Efficient Key Grouping for Near-Optimal Load Balancing in Stream Processing Systems*

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

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

Emmanuelle Anceaume
IRISA / CNRS
Rennes, France
emmanuelle.anceaume@irisa.fr

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

Bruno Sericola
Inria
Rennes, France
bruno.sericola@inria.fr

ABSTRACT

Key grouping is a technique used by stream processing frameworks to simplify the development of parallel stateful operators. Through key grouping a stream of tuples is partitioned in several disjoint sub-streams depending on the values contained in the tuples themselves. Each operator instance target of one sub-stream is guaranteed to receive all the tuples containing a specific key value. A common solution to implement key grouping is through hash functions that, however, are known to cause load imbalances on the target operator instances when the input data stream is characterized by a skewed value distribution. In this paper we present DKG, a novel approach to key grouping that provides near-optimal load distribution for input streams with skewed value distribution. DKG starts from the simple observation that with such inputs the load balance is strongly driven by the most frequent values; it identifies such values and explicitly maps them to sub-streams together with groups of less frequent items to achieve a near-optimal load balance. We provide theoretical approximation bounds for the quality of the mapping derived by DKG and show, through both simulations and a running prototype, its impact on stream processing applications.

*This work has been partially founded by the French ANR project SocioPlug (ANR-13-INFR-0003), by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01) and by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

DEBS '15, June 29 - July 03, 2015, Oslo, Norway

DOI: http://dx.doi.org/10.1145/2675743.2771827.

Categories and Subject Descriptors

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

Keywords

Stream Processing, Data Streaming, Key Grouping, Load Balancing

1. INTRODUCTION

Stream processing systems are today gaining momentum as a tool to perform analytics on continuous data streams. Their ability to produce analysis results with sub-second latencies, coupled with their scalability, makes them the preferred choice for many big data companies.

A stream processing application is commonly modeled as a direct acyclic graph where data operators, represented by nodes, are interconnected by streams of tuples containing data to be analyzed, the directed edges. Scalability is usually attained at the deployment phase where each data operator can be parallelized using multiple instances, each of which will handle 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. Two main approaches are commonly adopted: either tuples are randomly assigned to target instances (*random* or *shuffle grouping*) or the assignment is based on the specific value of data contained in the tuple itself (*key* or *field grouping*). Shuffle grouping is the preferred choice whenever the target operator is stateless as this approach is easy to implement (*e.g.,* with a round-robin mapping) and provides a nice load balancing among the operator's parallel instances.

When the target operator is stateful things get more complex as 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 which will affect 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 the case 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. Hash functions are convenient as they are compact, fast and deterministic. However, they are usually designed to uniformly spread values from their domain to available instances in their codomain; 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. A solution could lie in defining an explicit one-to-one mapping between input values and available target instances that could take into account the skewed frequency distribution and thus uniformly spread the load; this solution is considered impractical as it requires to have precise knowledge on the input value distribution (usually not known *a priori*) and imposes, at runtime, a memory footprint that is proportional to the number of possible values in the input domain (commonly a huge number if you consider, as an example, the domains of length-constrained strings or floating-point numbers).

In this paper 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 is based on the observation that when the values used to perform the grouping have skewed frequencies, *e.g.,* they can be approximated with a Zipfian distribution, the few most frequent values (the *heavy hitters*) drive the load distribution, while the remaining largest fraction of the values (the *sparse items*) appear so rarely in the stream that the relative impact of each of them on the global load balance is negligible. However, when considered in groups sparse items should be handled with care. On the basis of these observations DKG monitors the incoming stream to identify the heavy hitters and estimate their frequency, and thus the load they will impose on their target instance; sparse items are mapped, with a standard hash function, to a fixed set of buckets. After this initial training phase, whose length can be tuned, the final mapping is obtained by running a greedy multiprocessor scheduling algorithm 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. DKG is a practical solution as it uses efficient data streaming algorithms to estimate with a bounded error the frequency of the heavy hitters, and has a small and constant memory footprint that can be tuned by system administrators, thus overcoming the two main limitations of standard one-to-one mappings.

We show, through a theoretical analysis, that DKG provides on average near-optimal mappings using sub-linear space in the number of tuples read from the input stream in the learning phase and the support (value domain) of the tuples. In particular this analysis presents new results regarding the expected error made on the estimation of the frequency of heavy hitters.

We also extensively tested DKG both in a simulated environment with synthetic datasets and on a prototype implementation based on Apache Storm [17] running with real data. The experiments show that DKG outperforms standard hashing solutions when run over streams with slightly- to strongly-skewed value distributions. In particular, DKG is able to deliver very stable performance that do not vary strongly with the input, and that are always close to an approximate optimum that is obtainable only with full information on the input distribution. Our results also point out that DKG needs a short training phase before delivering the aforementioned performance and that it positively impacts real applications by significantly increasing the global throughput of the system.

After this introduction the next section states the system model we consider. Afterwards, Section 3 details DKG whose behavior is then theoretically analyzed in Section 4. Section 5 reports on our experimental evaluation and Section 6 analyzes the related works. Finally Section 7 concludes and hints on possible future work.

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 direct acyclic graph interconnecting processing elements (PE), represented by nodes, with data streams (DS), represented by edges. Each topology contains at least a *source*, *i.e.,* a PE connected only through outbound DSs, and a *sink*, *i.e.,* a PE connected only through inbound DSs. Each PE O can be parallelized by creating k independent instances $O_0, \cdots, O_{k-1}$ of it and by partitioning its inbound stream O^{in} in k sub-streams $O_0^{in}, \cdots, O_{k-1}^{in}$.

Data injected by the source is encapsulated in units called tuples. Each data stream is an unbounded sequence of tuples. Without loss of generality, here we assume that each tuple t is a finite set of key/value pairs that can be customized to represent complex data structures; furthermore, we assume for each possible key x that the corresponding value $t(x)$ is defined in a finite domain and that it is characterized by an unknown, but fixed, probability distribution.

DS partitioning can be performed in several different ways, but here we focus on the *key grouping* approach: given a key x, *i.e.,* the *grouping key*, all tuples of the DS containing the same value $t(x)$ are placed in the same sub-stream. The problem we target in this paper is how to perform the DS partitioning such that all sub-streams are load balanced, *i.e.,* all sub-streams contain, on average, the same number of tuples.

A common solution to this problem employed by several SPSs is based on an hash function h: the index of the target sub-stream for a tuple t is given by $h(t(v)) \mod k$. This solution is simple and highly scalable as the definition of h

is the only information that needs to be known by the PE that partitions the stream. On the other side, it can produce skewed load on the parallel instances[1] 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 [4, 12]. A second common 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, 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 assumption on a-priori knowledge on value distribution does not usually hold, and (ii) the map would easily grow to unmanageable sizes, even for typical value domains (*e.g.,* strings).

