The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines

Muhammad Anis Uddin Nasir[#1], Gianmarco De Francisci Morales[*2], David García-Soriano[*3]
Nicolas Kourtellis[*4], Marco Serafini[$5]

[#]*KTH Royal Institute of Technology, Stockholm, Sweden*
[*]*Yahoo Labs, Barcelona, Spain*
[$]*Qatar Computing Research Institute, Doha, Qatar*
[1]anisu@kth.se, [2]gdfm@apache.org, [3]davidgs@yahoo-inc.com
[4]kourtell@yahoo-inc.com, [5]mserafini@qf.org.qa

***Abstract*—We study the problem of load balancing in distributed stream processing engines, which is exacerbated in the presence of skew. We introduce PARTIAL KEY GROUPING (PKG), a new stream partitioning scheme that adapts the classical "power of two choices" to a distributed streaming setting by leveraging two novel techniques: *key splitting* and *local load estimation*. In so doing, it achieves better load balancing than key grouping while being more scalable than shuffle grouping.**

We test PKG on several large datasets, both real-world and synthetic. Compared to standard hashing, PKG reduces the load imbalance by up to several orders of magnitude, and often achieves nearly-perfect load balance. This result translates into an improvement of up to 60% in throughput and up to 45% in latency when deployed on a real Storm cluster.

I. INTRODUCTION

Distributed stream processing engines (DSPEs) such as S4,[1] Storm,[2] and Samza[3] have recently gained much attention owing to their ability to process huge volumes of data with very low latency on clusters of commodity hardware. Streaming applications are represented by directed acyclic graphs (DAG) where vertices, called *processing elements* (PEs), represent operators, and edges, called *streams*, represent the data flow from one PE to the next. For scalability, streams are partitioned into sub-streams and processed in parallel on a replica of the PE called *processing element instance* (PEI).

Applications of DSPEs, especially in data mining and machine learning, typically require accumulating state across the stream by grouping the data on common fields [1, 2]. Akin to MapReduce, this grouping in DSPEs is usually implemented by partitioning the stream on a *key* and ensuring that messages with the same key are processed by the same PEI. This partitioning scheme is called *key grouping*. Typically, it maps keys to sub-streams by using a hash function. Hash-based routing allows each source PEI to route each message solely via its key, without needing to keep any state or to coordinate among PEIs. Alas, it also results in load imbalance as it represents a "single-choice" paradigm [3], and because it disregards the popularity of a key, i.e., the number of messages with the same key in the stream, as depicted in Figure 1.

[1]https://incubator.apache.org/s4
[2]https://storm.incubator.apache.org
[3]https://samza.incubator.apache.org

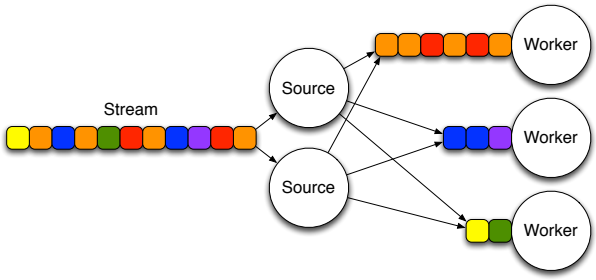

Fig. 1: Load imbalance generated by skew in the key distribution when using key grouping. The color of each message represents its key.

Large web companies run massive deployments of DSPEs in production. Given their scale, good utilization of the resources is critical. However, the skewed distribution of many workloads causes a few PEIs to sustain a significantly higher load than others. This suboptimal load balancing leads to poor resource utilization and inefficiency.

Another partitioning scheme called *shuffle grouping* achieves excellent load balancing by using a round-robin routing, i.e., by sending a message to a new PEI in cyclic order, irrespective of its key. However, this scheme is mostly suited for stateless computations. Shuffle grouping may require an additional aggregation phase and more memory to express stateful computations (Section II). Additionally, it may cause a decrease in accuracy for data mining algorithms (Section VI).

In this work, we focus on the problem of load balancing of stateful applications in DSPEs when the input stream follows a skewed key distribution. In this setting, load balancing is attained by having upstream PEIs create a balanced partition of messages for downstream PEIs, for each edge of the DAG. Any practical solution for this task needs to be both *streaming* and *distributed*: the former constraint enforces the use of an online algorithm, as the distribution of keys is not known in advance, while the latter calls for a decentralized solution with minimal coordination overhead in order to ensure scalability.

To address this problem, we leverage the "power of two choices" [4] (PoTC), whereby the system picks the least loaded out of two candidate PEIs for each key. However, to maintain the semantics of key grouping while using PoTC (i.e., so that one key is handled by a single PEI), sources would need to track which of the two possible choices has been made for each key. This requirement imposes a coordination overhead every time a new key appears, so that all sources agree on the choice. In addition, sources should then store this choice in a routing table. Each edge in the DAG would thus require a routing table for every source, each with one entry per key. Given that a typical stream may contain billions of keys, this solution is not practical.

Instead, we propose to relax the key grouping constraint and allow each key to be handled by *both* candidate PEIs. We call this technique *key splitting*; it allows us to apply PoTC without the need to agree on, or keep track of, the choices made. As shown in Section V, key splitting guarantees good load balance even in the presence of skew.

A second issue is how to estimate the load of a downstream PEI. Traditional work on PoTC assumes global knowledge of the current load of each server, which is challenging in a distributed system. Additionally, it assumes that all messages originate from a single source, whereas messages in a DSPE are generated in parallel by multiple sources.

In this paper we prove that, interestingly, a simple *local load estimation* technique, whereby each source independently tracks the load of downstream PEIs, performs very well in practice. This technique gives results that are almost indistinguishable from those given by a global load oracle.

The combination of these two techniques (key splitting and local load estimation) enables a new stream partitioning scheme named PARTIAL KEY GROUPING.

In summary, we make the following contributions.

- We study the problem of load balancing in modern distributed stream processing engines.
- We show how to apply PoTC to DSPEs in a principled and practical way, and propose two novel techniques to do so: key splitting and local load estimation.
- We propose PARTIAL KEY GROUPING, a novel and simple stream partitioning scheme that applies to any DSPE. When implemented on top of Apache Storm, it requires a single function and less than 20 lines of code.[4]
- We measure the impact of PKG on a real deployment on Apache Storm. Compared to key grouping, it improves the throughput of an example application on real-world datasets by up to 60%, and the latency by up to 45%.

II. PRELIMINARIES AND MOTIVATION

We consider a DSPE running on a cluster of machines that communicate by exchanging messages following the flow of a DAG, as discussed. In this work, we focus on balancing the data transmission along a single edge in a DAG. Load balancing across the whole DAG is achieved by balancing along each edge independently. Each edge represents a single stream of data, along with its partitioning scheme. Given a stream under consideration, let the set of upstream PEIs (sources) be $\mathcal{S}$, and the set of downstream PEIs (workers) be $\mathcal{W}$, and their sizes be $|\mathcal{S}| = S$ and $|\mathcal{W}| = W$ (see Figure 1).

The input to the engine is a sequence of messages $m = \langle t, k, v \rangle$ where t is the timestamp at which the message is received, $k \in \mathcal{K}, |\mathcal{K}| = K$ is the message key, and v is the value. The messages are presented to the engine in ascending order by timestamp.

