavy hitters and buckets of sparse items on all instances $O_p \in \mathcal{K}$. Then, the mapping produced by BPART/DKG would balance the sum of the cumulated execution times of the tuples assigned to an instance, minimizing $\max_{p \in [k]} W_p$, with $W_p = \sum_{t_j \in \sigma_{O_p}} w_p(t_j)$.

We believe that among the other 9 optimizations listed by Hirzel *et al.* [56], some (*e.g.,* batching and placement) would benefit from applying a similar reasoning.

Distributed and decentralized data streaming model An issue that we did not handle in this thesis is the decentralization of the distributed data streaming model. While a distributed but centralized algorithm makes sense in many scenarios, there are at least as many applications where we cannot afford a single point of failure, *i.e.,* the coordinator. A trivial solution is to instantiate a coordinator at each node, however this implicitly introduces in the communication complexity a linear factor in to the number of nodes in the system. So far, to the best of our knowledge, there are no efficient and fully decentralized algorithms, *i.e.,* with a communication complexity sub-linear in the size of the system. This is a challenging open question that can have several applications, starting with IoT and fog computing.

As such, an interesting research direction is the design of windowed and decentralized algorithms to solve the classical problems studied in the data streaming community. An even more interesting contribution would be an universal construction to decentralize efficiently some specific classes of data streaming data structures, such as sketches.

Sketches and sampling In Chapter 2 we introduce the two main approaches used to monitor massive data streams in real time: sampling and summaries (*e.g.,* sketches). Both have some weaknesses and strengths, in general sampling provides more accurate results, but may be more memory consuming and can be easily subverted by an adversary. Conversely, summaries may have weaker accuracy bounds (especially considering the worst case) but are resilient to adversaries and have a small memory footprint. As it has often yielded great results in many computer science fields, also in this case combining both techniques may help achieving the mutual strengths while not incurring in the respective weaknesses.

Data streaming, machine learning and data mining As mentioned in the introduction, Big Data has also sprung a lot of research in machine learning, data mining, as well as in modelling and querying uncertain information. On the other hand, data streaming provides an efficient and compact way to capture specific properties of streaming data. We believe that data streaming techniques may be leveraged in order to guide the information retrieval algorithms, reduce the dimensions and/or amount of data that has to be analysed, as well as to filter out the noise from the input data. It may also be interesting to devise techniques to seamlessly query full data or data stored in synopses.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Conference on Innovative Data Systems Research*, CIDR, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable Summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS, 2012.

[4] Akamai Technologies Inc. `http://www.akamai.com/`.

[5] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, STOC, 1996.

[6] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-scale Stream Processing Systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS, 2006.

[7] E. Anceaume and Y. Busnel. A Distributed Information Divergence Estimation over Data Streams. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):478–487, 2014.

[8] E. Anceaume, Y. Busnel, and N. Rivetti. Estimating the Frequency of Data Items in Massive Distributed Streams. In *Proceedings of the 4th IEEE Symposium on Network Cloud Computing and Applications*, NCCA, 2015.

[9] E. Anceaume, Y. Busnel, N. Rivetti, and B. Sericola. Identifying Global Icebergs in Distributed Streams. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems*, SRDS, 2015.

[10] E. Anceaume, Y. Busnel, E. Schulte-Geers, and B. Sericola. Optimization results for a generalized coupon collector problem. *Journal of Applied Probability*, 53(2), 2016.

[11] E. Anceaume, Y. Busnel, and B. Sericola. New results on a generalized coupon collector problem using Markov chains. *Journal of Applied Probability*, 52(2), 2015.

[12] A. Arasu and G. S. Manku. Approximate Counts and Quantiles over Sliding Windows. In *Proceedings of the 23rd ACM Symposium on Principles of Database Systems*, PODS, 2004.

[13] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems*, IOPADS, 1999.

[14] K. Ashton. That "Internet of Things" Thing. In the real world, things matter more than ideas. RFID Journal, `http://www.rfidjournal.com/articles/view?4986`, 2009.

[15] 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, 2004.

[16] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM, 2002.

[17] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the Power of Randomization in Online Algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, STOC, 1990.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM, 1999.

[19] CAIDA UCSD. "Anonymized Internet Traces 2008" Dataset. `http://www.caida.org/data/passive/passive_2008_dataset.xml`, 2008.

[20] CAIDA UCSD. "DDoS Attack 2007" Dataset. `http://www.caida.org/data/passive/ddos-20070804_dataset.xml`, 2010.

