When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing

Muhammad Anis Uddin Nasir[♯], Gianmarco De Francisci Morales[◇], Nicolas Kourtellis[‡], Marco Serafini[◇]

[♯]*KTH Royal Institute of Technology, Stockholm, Sweden*
[‡]*Telefonica Research, Barcelona, Spain*
[◇]*Qatar Computing Research Institute, Doha, Qatar*
anisu@kth.se, gdfm@acm.org, nicolas.kourtellis@telefonica.com, mserafini@qf.org.qa

***Abstract*—Carefully balancing load in distributed stream processing systems has a fundamental impact on execution latency and throughput. Load balancing is challenging because real-world workloads are skewed: some tuples in the stream are associated to keys which are significantly more frequent than others. Skew is remarkably more problematic in large deployments: having more workers implies fewer keys per worker, so it becomes harder to "average out" the cost of hot keys with cold keys.**

We propose a novel load balancing technique that uses a heavy hitter algorithm to efficiently identify the hottest keys in the stream. These hot keys are assigned to $d \geq 2$ choices to ensure a balanced load, where d is tuned automatically to minimize the memory and computation cost of operator replication. The technique works online and does not require the use of routing tables. Our extensive evaluation shows that our technique can balance real-world workloads on large deployments, and improve throughput and latency by 150% and 60% respectively over the previous state-of-the-art when deployed on Apache Storm.

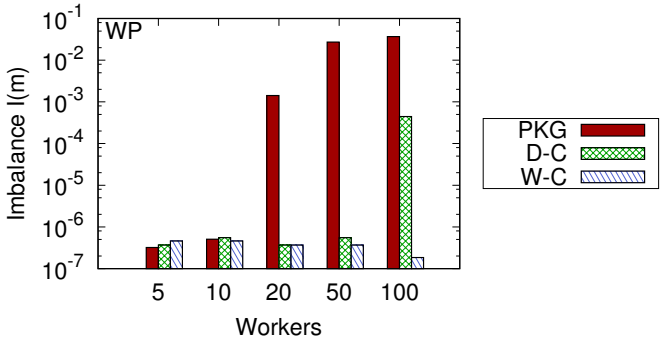

Fig. 1: Imbalance due to skew is more challenging to handle at large scale. On this dataset from Wikipedia, PKG is able to achieve low imbalance only at small scales, while the techniques proposed in this paper, D-Choices (D-C) and W-Choices (W-C), fare better at large scales.

I. Introduction

Stream processing is currently undergoing a revival,[1] mostly as a result of recent advances in Distributed Stream Processing Engines (DSPEs) such as Storm,[2] S4,[3] and Samza.[4] The capabilities of modern engines even allow to take a unified approach to streaming and batch computations, such as in Flink [2] and Google Dataflow [1]. These technologies promise high efficiency combined with high throughput at low latency.

The versatility achieved by DSPEs has enabled developers to build a plethora of streaming applications. Examples range from continuous query processing to online machine learning, from ETL pipelines to recommender systems [8, 9]. While the data streams that flow through these systems originate from disparate application domains, they have one common denominator: many real-world data streams are highly skewed. Skew is one of the main obstacles to scaling distributed computation, mainly because it creates load imbalance, and the primary focus of this work.

To better appreciate the problem and its context, we first need to understand how DSPE applications are structured. Streaming applications are commonly represented in the form of directed graphs that represent the *data flow* of the application. The vertices of the graph are data transformations (operators), and its edges are channels that route data between operators. The data flowing along these edges is a stream, represented as a sequence of *tuples*, each associated with a *key*. To achieve high throughput, modern distributed engines leverage data parallelism by creating several instances of an operator (workers) that process independent sub-streams. These sub-streams are created by partitioning the incoming stream across instances of the downstream operator via *grouping schemes*.

Among the existing grouping schemes, *key grouping* is often used for stateful operators. This scheme guarantees that all the tuples with the same key are processed by the same operator instance, akin to MapReduce. Key grouping is commonly implemented via hashing, which is known to cause load imbalance across multiple workers, especially in presence of skew in the input key distribution [3, 13].

Some solutions to balance load in DSPEs in presence of skew have recently been proposed [15, 10, 17, 14]. In particular, randomized techniques based on the "power of two choices" [3], such as Partial Key Grouping [15], have found their way into mainstream DSPEs such as Apache Storm, thanks to their combined simplicity and effectiveness. Unlike key grouping, PKG associates each key to two possible operator instances, and selects the least loaded of the two whenever a tuple for a given key must be processed. Techniques leveraging the power of two choices can adapt dynamically to workloads, they do not require the DSPE to support operator migration, and route keys to workers using hash functions rather than routing tables, which can become very large and must be replicated consistently across multiple upstream operators.

[1] http://radar.oreilly.com/2015/04/a-real-time-processing-revival.html
[2] http://storm.apache.org
[3] http://incubator.apache.org/s4
[4] http://samza.apache.org

In this paper, we tackle a serious scalability problem of these techniques: using only two choices is not sufficient in very large deployments. The source of the problem is that these techniques require the amount of skew in the distribution of keys to be inversely proportional to the size of the deployment. For example, PKG requires the fraction of load generated by the most frequent key to be less than the combined ideal (i.e., average) load supported by two workers. As the size of the deployment grows, the average fraction of load per worker decreases, so the fraction of load generated by the most frequent key must also decrease for the approach to be effective. Intuitively, the imbalance exists because at larger scales it becomes increasingly difficult to average out the load among multiple keys. Figure 1 shows this problem occurring while processing a real-world dataset from Wikipedia. While at small scales (5 or 10) PKG manages to keep the system balanced, at larger scales (20, 50 or 100 workers) the imbalance grows significantly and gets close to 10% (Section II formally defines the notion of imbalance).

The scalability problem is exacerbated for workloads that are extremely skewed, as for example in graph streaming applications where some vertices are much more popular than others.[5] Consider, for example, that under a Zipf distribution (which models the occurrence of words in natural language) with exponent $z = 2.0$, the most frequent key represents nearly 60% of the occurrences, and thus PKG is unable to guarantee ideal load balance for any deployment larger than 3 workers, as shown in the original analysis [15].

The straightforward alternative to using two choices is *shuffle grouping*, a grouping scheme which distributes the messages in a round-robin fashion irrespective of the keys. Shuffle grouping is the best choice for stateless operators. However, it comes with a high cost when used for stateful operators: it is necessary to replicate the state associated with keys potentially on each worker, since each worker could have to process each key. The memory overhead of shuffle grouping can thus become directly proportional to the number of workers used, therefore hindering scalability.

We propose two new streaming algorithms for load balancing, called *D-Choices* and *W-Choices*, that enable DSPEs to scale to large deployments while supporting stateful operators. As shown in Figure 1, *D-Choices* and *W-Choices* achieve very low imbalance (smaller than 0.1%) even on very large clusters. Our algorithms reduce the memory footprint by estimating the minimum number of choices per key for cases where two are not enough. The intuition behind our techniques is that, for highly skewed inputs, most of the load on the system is generated by a handful of "hot" keys. Thus, while the long tail of low-frequency keys can be easily managed with two choices, the few elements in the head needs additional choices. Using more than two choices requires answering two main questions: (i) *"which are the keys that need more than two choices?"* and (ii) *"how many choices are needed for these keys?"*.

Our proposed method identifies and partitions the most frequent keys in the distribution, which are the main reason for load imbalance at scale. It uses a streaming algorithm to identify "heavy hitter" keys that require a larger number of workers, and allows them a number of choices $d \geq 2$ which is minimal yet sufficient for load balancing. For the rest of the keys it uses the standard two choices. To find the heavy hitters, we leverage the well-known SpaceSaving algorithm [12], and its recent generalizations to the distributed setting [5].

The threshold used to find the heavy hitters is an important parameter in determining the memory cost of the approach, since these keys are mapped to a larger number of workers. Our evaluation examines a range of potential thresholds and finds that a single threshold is sufficient for most settings. This threshold requires only a very small number of keys to be assigned to more than two workers. In fact, if the stream is very skewed the hot keys are only a handful, otherwise if the stream is not very skewed, two choices are sufficient for most keys, even for some of the most frequent ones.