A *stream partitioning* function $P_t : \mathcal{K} \to \mathbb{N}$ 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. Each worker is associated to one or more keys.

We use a definition of *load* similar to others in the literature (e.g., Flux [5]). At time t, the load of a worker i is the number of messages handled by the worker up to t:

$$L_i(t) = |\{\langle \tau, k, v \rangle : P_\tau(k) = i \wedge \tau \leq t\}|$$

In principle, depending on the application, two different messages might impose a different load on workers. However, in most cases these differences even out and modeling such application-specific differences is not necessary.

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

$$I(t) = \max_i(L_i(t)) - \operatorname*{avg}_i(L_i(t)), \text{ for } i \in \mathcal{W}$$

We tackle the problem of identifying a stream partitioning function that minimizes the imbalance, while at the same time avoiding the downsides of shuffle grouping.

A. Existing Stream Partitioning Functions

Data is sent between PEs by exchanging messages over the network. Several primitives are offered by DSPEs for sources to partition the stream, i.e., to route messages to different workers. There are two main primitives of interest: *key grouping* (KG) and *shuffle grouping* (SG).

KG ensures that messages with the same key are handled by the same PEI (analogous to MapReduce). It is usually implemented through hashing.

SG routes messages independently, typically in a round-robin fashion. SG provides excellent load balance by assigning an almost equal number of messages to each PEI. However, no guarantee is made on the partitioning of the key space, as each occurrence of a key can be assigned to any PEIs. SG is the perfect choice for *stateless* operators. However, with *stateful* operators one has to handle, store and aggregate multiple partial results for the same key, thus incurring additional costs.

In general, when the distribution of input keys is skewed, the number of messages that each PEI needs to handle can vary greatly. While this problem is not present for stateless operators, which can use SG to evenly distribute messages, stateful operators implemented via KG suffer from load imbalance. This issue generates a degradation of the service level, or reduces the utilization of the cluster which must be provisioned to handle the peak load of the single most loaded server.

[4] Available at https://github.com/gdfm/partial-key-grouping

Example. To make the discussion more concrete, we introduce a simple application that will be our running example: *streaming top-k word count*. This application is an adaptation of the classical MapReduce word count to the streaming paradigm where we want to generate a list of top-k words by frequency at periodic intervals (e.g., each T seconds). It is also a common application in many domains, for example to identify trending topics in a stream of tweets.

Implementation via key grouping. Following the MapReduce paradigm, the implementation of word count described by Neumeyer et al. [6] or Noll [7] uses KG on the source stream. The counter PE keeps a running counter for each word. KG ensures that each word is handled by a single PEI, which thus has the total count for the word in the stream. At periodic intervals, the counter PEIs send their top-k counters to a single downstream aggregator to compute the top-k words. While this application is clearly simplistic, it models quite well a general class of applications common in data mining and machine learning whose goal is to create a model by tracking aggregated statistics of the data.

Clearly KG generates load imbalance as, for instance, the PEI associated to the key "the" will receive many more messages than the one associated with "Barcelona". This example captures the core of the problem we tackle: the distribution of word frequencies follows a Zipf law where few words are extremely common while a large majority are rare. Therefore, an even distribution of keys such as the one generated by KG results in an uneven distribution of messages.

Implementation via shuffle grouping. An alternative implementation uses shuffle grouping on the source stream to get partial word counts. These counts are sent downstream to an aggregator every T seconds via key grouping. The aggregator simply combines the counts for each key to get the total count and selects the top-k for the final result.

Using SG requires a slightly more complex logic but it generates an even distribution of messages among the counter PEIs. However, it suffers from other problems. Given that there is no guarantee which PEI will handle a key, each PEI potentially needs to keep a counter for *every* key in the stream. Therefore, the memory usage of the application grows linearly with the parallelism level. Hence, it is not possible to scale to a larger workload by adding more machines: the application is not scalable in terms of memory. Even if we resort to approximation algorithms, in general, the error depends on the number of aggregations performed, thus it grows linearly with the parallelism level. We analyze this case in further detail along with other application scenarios in Section VI.

B. Key grouping with rebalancing

One common solution for load balancing in DSPEs is operator migration [5, 8, 9, 10, 11, 12]. Once a situation of load imbalance is detected, the system activates a rebalancing routine that moves part of the keys, and the state associated with them, away from an overloaded server. While this solution is easy to understand, its application in our context is not straightforward for several reasons.

Rebalancing requires setting a number of parameters such as how often to check for imbalance and how often to rebalance. These parameters are often application-specific as they involve a trade-off between imbalance and rebalancing cost that depends on the size of the state to migrate.

Further, implementing a rebalancing mechanism usually requires major modifications of the DSPE at hand. This task may be hard, and is usually seen with suspicion by the community driving open source projects, as witnessed by the many variants of Hadoop that were never merged back into the main line of development [13, 14, 15].

In our context, rebalancing implies migrating keys from one sub-stream to another. However, this migration is not directly supported by the programming abstractions of some DSPEs. Storm and Samza use a *coarse-grained* stream partitioning paradigm. Each stream is partitioned into as many sub-streams as the number of downstream PEIs. Key migration is not compatible with this partitioning paradigm, as a key cannot be uncoupled from its sub-stream. In contrast, S4 employs a *fine-grained* paradigm where the stream is partitioned into one sub-stream per key value, and there is a one-to-one mapping of a key to a PEI. The latter paradigm easily supports migration, as each key is processed independently.

A major problem with mapping keys to PEIs explicitly is that the DSPE must maintain several routing tables: one for each stream. Each routing table has one entry for each key in the stream. Keeping these tables is impractical because the memory requirements are staggering. In a typical web mining application, each routing table can easily have billions of keys. For a moderately large DAG with tens of edges, each with tens of sources, the memory overhead easily becomes prohibitive.

Finally, as already mentioned, for each stream there are several sources sending messages in parallel. Modifications to the routing table must be consistent across all sources, so they require coordination, which creates further overhead. For these reasons we consider an alternative approach to load balancing.

III. Partial Key Grouping

The problem described so far currently lacks a satisfying solution. To solve this issue, we resort to a widely-used technique in the literature of load balancing: the so-called *"power of two choices"* (PoTC). While this technique is well-known and has been analyzed thoroughly both from a theoretical and practical perspective [16, 17, 18, 19, 4, 20], its application in the context of DSPEs is not straightforward and has not been previously studied.

Introduced by Azar et al. [17], PoTC is a simple and elegant technique that allows to achieve load balance when assigning units of load to workers. It is best described in terms of "balls and bins". Imagine a process where a stream of balls (units of work) is distributed to a set of bins (the workers) as evenly as possible. The *single-choice paradigm* corresponds to putting each ball into one bin selected uniformly at random. By contrast, the power of two choices selects two bins uniformly at random, and puts the ball into the least loaded one. This

simple modification of the algorithm has powerful implications that are well known in the literature (see Sections IV, VII).

Single choice. The current solution used by all DSPEs to partition a stream with key grouping corresponds to the single-choice paradigm. The system has access to a single hash function $\mathcal{H}_1(k)$. The partitioning of keys into sub-streams is determined by the function $P_t(k) = \mathcal{H}_1(k) \bmod W$, where $\bmod$ is the modulo operator.