For the sake of clarity, and without loss of generality, here we restrict our model to a single DS O^{in} of the topology with its producer PE P and its consumer PE O. In the rest of this work we deal with streams of unary tuples with a single non null positive integer value representing the values of the grouping key(s). For instance, let $\text{KEY}(t)$ be a function that returns a positive integer value representing the grouping key(s) value(s) of tuple t, *i.e.,* $\text{KEY}(t) = \phi(\{t(x) \mid x \in \mathcal{X}\})$ where $\mathcal{X}$ is the set of grouping keys, then the stream $O^{k,in}$ we consider is the output stream of $\text{KEY}(t)$ executed on the tuples carried by the stream O^{in}. We assume that values of $O^{k,in}$ are characterized by a possibly skewed distribution and that each sequence extracted from $O^{k,in}$ has the same statistical characteristics of the whole stream (*i.e.,* no adversary in the model). Abusing the notation, we denote both the tuple and the value encapsulated by tuple as t .

3. Distribution-aware Key Grouping

In this section we present our solution consisting of a three-phase algorithm. (i) In the first phase the algorithm becomes aware of the stream distribution (*learning* phase). (ii) The following phase builds a global mapping function taking into account the previously gathered knowledge of the stream (*build* phase). (iii) The last phase uses the built global mapping function to perform key grouping with near-optimal balancing (*deployment* phase).

As previously motivated, we cannot afford to store the entire input data stream in order to analyze it, or cannot make multiple passes over it to keep pace with the rate of the stream. As such we rely on data streaming algorithms, which have shown their highly desirable properties in data intensive applications to compute different kind of basic statistics, including the number of different items in a given stream [3, 7, 11], the frequency moments [1], the most frequent data items [1, 5], or to provide uniform sampling [2].

3.1 Background

Data Streaming model — We present the the data stream model [14], under which we analyze our algorithms and derive bounds. A stream is a sequence of elements $\langle t_1, t_2, \ldots, t_m \rangle$ called tuples or items, which are drawn from a large universe $[n] = \{1, \ldots, n\}$. In the following, the size (or length) of the stream is denoted by m. Notice that in this work we do not consider m as the size of the whole stream, but as the size of the learning set, *i.e.,* how long the first phase lasts. In the following we denote by p_i the unknown probability of occurrence of item i in the stream and with f_i the unknown frequency of item i, *i.e.,* the mean number of occurrences of i in the stream of size m.

Heavy Hitters — An item i of a stream is called *heavy hitter* if the empirical probability p_i with which item i appears in the stream satisfies $p_i \geq \Theta$ for some given threshold $0 < \Theta \leq 1$. In the following we call *sparse items* the items of the stream that are not heavy hitters.

Space Saving — Metwally *et al.* [13] have proposed a deterministic counter-based solution to estimate the frequency of heavy hitters. This algorithm, called Space Saving, takes two parameters: Θ and ε, such that $0 < \varepsilon < \Theta \leq 1$. It maintains $\lceil 1/\varepsilon \rceil$ $\langle tuple, counter \rangle$ couples, and returns all items whose frequency of occurrences are greater than or equal to Θm. Metwally *et al.* [13] have proven that after having received m items, the over-approximation made by the algorithm on the estimated frequency $\hat{f}_i$ of heavy hitter i verifies $\hat{f}_i - f_i \leq \varepsilon m$ for any $0 < \varepsilon < \Theta$.

2-Universal Hash Functions — We make use of hash functions randomly picked from a 2-universal hash functions family. A collection $\mathcal{H}$ of hash functions $h : \{1, \ldots, M\} \rightarrow \{0, \ldots, M'\}$ is said to be 2-universal if for every two different items $x, y \in [M]$, for all $h \in \mathcal{H}$, $\mathbb{P}\{h(x) = h(y)\} \leq \frac{1}{M'}$, which is the probability of collision obtained if the hash function assigned truly random values to any $x \in [M]$.

Greedy scheduling algorithm — 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. In this paper we adapt this problem to our setting. More formally we have (i) a set of buckets $b_i \in \mathcal{B}$, each of them with an associated frequency f_i, and (ii) a set of instances $r_j \in \mathcal{R}$ ($|\mathcal{R}| = k$), each of them with an associated load L_j. The load L_j is the sum of the frequencies of the buckets assigned to instance r_j. We want to associate each bucket $b_j \in \mathcal{B}$, minimizing the maximum load on the instances: $\max_{j=1,\ldots,k}(L_j)$. To solve efficiently this problem, known to be NP-hard, we make use of the classic Least Processing Time First (LPTF) approximation algorithm run over small inputs. The algorithm (referred as `scheduling` algorithm in the following) assigns the bucket $b_i \in \mathcal{B}$ with the largest frequency f_i to the instance r_j with the lowest load L_j, then removes b_i from $\mathcal{B}$ and repeats until $\mathcal{B}$ is empty. It is proven [9] that this algorithm provides a $(\frac{4}{3} - \frac{1}{3k})$-approximation of the optimal mapping.

3.2 DKG design

As previously mentioned, the unbalancing in key grouping is mainly due to the skewness of the input stream distribution. For instance, let i and j be the stream heavy hitters, most likely a good mapping should keep them separated. In addition, if $f_i > m/k$, the hash function should isolate i on an instance. However, a randomly chosen hash function will almost certainly assign other (sparse) items with i. Even worse, the hash function may end up putting i and j together.

To cope with these issues, DKG becomes aware of the stream distribution through a learning phase. It is then able

[1]With some abuse of terms, in the rest of the paper we will use as synonyms the sub-streams and the parallel instances of the target operator to which such sub-streams are inbound.

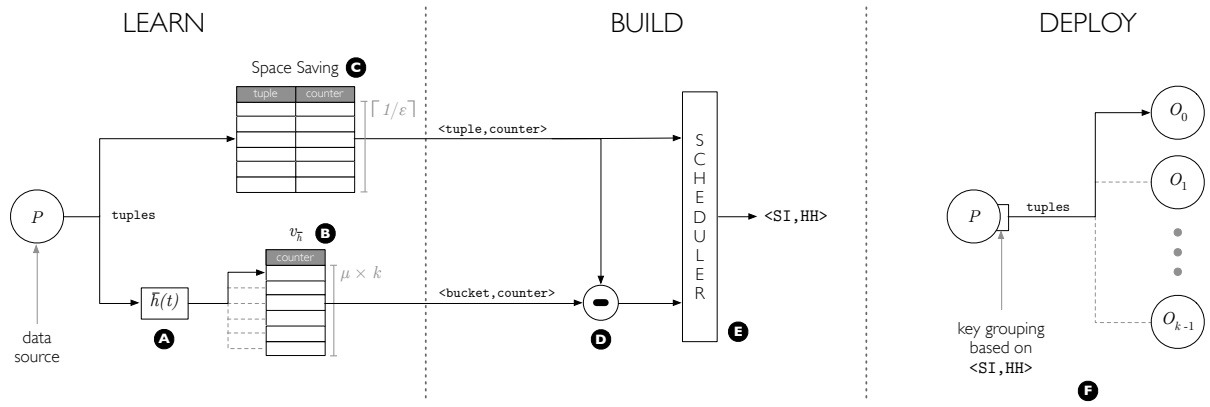


Figure 1: DKG architecture and working phases.

to build a global mapping function that avoids pathological configurations and that achieves close to optimal balancing. Listings 3.1 and 3.2 show the pseudo code for each phase.