Our two algorithms differ in the number of choices provided for the hot keys. W-Choices allows as many choices as the number of workers. Since the heavy hitter algorithm selects only few hot keys, this solution performs well if the state associated with each key is not extremely large. D-Choices is a more sophisticated strategy that tries to estimate the minimum number of choices d for hot keys that is sufficient to ensure a balanced load. The estimation considers several factors, such as the relative frequency of the hot keys, and the overlaps among the sets of choices of different keys. In addition, since the mapping of keys to workers responsible for processing them is typically done using hash functions, we need to take into account their collision probability. Our analysis provides a lower bound on the expected value of d which is very accurate. We verify our bound via an extensive evaluation that compares the value of d obtained from the analysis with the optimal value of d found empirically in several simulated settings.

We evaluate our methods on real workloads and show that they allow scaling to large deployments, whereas existing approaches using only two choices do not scale to 50 or more workers. This increased scalability translates into an improvement of up to 150% in throughput, and up to 60% in latency, over PKG when deployed on a real Apache Storm cluster. The improvements are even larger over key grouping: 230% in throughput and 75% in latency.

In summary, this paper makes the following contributions:

- We propose two new algorithms for load balancing of distributed stream processing systems; the algorithms are tailored for large-scale deployments and suited to handle the skew in real-world datasets;
- We provide an accurate analysis of the parameters that determine the cost and effectiveness of the algorithms;
- An extensive empirical evaluation[6] shows that our proposal is able to balance the load at large scale and in the presence of extreme skew;
- The reduced imbalance translates into significant improvements in throughput and latency over the state-of-the-art when deployed on Apache Storm.

[5] See http://konect.uni-koblenz.de/statistics/power for a repository of common graph datasets and their power law exponent.

[6] Code available at https://github.com/anisnasir/SLBSimulator and https://github.com/anisnasir/SLBStorm

II. Preliminaries

A. The Dataflow Model

Most modern DSPEs run on clusters of machines that communicate by exchanging messages. These DSPEs run streaming applications that are usually represented in the form of a directed acyclic graph (DAG), which consists of a set of vertices and a set of edges. The vertices in the DAG (called *operators*) are a set of data transformations that are applied on an incoming data stream, whereas the edges are data channels between vertices which connect the output of one operator to the input of the following one. In order to achieve high performances, DSPEs run multiple *instances* of these operators and let each instance process a portion of the input stream, i.e., a *sub-stream* (Figure 2).

For simplicity, we specify our notation by considering a single stream between a pair of operators, i.e., a single edge connecting two vertices. Given a stream under consideration, let the set of upstream operator instances be called *sources* and be denoted as $\mathcal{S}$, and the set of downstream operator instances be called *workers* and denoted as $\mathcal{W}$, and their sizes be $|\mathcal{S}| = s$ and $|\mathcal{W}| = n$. Although the number of messages is unbounded, at any given time we denote the number of messages in the input stream as m.

The input stream is fed into a DSPE as a sequence of messages $\langle t, k, v \rangle$, where t is a timestamp, k is a key, and v is a value. The keys in the messages are drawn from a (skewed) distribution $\mathcal{D}$ over a finite key space $\mathcal{K}$, i.e., some of the keys appear more frequently than others. Let p_k be the probability of drawing key k from $\mathcal{D}$. We define the *rank* of a key $r(k)$ as the number of keys that have a higher or equal probability of appearing according to $\mathcal{D}$ (with ties broken arbitrarily). We identify the key by its rank in $\mathcal{D}$, and therefore, by definition the probability of keys being drawn decreases as follows: $p_1 \geq p_2 \geq \ldots \geq p_{|\mathcal{K}|}$, with $\sum_{k \in \mathcal{K}} p_k = 1$.

B. Stream Partitioning

Sub-streams are created by partitioning the input stream via a *stream partitioning* function $P_t : \mathcal{K} \rightarrow \mathbb{N}$, which maps each key in the key space to a natural number, at a given time t. This number identifies the worker responsible for processing the message (we identify the set $\mathcal{W}$ of workers with a prefix of the naturals). Each worker is associated to one or more keys, and keys associated to the same worker are said to be *grouped* together by a *grouping scheme*. Henceforth, we consider three already existing grouping schemes:

- ***Key Grouping* (KG)**: This scheme ensures that messages with the same key are handled by the same downstream worker. KG is usually implemented via hashing.
- ***Shuffle Grouping* (SG)**: This scheme evenly distributes messages across the available downstream workers, thus ensuring ideal load balance. SG is usually implemented via round-robin selection of workers.
- ***Partial Key Grouping* (PKG)** [15]: The scheme ensures that messages with the same key are processed by at most two workers. It is implemented by using two separate hash functions that produce two candidate workers. The message is then routed to the least loaded of the two workers.

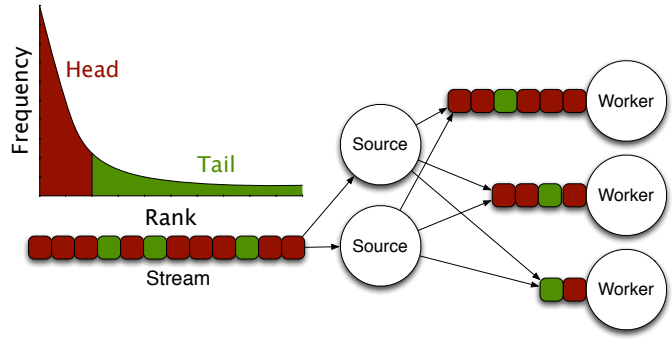

Fig. 2: Imbalance generated by high skew. Most of the load is due to the head of the distribution.

Load imbalance. We use a definition of *load* similar to others in the literature (e.g., Flux [18] and PKG [15]). At time t, the load of a worker w is the fraction of messages handled by the worker up to t:

$$L_w(t) = \frac{|\{\langle \tau, k, v \rangle \mid P_\tau(k) = w \wedge \tau \leq t\}|}{m}.$$

We define the *imbalance* at time t as the difference between the maximum and the average load of the workers:

$$I(t) = \max_w L_w(t) - \operatorname*{avg}_w L_w(t), \quad w \in \mathcal{W}.$$

Our goal is to find a stream partitioning function P_t that minimizes the maximum load $L(m)$ of the workers, which is equivalent to minimizing the imbalance $I(m)$ since the average load $1/n$ is independent of P_t.

III. Solution Overview

The solution we explore in this paper revolves around the idea of detecting at runtime the hot keys or "heavy hitters", which comprise the *head* of the key distribution, and treat them differently from the *tail* when it comes to mapping them on workers for processing. In particular, we track the head $\mathcal{H}$ of the key distribution in a distributed fashion across sources, and then split the keys belonging to the head across $d \geq 2$ workers.

To instantiate this solution, we need to define the size of the head (i.e., the cardinality of $\mathcal{H}$) and the number of choices per key that should be used (i.e., the value of d). These two quantities jointly affect the load on the workers. Given a skewed key distribution, there will be a combination of number of keys considered $|\mathcal{H}|$ and number of choices d, that minimize the overall load imbalance across workers, while adding the smallest possible overhead. However, such an ideal setup is difficult to reach at runtime, in a real system that analyzes streams of data with an unknown key distribution.

In order to tackle this multi-objective problem, we first define a threshold θ that separates the head from the tail of the distribution. Then, we study the problem of finding d, the number of choices for the head that produces a low load imbalance. Next, we discuss these two issues.

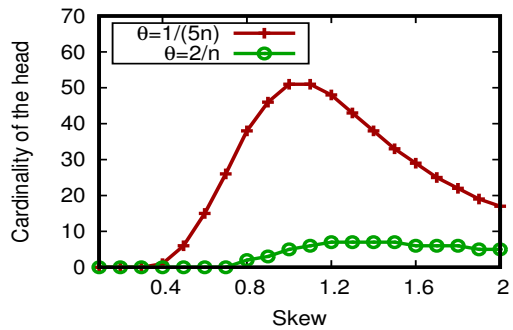

Fig. 3: Number of keys in the head of the distribution for 50 and 100 workers for a Zipf distribution with $z = 0.1 \ldots 2.0$, $|\mathcal{K}| = 10^4$ and $m = 10^7$.

A. Finding the Head

One important issue in our solution is how to define the threshold θ that splits the head of the distribution in the input stream from the tail. Formally, we define the head $\mathcal{H}$ as the set of keys that are above a frequency threshold:

$$\mathcal{H} = \{k \in \mathcal{K} \mid p_k \geq \theta\}.$$

As discussed in the introduction, algorithms based on the power of two choices make assumptions on the maximum skew in the key distribution. For instance, PKG assumes an upper bound on the frequency of the most frequent key. This bound is inversely proportional to the total number of choices in the system, i.e., to the total number of workers in our setting. We want to select the threshold θ such that the head includes all the keys violating the assumptions of PKG. Therefore, we focus on the space where PKG falls short, according to its analysis.