The single-choice paradigm is attractive because of its simplicity: the routing does not require to maintain any state and can be done independently in parallel. However, it suffers from a problem of load imbalance [4]. This problem is exacerbated when the distribution of input keys is skewed.

PoTC. When using the power of two choices, we have two hash functions $\mathcal{H}_1(k)$ and $\mathcal{H}_2(k)$. The algorithm maps each key to the sub-stream assigned to the least loaded worker between the two possible choices, that is: $P_t(k) = \operatorname{argmin}_i(L_i(t) : \mathcal{H}_1(k) = i \vee \mathcal{H}_2(k) = i)$.

The theoretical gain in load balance with two choices is exponential compared to a single choice. However, using more than two choices only brings constant factor improvements [17]. Therefore, we restrict our study to two choices.

PoTC introduces two additional complications. First, to maintain the semantics of key grouping, the system needs to *keep state* and track the choices made. Second, the system has to *know the load* of the workers in order to make the right choice. We discuss these two issues next.

A. Key Splitting

A naïve application of PoTC to key grouping requires the system to store a bit of information for each key seen, to keep track of which of the two choices needs to be used thereafter. This variant is referred to as *static* PoTC.

Static PoTC incurs some of the problems discussed for key grouping with rebalancing. Since the actual worker to which a key is routed is determined dynamically, sources need to keep a routing table with an entry per key. As already discussed, maintaining this routing table is often impractical.

In order to leverage PoTC and make it viable for DSPEs, we relax the requirement of key grouping. Rather than mapping each key to one of the two possible choices, we allow it to be mapped to *both choices*. Every time a source sends a message, it selects the worker with the lowest current load among the two candidates associated to that key. This technique, called *key splitting*, introduces several new trade-offs.

First, key splitting allows the system to operate in a decentralized manner, by allowing multiple sources to take decisions independently in parallel. As in key grouping and shuffle grouping, no state needs to be kept by the system and each message can be routed independently.

Key splitting enables far better load balancing compared to key grouping. It allows using PoTC to balance the load on the workers: by splitting each key on multiple workers, it handles the skew in the key popularity. Moreover, given that *all* its decisions are dynamic and based on the current load of the system (as opposed to static PoTC), key splitting adapts to changes in the popularity of keys over time.

Third, key splitting reduces the memory usage and aggregation overhead compared to shuffle grouping. Given that each key is assigned to *exactly two* PEIs, the memory to store its state is just a constant factor higher than when using key grouping. Instead, with shuffle grouping the memory grows linearly with the number of workers W. Additionally, state aggregation needs to happen only once for the two partial states, as opposed to $W - 1$ times in shuffle grouping. This improvement also allows to reduce the error incurred during aggregation for some algorithms, as discussed in Section VI.

From the point of view of the application developer, key splitting gives rise to a novel stream partitioning scheme called PARTIAL KEY GROUPING, which lies in-between key grouping and shuffle grouping.

Naturally, not all algorithms can be expressed via PKG. The functions that can leverage PKG are the same ones that can leverage a combiner in MapReduce, i.e., associative functions and monoids. Examples of applications include naïve Bayes, heavy hitters, and streaming parallel decision trees, as detailed in Section VI. On the contrary, other functions such as computing the median cannot be easily expressed via PKG.

Example. Let us examine the streaming top-k word count example using PKG. In this case, each word is tracked by two counters on two different PEIs. Each counter holds a partial count for the word, while the total count is the sum of the two partial counts. Therefore, the total memory usage is $2 \times K$, i.e., $O(K)$. Compare this result to SG where the memory is $O(WK)$. Partial counts are sent downstream to an aggregator that computes the final result. For each word, the application sends two counters, and the aggregator performs a constant time aggregation. The total work for the aggregation is $O(K)$. Conversely, with SG the total work is again $O(WK)$. Compared to KG, the implementation with PKG requires additional logic, some more memory and has some aggregation overhead. However, it also provides a much better load balance which maximizes the resource utilization of the cluster. The experiments in Section V prove that the benefits outweigh its cost.

B. Local Load Estimation

PoTC requires knowledge of the load of each worker to take its routing decision. A DSPE is a distributed system, and, in general, sources and workers are deployed on different machines. Therefore, the load of each worker is not readily available to each source.

Interestingly, we prove that no communication between sources and workers is needed to effectively apply PoTC. We propose a *local load estimation* technique, whereby each source independently maintains a local load-estimate vector with one element per worker. The load estimates are updated by using only local information of the portion of stream sent by each source. We argue that in order to achieve global load balance it is sufficient that each source independently balances the load it generates across all workers.

The correctness of local load estimation directly follows from our standard definition of load in Section II. The load on a worker L_i is simply the sum of the loads that each source j imposes on the given worker: $L_i(t) = \sum_{j \in \mathcal{S}} L_i^j(t)$. Each source j can keep an estimate of the load on each worker i based on the load it has generated L_i^j. As long as each source keeps its own portion of load balanced, then the overall load on the workers will also be balanced. Indeed, the maximum overall load is at most the sum of the maximum load that each source sees locally. It follows that the maximum imbalance is also at most the sum of the local imbalances.

IV. Analysis

We proceed to analyze the conditions under which PKG achieves good load balance. Recall from Section II that we have a set $\mathcal{W}$ of n workers at our disposal and receive a sequence of m messages $k_1, \ldots, k_m$ with values from a key universe $\mathcal{K}$. Upon receiving the i-th message with value $k_i \in \mathcal{K}$, we need to decide its placement among the workers; decisions are irrevocable. We assume one message arrives per unit of time. Our goal is to minimize the eventual maximum load $L(m)$, which is the same as minimizing the imbalance $I(m)$. A simple placement scheme such as shuffle grouping provides an imbalance of at most one, but we would like to limit the number of workers processing each key to $d \in \mathbb{N}^+$.

Chromatic balls and bins. We model our problem in the framework of balls and bins processes, where keys correspond to colors, messages to colored balls, and workers to bins. Choose d independent hash functions $\mathcal{H}_1, \ldots, \mathcal{H}_d \colon \mathcal{K} \to [n]$ uniformly at random. Define the Greedy-d scheme as follows: at time t, the t-th ball (whose color is k_t) is placed on the bin with minimum current load among $\mathcal{H}_1(k_t), \ldots, \mathcal{H}_d(k_t)$, i.e., $P_t(k_t) = \operatorname{argmin}_{i \in \{\mathcal{H}_1(k_t), \ldots, \mathcal{H}_d(k_t)\}} L_i(t)$. Recall that with key splitting there is no need to remember the choice for the next time a ball of the same color appears.

Observe that when $d = 1$, each ball color is assigned to a unique bin so no choice has to be made; this models hash-based key grouping. At the other extreme, when $d \gg n \ln n$, all n bins are valid choices, and we obtain shuffle grouping.

Key distribution. Finally, we assume the existence of an underlying discrete distribution $\mathcal{D}$ supported on $\mathcal{K}$ from which ball colors are drawn, i.e., $k_1, \ldots, k_m$ is a sequence of m independent samples from $\mathcal{D}$. Without loss of generality, we identify the set $\mathcal{K}$ of keys with $\mathbb{N}^+$ or, if $\mathcal{K}$ is finite of cardinality $K = |\mathcal{K}|$, with $[K] = \{1, \ldots, K\}$. We assume them ordered by decreasing probability: if p_i is the probability of drawing key i from $\mathcal{D}$, then $p_1 \geq p_2 \geq p_3 \ldots$ and $\sum_{i \in \mathcal{K}} p_i = 1$. We also identify the set $\mathcal{W}$ of bins with $[n]$.