DKG (Figure 1 point A) chooses a hash function $\bar{h} : \{1,\dots,n\} \to \{1,\dots,k\mu\}$ 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 $\bar{h}$ co-domain) than instances, we can map buckets to instances minimizing the unbalancing. More in details, DKG feeds to the previously presented `scheduling` algorithm (*cf.,* Section 3.1) the buckets of $\bar{h}$ with their frequencies as the set $\mathcal{B}$ and the number of instances k. The frequencies of $\bar{h}$'s buckets are computed in the learning phase as follows. DKG keeps an array $v_{\bar{h}}$ of size $k\mu$ (Line 3). When receiving tuple t, DKG increments the cell associated through $\bar{h}$ with t (Line 9, Figure 1 point B). In other words $v_{\bar{h}}$ represents how $\bar{h}$ maps the stream tuples to its own buckets.

Listing 3.1: DKG Learning phase

```
1: init ( Θ, ε, k, μ ) do
2:     h̄ : {1,...,n} → {1,...,kμ}: a randomly chosen 2-universal hash functions.
3:     v_h̄ array of size μ × k.
4:     SpaceSaving(Θ, ε): a Space Saving algorithm instance.
5:     HH associative array mapping heavy hitters to instances.
6:     SI associative array mapping h̄ buckets to instances.
7: end init
8: function LEARN(t : tuple)
9:     v_h̄[ h̄(t) ] ← v_h̄[ h̄(t) ] + 1
10:    SpaceSaving.UPDATE(t)
11: end function
```

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 `scheduling` algorithm should be able to produce a good mapping. To approximate this ideal configuration we have to remove all the heavy hitters. As such, DKG 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, DKG updates the `Space Saving` algorithm instance with the values of t (Line 10, Figure 1 point C). At the end of the learning phase, the `Space Saving` algorithm will maintain the most frequent values of t and their estimated frequencies. Then (Figure 1 point D), DKG removes the frequency count of each heavy hitter from $v_{\bar{h}}$ (Line 15). Finally (Figure 1e) , it feeds to the `scheduling` algorithm the $v_{\bar{h}}$ array and the heavy hitters from the `Space Saving` algorithm, with the frequencies of both, as the set of bucket $\mathcal{B}$. The `scheduling` algorithm will return an approximation of the optimal mapping from $\bar{h}$ buckets and detected heavy hitters to instances (Line 17).

When receiving tuple t in the deployment phase (Figure 1 point F), the resulting global mapping function: (i) checks if t is a frequent item and returns the associated instance (Line 21). (ii) Otherwise it computes the hash of t: $\bar{h}(t)$, and returns the instance associated with the resulting bucket (Line 23).

Listing 3.2: DKG Deployment phase

```
12: function BUILD()
13:     SS ← SpaceSaving.QUERY()
14:     for all ⟨ℓ, f̂_ℓ⟩ ∈ SS do
15:         v_h̄[ h̄(ℓ) ] ← v_h̄[ h̄(ℓ) ] − f̂_ℓ
16:     end for
17:     ⟨ SI, HH ⟩ ← scheduling( v_h̄, SS )
18: end function
19: function GETINSTANCE(t : tuple)
20:     if t ∈ HH then
21:         return HH[ t ]
22:     else
23:         return SI[ h̄(t) ]
24:     end if
25: end function
```

THEOREM 3.1 (TIME COMPLEXITY OF DKG). *The time complexity of DKG is $\mathcal{O}(\log 1/\varepsilon)$ per update in the learning phase, $\mathcal{O}\left((k\mu + 1/\varepsilon)\log(k\mu + 1/\varepsilon)\right)$ to build the global mapping function, and $\mathcal{O}(1)$ to return the instance associated with a tuple.*

THEOREM 3.2 (MEMORY REQUIREMENT OF DKG). *The space complexity of DKG is $\mathcal{O}\left((k\mu + 1/\varepsilon)\log m + \log n\right)$ bits in the learning phase and to build the global mapping function. To store the global mapping function, DKG requires $\mathcal{O}\left((k\mu + 1/\Theta)\log n\right)$ bits.*

4. THEORETICAL ANALYSIS

Data-streaming algorithms strongly rely on pseudo-random functions that map elements of the stream to uniformly distributed image values to keep the essential information of the input stream, regardless of the distribution of the elements of the stream. Most of the research done so far has assumed that the input stream is manipulated by an adversary. Such worst case scenario rarely occurs in many real-world applications and do not allow the capture of a notion of average case analysis [10]. Rather, and as motivated in Section 2, we suppose that data set follow a Zipfian distribution and we consider that the order of the items in the stream is random.

We characterize the mean error made by our algorithm with respect to the *optimal* mapping according to the *percentage of imbalance* metric [16], which is defined as follows:

$$\lambda(L) = \left(\frac{\max_{i=1,\ldots,k}(L_i)}{\overline{L}} - 1 \right) \times 100\% \qquad (1)$$

where L represents the load vector of instances, L_i is the load on instance i and $\overline{L}$ is the mean load over all k instances. We first derive in Theorem 4.1 the expected error made by the `Space Saving` algorithm. In the following m denotes the size of the portion of the stream read during the learning phase.

THEOREM 4.1 (MEAN ERROR OF SPACE SAVING). *For any $0 < \varepsilon < \Theta$, the expected error made by `Space Saving` algorithm to estimate the frequency of any heavy hitter i is bounded by $1 - \varepsilon$, that is,*

$$0 \leq \mathbb{E}\{\hat{f}_i\} - f_i \leq 1 - \varepsilon,$$

where $\hat{f}_i$ represents the estimated frequency of item i.