The first bound provided by the analysis states that if $p_1 > 2/n$, then the expected imbalance at time m is lower-bounded by $(\frac{p_1}{2} - \frac{1}{n})m$, which grows linearly with m [15]. This imbalance exists because the load generated by the most frequent key exceeds the capacity of two workers. Therefore, clearly, all the keys that exceed the capacity of two workers belong to the head, and thus, $\theta \leq 2/n$. The second bound in the analysis states that if $p_1 \leq 1/5n$, then the imbalance generated by PKG is bounded, with probability at least $1 - 1/n$. Therefore, we do not need to track keys whose frequency is smaller than this bound, and so $\theta \geq 1/5n$.

Given these theoretical bounds, we evaluate thresholds in the range $1/5n \leq \theta \leq 2/n$. As shown in Section V, most values in this range provide a satisfactory level of imbalance, so we pick a conservative value of $1/5n$ as the default. Albeit conservative, this value still results in a small cardinality for the head. Even if keys follow a uniform distribution where all keys have probability $1/5n$, the head will be comprised only by $5n$ keys. In more skewed distributions, the size is substantially smaller. Figure 3 shows the cardinality of $\mathcal{H}$ for several Zipf distributions, for the two extremes of the range of θ.

These numbers show that we we need to pay the price of added replication only for a small number of keys. In order to make our proposal work completely online, we use a streaming algorithm to track the head and its frequency at runtime. In particular, we use the SpaceSaving algorithm [12], a counter-based algorithm for heavy hitters which can be extended to work in a distributed setting [5].

B. Number of Choices for each Head Key

Recall that we want to manage the head of the distribution differently from the tail, in order to achieve better load balance across workers. In our proposed solution, both the head and tail make use of a Greedy-d process, but apply different d per key. Next we outline the basic principle behind this process.

Greedy-d. Let $\mathcal{F}_1, \ldots, \mathcal{F}_d$ be d independent hash functions that map $\mathcal{K} \to [n]$ uniformly at random. We define the Greedy-d process as follows: at time t, the t-th message (whose key is k_t) is placed on the worker with minimum current load among the candidate ones $\mathcal{F}_1(k_t), \ldots, \mathcal{F}_d(k_t)$, i.e., $P_t(k_t) = \operatorname{argmin}_{i \in \{\mathcal{F}_1(k_t), \ldots, \mathcal{F}_d(k_t)\}} L_i(t)$, where ties are broken arbitrarily.

Our method leverages this process to map keys to workers by hashing a key with a set of d hash functions, and assigning the key to the least loaded of the d available options. In our solution, the tail has $d = 2$ choices as in PKG. However, the head can have $d \geq 2$, depending on the frequency of the keys.

We propose two different algorithms, depending on the value of d (see Algorithm 1). These algorithms differ in their memory and aggregation overhead.

- ***D-Choices:*** This algorithm adapts to the particular distribution of frequencies in the head, and uses a Greedy-d process to handle the head, with $d < n$. We show analytically how to pick d in the following section.
- ***W-Choices:*** This algorithm allows $n = |\mathcal{W}|$ workers for the head. Conceptually, it is equivalent to setting $d \gg n \ln n$ for a Greedy-d scheme, since this condition guarantees that there is at least one item in each bucket when using an ideal hash function. In practice, there is no need to hash the keys in the head, and the algorithm can simply pick the least loaded among all workers.

The pseudocode of the algorithm used by senders to determine the destination worker for their output messages is reported in Algorithm 1. UPDATESPACESAVING invokes the heavy hitters algorithm of [12] and returns the set of the current heavy hitters. FINDOPTIMALCHOICES determines the number of choices for heavy hitter keys as explained in Section IV-A. MINLOAD selects the id of the worker with lowest load among the ones passed as arguments. The load is determined based only on local information available at the sender, as in [15]. The algorithm uses d different hash functions $\mathcal{F}_i$.

We also test our algorithms against a simple but competitive baseline, which has the same overhead as W-Choices:

- ***Round-Robin:*** This algorithm performs round-robin assignment for the head across all available n workers. In comparison to W-Choices, this algorithm assigns the head to workers in a load-oblivious manner.

IV. ANALYSIS

In this section, we analyze the D-Choices algorithm in order to find the number of choices d that enables achieving load balance. In addition, we discuss the memory requirement and overhead for the different algorithms.

Algorithm 1: Stream partitioning algorithm.

```
upon message m = ⟨k, v⟩
    H ← UpdateSpaceSaving(k)
    d ← 2 // Default as in PKG
    if k ∈ H then
        if D-CHOICES then
            d ← FindOptimalChoices()
        else if W-CHOICES then
            d ← n
    w ← MinLoad(F_1(k), ..., F_d(k))
    send(w, m)
```

A. Setting d for D-Choices

We first determine how to express d analytically and then discuss how the function FindOptimalChoices computes a solution for that expression.

For this analysis, we assume that the key distribution $\mathcal{D}$ and the threshold θ are given. Recall that we aim at minimizing d in order to reduce the memory and aggregation overheads created by using a large number of choices, which are roughly proportional to $d \times |\mathcal{H}|$. Therefore, we would like to know which is the smallest number of choices d that can lead to imbalance below some threshold ϵ.

More formally, our problem can be formulated as the following minimization problem:

$$\underset{d}{\text{minimize}} \quad f(d; \mathcal{D}, \theta) = d \times |\mathcal{H}_{\mathcal{D},\theta}|\,, \tag{1}$$

$$\text{subject to} \quad \underset{d}{\mathbb{E}}\left[I(m)\right] \leq \epsilon\,. \tag{2}$$

In the problem formulation, we emphasize that $\mathcal{H}$ is a function of $\mathcal{D}$ and θ. Because both $\mathcal{D}$ and θ are given, the objective function of the problem reduces to simply minimizing d. Additionally, we employ a constraint on the expected imbalance generated when using the Greedy-d process for $\mathcal{H}$.

In order to attack the problem, we would like to express the imbalance $I(m)$ as a function of d. However, the load on each worker L_i, from which the imbalance is computed, is a random variable that depends on the key assignment and the specific distribution, both of which are random. The Greedy-d process adapts dynamically to the load conditions, so it is hard to express the load analytically as a function of d. Instead, we perform a *lower bound* analysis on the expected load, and find analytically what are the *necessary* conditions to obtain a feasible solution. We then show empirically in Section V that the optimal values for d are very close to this lower bound.

Proposition 4.1: If the constraint of the optimization problem in Eqn. (2) holds, then:

$$\sum_{i \leq h} p_i + \left(\frac{b_h}{n}\right)^d \sum_{h < i \leq |\mathcal{H}|} p_i + \left(\frac{b_h}{n}\right)^2 \sum_{i > |\mathcal{H}|} p_i \leq b_h \left(\frac{1}{n} + \epsilon\right), \tag{3}$$

$$\text{where } b_h = n - n\left(\frac{n-1}{n}\right)^{h \times d}, \quad \forall k_h \in \mathcal{H}.$$

Proof: Let L_w be the load of worker w. By definition, $\text{avg}_w(L_w) = 1/n$ since we consider normalized load that sums to 1 over all workers. Since $I(m) = \max_w(L_w) - \text{avg}_w(L_w)$ by definition, and therefore $L_w - 1/n \leq I(m)$, we can rewrite Eqn. (2) as:

$$L_w - \frac{1}{n} \leq \epsilon, \quad \forall w \in \mathcal{W}. \tag{4}$$

To continue our analysis, we would like to express the load on each worker as a function of the key distribution, however the load is a random variable that depends on the mix of keys that end up on the given worker, and the dynamic conditions of the other workers. Nevertheless, while we cannot say anything about the load of a single worker, we can lower bound the (expected) load of a set of workers that handle a given key.

Let $\mathcal{W}_i = \{\mathcal{F}_1(k_i), \ldots, \mathcal{F}_d(k_i)\}$ be the set of workers that handle k_i. Because workers are chosen via hashing, some of the d choices might collide, and thus it might be that $|\mathcal{W}_i| \leq d$. Let $b = |\mathcal{W}_i|$ be the expected size of $\mathcal{W}_i$.