A. Imbalance with Partial Key Grouping

Comparison with standard problems. As long as we keep getting balls of different colors, our process is identical to the standard Greedy-d process of Azar et al. [17]. This occurs with high probability provided that m is small enough. But for sufficiently large m (e.g., when $m \geq \frac{1}{p_1}$), repeated keys will start to arrive. Recall that for any number of choices $d \geq 2$, the maximum imbalance after throwing m balls *of different colors* into n bins with the standard Greedy-d process is $\frac{\ln \ln n}{\ln d} + \frac{m}{n} + O(1)$. Unfortunately, such strong bounds (independent of m) cannot apply to our setting. To gain some intuition on what may go wrong, consider the following examples where d=2.

Note that for the maximum load not to be much larger than the average load, the number of bins used must not exceed $O(1/p_1)$, where p_1 is the maximum key probability. Indeed, at any time we expect the two bins $h_1(1), h_2(1)$ to contain together at least a p_1 fraction of all balls, just counting the occurrences of a single key. Hence the expected maximum load among the two grows at a rate of at least $p_1/2$ per unit of time, while the overall average load increases by exactly $\frac{1}{n}$ per unit of time. Thus, if $p_1 > 2/n$, the expected imbalance at time m will be lower bounded by $(\frac{p_1}{2} - \frac{1}{n})m$, which grows *linearly* with m. This holds irrespective of the placement scheme used.

However, requiring $p_1 \leq 2/n$ is not enough to prevent imbalance $\Omega(m)$. Consider the uniform distribution over n keys. Let $B = \bigcup_{i \leq n} \{\mathcal{H}_1(i), \mathcal{H}_2(i)\}$ be the set of all bins that belong to one of the potential choices for some key. As is well-known, the expected size of B is $n - n\left(1 - \frac{1}{n}\right)^{2n} \approx n(1 - \frac{1}{e^2})$. So all n keys use only an $(1 - \frac{1}{e^2}) \approx 0.865$ fraction of all bins, and roughly $0.135n$ bins will remain unused. In fact the imbalance after m balls will be at least $\frac{m}{0.865n} - \frac{m}{n} \approx 0.156m$. The problem is that most concrete instantiations of our two random hash functions cause the existence of an "overpopulated" set B of bins inside which the average bin load must grow faster than the average load across all bins. (In fact, this case subsumes our first example above, where B was $\{\mathcal{H}_1(1), \mathcal{H}_2(1)\}$.)

Finally, even in the absence of overpopulated bin subsets, some inherent imbalance is due to deviations between the empirical and true key distributions. For instance, suppose there are two keys $1, 2$ with equal probability $\frac{1}{2}$ and $n = 4$ bins. With constant probability, key 1 is assigned to bins $1, 2$ and key 2 to bins $3, 4$. This situation looks perfect because the Greedy-2 choice will send each occurrence of key 1 to bins $1, 2$ alternately so the loads of bins $1, 2$ will always equal up to ± 1. However, the number of balls with key 1 seen is likely to deviate from $m/2$ by roughly $\Theta(\sqrt{m})$, so either the top two or the bottom two bins will receive $m/4 + \Omega(\sqrt{m})$ balls, and the imbalance will be $\Omega(\sqrt{m})$ with constant probability.

In the remainder of this section we carry out our analysis, which broadly construed asserts that the above are the only impediments to achieve good balance.

Statement of results. We noted that once the number of bins exceeds $2/p_1$ (where p_1 is the maximum key frequency), the maximum load will be dominated by the loads of the bins to which the most frequent key is mapped. Hence the main case of interest is where $p_1 = O(\frac{1}{n})$.

We focus on the case where the number of balls is large compared to the number of bins. The following results show that partial key grouping can significantly reduce the maximum load (and the imbalance), compared to key grouping.

Theorem 4.1: Suppose we use n bins and let $m \geq n^2$. Assume a key distribution $\mathcal{D}$ with maximum probability $p_1 \leq \frac{1}{5n}$.

Then the imbalance after m steps of the Greedy-d process satisfies, with probability at least $1 - \frac{1}{n}$,

$$I(m) = \begin{cases} O\left(\frac{m}{n} \cdot \frac{\ln n}{\ln \ln n}\right), & \text{if } d = 1 \\ O\left(\frac{m}{n}\right), & \text{if } d \geq 2 \end{cases}.$$

As the next result shows, the bounds above are best-possible.[5]

Theorem 4.2: There is a distribution $\mathcal{D}$ satisfying the hypothesis of Theorem 4.1 such that the imbalance after m steps of the Greedy-d process satisfies, with probability at least $1 - \frac{1}{n}$,

$$I(m) = \begin{cases} \Omega\left(\frac{m}{n} \cdot \frac{\ln n}{\ln \ln n}\right), & \text{if } d = 1 \\ \Omega\left(\frac{m}{n}\right), & \text{if } d \geq 2 \end{cases}.$$

We omit the proof of Theorem 4.2 (it follows by considering a uniform distribution over $5n$ keys). The next section is devoted to the proof of the upper bound, Theorem 4.1.

B. Proof

Concentration inequalities. We recall the following results, which we need to prove our main theorem.

Theorem 4.3 (Chernoff bounds): Suppose $\{X_i\}$ is a finite sequence of independent random variables with $X_i \in [0, M]$ and let $Y = \sum_i X_i$, $\mu = \sum_i \mathbb{E}[X_i]$. Then for all $\beta \geq \mu$,

$$\Pr[Y \geq \beta] \leq C(\mu, \beta, M),$$

where

$$C(\mu, \beta, M) \triangleq \exp\Big(-\frac{\beta \ln(\frac{\beta}{e\mu}) + \mu}{M}\Big).$$

Theorem 4.4 (McDiarmid's inequality): Let $X_1, \ldots, X_n$ be a vector of independent random variables and let f be a function satisfying $|f(a) - f(a')| \leq 1$ whenever the vectors a and a' differ in just one coordinate. Then

$$\Pr[f(X_1, \ldots, X_n) > \mathbb{E}[f(X_1, \ldots, X_n)] + \lambda] \leq \exp(-2\lambda^2).$$

The μ_r measure of bin subsets. For every nonempty set of bins $S \subseteq [n]$ and $1 \leq r \leq d$, define

$$\mu_r(S) = \sum \{p_i \mid \{\mathcal{H}_1(i), \ldots, \mathcal{H}_r(i)\} \subseteq B\}.$$

We will be interested in $\mu_1(B)$ (which measures the probability that a random key from $\mathcal{D}$ will have its choice inside B) and $\mu_d(B)$ (which measures the probability that a random key from $\mathcal{D}$ will have all its choices inside B). Note that $\mu_1(B) = \sum_{j \in B} \mu_1(\{j\})$ and $\mu_d(B) \leq \mu_1(B)$.