[21] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The State of the Art in Locally Distributed Web-server Systems. *ACM Computing Surveys*, 34(2):263–311, 2002.

[22] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS, 2016.

[23] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB, 2003.

[24] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[25] A. Chakrabarti, G. Cormode, and A. McGregor. A Near-optimal Algorithm for Computing the Entropy of a Stream. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, SODA, 2007.

[26] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2003.

[27] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP, 2002.

[28] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys, 2015.

[29] G. Cormode. Continuous Distributed Monitoring: A Short Survey. In *Proceedings of the 1st International Workshop on Algorithms and Models for Distributed Event Processing*, AlMoDEP '11, 2011.

[30] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[31] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, SODA, 2008.

[32] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for Distributed Functional Monitoring. *ACM Transactions on Algorithms*, 7(2):21:1–21:20, 2011.

[33] G. Cormode and K. Yi. Tracking Distributed Aggregates over Time-based Sliding Windows. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*, SSDBM, 2012.

[34] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[36] DEBS 2013. Grand Challenge. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails`.

[37] DOMO. Data Never Sleeps 4.0. `https://www.domo.com/`, 2016.

[38] G. Einziger and R. Friedman. TinyLFU: A Highly Efficient Cache Admission Policy. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP, 2014.

[39] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[40] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[41] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39:207–229, 1992.

[42] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[43] S. Ganguly, M. Garafalakis, R. Rastogi, and K. Sabnani. Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS, 2007.

[44] B. Gedik. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal*, 23(4):517–539, 2014.

[45] P. B. Gibbons and S. Tirthapura. Estimating Simple Functions on the Union of Data Streams. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, SPAA, 2001.

[46] P. B. Gibbons and S. Tirthapura. Distributed Streams Algorithms for Sliding Windows. *Theory of Computing Systems*, 37(3):457–478, 2004.

[47] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying Frequent Items in Sliding Windows over On-line Packet Streams. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC, 2003.

[48] Google Inc. YouTube. `https://www.youtube.com/`.

[49] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[50] O. Green, R. McColl, and D. A. Bader. A Fast Algorithm for Streaming Betweenness Centrality. In *10th International Conference on Privacy, Security, Risk and Trust, and 4th International Conference on Social Computing*, SocialCom-PASSAT, 2012.

[51] S. Guha and A. McGregor. Quantile Estimation in Random-Order Streams. *SIAM Journal on Computing*, 38(5), 2009.

[52] V. Gupta, M. Harchol-balter, K. Sigman, and W. Whitt. Analysis of jointheshortest-queue routing for web server farms. In *Proceedings of the 25th IFIP WG 7.3 International Symposium on Computer Modeling, Measurement and Evaluation*, PERFORMANCE, 2007.

[53] D. Gusfield. Bound the naive Multiple Machine Scheduling with Release Times Deadlines. *Journal of Algorithms*, 5(1):1–6, 1984.

[54] 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, 2014.

[55] 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, 2014.

[56] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys*, 46(4):41–34, 2014.

[57] N. Homem and J. P. Carvalho. Finding top-k elements in data streams. *Information Sciences*, 180(24):4958 – 4974, 2010.

[58] N. Homem and J. P. Carvalho. Finding top-k elements in a time-sliding window. *Evolving Systems*, 2(1):51–70, 2011.

[59] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2006.

[60] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, 2002.

[61] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload Management in Data Stream Processing Systems with Latency Guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing*, Feedback Computing, 2012.

[62] D. M. Kane, J. Nelson, and D. P. Woodruff. An Optimal Algorithm for the Distinct Elements Problem. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems*, PODS, 2010.

[63] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC, 1997.

[64] R. Kumar, J. Novak, P. Raghavan, and A. Tomkins. On the Bursty Evolution of Blogspace. *World Wide Web*, 8(2):159–178, 2005.

[65] Linked Data Benchmark Council. Social Network Benchmark. `http://ldbcouncil.org/benchmarks/snb`.

[66] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE, 2005.

[67] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB, 2002.

[68] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST, 2003.

[69] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT, 2005.

[70] J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143–152, 1982.

[71] A. Mukhopadhyay and R. Mazumdar. Analysis of Randomized Join-The-Shortest-Queue (JSQ) Schemes in Large Heterogeneous Processor Sharing Systems. *IEEE Transactions on Control of Network Systems*, PP(99):1–1, 2015.

[72] S. Muthukrishnan. *Data streams: algorithms and applications*. Now Publishers Inc, 2005.