We now derive a lower bound on the cumulative (asymptotic) load on this set of workers. This load consists of p_i (by definition), plus the load generated by any key k_j whose choices *completely collide* with k_i, i.e., $\mathcal{W}_j \subseteq \mathcal{W}_i$. Note that keys that partially collide with k_i may also add up to the load of the workers in $\mathcal{W}_i$, but we can ignore this additional load when deriving our lower bound. The completely colliding keys k_j may come from the head or from the tail. We take into account these two cases separately and express a lower bound on the expected cumulative load on $\mathcal{W}_i$ as:

$$\mathbb{E}\left[\sum_{w \in \mathcal{W}_i} L_w\right] \geq p_i + \left(\frac{b}{n}\right)^d \sum_{\substack{j \leq |\mathcal{H}| \\ j \neq i}} p_j + \left(\frac{b}{n}\right)^2 \sum_{j > |\mathcal{H}|} p_j. \tag{5}$$

This equation expresses the frequency of k_i (p_i) plus the (expected) fraction of the distribution that *completely* collide with k_i. A key in the head has d independent choices, while a key in the tail has only 2 independent choices. Each choice has a probability of colliding with $\mathcal{W}_i$ of b/n where b is the size of $\mathcal{W}_i$, as we assume ideal hash functions.

We now use this lower bound to express a necessary condition for Eqn. (4), and thus for Eqn. (3). By summing L_w for all $w \in \mathcal{W}_i$, Eqn. (4) implies the following condition:

$$\mathbb{E}\left[\sum_{w \in \mathcal{W}_i} L_w\right] \leq \frac{b}{n} + b \cdot \epsilon. \tag{6}$$

A *necessary* condition for this inequality to hold can be obtained by using the lower bound of Eqn. (5), which results in the following expression:

$$p_i + \left(\frac{b}{n}\right)^d \sum_{\substack{j \leq |\mathcal{H}| \\ j \neq i}} p_j + \left(\frac{b}{n}\right)^2 \sum_{j > |\mathcal{H}|} p_j \leq \frac{b}{n} + b \cdot \epsilon. \tag{7}$$

We can generalize the previous condition by requiring it to hold on the set of workers handling any prefix $p_1, \ldots, p_h$ of h keys of the head. This results in a larger number of necessary conditions and thus in a potentially tighter bound, depending on the distribution of keys. Similarly to Eqn. (6), we obtain a set of constraints, one for each prefix of length h:

$$\mathbb{E}\left[\sum_{w\in\bigcup_{i\leq h}\mathcal{W}_i} L_w\right] \leq \frac{b_h}{n} + b_h \cdot \epsilon, \quad \forall k_h \in \mathcal{H}, \tag{8}$$

where $b_h = \mathbb{E}[|\bigcup_{h\leq i}\mathcal{W}_h|]$ is the expected number of workers assigned to a prefix of the head of length h. We can use once again the lower bound of Eqn. (5) to obtain a necessary condition for the previous expression to hold as follows:

$$\sum_{i\leq h} p_i + \left(\frac{b_h}{n}\right)^d \sum_{h<i\leq|\mathcal{H}|} p_i + \left(\frac{b_h}{n}\right)^2 \sum_{i>|\mathcal{H}|} p_i \leq \tag{9}$$
$$\leq \mathbb{E}\left[\sum_{w\in\bigcup_{i\leq h}\mathcal{W}_i} L_w\right] \leq \frac{b_h}{n} + b_h \cdot \epsilon, \quad \forall k_h \in \mathcal{H}.$$

We derive b_h in Appendix A as follows:

$$b_h = \mathbb{E}\left[\left|\bigcup_{i\leq h}\mathcal{W}_i\right|\right] = n - n\left(\frac{n-1}{n}\right)^{h\times d}. \tag{10}$$

Finally, after substituting b_h we obtain the final set of bounds of Eqn. (3), one for each prefix of the head $\mathcal{H}$. ■

Solving analytically for d is complex, so we use an alternative approach to implement FINDOPTIMALCHOICES. We simply start from $d = p_1 \times n$, which is a simple lower bound on d that derives from the fact that we need $p_1 \leq d/n$, and increase d until all constraints are satisfied. In practice, the tight constraints are the ones for $h = 1$, the most frequent key, and for $h = |\mathcal{H}|$, when considering the whole head. The latter, is especially tight for very skewed distributions, where the head represents a large fraction of the total load. In this case, we need $b_h \approx n$ with high probability, but, as well know, this happens only when $h \times d \gg n \ln n$. Given that $\max h$ is small, as the distribution is skewed and $\mathcal{H}$ has small cardinality, we need a large d. However, having $d \geq n$ is not sensible, so when this condition is reached, the system can simply switch to the W-Choices algorithm.

B. Memory Overhead

We now turn our attention to the cost of the algorithms under study, as measured by the memory overhead required to achieve load balance. Henceforth, we refer mainly to the memory overhead, however note that when splitting a key in d separate partial states, if reconciliation is needed, there is also an aggregation cost proportional to d. The two costs are proportional, so we focus our attention only on the first one.

Overhead on Sources. All three algorithms have the same cost on the sources. The sources are responsible for forwarding messages to the workers. Each source maintains a load vector storing local estimations of the load of every worker. This estimation is based only on the load each source sends, and it is is typically a very accurate approximation of the actual global load of workers obtained considering all sources [15]. Storing the load vector requires $\mathcal{O}(n)$ memory. Thus, the total memory required for estimating the load of workers at each source is $s \times n$. These costs are the same for all algorithms, even though

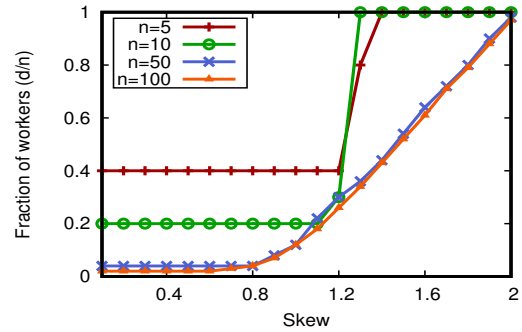

Fig. 4: Fraction of workers used by D-Choices for the head as a function of skew $z \in \{0.1, \ldots, 2.0\}$. Values for $|\mathcal{K}| = 10^4$, $m = 10^7$, and $\epsilon = 10^{-4}$.

Round-Robin uses the load information only for the tail. Alongside the load estimation, each source runs an instance of the SpaceSaving algorithm to estimate the frequency of the head keys in the input streams. The SpaceSaving algorithm requires $\mathcal{O}(1)$ memory and $\mathcal{O}(1)$ processing time per message.

Overhead on Workers. Without loss of generality, let us assume the memory required to maintain the state for each key is unitary. Recall the set of choices available per key:[7]

- For $d = 2$, each key is assigned to at most two workers, as in PKG. The upper bound on the memory requirement to store $|\mathcal{K}|$ unique items is $2 \times |\mathcal{K}|$.
- For $2 < d < n$, each key is assigned to at most d workers, as in D-Choices. The upper bound on the memory requirement is $d \times |\mathcal{H}| + 2 \times |\mathcal{K} \setminus \mathcal{H}|$.
- For $d \gg n \ln n$, each key is assigned to all the n workers, as in W-Choices and Round-Robin. The upper bound on the memory requirement is $n \times |\mathcal{H}| + 2 \times |\mathcal{K} \setminus \mathcal{H}|$.

From these formulas, it is clear that D-Choices always has a smaller overhead than the other algorithms, although it is larger than PKG. But how much is the overhead in theory? By using Inequality (3), we can find the appropriate value for d, in case the distribution is known apriori. Figure 4 shows the overhead of D-Choices for the required d, as a fraction of the number of workers, as a function of the skew. We vary the exponent of a Zipf distribution $z \in \{0.1, \ldots, 2.0\}$, with $|\mathcal{K}| = 10^4$ unique keys, and $m = 10^7$ messages. The plot shows that D-Choices reduces the number of workers compared to W-Choices and Round-Robin. Especially at larger scales, $n = 50$ and $n = 100$ workers, d is always smaller than n.

Lastly, Figure 5 and Figure 6 show the estimated memory used by D-Choices and W-Choices with respect to PKG and SG respectively. All the algorithms (including PKG) need an aggregation phase, so the comparison is fair. We estimate the memory for PKG and SG using $\text{mem}_{\text{PKG}} = \sum_{k\in\mathcal{K}} \min(f_k, 2)$ and $\text{mem}_{\text{SG}} = \sum_{k\in\mathcal{K}} \min(f_k, n)$ respectively.