Lemma 4.5: For every $B \subseteq [n]$, $\mathbb{E}[\mu_1(B)] = \frac{|B|}{n}$ and, if $p_1 \leq \frac{1}{n}$,

$$\Pr\left[\mu_1(B) \geq \frac{|B|}{n}(e\lambda)\right] \leq \left(\frac{1}{\lambda^\lambda}\right)^{|B|}.$$

[5]However, the imbalance can be much smaller than the worst-case bounds from Theorem 4.1 if the probability of most keys is much smaller than p_1, which is the case in many setups.

Proof: The first claim follows from linearity of expectation and the fact that $\sum_i p_i = 1$. For the second, let $|B| = k$. Using Theorem 4.3, $\Pr\left[\mu_1(B) \geq \frac{k}{n}(e\lambda)\right]$ is at most

$$C\left(\frac{k}{n}, \frac{k}{n}e\lambda, p_1\right) \leq \exp\left(-\frac{k}{np}e\lambda \ln \lambda\right) \leq \exp(-k\lambda \ln \lambda),$$

since $np_1 \leq 1$. ∎

Lemma 4.6: For every $B \subseteq [n]$, $\mathbb{E}[\mu_d(B)] = \left(\frac{|B|}{n}\right)^d$ and, provided that $p_1 \leq \frac{1}{5n}$,

$$\Pr\left[\mu_d(B) \geq \frac{|B|}{n}\right] \leq \left(\frac{e|B|}{n}\right)^{5|B|}.$$

Proof: Again the first claim is easy. For the second, let $|B| = k$. Using Theorem 4.3, $\Pr\left[\mu_d(B) \geq \frac{k}{n}\right]$ is at most

$$\begin{aligned} C\left(\left(\frac{k}{n}\right)^d, \frac{k}{n}, p_1\right) &\leq \exp\left(-\frac{k(d-1)}{np_1} \ln\left(\frac{n}{ek}\right)\right) \\ &\leq \exp\left(-5k \ln\left(\frac{n}{ek}\right)\right) \end{aligned}$$

since $np_1 \leq \frac{1}{5}$. ∎

Corollary 4.7: Assume $p_1 \leq \frac{1}{4n}$, $d \geq 2$. Then, with high probability,

$$\max\left\{\frac{\mu_d(B)}{|B|/n} \middle| B \subseteq [n], |B| \leq \frac{n}{5}\right\} \leq 1.$$

Proof: We use Lemma 4.5 and the union bound. The probability that the claim fails to hold is bounded by

$$\begin{aligned} \sum_{|B| \leq n/5} \Pr\left[\mu_d(B) \geq \frac{k}{n}\right] &\leq \sum_{k \leq n/5} \binom{n}{k} \left(\frac{ek}{n}\right)^{5k} \\ &\leq \sum_{k \leq n/5} \left(\frac{en}{k}\right)^k \left(\frac{ek}{n}\right)^{5k} = o\left(\frac{1}{n}\right), \end{aligned}$$

where we used $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$, valid for all k. ∎

For a scheduling algorithm $\mathcal{A}$ and a set $B \subseteq [n]$ of bins, write $L_B^{\mathcal{A}}(t) = \max_{j \in B} L_j(t)$ for the maximum load among the bins in B after t balls have been processed by $\mathcal{A}$.

Lemma 4.8: Suppose there is a set $A \subseteq [n]$ of bins such that for all $T \subseteq A$, $\mu_d(T) \leq \frac{|T|}{n}$. Then $\mathcal{A} =$ Greedy-d satisfies $L_A^{\mathcal{A}}(m) = O(\frac{m}{n}) + L_{[n] \setminus A}^{\mathcal{A}}(m)$ with high probability.

Proof: We use a coupling argument. Consider the following two independent processes $\mathcal{P}$ and $\mathcal{Q}$: $\mathcal{P}$ proceeds as Greedy-d, while $\mathcal{Q}$ picks the bin for each ball independently at random from $[n]$ and increases its load. Consider any time t at which the load vector is $\omega_t \in \mathbb{N}^n$ and $M_t = M(\omega_t)$ is the set of bins with maximum load. After handling the t-th ball, let X_t denote the event that $\mathcal{P}$ increases the maximum load in A because the new ball has all choices in $M_t \cap A$, and Y_t denote the event that $\mathcal{Q}$ increases the maximum load in A. Finally, let Z_t denote the event that $\mathcal{P}$ increases the maximum load in A because the new ball has some choice in $M_t \cap A$ and some choice in $M_t \setminus A$, but the load of one of its choices in $M_t \cap A$ is no larger. We identify these events with their indicator random variables.

Note that the maximum load in A at the end of Process $\mathcal{P}$ is $L_A^{\mathcal{P}}(m) = \sum_{t\in[m]}(X_t + Z_t)$, while at the end of Process $\mathcal{Q}$ is $L_A^{\mathcal{Q}}(m) = \sum_{t\in[m]} Y_t$. Conditioned on any load vector ω_t, the probability of X_t is

$$\Pr[X_t \mid \omega_t] = \mu_d(M_t \cap A) \leq \frac{|M_t \cap A|}{n} \leq \frac{|M_t|}{n} = \Pr[Y_t \mid \omega_t],$$

So $\Pr[X_t \mid \omega_t] \leq \Pr[Y_t \mid \omega_t]$, which implies that for any $b \in \mathbb{N}$, $\Pr[\sum_{t\in[m]} X_t \leq b] \geq \Pr[\sum_{t\in[m]} Y_t \leq b]$. But with high probability, the maximum load of Process $\mathcal{Q}$ is $b = O(m/n)$, so $\sum_t X_t = O(m/n)$ holds with at least the same probability. On the other hand, $\sum_t Z_t \leq L^{\mathcal{P}}_{[n]\setminus A}(m)$ because each occurrence of Z_t increases the maximum load on A, and once a time t is reached such that $L_A^{\mathcal{P}}(t) > L^{\mathcal{P}}_{[n]\setminus A}(m)$, event Z_t must cease to happen. Therefore $L_A^{\mathcal{P}}(m) = \sum_{t\in[m]} X_t + \sum_{t\in[m]} Z_t \leq O(m/n) + L^{\mathcal{P}}_{[n]\setminus A}(m)$, yielding the result. ∎

Proof of Theorem 4.1: Let

$$A = \left\{ j \in [n] \mid \mu_1(\{j\}) \geq \frac{3e}{n} \right\}.$$

Observe that every bin $j \notin A$ has $\mu_1(\{j\}) < \frac{3e}{n}$ and this implies that, conditioned on any choice of hash functions, the maximum load of all bins outside A is at most $\frac{20m}{n}$ with high probability.[6] Therefore our task reduces to showing that the maximum load of the bins in A is $O(\frac{m}{n})$.

Consider the sequence $X_1, \ldots, X_K$ of random variables given by $X_i = \mathcal{H}_1(i)$, and let $f(X_1, X_2, \ldots, X_K) = |A|$ denote the number of bins j with $\mu_1(\{j\}) \geq \frac{3e}{n}$. By Lemma 4.5, $\mathbb{E}[|A|] = \mathbb{E}[f] \leq \frac{1}{27}$. Moreover, the function f satisfies the hypothesis of Theorem 4.4.We conclude that, with high probability, $|A| \leq \frac{n}{5}$.