PROOF. We denote by $p = (p_1, \ldots, p_n)$ the probability distribution of the occurrence of the items $1, \ldots, n$ in the input stream. For $\ell = 1, \ldots, m$, we denote by Y_ℓ the value of the item at position ℓ in the stream. These random variables are supposed to be independent and identically distributed with probability distribution p. We thus have $\mathbb{P}\{Y_\ell = i\} = p_i$ and $f_i = mp_i$. Items in the stream are successively selected at each discrete time $\ell \geq 1$, and are inserted in the `Space Saving` buffer (if not already present) according to the `Space Saving` algorithm. Let $c = 1/\varepsilon > 1/\Theta$ be the size of `Space Saving`. In the following $\mathtt{SS}(\ell)$ denote the content of `Space Saving` at discrete time ℓ. Let $\mathcal{H}$ be the set of the heavy hitters, that is the set of all items i whose probability of occurrence p_i verifies $p_i \geq \Theta$. We necessarily have $|\mathcal{H}| < c$. It has been proven in [13] that at time m all the items $i \in \mathcal{H}$ are present in $\mathtt{SS}(m)$. It follows that for every $i \in \mathcal{H}$ there exists a random time $T_i \leq m$ such that item i is inserted for the last time in `SS` at time T_i (*i.e.,* item i was not present in the `Space Saving` buffer at time $T_i - 1$ and thus replaces another item), and is never removed from `SS` afterwards. More precisely, we have

$$T_i = \inf\{\ell \geq 1 \mid i \in \mathtt{SS}(r) \text{ for every } r = \ell, \ldots, m\}.$$

For every $j \in \mathtt{SS}(\ell)$, we denote by $V_j(\ell)$ the counter value of item j at discrete time ℓ. For every $i \in \mathcal{H}$, by definition of T_i item i is not in the buffer at time $T_i - 1$, thus we have,

$$V_i(T_i) = \min\{V_j(T_i - 1),\ j \in \mathtt{SS}(T_i - 1)\} + 1,$$

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

$$V_i(\ell) = V_i(\ell - 1) + 1_{\{Y_\ell = i\}}.$$

Putting together these results, we get

$$V_i(m) = \min\{V_j(T_i - 1),\ j \in \mathtt{SS}(T_i - 1)\} + 1 + N_i(T_i + 1, m),$$

where $N_i(s, s') = \sum_{\ell=s}^{s'} 1_{\{Y_\ell = i\}}$ is the number of occurrences of item i in the stream between discrete instants s and s'. It is easy to see that $N_i(s, s')$ has a binomial distribution with parameters $s' - s + 1$ and p_i. This relation can also be written as

$$\begin{aligned} 0 &\leq V_i(m) - N_i(1, m) \\ &= \min\{V_j(T_i - 1),\ j \in \mathtt{SS}(T_i - 1)\} + 1 - N_i(1, T_i). \end{aligned}$$

Note that this also implies that $N_i(1, T_i) \leq \min\{V_j(T_i - 1),\ j \in \mathtt{SS}(T_i - 1)\} + 1$.

Since for every $\ell \geq 1$, we have $\min\{V_j(\ell),\ j \in \mathtt{SS}(\ell)\} \leq \ell/c$, we obtain, taking the expectations,

$$0 \leq \mathbb{E}\{V_i(m)\} - mp_i \leq \mathbb{E}\{T_i - 1\}/c + 1 - \mathbb{E}\{T_i\}p_i.$$

This leads to

$$0 \leq \mathbb{E}\{V_i(m)\} - mp_i \leq 1 - 1/c + \mathbb{E}\{T_i\}(1/c - p_i).$$

Since, by the Space Saving algorithm, $V_i(m)$ represents the estimated frequency of any heavy hitter i of the input stream, and since $1/c - p_i \leq 0$ and $c = 1/\varepsilon$, we finally get [2]

$$0 \leq \mathbb{E}\{V_i(m)\} - mp_i \leq 1 - \varepsilon.$$

□

Using the Markov inequality and the fact that $p_i \geq 1/c$, we get for all $\varepsilon > 0$,

$$\mathbb{P}\left\{ \frac{V_i(m) - mp_i}{mp_i} \geq \varepsilon \right\} \leq \frac{c - 1}{\varepsilon c m p_i} \leq \frac{1 - \varepsilon}{\varepsilon^2 m}.$$

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

$$\lim_{m \longrightarrow \infty} \mathbb{P}\left\{ \frac{V_i(m) - mp_i}{mp_i} \geq \varepsilon \right\} = 0.$$

THEOREM 4.2 (ACCURACY OF DKG). *DKG provides an $(1 + \Theta)$-optimal mapping for key grouping using $\mathcal{O}(k\mu \log m + \log n)$ bits of memory in the learning phase and $\mathcal{O}(k\mu \log n)$ 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 $v_{\overline{h}}$, after having removed from these counters the weight of the heavy hitters (*cf.,* Line 15 of Listing 3.2). The value of every counter $v_{\overline{h}}[i]$, $1 \leq i \leq \mu k$, is equal to the sum of the exact frequencies of all the received items t such that $\overline{h}(t) = i$, 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 $v_{\overline{h}}[i]$. We have

$$X_i = \sum_{j=1}^{n} f_j \mathbb{1}_{\{\overline{h}(j) = i\}} - \sum_{j \in \mathcal{H}} \hat{f}_j \mathbb{1}_{\{\overline{h}(j) = i\}},$$

[2]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.

where, as defined above, $\hat{f}_j$ represents the frequency of heavy hitter j 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,

$$\mathbb{E}\{X_i\} = \sum_{j=1}^{n} \mathbb{E}\left\{f_j \mathbb{1}_{\{\overline{h}(j)=i\}}\right\} - \sum_{j\in\mathcal{H}} \mathbb{E}\left\{\hat{f}_j \mathbb{1}_{\{\overline{h}(j)=i\}}\right\} \leq \frac{m - \sum_{j\in\mathcal{H}} \mathbb{E}\left\{\hat{f}_j\right\}}{k\mu}.$$

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

$$\mathbb{E}\{X_i\} \leq \frac{m - \sum_{j\in\mathcal{H}} f_j}{k\mu} \leq \frac{m_{sp}}{k\mu}.$$

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 $v_{\overline{h}}$. 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 (1)) of the mapping provided by DKG with respect to the optimal one. Let L^{DKG} denote the load vector induced by the mapping provided by DKG and L^d denote the optimal one. The error Δ introduced by DKG is given by

$$\Delta = \lambda(L^{DKG}) - \lambda(L^d) = \frac{\max_{i=1,\ldots,k} L_i^{DKG} - \max_{i=1,\ldots,k} L_i^d}{\overline{L}}.$$

The analysis of the expected value of Δ is split into three cases. We do not make any assumption regarding the statistical moment of the input distribution and thus in a conservative way, we set $\Theta \leq 1/k$. Recall that any item whose relative frequency (*i.e.,* probability of occurrence) is greater than or equal to Θ is considered as an heavy hitter.