We derive three main observations from the results of Figure 5 and Figure 6. First, both D-Choices and W-Choices require an amount of memory similar to PKG when the number of workers is small. However, this behavior changes when the number of workers increases. For instance, for 100 workers

[7] Ignoring that some keys in the tail have frequency smaller than n.

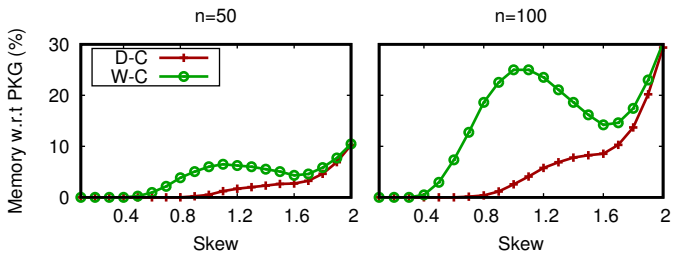

Fig. 5: Memory overhead for D-Choices (D-C) and W-Choices (W-C) with respect to PKG as a function of skew, for different number of workers $n \in \{50, 100\}$. Values for $|\mathcal{K}| = 10^4$ and $\epsilon = 10^{-4}$. In the worst case, D-C and W-C use at most 30% more memory than PKG, and D-C uses considerably less memory than W-C in most cases.

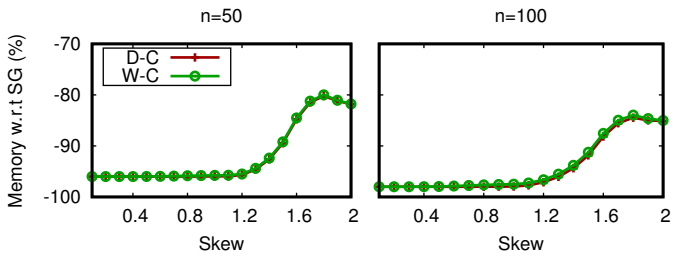

Fig. 6: Memory overhead for D-Choices (D-C) and W-Choices (W-C) with respect to SG as a function of skew, for different number of workers $n \in \{50, 100\}$. Values for $|\mathcal{K}| = 10^4$ and $\epsilon = 10^{-4}$. In the worst case, D-C and W-C still use 80% less memory than SG.

W-Choices requires up to 25% more memory compared to PKG. Second, D-Choices requires less memory than W-Choices when the skew is moderately high (note that ratios lower than 10^{-2} are not plotted). Nevertheless, both curves approach the same memory overhead at very high skew.

Note that the bump in Fig. 5 primarily depends on the size of the head (as shown in Fig 3) and how a Zipf distribution with finite support is defined. Intuitively, as the skew grows larger, initially more keys pass the frequency threshold θ, but later only a few keys become dominant (to the limit, only one key is in the head with $p_1 \to 1$).

Third, the overhead compared to SG is negligible, especially at larger scales. Both D-Choices and W-Choices require only a fraction of the cost of shuffle grouping, fulfilling one of the two desiderata for our solution. The next section shows both proposed algorithms also achieve low imbalance, which is their main purpose.

V. Evaluation

In Section IV-B we have derived an analysis of the memory overhead of our proposed techniques. We now assess their load balancing performance using both simulations and a real deployment. In so doing, we answer the following questions:

Q1: How to decide the threshold θ that defines the head of the distribution?

Q2: How close is the estimated value of d for D-C to an empirical optimum?

TABLE I: Summary of the datasets used in the experiments: number of messages, number of keys, and probability of the most frequent key (p_1).

Dataset	Symbol	Messages	Keys	$p_1(\%)$
Wikipedia	WP	22M	2.9M	9.32
Twitter	TW	1.2G	31M	2.67
Cashtags	CT	690k	2.9k	3.29
Zipf	ZF	10^7	10^4,10^5,10^6	$\propto \frac{1}{\sum x^{-z}}$

TABLE II: Notation for the algorithms.

Symbol	Algorithm	Head vs. Tail
D-C	D-Choices	Specialized on head
W-C	W-Choices	
RR	Round-Robin	
PKG	Partial Key Grouping	Treats all keys equally
SG	Shuffle Grouping	

Q3: How do the proposed algorithms compare in terms of load imbalance?

Q4: What is the overall effect of proposed method on the throughput and latency of a real DSPE?

A. Experimental Setup

Datasets. Table I summarizes the datasets used. We use three real-world datasets, extracted from Wikipedia and Twitter. The Wikipedia dataset (WP)[8] is a log of the pages visited during a day in January 2008 [21]. Each visit is a message and the page's URL represents its key. The Twitter dataset (TW) is a sample of tweets crawled during July 2012. Each tweet is split into words, which are used as the key for the message. Additionally, we use a Twitter dataset that is comprised of tweets crawled in November 2013. The keys for the messages are the *cashtags* in these tweets. A *cashtag* is a ticker symbol used in the stock market to identify a publicly traded company preceded by the dollar sign (e.g., $AAPL for Apple). This dataset allows to study the effect of drift in the skew of the key distribution. Finally, we generate synthetic datasets (ZF) with keys drawn from Zipf distributions with exponent in the range $z \in \{0.1, \ldots, 2.0\}$, and with a number of unique keys in the range $|\mathcal{K}| \in \{10^4, 10^5, 10^6\}$.

Simulation. We process the datasets by simulating the simplest possible DAG, comprising one set of sources ($\mathcal{S}$), one set of workers ($\mathcal{W}$), and one intermediate partitioned stream from sources to workers. The DAG reads the input stream from multiple independent sources via shuffle grouping. The sources forward the received messages to the workers downstream via the intermediate stream, on which we apply our proposed grouping schemes. Our setting assumes that the sources perform data extraction and transformation, while the workers perform data aggregation, which is the most computationally expensive part of the DAG, and the focus of the load balancing.

[8] http://www.wikibench.eu/?page_id=60

TABLE III: Default parameters for the algorithms.

Parameter	Description	Values
n	Number of workers	5, 10, 20, 50, 100
s	Number of sources	5
ϵ	Imbalance tolerance (D-Choices)	10^{-4}
θ	Threshold defining the head	$\frac{2}{n}, \ldots, \frac{1}{8n}$

Algorithms. Table II defines the notations used for different algorithm, while Table III defines values of different parameters that we use for the experiments. For most of the experiments we use load imbalance as defined in Section II as a metric to evaluate the performance of the algorithms. Unlike the algorithms in Table II, other related load balancing algorithms [18, 7, 23, 4, 6] require the DSPE to support operator migration. Many DSPEs, such as Apache Storm, do not support migration, so we omit these algorithms from the evaluation.

B. Experimental Results

Q1. To find the right threshold θ, we run an experiment with the W-Choices algorithm. We use W-C because it is potentially the best one in terms of imbalance, as it is able to use the entire set of workers to distribute the load. We compare it with Round-Robin, which has the same cost in terms of memory. We vary the threshold by halving it, starting from $2/n$ down to $1/8n$. For this experiment, we use the ZF dataset in order to verify the effect of skew.

Figure 7 shows the load imbalance as a function of skew, for different number of workers. For both algorithms, reducing the threshold (i.e., considering a larger head) reduces the imbalance, and increasing the skew (i.e., larger z) increases the imbalance, as expected. However, W-C achieves ideal load balance for any threshold $\theta \leq 1/n$, while RR exhibits a larger gradient of results. This behavior is more evident at larger scales, where RR starts generating imbalance even under modest levels of skew.

The difference in results between the two algorithms is entirely due to the fact that W-C uses a load-sensitive algorithm to balance the head, which adapts by taking into consideration also the load of the tail. Conversely, RR balances the head and tail separately. As a result, while the head is perfectly balanced, the tail has less flexibility in how to be balanced.

As additional evidence, Figure 8 shows the workload generated by head and tail separately for PKG, W-C, and RR. To ease visual inspection, we show the case for $n = 5$; the other cases follow a similar pattern. This plot is obtained for $\theta = 1/8n$, i.e., the lowest threshold in our experiments. As expected, the most frequent key in PKG overloads the two workers responsible for handling it, thus giving rise to high load imbalance. Instead, W-C mixes the head and tail across all the workers, and achieves ideal load balance. Finally, RR splits the head evenly across the workers, but falls short of perfect balance, as represented by the horizontal dotted line.

In summary, even for low thresholds, W-C behaves better than RR, and a threshold of $1/n$ is enough for W-C to achieve low imbalance in any setting of skew and scale. In the following experiments we set the threshold to its default value $\theta = 1/5n$. This conservative setting gives the algorithms the largest amount of freedom in setting d, and guarantees that any imbalance is due to the choice of d rather than the choice of θ, which determines the cardinality of $\mathcal{H}$.