Now assume that the thesis of Corollary 4.7 holds, which happens except with probability $o(1/n)$. Then we have that for all $B \subseteq A$, $\mu_d(B) \leq \frac{|B|}{n}$. Thus Lemma 4.8 applies to A. This means that after throwing m balls, the maximum load among the bins in A is $O(\frac{m}{n})$, as we wished to show. ∎

V. Evaluation

We assess the performance of our proposal by using both simulations and a real deployment. In so doing, we answer the following questions:

Q1: What is the effect of key splitting on PoTC?

Q2: How does local estimation compare to a global oracle?

Q3: How robust is Partial Key Grouping?

Q4: What is the overall effect of Partial Key Grouping on applications deployed on a real DSPE?

A. Experimental Setup

Datasets. Table I summarizes the datasets used. We use two main real datasets, one from *Wikipedia* and one from *Twitter*. These datasets were chosen for their large size, their different degree of skewness, and because they are representative of Web and online social network domains. The Wikipedia dataset (WP)[7] is a log of the pages visited during a day in January 2008. 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 parsed and split into its words, which are used as the key for the message.

TABLE I: Summary of the datasets used in the experiments: number of messages, number of keys and percentage of messages having 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
Synthetic 1	LN_1	10M	16k	14.71
Synthetic 2	LN_2	10M	1.1k	7.01
LiveJournal	LJ	69M	4.9M	0.29
Slashdot0811	SL_1	905k	77k	3.28
Slashdot0902	SL_2	948k	82k	3.11

An additional Twitter dataset comprises a sample 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). Popular cash tags change from week to week. This dataset allows to study the effect of shift of skew in the key distribution.

We also generate two synthetic datasets (LN_1, LN_2) with keys following a log-normal distribution, a commonly used heavy-tailed skewed distribution [21]. The parameters of the distribution (μ_1=1.789, σ_1=2.366; μ_2=2.245, σ_2=1.133) come from an analysis of Orkut, and try to emulate workloads from the online social network domain [22].

Finally, we experiment on three additional datasets comprised of directed graphs[8] (LJ, SL_1, SL_2). We use the edges in the graph as messages and the vertices as keys. These datasets are used to test the robustness of PKG to skew in partitioning the stream *at the sources*, as explained next. They also represent a different kind of application domain: streaming graph mining.

Simulation. We process the datasets by simulating the DAG presented in Figure 1. The stream is composed of timestamped keys that are read by multiple independent sources ($\mathcal{S}$) via shuffle grouping, unless otherwise specified. The sources forward the received keys to the workers ($\mathcal{W}$) downstream. In our simulations we assume that the sources perform data extraction and transformation, while the workers perform data aggregation, which is the most computationally expensive part of the DAG. Thus, the workers are the bottleneck in the DAG and the focus for the load balancing.

B. Experimental Results

Q1. We measure the imbalance in the simulations when using the following techniques:

[6] This is by majorization with the process that just throws every ball to the *first* choice; see, e.g, Azar et al. [17].

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

[8] http://snap.stanford.edu/data

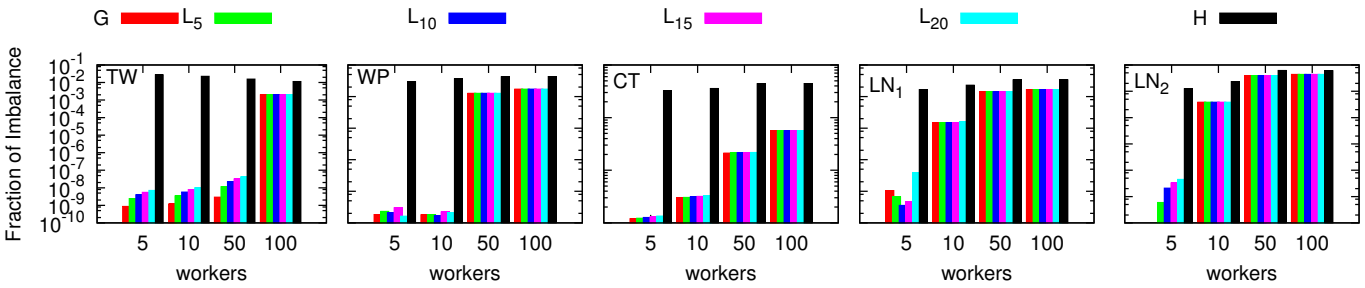

Fig. 2: Fraction of average imbalance with respect to total number of messages for each dataset, for different number of workers and number of sources.

TABLE II: Average imbalance when varying the number of workers for the Wikipedia and Twitter datasets.

Dataset	WP				TW			
W	5	10	50	100	5	10	50	100
PKG	0.8	2.9	5.9e5	8.0e5	0.4	1.7	2.74	4.0e6
Off-Greedy	0.8	0.9	1.6e6	1.8e6	0.4	0.7	7.8e6	2.0e7
On-Greedy	7.8	1.4e5	1.6e6	1.8e6	8.4	92.7	1.2e7	2.0e7
PoTC	15.8	1.7e5	1.6e6	1.8e6	2.2e4	5.1e3	1.4e7	2.0e7
Hashing	1.4e6	1.7e6	2.0e6	2.0e6	4.1e7	3.7e7	2.4e7	3.3e7

H: Hashing, which represents standard key grouping (KG) and is our main baseline. We use a 64-bit Murmur hash function to minimize the probability of collision.

PoTC: Power of two choices *without* using key splitting, i.e., traditional PoTC applied to key grouping.

On-Greedy: Online greedy algorithm that picks the least loaded worker to handle a new key.

Off-Greedy: Offline greedy sorts the keys by decreasing frequency and executes On-Greedy.

PKG: PoTC with key splitting.

Note that PKG is the only method that uses key splitting. Off-Greedy knows the whole distribution of keys so it represents an unfair comparison for online algorithms.

Table II shows the results of the comparison on the two main datasets WP and TW. Each value is the average imbalance measured throughout the simulation. As expected, hashing performs the worst, creating a large imbalance in all cases. While PoTC performs better than hashing in all the experiments, it is outclassed by On-Greedy on TW. On-Greedy performs very close to Off-Greedy, which is a good result considering that it is an online algorithm. Interestingly, PKG performs even better than Off-Greedy. Relaxing the constraint of KG allows to achieve a load balance comparable to offline algorithms.

We conclude that PoTC alone is not enough to guarantee good load balance, and key splitting is fundamental not only to make the technique practical in a distributed system, but also to make it effective in a streaming setting. As expected, increasing the number of workers also increases the average imbalance. The behavior of the system is binary: either well balanced or largely imbalanced. The transition between the two states happens when the number of workers surpasses the limit $O(1/p_1)$ described in Section IV, which happens around 50 workers for WP and 100 for TW.

Q2. Given the aforementioned results, we focus our attention on PKG henceforth. So far, it still uses global information about the load of the workers when deciding which choice to make. Next, we experiment with local estimation, i.e., each source performs its own estimation of the worker load, based on the sub-stream processed so far.

We consider the following alternatives:

G: PKG with global information of worker load.

L: PKG with local estimation of worker load and different number of sources, e.g., L_5 denotes $S = 5$.

LP: PKG with local estimation and *periodic probing* of worker load every T_p minutes. For instance, L_5P_1 denotes $S = 5$ and $T_p = 1$. When probing is executed, the local estimate vector is set to the actual load of the workers.