Case 1 $\exists i \in \mathcal{H}$ such that $f_i > \overline{L} = m/k$. Let ℓ denote the item with the largest frequency. We have $\lambda(L^d) = f_\ell k/m$ since the optimal solution cannot do better than scheduling item ℓ alone to a single instance. We also have $\lambda(L^{DKG}) = f_\ell k/m$ since from above $\hat{f}_\ell$ ε-approximates f_ℓ with probability close to 1 (by construction, $\varepsilon \gg 1/m$), and thus `scheduling` also isolates item ℓ to a single instance. Thus the error Δ introduced by DKG in Case 1 is null.

Case 2 $\forall i \in \mathcal{H}$, we have $\Theta m \leq f_i \leq \overline{L}$. By construction of `scheduling` (that maps each tuple into the less loaded instance in the decreasing frequency order), and by the fact that the value of each counter of $v_{\overline{h}}$ is equal to $m_{sp}/(k\mu)$, we have that $\arg\max_{i=1,\ldots,k} L_i^{DKG}$ corresponds to the last instance chosen by `scheduling`. 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 instance that has been chosen previously by `scheduling` without being the smallest loaded one). Let L' be the load vector before this last mapping. We have in average

$$\sum_{i=1}^{k} L'_i = \sum_{i=1}^{k} L_i^{DKG} - \frac{m_{sp}}{k\mu}.$$

Thus, one can easily check that $\min_{i=1,\ldots,k} L'_i$ cannot exceed $(m_{sp} - m_{sp}/(k\mu))/k = (k\mu - 1)m_{sp}/k^2\mu$, leading to

$$\max_{i=1,\ldots,k} L_i^{DKG} - \overline{L} = \left(\min_{i=1,\ldots,k} L'_i + \frac{m_{sp}}{k\mu}\right) - \frac{m}{k} \leq \frac{(\mu+1)m_{sp}}{k\mu}. \quad (2)$$

Moreover, by assumption, the distribution of the items in the stream follows a Zipfian distribution with an unknown parameter α (see Section 2). This means that, for any $i \in \{1,\ldots,n\}$, $p_i = 1/(i^\alpha H_{n,\alpha})$ where $H_{n,\alpha}$ is the n-th generalized harmonic number, that is $H_{n,\alpha} = \sum_{j=1}^{n} 1/j^\alpha$ (we assume without loss of generality that items are ranked according to their decreasing probability). We have

$$m_{sp} = m - \sum_{i\in\mathcal{H}} f_i = m\left(1 - \frac{H_{|\mathcal{H}|,\alpha}}{H_{n,\alpha}}\right). \quad (3)$$

Recall that $|\mathcal{H}| = |\{i \in \{1,\ldots,n\} \mid p_i \geq \Theta\}|$. Thus, we get

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

From Relations 2 and 3 we have

$$\max_{i=1,\ldots,k} L_i^{DKG} - \overline{L} \leq \frac{(\mu+1)\left(H_{n,\alpha} - H_{|\mathcal{H}|,\alpha}\right)m}{k\mu H_{n,\alpha}} \leq \frac{(\mu+1)m}{k\mu}.$$

Given the fact that $\max_{i=1,\ldots,k} L_i^d \geq \overline{L} = m/k$ by definition, the expected error introduced by DKG is given by

$$\mathbb{E}\{\Delta_2\} \leq \frac{\max_{i=1,\ldots,k} L_i^{DKG} - \overline{L}}{\overline{L}} \leq \frac{\mu+1}{\mu}.$$

Case 3 $\forall i$ in the input stream of length m, we have $f_i < \Theta m$. In that case, there are no heavy hitters, and thus $m_{sp} = m$. This case is handled similarly as Case 2 by considering that the largest potential item has a frequency equal to $\Theta m - 1$. Thus $\min_{i=1,\ldots,k} L'_i$ cannot exceed $(m - (m/k\mu + \Theta m - 1)/k = (k(m\mu + \Theta m\mu - \mu) - m)/(k^2\mu)$, leading to

$$\max_{i=1,\ldots,k} L_i^{DKG} - \overline{L} = \left(\min_{i=1,\ldots,k} L'_i + \frac{m}{k\mu} + \Theta m - 1\right) - \frac{m}{k} \leq \frac{(\Theta\mu k + 1)m}{k\mu}.$$

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

$$\mathbb{E}\{\Delta_3\} \leq \frac{\max_{i=1,\ldots,k} L_i^{DKG} - \overline{L}}{\overline{L}} \leq \frac{1 + \mu k\Theta}{\mu}.$$

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

$$\mathbb{E}\{\Delta\} \leq \max\{\mathbb{E}\{\Delta_1\}, \mathbb{E}\{\Delta_2\}, \mathbb{E}\{\Delta_3\}\} \leq \max\{0, \frac{1+\mu}{\mu}, \frac{1+\mu k\Theta}{\mu}\}.$$

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

$$\mathbb{E}\{\Delta\} \leq 1 + \Theta \quad (4)$$

□

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance obtainable by using DKG to perform key grouping. We will first describe the general setting used to run the tests and will then discuss results obtained through simulations (Section 5.2) and implementing DKG as a custom grouping function in Apache Storm (Section 5.3).

5.1 Setup

Evaluation metrics — To characterize how unevenly work is distributed we take into account two well known [16] load balance metrics:

Percentage of imbalance **(λ)** measures 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 1.

Load standard deviation **(σ)** measures if the load of the instances tends to be closer (small values of σ) or farther (large values of σ) from the mean. It's the average number of tuples that each instance handles in excess or deficiency with respect to the mean.

CPU Load measured in Hz.

Throughput measured in tuples processed per second.

While λ measures a global performance of the system, σ provides a per-instance perspective. 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.

Datasets — In our tests we considered both synthetic and real datasets. For the synthetic datasets we generated streams of integer values (items) representing the values of the tuples. We consider streams made of $100,000$ tuples each 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 evaluation was performed on the last $20,000$ tuples of the stream (validation set). 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 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). However, in most of the plots presented in the following the worst performance figures among all the runs are reported as well.

As a real dataset we considered data provided by the DEBS 2013 Grand Challenge [6]. 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 denote this data set as *DEBS trace* in the following.

Algorithm	a	b
O-apx	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

Table 1: Linear regression coefficients for the plots in Figure 2

Tested solutions — To compare the performance of DKG, we considered four key grouping algorithms:

Optimal approximation **(O-apx)** is an execution of the `scheduling` algorithm introduced in Section 3.1 with complete information on the distribution of the stream portion used as validation set. From this point of view we consider O-apx performance 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.

Modulo returns $t \mod k$ and represents a common implementation for key grouping in many SPSs; in particular it is the implementation of field grouping in Apache Storm.