Q2. In this experiment, we evaluate the parameter d for D-C. We compare how far the d value as computed by the D-C algorithm is from an empirically-determined optimal value. To find the optimum, we execute the Greedy-d algorithm with all the possible values of d, i.e., $2, \ldots, n$ and find the minimum d that is required to match the imbalance achieved by W-C.

Figure 9 shows this optimum in comparison to the value of d for D-C as computed by the algorithm described in Section IV. The results clearly show that D-C is very close to the optimal d, thus supporting our analysis. In all cases, D-C uses a d slightly larger than the minimum, thus providing good load balance at low cost.

Q3. We now examine the performance of the proposed algorithms in terms of load imbalance. We use the ZF datasets to control for the skew, and additionally, we evaluate the proposed algorithms on several real-world datasets, i.e., WP, TW and CT. We compare D-C and W-C to RR and PKG as baselines.

Figure 10 shows the average imbalance as a function of skew for several possible settings of number of workers and number of keys. The number of keys does not influence the imbalance, thus suggesting that $|\mathcal{K}| = 10^4$ is a large enough number to give the algorithms enough freedom in their placement, for these particular distributions. On the other hand, skew and scale clearly play an important role. The problem becomes harder as both z and n increase. W-C is the best performer in terms of imbalance, as it manages to keep it constantly low irrespective of the setting. D-C and RR perform similarly, and both manage to keep the imbalance much lower than PKG for larger values of skew and scale. However, recall that D-C does so at a fraction of the cost of RR.

Figure 11 shows similar results for the real-world datasets (we omit RR as it is superseded by W-C). All algorithms perform equally well when the number of workers is small. Instead, at larger scales ($n = 20, 50, 100$) PKG generates higher load imbalance compared to D-C and W-C. As expected, W-C provides better load balance compared to D-C. However, recall that for D-C the level of imbalance is also determined by the tolerance ϵ. As each source executes the algorithm independently, in the worst case we should expect an imbalance of $s \times \epsilon$, plotted as a horizontal dotted line.

The Twitter cashtags dataset (CT) behaves differently from TW and WP. This dataset is characterized by high concept drift, that is, the distribution of keys changes drastically throughout time. This continuous change poses additional challenges to our method, especially for the heavy hitters algorithm that tracks the head of the distribution. Overall, the dataset is harder to balance for all methods, however the relative performance of the various algorithms is still as expected. D-C works better than PKG at larger scales, and W-C provides even better load balance.

Finally, Figure 12 allows to observe the behavior of the proposed algorithms over time. This experiment enables us to study the effect of drift in the key distribution. We compare the

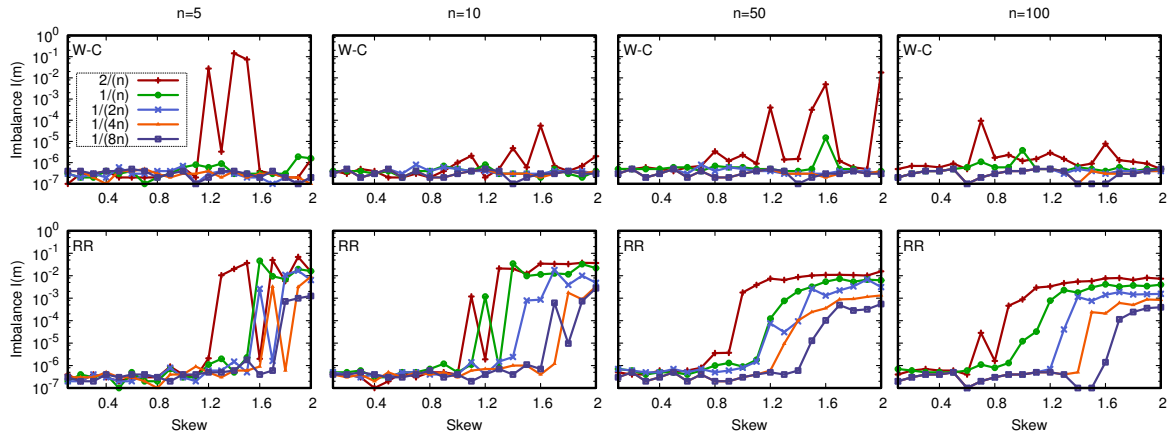

Fig. 7: Load imbalance as a function of skew z, for different thresholds θ, for W-C and RR. W-C provides a better load balance than RR at high skew, especially for larger scale deployments, while having the same memory overhead. For this experiment, the stream has $|\mathcal{K}| = 10^4$ unique keys, and $m = 10^7$ messages.

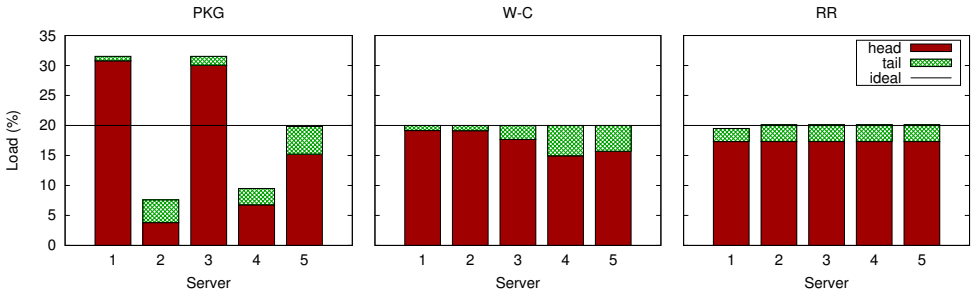

Fig. 8: Load generated by head and tail when using PKG, W-C, and RR. The horizontal dotted line represents the ideal even distribution of the load ($1/n$). For this experiment, the threshold is set to $\theta = 1/8n$, and the stream has $m = 10^7$ messages with $|\mathcal{K}| = 10^4$ drawn from a Zipf distribution with exponent $z = 2.0$.

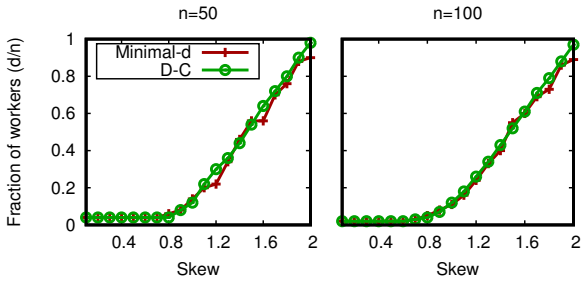

Fig. 9: Comparison of the d value for D-C to the minimum value of d required to match the imbalance of W-C (Min-d). The value computed by D-C is very close to the minimum for all settings.

imbalance over time for PKG, D-C, and W-C on the real-world datasets. The plot shows the imbalance for several settings, and confirms our previous observations. Larger scales are challenging for PKG, but are handled well by D-C and W-C, and the concept drift in the CT dataset makes the task harder. Nevertheless, overall the imbalance remains relatively stable throughout the simulation.

Q4. To quantify the impact of our proposed techniques, D-C and W-C, on a real DSPE, we implement them in Apache Storm and deploy them on a cluster. We compare them with the standard groupings: KG, SG, and PKG. The Storm cluster consists of 9 machines (1 master and 8 slaves), where each machine has 16 VCPUs. Each slave machine has 16 execution slots (for a total of $8 \times 16 = 128$ slots in the cluster).

We use a simple streaming application that consist of two operators: sources that generate the input stream, and workers that perform aggregation on the stream. Concretely, we use 48 sources and 80 workers. The sources generate the stream from a Zipf distribution with skew $z \in \{1.4, 1.7, 2.0\}$, $|\mathcal{K}| = 10^4$, and $m = 2 \times 10^6$. This aggregation-like application is representative of many data mining algorithms, such as computing statistics for classification, or extracting frequent patterns. To emulate some CPU consumption, we add a fixed delay of 1ms to the processing of each message (this delay represents $1/10$-th of a disk seek[9]). We choose this value for the delay to bring the application to the saturation point of the cluster. We report overall throughput and end-to-end latency by taking the average over multiple iterations.

Figure 13 shows the throughput for the experiment. As expected, KG achieves the lowest throughput among all the groupings. PKG performs better than KG, however it is not able to match SG, and pays a price for the imbalance. Instead, our proposed techniques (D-C and W-C) are able to match the throughput of SG (at a fraction of its cost). As expected, increasing the skew makes the problem harder for both KG and PKG. In the best case, D-C and W-C achieve a throughput 1.5 times higher than PKG, and 2.3 times higher than KG.