Figure 2 shows the average imbalance (normalized to the size of the dataset) with different techniques, for different number of sources and workers, and for several datasets. The baseline (H) always imposes very high load imbalance on the workers. Conversely, PKG with local estimation (L) has always a lower imbalance. Furthermore, the difference from the global variant (G) is always less than one order of magnitude. Finally, this result is robust to changes in the number of sources.

Figure 3 displays the imbalance of the system through time $I(t)$ for TW, WP and CT, 5 sources, and for $W = 10$ and 50. Results for $W = 5$ and $W = 100$ are omitted as they are similar to $W = 10$ and $W = 50$, respectively. PKG with global information (G) and its variant with local estimation (L_5) perform best. Interestingly, even though both G and L achieve very good load balance, their choices are quite different. In an experiment measuring the agreement on the destination of each message, G and L have only 47% Jaccard overlap. Hence, L reaches a local minimum which is very close in value to the one obtained by G, although different. Also in this case, good balance can only be achieved up to a number of workers that depends on the dataset. When that number is exceeded, the imbalance increases rapidly, as seen in the cases of WP and partially for CT for $W = 50$, where all techniques lead to the same high load imbalance.

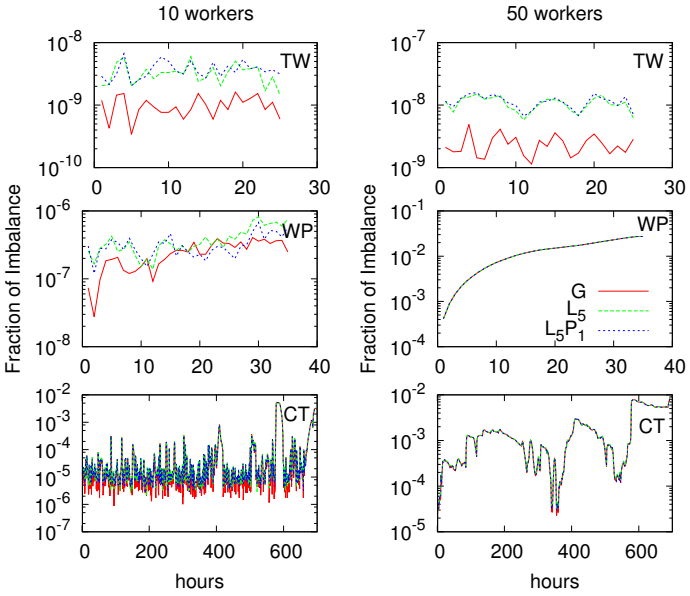

Fig. 3: Fraction of imbalance through time for different datasets, techniques, and number of workers, with $S = 5$.

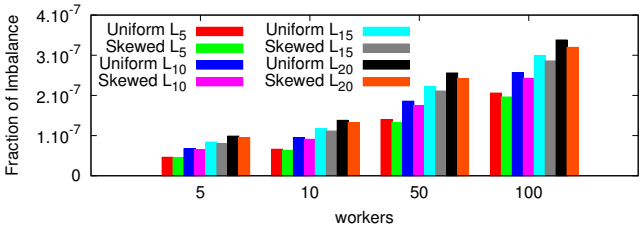

Fig. 4: Fraction of average imbalance with uniform and skewed splitting of the input keys on the sources when using the LJ graph.

Finally, we compare our local estimation strategy with a variant that makes use of periodic probing of workers' load every minute (L_5P_1). Probing removes any inconsistency in the load estimates that the sources may have accumulated. However, interestingly, this technique does not improve the load balance, as shown in Figure 3. Even increasing the frequency of probing does not reduce imbalance (results not shown in the figure for clarity). In conclusion, local information is sufficient to obtain good load balance, therefore it is not necessary to incur the overhead of probing.

Q3. To operationalize this question, we use the directed graphs datasets. We use KG to distribute the messages to the sources to test the robustness of PKG to *skew in the sources*, i.e., when each source forwards an uneven part of the stream. We simulate a simple application that computes a function of the incoming edges of a vertex (e.g., in-degree, PageRank). The input keys for the source PE is the source vertex id, while the key sent to the worker PE is the destination vertex id, that is, the source PE inverts the edge. This schema projects the out-degree distribution of the graph on sources, and the in-degree distribution on workers, both of which are highly skewed.

Figure 4 shows the average imbalance for the experiments with a skewed split of the keys to sources for the LJ social graph (results on SL_1 and SL_2 are similar to LJ and are omitted due to space constraint). For comparison, we include the results when the split is performed *uniformly* using shuffle grouping of keys on sources. On average, the imbalance generated by the skew on sources is similar to the one obtained with uniform splitting. As expected, the imbalance slightly increases as the number of sources and workers increase, but, in general, it remains at very low absolute values.

To answer Q3, we additionally experiment with drift in the skew distribution by using the cashtag dataset (CT). The bottom row of Figure 3 demonstrates that all techniques achieve a low imbalance, even though the change of key popularity through time generates occasional spikes.

In conclusion, PKG is robust to skew on the sources, and can therefore be chained to key grouping. It is also robust to the drift in key distribution common of many real-world streams.

Q4. We implement and test our technique on the streaming top-k word count example, and perform two experiments to compare PKG, KG, and SG on WP. We choose word count as it is one of the simplest possible examples, thus limiting the number of confounding factors. It is also representative of many data mining algorithms as the ones described in Section VI (e.g., counting frequent items or co-occurrences of feature-class pairs). Due to the requirement of real-world deployment on a DSPE, we ignore techniques that require coordination (i.e., PoTC and On-Greedy). We use a topology configuration of a single source along with 9 workers (counters) running on a storm cluster of 10 virtual servers. We report overall throughput, end-to-end latency, and memory usage.

In the first experiment, we emulate different levels of CPU consumption per key by adding a fixed delay to the processing. We prefer this solution over implementing a specific application in order to be able to control the load on the workers. We choose a range that is able to bring our configuration to a saturation point, although the raw numbers would vary for different setups. Even though real deployments rarely operate at saturation point, PKG allows better resource utilization, therefore supporting the same workload on a smaller number of machines. In this case, the minimum delay (0.1ms) corresponds approximately to reading 400kB sequentially from memory, while the maximum delay (1ms) to $\frac{1}{10}$-th of a disk seek.[9] Nevertheless, even more expensive tasks exist: parsing a sentence with NLP tools can take up to 500ms.[10]

The system does not perform aggregation in this setup, as we are only interested in the raw effect on the workers. Figure 5(a) shows the throughput achieved when varying the CPU delay for the three partitioning strategies. Regardless of the delay, SG and PKG perform similarly, and their throughput is higher than KG. The throughput of KG is reduced by $\approx 60\%$

[9] http://brenocon.com/dean_perf.html
[10] http://nlp.stanford.edu/software/parser-faq.shtml\#n

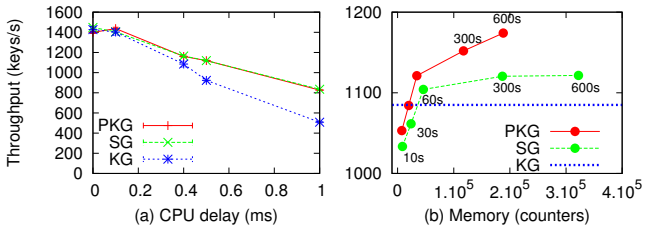

Fig. 5: (a) Throughput for PKG, SG and KG for different CPU delays. (b) Throughput for PKG and SG vs. average memory for different aggregation periods.

when the CPU delay increases tenfold, while the impact on PKG and SG is smaller ($\approx 37\%$ decrease). We deduce that reducing the imbalance is critical for clusters operating close to their saturation point, and that PKG is able to handle bottlenecks similarly to SG and better than KG. In addition, the imbalance generated by KG translates into longer latencies for the application. When the workers are heavily loaded, the average latency with KG is up to 45% larger than with PKG. Finally, the benefits of PKG over SG regarding memory are substantial. Overall, PKG ($3.6M$ counters) requires about 30% more memory than KG ($2.9M$ counters), but about half the memory of SG ($7.2M$ counters).