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. Since Modulo often performs similarly to Universal, in general we omit the former in the results.

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 does not hold. In all tests discussed in the following 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.

5.2 Simulation Results

Imbalance (λ) with respect to the number of instances (k) — Figure 2 shows the imbalance as a function of k for DKG, Universal, O-apx 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

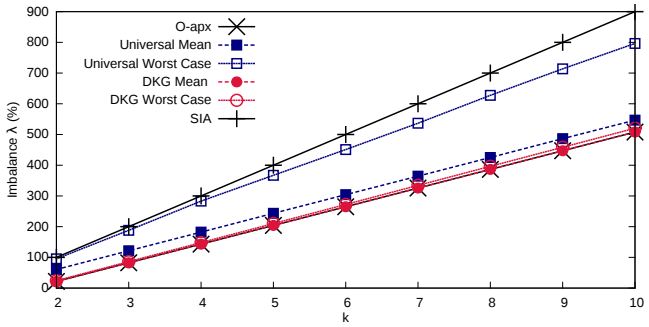


Figure 2: Imbalance (λ) as a function of k with Zipf-2 ($\Theta = 0.1$ and $\mu = 2$)

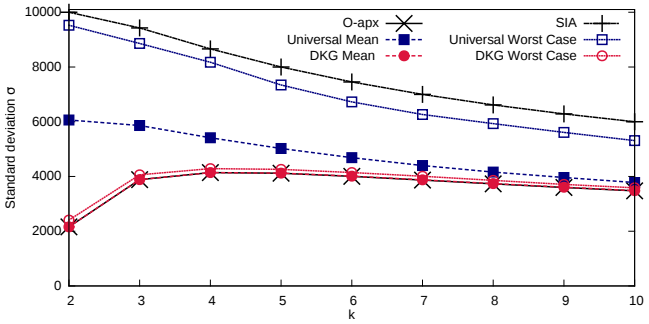


Figure 3: Standard deviation (σ) as a function of k with Zipf-2 ($\Theta = 0.1$ and $\mu = 2$)

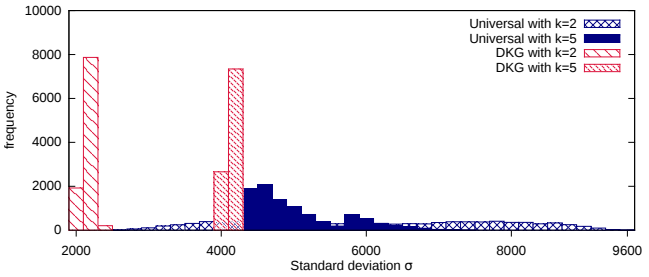


Figure 4: Standard deviation (σ) distribution for Zipf-2 ($\Theta = 0.1$ and $\mu = 2$)

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 1), justifies the monotonically increasing behavior of all curves.

The most important observation we can do by looking at the graph is that DKG mean matches O-apx and how little is the gap between DKG worst case and O-apx. This means that DKG is not susceptible to unlucky choices made with the used hash functions. Furthermore, there is a noteworthy gap between DKG and Universal mean values, while Universal worst case is closer to SIA than to O-apx. 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 1 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 O-apx while DKG worst case has a value of a close to O-apx; besides, Universal mean has not the same b of O-apx, while DKG mean matches O-apx. Notice that the results shown for DKG mean fall into the first case of the proof of Theorem 4.2 (*cf.,* Section 4), 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. Due to space limitations we omit the plots 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 3.2).

Standard deviation (σ) with respect to the number of instances (k) — Figure 3 shows the standard deviation for the same configuration of Figure 2. The main difference with respect to Figure 2 is in the trend of the plots. σ is normalized by k and represents the average number of unbalanced items per instance. As such, σ for SIA decreases with increasing values of k, as the number of unbalanced items is fixed. On the other hand, σ for O-apx 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, σ for SIA decreases faster than for O-apx, thus increasing k shrinks the gap between SIA and O-apx, as well as reducing DKG's gain.

Figures 4 shows the distribution of σ values achieved by DKG and Universal for $k \in \{2, 5\}$ in the same runs of Figure 3. Each bar represents how many times a σ 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: the values are concentrated in a small interval of σ for both values of k. Conversely, Universal has a large variance, for instance it spans from 2800 to 9600 items for $k = 2$.

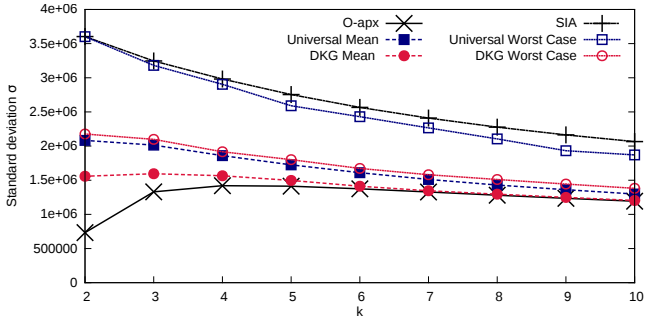


Figure 5: Standard deviation (λ) as a function of k with Zipf-1 ($\Theta = 0.1$ and $\mu = 2$) with quadratic cost

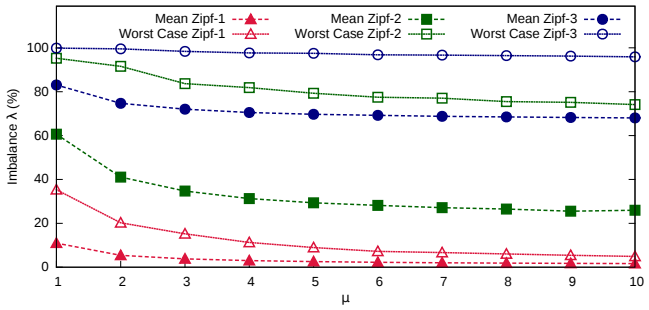


Figure 6: DKG imbalance as a function of μ ($\Theta = 1.0$ and $k = 2$)

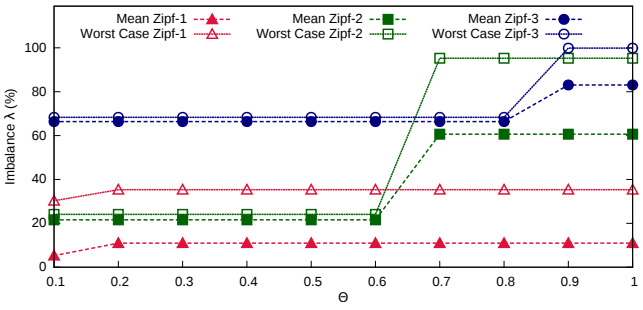


Figure 7: DKG imbalance as a function of Θ ($\mu = 1.0$ and $k = 2$)