[9] http://brenocon.com/dean_perf.html

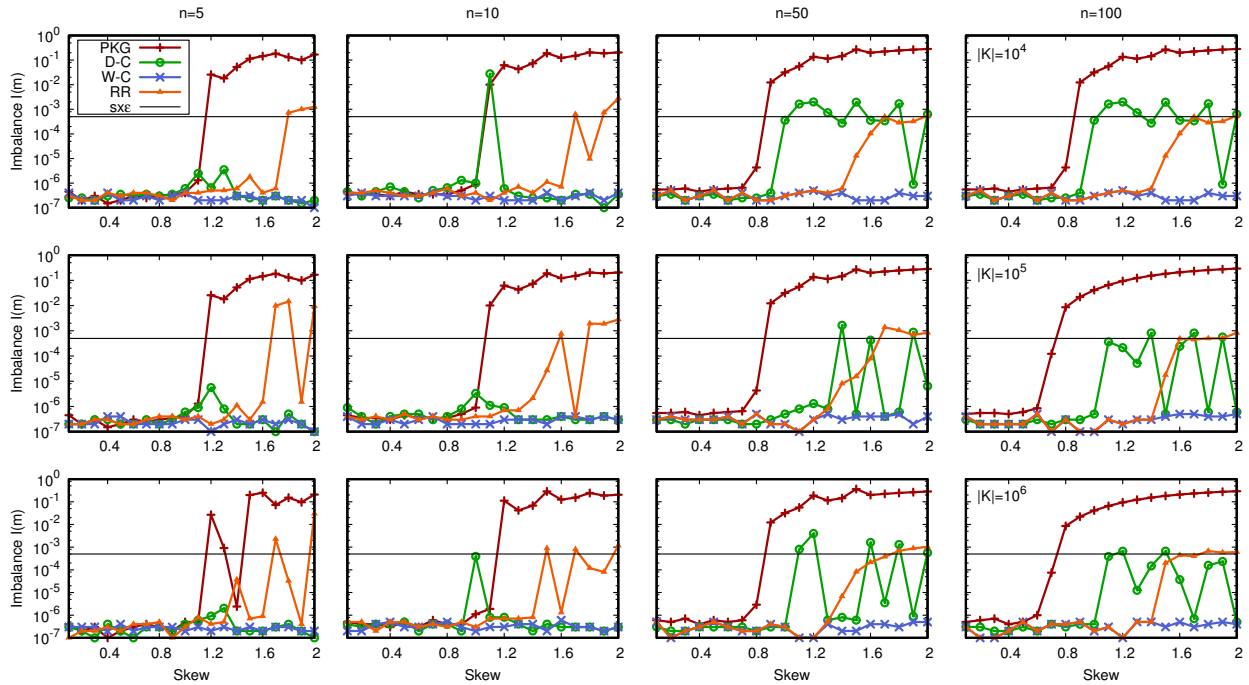

Fig. 10: Load imbalance on the ZF datasets for PKG, D-C, W-C, and RR as a function of skew. The plots are shown for several combinations of settings of number of workers n and number of unique keys $|\mathcal{K}|$ (number of messages $m = 10^7$).

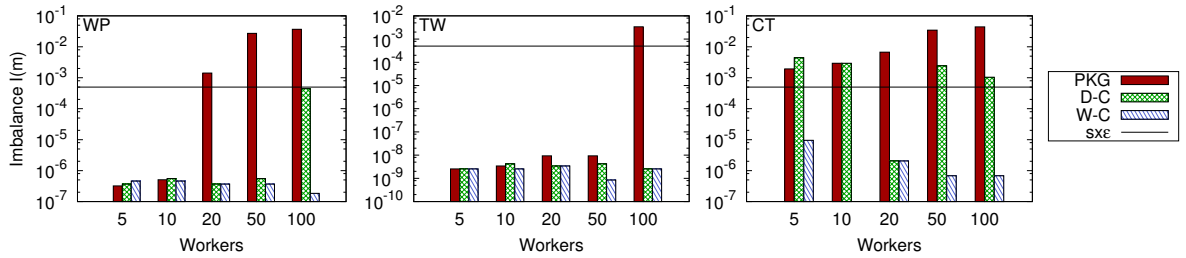

Fig. 11: Imbalance on the real-world datasets WP, TW and CT for PKG, D-C, and W-C as a function of the number of workers. Larger scale deployments are inherently harder to balance.

Finally, Figure 14 shows the difference among the algorithms in terms of latency. The plot reports the maximum of the per-worker average latencies, along with the 50th, 95th, and 99th percentiles across all workers. This message latency primarily depends on the time each message waits in the input queue of the worker before being processed, since we keep computation time per message constant to 1ms.

While with modest skew the difference is small, for larger skews KG suffers from very high latency. Given that the worker handling the most frequent key needs to process a large fraction of the stream, longer queues form at this worker, and this result is to be expected. Clearly PKG fares better, introducing about 50% of the latency of KG at the highest skew. However, for all settings it is not able to match the latency provided by SG, which can be up to 3 times smaller. Conversely, both D-C and W-C are able to provide latencies very close to the ones obtained by SG. In the best case, D-C and W-C reduce the 99th percentile latency of PKG by 60%, and the latency of KG by over 75%.

Overall, both D-Choices and W-Choices are able to achieve the same throughput and latency of shuffle grouping at just a fraction of the cost in terms of memory. These results suggest that both algorithms could easily replace the default implementation of shuffle grouping in most DSPEs, and provide a form of "worker affinity" for states related to a given key without compromising on the load balance.

VI. Related Work

Recently, there has been considerable interest in the load balancing problem for DSPEs when faced with a skewed input stream [10, 17]. Similarly to our work, previous solutions employ heavy hitter algorithms, and provide load balancing for skewed inputs when partitioned on keys. Gedik [10] develops a partitioning function (a hybrid between explicit mapping and consistent hashing) for stateful data parallelism in DSPEs that leverages item frequencies to control migration cost and imbalance in the system. The author proposes heuristics to discover new placement of keys on addition of parallel channels in the cluster, while minimizing the load imbalance and the migration cost. Rivetti et al. [17] propose an algorithm for a similar problem with an offline training phase to learn the best explicit mapping. Comparatively, we propose a light-weight streaming algorithm that provide strong load balancing guarantees, and

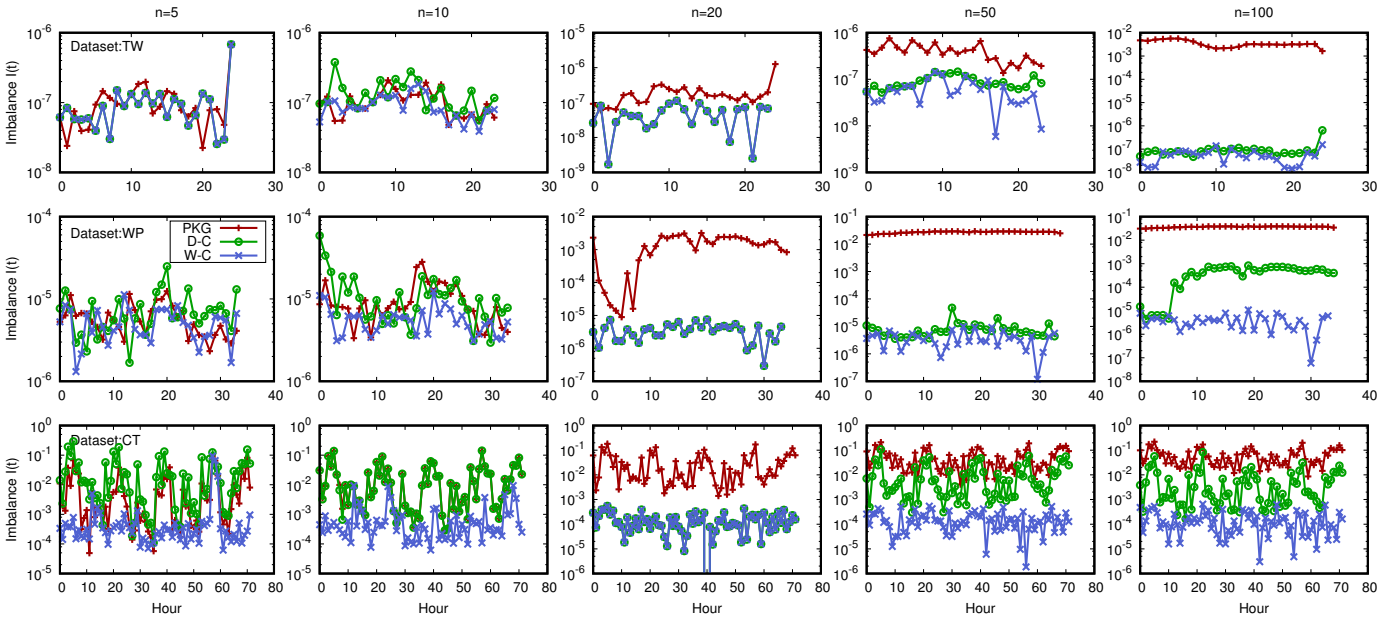

Fig. 12: Load imbalance over time for the real-world datasets TW, WP, and CT.