[73] M. A. U. Nasir, G. D. F. Morales, D. G. Soriano, N. Kourtellis, and M. Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE, 2015.

[74] Netflix. `https://www.netflix.com/`.

[75] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW, 2010.

[76] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis. Sketch-based Querying of Distributed Sliding-Window Data Streams. *Proceedings of the VLDB Endowment*, 5(10):992–1003, 2012.

[77] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS, 2012.

[78] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Dhedding in TelegraphCQ. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE, 2005.

[79] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems. In *Proceedings of the 17th ACM/IFIP/USENIX International Middleware Conference*, Middleware, 2016.

[80] N. Rivetti, Y. Busnel, and A. Mostefaoui. Efficiently Summarizing Distributed Data Streams over Sliding Windows. In *Proceedings of the 14th IEEE International Symposium on Network Computing and Applications*, NCA, 2015.

[81] N. Rivetti, Y. Busnel, and L. Querzoni. Load-aware Shedding in Stream Processing Systems. In *Proceedings of the 10th ACM International Conference on Distributed Event-Based Systems*, DEBS, 2016.

[82] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient Key Grouping for Near-Optimal Load Balancing in Stream Processing Systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS, 2015.

[83] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu. Elastic Scaling of Data Parallel Operators in Stream Processing. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing*, IPDPS, 2009.

[84] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries. *ACM Transactions on Database Systems*, 33(1):5:1–5:44, 2008.

[85] 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, 2003.

[86] 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*, VLDB, 2007.

[87] The Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[88] The Apache Software Foundation. Apache Pig. `http://pig.apache.org/`.

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

[90] The Weather Channel. `https://weather.com/`.

[91] 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, 2006.

[92] Voucher Cloud. Every Day Big Data Statistics – 2.5 Quintillion Bytes of Data Created Daily. `http://www.vcloudnews.com/`, 2016.

[93] K. Yi and Q. Zhang. Optimal Tracking of Distributed Heavy Hitters and Quantiles. *Algorithmica*, 65:206–223, 2013.

[94] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2012.

[95] L. Zhang and Y. Guan. Variance Estimation over Sliding Windows. In *Proceedings of the 26h ACM Symposium on Principles of Database Systems*, PODS, 2007.

[96] Q. Zhao, A. Lall, M. Ogihara, and J. Xu. Global iceberg detection over Distributed Streams. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE, 2010.

[97] Q. Zhao, M. Ogihara, H. Wang, and J. Xu. Finding Global Icebergs over Distributed Data Sets. In *Proceedings of the 25th ACM SIGACT- SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS, 2006.

[98] S. Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems.* PhD thesis, UC Berkeley, 1987.

List of Notations

Roman lower case letters

b	jobs	p. 47
c	number of columns of the $\mathfrak{F}$, $\mathfrak{F}_p$, $\mathfrak{W}$ and $\mathfrak{W}_p$ matrices	p. 17, 18, 28, 90, 108
d	number of tuples dropped by a load shedding algorithm	p. 107
e	euler's constant	p. 18
$\mathbf{f}$	frequency vector of stream σ	p. 12
f_t	frequency of item t in σ	p. 12
$\mathbf{f}_{-t}$	frequency vector of stream σ without the scalar f_t	p. 18
$\mathbf{f}_i$	frequency vector of sub-stream σ_i	p. 13
$f_{t,i}$	frequency of item t in sub-stream σ_i	p. 13
g	number of $\mathcal{G}_\kappa$ buffers	p. 38
h	hash function	p. 15
$\overline{h}$	BPART sparse items hash function	p. 47
i	generic S_i node or instance index	p. 13, 73
j	index in $[m]$ of the stream σ	p. 12
k	number of partitions of $[n]$ or operator O instances	p. 46, 73
$\ell(t_j)$	completion time of tuple t_j	p. 73
m	size or length of the stream σ	p. 12
$[m]$	index sequence $\{1, \ldots, m\}$ of the stream σ	p. 12
m_i	size or length of the sub-stream σ_i	p. 13
$\max_\omega$	maximum ω_t value	p. 29
$\max_w$	maximum tuple execution time	p. 95, 114
$\min_\omega$	minimum ω_t value	p. 29
$\min_w$	minimum tuple execution time	p. 95, 114
n	number of distinct items in the stream σ	p. 12
$[n]$	universe $\{1, \ldots, n\}$ of the stream σ	p. 12
n_κ	maximum size of $\mathcal{G}_\kappa$	p. 38
n_w	number of tuple execution time values	p. 95, 114
p	generic O_p partition or operator instance index	p. 46, 73
$\mathbf{p}$	empirical probability distribution of the stream σ	p. 12
p_t	empirical probability of item t in the stream σ	p. 12
$\mathbf{p}_i$	empirical probability distribution of the sub-stream σ_i	p. 13
$p_{t,i}$	empirical probability of item t in the sub-stream σ_i	p. 13
$\widehat{p}_{t,i}$	estimated probability of item t in the sub-stream σ_i	p. 38