Impact of the cost function — Figure 5 shows the standard deviation as a function of k for DKG, Universal, O-apx and SIA, with Zipf-1 ($\Theta = 0.1$ and $\mu = 2$), considering a quadratic cost function. In other words, the load induced by tuple j on an instance is f_j^2. The `scheduling` algorithm can take this into account by squaring the load value of the buckets. This approach overestimates the load of the buckets containing sparse items (*i.e.,* the buckets associated with $v_{\bar{h}}$), considering the square of the whole bucket load, instead of the sum of the squares of the load induced by the tuples hitting the bucket. This rough approximation has a sizeable impact on DKG performance. DKG mean does not stick to O-apx for $k < 5$ and there is a sizeable gap between DKG mean and worst case. However, Universal in average does only slightly better than DKG worst case, while Universal worst case is pretty close to SIA. Notice that we can reduce the overestimation on the sparse items by increasing μ (thus decreasing the value of each $v_{\bar{h}}$ entry). Alternatively, we could estimate the average number of distinct tuples in each $v_{\bar{h}}$ entry (this can be easily achieved leveraging [3]) and use this value to estimate more precisely the load induced by each bucket.

Impact of μ — Figure 6 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 `scheduling` 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 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 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 `scheduling` algorithm does not have enough freedom to improve the sparse item mapping.

We do not show O-apx in Figure 7, 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 O-apx imbalance. With Zipf-3 and Zipf-2, the theoretical analysis (*cf.,* Section 4) 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 8 show DKG mean and worst case standard deviations (σ) 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 8a and 8b confirm that, as for imbalance, with non- or lightly-skewed distributions, μ drives the performance, while Θ has a negligible impact. Figure 8c, 8d, 8c and 8d 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.

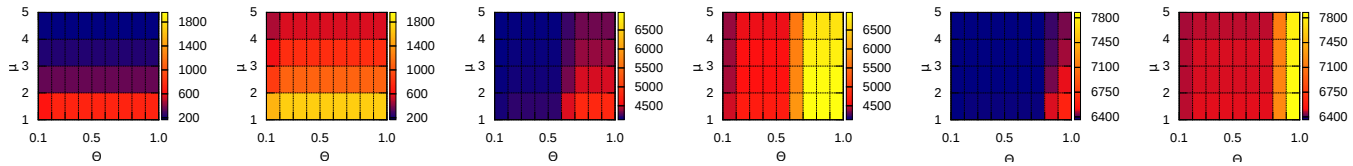
Impact of the training set's size m — Figure 9 shows DKG mean and worst case standard deviation (σ) for Zipf-1, Zipf-2 and Zipf-3 as a function of the training set's size m ($\Theta = 0.1$, $\mu = 2$ and $k = 2$).

This plot shows that DKG learns quite fast: less than 100 items for Zipf-3, less than 1.000 items for Zipf-2 and less than 10.000 items for Zipf-1 are enough for its data structures to converge. DKG learns faster with strongly skewed distribution, i.e. it learns faster and produces a close to optimal balancing when the distribution is potentially more harmful.

5.3 Prototype

To evaluate the impact of DKG on real applications we implemented it[3] as a custom grouping function within the Apache Storm [17] framework. More in details, DKG implements the CustomStreamGrouping interface offered by the Storm API, defining two methods: prepare() and

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



(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 8: DKG standard deviation (σ) as a function of both Θ and μ for Zipf-1, Zipf-2 and Zipf-3 ($k = 5$)

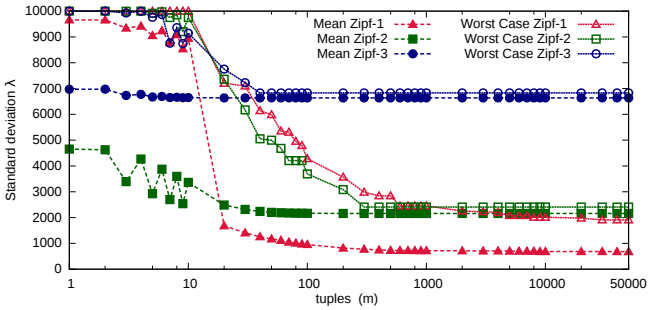


Figure 9: DKG standard deviation (σ) as a function of the training set's size m ($\Theta = 0.1$, $\mu = 2$ and $k = 2$)

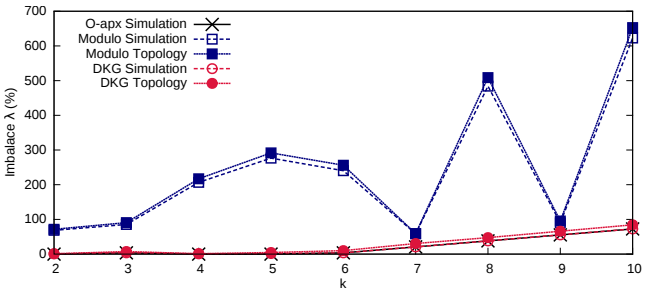


Figure 10: Imbalance (λ) for the *DEBS trace* as a function of k ($\Theta = 0.1$, $\varepsilon = 0.05$ and $\mu = 2$)

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 3.1). In particular they take Θ, ε, μ, k, the size of the learning set m, and the function KEY(t) as parameters. KEY(t) 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.,* Listings 3.1 and 3.2): 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 [6]: 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.

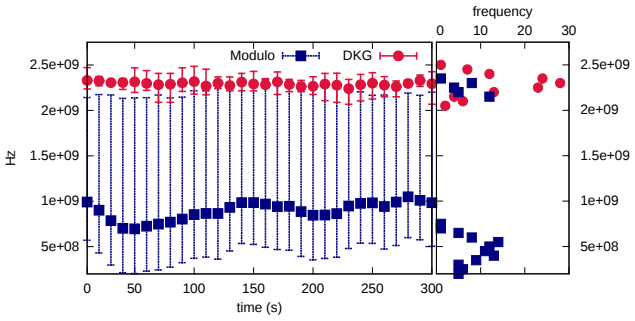
The test topology is made of a source (*spout* in Storm jargon) and an processing element (*bolt*) with k instances (*tasks*). To avoid I/O to be a bottleneck for our tests, the source store the whole sensors reading data file in memory. For each reading, it emits 4 tuples (one for each granularity level) towards the processing element 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 processing element 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 (Modulo) and the DKG prototype.