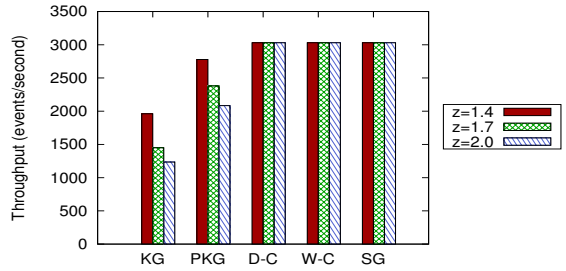

Fig. 13: Throughput on a cluster deployment on Apache Storm for KG, PKG, SG, D-C, and W-C on the ZF dataset, for sample values of z, with $n = 80$, $|\mathcal{K}| = 10^4$, and $m = 2 \times 10^6$.

which does not require training or active monitoring of the load imbalance. We show that by splitting the head of the distribution on multiple workers, and the tail distribution on at-most two workers, the techniques in this paper achieve nearly ideal load balance.

The load balancing problem has been extensively studied in the theoretical research community [3, 20, 22]. There have also been several proposals to solve the load balancing problem for peer-to-peer systems [16, 11]. The E-Store system explores the idea of handling hot tuples separately from cold tuples in the context of an elastic database management system [19]. Most existing load balancing techniques for DSPEs are analogous to key grouping with rebalancing [18, 7, 23, 4, 6]. Flux monitors the load of each operator, ranks servers by load, and migrates operators from the most loaded to the least loaded servers [18]. Aurora* and Medusa propose policies to operator migration in DSPEs and federated DSPEs [7]. Borealis uses a similar approach, but also aims at reducing the correlation of load spikes among operators placed on the same server [23]. This correlation is estimated by using load samples taken in the recent past. Similarly, Balkesen et al. [4] propose a frequency-aware hash-based partitioning to achieve load balance. Castro Fernandez et al. [6] propose integrating common operator state management techniques for both checkpointing and migration. Differently from these techniques, the ones proposed in this work do not require any rebalancing, operator migration, or explicit mapping of the keys to servers (which necessitates potentially huge routing tables).

VII. Conclusion

We studied the problem of load balancing for distributed stream processing engines when deployed at large scale. In doing so, we analyzed how highly skewed data streams exacerbate the problem, and how this affects the throughput and latency of an application. We showed that existing state-of-the-art techniques fall short in these extreme conditions.

We proposed two novel techniques for this tough problem: D-Choices and W-Choices. These techniques employ a streaming algorithm to detect heavy hitter for tracking the hot keys in the stream, which constitute the head of the distribution of keys, and allows those hot keys to be processed on larger set of workers. In particular, W-Choices allows a head key to be processed on entire set of workers, while D-Choices places a key on a smaller number of workers. This number is determined by using an easy-to-compute lower bound derived analytically and verified empirically.

We evaluated our proposal via extensive experimentation that covers simulations with synthetic and real datasets, and deployment on a cluster running Apache Storm. Results show that the techniques in this paper achieve better load balance compared to the state-of-the-art approaches. This improvement in balance translates into a gain in throughput of up to 150%, and latency of up to 60% over PKG for the cluster setup. When compared to key grouping, the gains are even higher: 230% for throughput and 75% for latency. Overall, D-Choices and W-Choices achieve the same throughput and latency as shuffle grouping, at just a fraction of the cost in terms of memory.

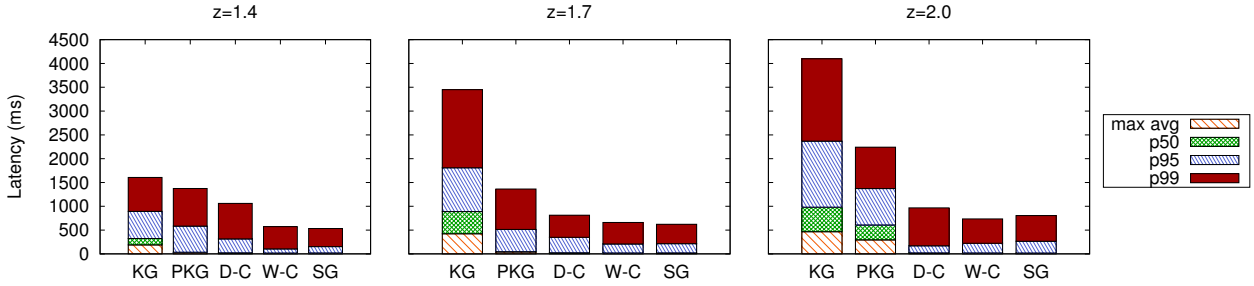

Fig. 14: Latency (average and percentiles) on a cluster deployment on Apache Storm for KG, PKG, SG, D-C, and W-C on the ZF datasets, for sample values of z, with $n = 80$, $|\mathcal{K}| = 10^4$, and $m = 2 \times 10^6$.

References

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8(2):1792–1803, 2015.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6): 939–964, 2014.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[4] C. Balkesen, N. Tatbul, and M. T. Özsu. Adaptive input admission and management for parallel stream processing. In *DEBS*, pp. 15–26. ACM, 2013.

[5] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM TODS*, 35(4):1–28, 2010.

[6] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pp. 725–736, 2013.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pp. 257–268, 2003.

[8] G. De Francisci Morales. SAMOA: A Platform for Mining Big Data Streams. In *RAMSS*, 2013.

[9] G. De Francisci Morales and A. Bifet. SAMOA: Scalable Advanced Massive Online Analysis. *JMLR*, 16(Jan):149–153, 2015.

[10] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, 2014.

[11] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA*, pp. 36–43, 2004.

[12] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pp. 398–412, 2005.

[13] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE TPDS*, 12(10):1094–1104, 2001.

[14] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. Partial key grouping: Load-balanced partitioning of distributed streams. *CoRR*, abs/1510.07623, 2015.

[15] M. A. U. Nasir, G. De Francisci Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pp. 137–148, 2015.

[16] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Peer-to-Peer Systems II*, pp. 68–79. Springer, 2003.

[17] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *DEBS*, pp. 80–91, 2015.

[18] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pp. 25–36, 2003.

[19] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8 (3):245–256, 2014.

[20] K. Talwar and U. Wieder. Balanced allocations: the weighted case. In *STOC*, pp. 256–265, 2007.

[21] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830i–1845, 2009.

[22] U. Wieder. Balanced allocations with heterogenous bins. In *SPAA*, pp. 188–193, 2007.

[23] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pp. 791–802, 2005.

Appendix A
Expected Size of a Worker Set

The process of chosing the workers for a key, is akin to placing d items, uniformly at random and with replacement, into n possible slots, with $d < n$ (in our case, if $d = n$ we should switch to the W-Choices algorithm). We now derive the expected number of slots b filled with at least one item.

As usual, it is easier to answer the complementary question: how many slots are left empty? Each slot has probability $p_{\text{empty}} = \left(1 - \frac{1}{n}\right)^d$ to be left empty at the end of d independent placements. Therefore, the expected number of empty slots is

$$\mathbb{E}[X_{\text{empty}}] = \sum_{i=1}^{n} p_{\text{empty}} = n \left(\frac{n-1}{n}\right)^d.$$

The number of full slots can be computed by subtracting the number of empty slots from the total number of slots,

$$b = \mathbb{E}[X_{\text{full}}] = n - \mathbb{E}[X_{\text{empty}}] = n - n \left(\frac{n-1}{n}\right)^d.$$

The expression of b_h can be obtained by replacing d with $h \times d$ in the previous expression, since we want to place $h \times d$ items.