$q(t_j)$	queuing time of tuple t_j	p. 73
$\widehat{q}(t_j)$	estimated queuing time of tuple t_j	p. 108
r	number of rows of the $\mathfrak{F}$, $\mathfrak{F}_p$, $\mathfrak{W}$ and $\mathfrak{W}_p$ matrices	p. 17, 18, 28, 90, 108
s	number of $S_i \in \mathcal{I}$ nodes or operator instances	p. 13, 73
t	generic item or tuple value in $[n]$ of the stream σ	p. 12, 73
$\mathbf{v}_{\overline{h}}$	BPART sparse items frequency vector	p. 48
$w_p(t_j)$	execution time of tuple t_j on operator instance O_p	p. 73
$\widehat{w}_p(t_j)$	estimated execution time of tuple t_j on instance O_p	p. 89, 108

Roman upper case letters

D	dropped ratio d/m	p. 115
$F_\imath$	$\imath$-th frequency moment	p. 17
$H_\imath$	$\imath$-th harmonic function	p. 40
L	total completion latency	p. 73
$\overline{L}$	ttoal average completion latency	p. 73
M	window size	p. 12
M^{OSG}	OSG window size parameter	p. 90
M^{LAS}	LAS window size parameter	p. 109
O	operator	p. 73
O_p	part or operator instance	p. 46, 73
Q	total queuing time	p. 73
$\overline{Q}$	total average queuing time	p. 73, 107
$\overline{Q}^{\max}$	LAS average queuing time threshold parameter	p. 107
S	node or operator	p. 13, 73
S_i	i-th node or operator instance in $\mathcal{I}$	p. 13, 73
$T_{a,n}$	coupon collector value	p. 42
W	total execution time	p. 73
$\mathbf{W}$	parts weights vector	p. 49
$\widehat{W}$	estimated total execution time	p. 108
W_p	weight or execution time on partition or operator O_p	p. 46, 89
$\widehat{W}_p$	estimated total execution time on operator O_p	p. 89
$\overline{W}$	total average weight or execution time	p. 49, 73

Roman calligraphic letters

$\mathcal{B}$	set of jobs	p. 47
$\mathcal{D}$	set of dropped tuples	p. 107
$\mathfrak{F}$	`Count-Sketch`, `Count-Min`, VALES and LAS frequency matrix	p. 17, 18, 28, 108
$\mathfrak{F}_\mathfrak{p}$	OSG frequency matrix on operator O_p	p. 90
$\mathcal{G}_{\kappa,i}$	κ-th DHHE buffer on node S_i	p. 38
$\mathcal{H}$	set of 2-universal hash function	p. 15

$\mathcal{HH}$	set of heavy hitters	p. 33
$\mathcal{I}$	set of s nodes or instances S_i	p. 13, 73
$\mathcal{J}_i$	DHHE detected heavy hitter buffer on node S_i	p. 40
$\mathcal{K}$	set of partitions or instances O_p	p. 46, 73
$\mathfrak{S}$	Proportional WCM, OSG and LAS snapshot matrix	p. 21, 90, 109
$\mathcal{SI}$	set of sparse items	p. 33
$\mathfrak{W}$	VALES and LAS value matrix	p. 28, 108
$\mathfrak{W}_p$	OSG value matrix on operator O_p	p. 90