Imbalance (λ) with respect to the number of instances (k) — Figure 10 shows the imbalance as a function of k for O-apx, Modulo and DKG with the *DEBS trace*. For Modulo 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 processing element instances). We can notice that, for both DKG and Modulo, results from the tested topology closely match those provided by the simulations. DKG sticks very close to O-apx for all k, and, except for $k \in \{7, 9\}$, largely outperforms Modulo. Furthermore, we can clearly see the hardly predictable behavior faced when using a vanilla hash function. For $k \in \{2, \ldots, 5\}$, O-apx 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 O-apx, as well as DKG, imbalance grows with k.

Cpu usage and throughput — Figure 11a shows the cpu usage (Hz) over time (300 seconds of execution) for DKG and Modulo 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 Modulo, while the mean cpu usage for Modulo is much smaller. This was expected as there is a large resource underutilization with Modulo. Figure 11b 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.



(a) Timeline (b) Distribution

Figure 11: Cpu usage (Hz) for 300s of execution ($\Theta = 0.1$, $\varepsilon = 0.05$, $\mu = 2$ and $k = 4$)

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 Modulo 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 Modulo some instance (in particular 3) are underused, leaving most of the load on fewer instances (in particular 1).

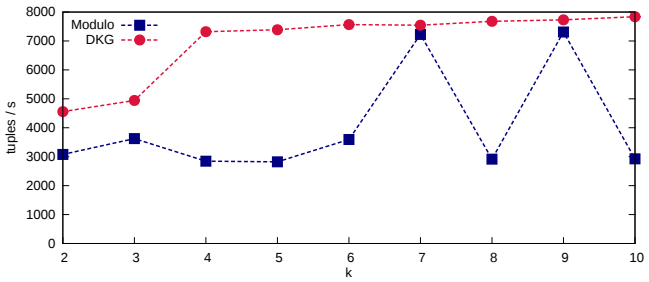


Figure 12: Throughput (tuples/s) for the *DEBS trace* as a function of k ($\Theta = 0.1$, $\varepsilon = 0.05$ and $\mu = 2$)

This improvement in resource usage translates directly into a larger throughput and reduced execution time, as clearly shown by Figure 12; in particular, in our experiments, DKG delivered 2× the throughput of Modulo for $k \in \{4, 5, 6, 8, 10\}$.

6. RELATED WORK

Load balancing in distributed computing is a well known problem that has been extensively studied since the 80s. 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 the the last year new interest in this field was spawned by research on how key grouping can be improved targeting better load balancing.

Gedik in [8] proposed a solution that is close to our from several perspectives. In particular, he proposes to use a *lossy counting* algorithm to keep track of the most frequent items that are then explicitly mapped to target sub-streams. 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. Our experimental evaluation showed that this change in the approach provides a marked improvement in performance.

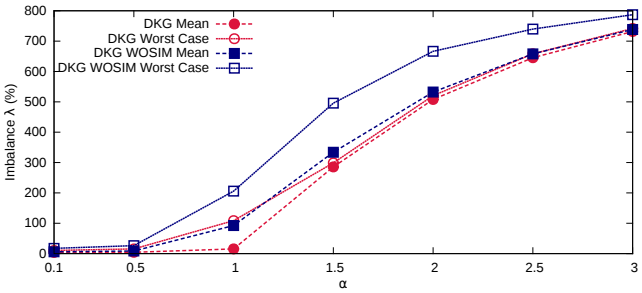


Figure 13: Imbalance (λ) as a function of α ($\Theta = 0.1$, $\varepsilon = 0.05$ $\mu = 2$ and $k = 10$)

In particular, Figure 13 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 $\bar{h}$ 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. As the curves show, DKG always outperforms the WOSIM version for all tested values of α. One further important difference between this work and [8] 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.

Nasir et al. in [15] 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 t to the less loaded of the two sub-streams associated with t; this is roughly equivalent to working on a modified input stream where each value has half the load of the original stream. Differently from our solution, the algorithm proposed in [15] 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 [15] to provide even better performance.

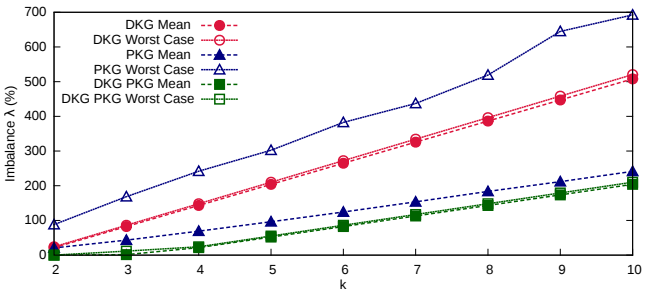


Figure 14: Imbalance (λ) as a function of k ($\Theta = 0.1$, $\mu = 2$ and $\alpha = 2$)

Figure 14 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 `scheduling` algorithm provides two different mappings for each heavy hitter. When hashing a heavy hitter, DKG PKG will return 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 [15]. 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 performances, stressing again how our solution is able to provide stable performance irrespective of exogenous factors.

7. CONCLUSIONS

In this paper we presented a novel key grouping solution for parallel and distributed stream processing frameworks. We theoretically proved that our solution is able to provide $(1 + \Theta)$-optimal load balancing when applied on skewed input streams characterized by a Zipfian distribution (on the key used for grouping). Through an extensive experimental evaluation we showed that the proposed solution outperforms standard approaches to field grouping (*i.e.,* modulo and universal hash functions) and how this positively impacts a real stream processing application.

In the near future we plan to further improve our proposal by solving two issues; firstly, our solution, as proposed in this paper, works as an offline algorithm that must learn on a dataset before providing the desired mapping. While this is an acceptable deployment approach for many application domains, it would not fit a scenario where the value distribution in the input stream changes over time; from this point of view we think that the approach employed in [8] to adapt the mapping at runtime, taking into account the cost of operator migration, could be integrated in our solution as well. A second issue that we plan to tackle is related to how the learning phase can be executed in a parallel fashion; a trivial solution would be to share the local state of the algorithm learning phase executed on parallel instances to build a global state that can then be used to calculate the correct final mapping. While this solution is technically feasible, further investigation could possibly lead to more efficient approaches with a reduced synchronization overhead.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th annual ACM Symposium on Theory of computing (STOC)*, 1996.

[2] E. Anceaume, Y. Busnel, and B. Sericola. Uniform node sampling service robust against collusions of malicious nodes. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[3] 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.

[4] 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)*, volume 1, 1999.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1), 2004.

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

[7] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2), 1985.

[8] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4), 2014.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2), 1969.

[10] S. Guha and A. McGregor. Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5), 2009.

[11] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct element problem. In *Proceedings of the Symposium on Principles of Databases (PODS)*, 2010.

[12] R. Kumar, J. Novak, P. Raghavan, and A. Tomkins. On the bursty evolution of blogspace. *World Wide Web*, 8(2), 2005.

[13] 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.

[14] Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc., 2005.

[15] 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.

[16] 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.

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