Greek lower case letters

α	Zipf distribution exponent	p. 51
β	Splitter WCM tolerance threshold parameter	p. 23
γ	Splitter WCM sub-cell size threshold parameter	p. 23
δ	(ε, δ)-approximation error probability parameter	p. 14
ε	(ε, δ)-approximation precision parameter	p. 14
η	LAS relative distance	p. 109
η_p	OSG relative distance	p. 90
$\overline{\eta}$	OSG/LAS relative distance threshold	p. 90, 109
θ	heavy hitter threshold	p. 33
κ	DHHE generic $\mathcal{G}_\kappa$ buffer index	p. 38
λ	percentage of imbalance	p. 49
μ	DKG co-domain multiplier parameter	p. 47
ν_κ	DHHE $\mathcal{G}_\kappa$ size threshold	p. 38
π	Splitter WCM number of splits	p. 25
$\overline{\pi}$	Splitter WCM number of splits upper bound	p. 25
ρ	DHHE $\mathcal{G}_\kappa$ buffer size parameter	p. 38
σ	stream	p. 12
σ_i	i-th sub-stream of σ	p. 13
$\sigma(M)$	active window in σ of size M	p. 12
$\sigma_{S \to O}$	stream from operator S to O	p. 73
$\sigma_{S_i \to O_p}$	sub-stream from operator instance S_i to O_p	p. 73
σ_{S_i}	outbound sub-stream of operator instance S_i	p. 73
σ_{O_p}	inbound sub-stream of operator instance O_p	p. 73
τ_κ	DHHE $\mathcal{G}_\kappa$ buffer timer	p. 38
ω_{t_j}	value carried by or weight of tuple t_j	p. 18,46

Greek upper case letters

Δ_p	OSG operator feedback	p. 92
Δ	LAS operator feedback	p. 110
Λ_L	completion latency speedup	p. 96
Λ_d	shedding speedup	p. 115

List of Figures, Listings and Tables

List of Theorems and Problems

UNIVERSITE BRETAGNE LOIRE

SAPIENZA UNIVERSITÀ DI ROMA

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Nicoló RIVETTI
DI VAL CERVO

Analyse efficace de flux de données et applications au traitement des grandes masses de données

Efficient Stream Analysis and its Application to Big Data Processing

Résumé

L'analyse de flux de données est utilisée dans beaucoup de contexte où la masse des données et/ou le débit auquel elles sont générées, excluent d'autres approches (par exemple le traitement par lots). Le modèle flux fourni des solutions aléatoires et/ou fondées sur des approximations pour calculer des fonctions d'intérêt sur des flux (repartis) de n-uplets, en considérant le pire cas, et en essayant de minimiser l'utilisation des ressources. En particulier, nous nous intéressons à deux problèmes classiques : l'estimation de fréquence et les poids lourds.
Un champ d'application moins courant est le traitement de flux qui est d'une certaine façon un champ complémentaire aux modèle flux. Celui-ci fournis des systèmes pour effectuer des calculs génériques sur les flux en temps réel souple, qui passent à l'échèle. Cette dualité nous permet d'appliquer des solutions du modèle flux pour optimiser des systèmes de traitement de flux.
Dans cette thèse, nous proposons un nouvel algorithme pour la détection d'éléments surabondants dans des flux repartis, ainsi que deux extensions d'un algorithme classique pour l'estimation des fréquences des items. Nous nous intéressons également à deux problèmes : construire un partitionnement équitable de l'univers des n-uplets par rapport à leurs poids et l'estimation des valeurs de ces n-uplets. Nous utilisons ces algorithmes pour équilibrer et/ou délester la charge dans les systèmes de traitement de flux.

Mots clés

Modèle Flux, Approximation et Processus Aléatoire, Eléments Surabondants, Estimation de Fréquences, Sureté de Fonctionnement des Réseaux, Traitement de Flux en Temps Réel, Equilibrage de Charge, Délestage de charge.

Abstract

Nowadays stream analysis is used in many context where the amount of data and/or the rate at which it is generated rules out other approaches (*e.g.,* batch processing). The data streaming model provides randomized and/or approximated solutions to compute specific functions over (distributed) stream(s) of data-items in worst case scenarios, while striving for small resources usage. In particular, we look into two classical and related data streaming problems: frequency estimation and (distributed) heavy hitters.
A less common field of application is stream processing which is somehow complementary and more practical, providing efficient and highly scalable frameworks to perform soft real-time generic computation on streams, relying on cloud computing. This duality allows us to apply data streaming solutions to optimize stream processing systems.
In this thesis, we provide a novel algorithm to track heavy hitters in distributed streams and two extensions of a well-known algorithm to estimate the frequencies of data items. We also tackle two related problems and their solution: provide even partitioning of the item universe based on their weights and provide an estimation of the values carried by the items of the stream. We then apply these results to both network monitoring and stream processing. In particular, we leverage these solutions to perform load shedding as well as to load balance parallelized operators in stream processing systems.

Key Words

Data Streaming, Randomized Approximation, Heavy Hitter, Frequency Estimation, Network Dependability, Stream Processing, Load Balancing, Load Shedding