In the second experiment, we fix the CPU delay to 0.4ms per key, as it is the saturation point for KG in our setup. We activate the aggregation of counters at different time intervals T to emulate different application policies for when to receive up-to-date top-k word counts. In this case, PKG and SG need additional memory compared to KG to keep partial counters. Shorter aggregation periods reduce the memory requirements, as partial counters are flushed often, at the cost of a higher number of aggregation messages. Figure 5(b) shows the relationship between throughput and memory overhead for PKG and SG. The throughput of KG is shown for comparison. For all values of aggregation period, PKG achieves higher throughput than SG, with lower memory overhead and similar average latency per message. When the aggregation period is above 30s, the benefits of PKG compensate its extra overhead and its overall throughput is higher than when using KG.

VI. Applications

PKG is a novel programming primitive for stream partitioning and not every algorithm can be expressed with it. In general, all algorithms that use shuffle grouping can use PKG to reduce their memory footprint. In addition, many algorithms expressed via key grouping can be rewritten to use PKG in order to get better load balancing. In this section we provide a few such examples of common data mining algorithms, and show the advantages of PKG. Henceforth, we assume that each message contains a data point for the application, e.g., a feature vector in a high-dimensional space.

A. Naïve Bayes Classifier

A naïve Bayes classifier is a probabilistic model that assumes independence of features. It estimates the probability of a class C given a feature vector X by using Bayes' theorem. In practice, the classifier works by counting the frequency of co-occurrence of each feature and class values.

The simplest way to parallelize this algorithm is to spread the counters across several workers via vertical parallelism, i.e., each feature is tracked independently in parallel. Following this design, the algorithm can be implemented by the same pattern used for the KG example in Section II-A. Sparse datasets often have a skewed distribution of features, e.g., for text classification. Therefore, this implementation suffers from the same load imbalance, which PKG solves.

Horizontal parallelism can also be used to parallelize the algorithm, i.e., by shuffling messages to separate workers. This implementation uses the same pattern as the DAG in the SG example in Section II-A. The count for a single feature-class pair is distributed across several workers, and needs to be combined at prediction (query) time. This combination requires broadcasting the query to all the workers, as a feature can be tracked by any worker. This implementation, while balancing the work better than key grouping, requires an expensive query stage that may be affected by stragglers.

PKG tracks each feature on two workers and avoids replicating counters on all workers. Furthermore, the two workers are deterministically assigned for each feature. Thus, at query time, the algorithm needs to probe only two workers for each feature, rather than having to broadcast it to all the workers. The resulting query phase is less expensive and less sensitive to stragglers than with shuffle grouping.

B. Streaming Parallel Decision Tree

A decision tree is a classification algorithm that uses a tree-like model where nodes are tests on features, branches are possible outcomes, and leafs are class assignments.

Ben-Haim and Tom-Tov [1] propose an algorithm to build a streaming parallel decision tree that uses approximated histograms to find the test value for continuous features. Messages are shuffled among W workers. Each worker generates histograms independently for its sub-stream, one histogram for each feature-class-leaf triplet. These histograms are then periodically sent to a single aggregator that merges them to get an approximated histogram for the whole stream. The aggregator uses this final histogram to grow the model by taking split decisions for the current leaves in the tree. Overall, the algorithm keeps $W \times D \times C \times L$ histograms, where D is the number of features, C is the number of classes, and L is the current number of leaves.

The memory footprint of the algorithm depends on W, so it is impossible to fit larger models by increasing the parallelism. Moreover, the aggregator needs to merge $W \times D \times C$ histograms each time a split decision is tried, and merging the histograms is one of the most expensive operations.

Instead, PKG reduces both the space complexity and aggregation cost. If applied on the features of each message, a single

feature is tracked by two workers, with an overall cost of only $2 \times D \times C \times L$ histograms. Furthermore, the aggregator needs to merge only two histograms per feature-class-leaf triplet. This scheme allows to alleviate memory pressure by adding more workers, as the space complexity does not depend on W.

C. Heavy Hitters and Space Saving

The heavy hitters problem consists in finding the top-k most frequent items occurring in a stream. The SPACESAVING [23] algorithm solves this problem approximately in constant time and space. Recently, Berinde et al. [2] have shown that SPACESAVING is space-optimal, and how to extend its guarantees to merged summaries. This result allows for parallelized execution by merging partial summaries built independently on separate sub-streams.

In this case, the error bound on the frequency of a single item depends on a term representing the error due to the merging, plus another term which is the sum of the errors of each individual summary for a given item i:

$$| \hat{f}_i - f_i | \leq \Delta_f + \sum_{j=i}^{W} \Delta_j$$

where f_i is the true frequency of item i and $\hat{f}_i$ is the estimated one, each Δ_j is the error from summarizing each sub-stream, while Δ_f is the error from summarizing the whole stream, i.e., from merging the summaries.

Observe that the error bound depends on the parallelism level W. Conversely, by using KG, the error for an item depends only on a single summary, thus it is equivalent to the sequential case, at the expense of poor load balancing.

Using PKG we achieve both benefits: the load is balanced among workers, and the error for each item depends on the sum of only two error terms, regardless of the parallelism level. However, the individual error bounds may depend on W.

VII. RELATED WORK

Various works in the literature either extend the theoretical results from the power of two choices, or apply them to the design of large-scale systems for data processing.

Theoretical results. Load balancing in a DSPE can be seen as a balls-and-bins problem, where m balls are to be placed in n bins. The power of two choices has been extensively researched from a theoretical point of view for balancing the load among machines [4, 20]. Previous results consider each ball equivalent. For a DSPE, this assumption holds if we map balls to messages and bins to servers. However, if we map balls to *keys*, more popular keys should be consider to be heavier. [21] tackle the case where each ball has a weight drawn independently from a fixed weight distribution $\mathcal{X}$. They prove that, as long as $\mathcal{X}$ is "smooth", the expected imbalance is independent of the number of balls. However, the solution assumes that $\mathcal{X}$ is known beforehand, which is not the case in a streaming setting. Thus, in our work we take the standard approach of mapping balls to messages.

Another assumption common in previous works is that there is a single source of balls. Existing algorithms that extend POTC to multiple sources execute several rounds of intra-source coordination before taking a decision [16, 19, 24]. Overall, these techniques incur a significant coordination overhead, which becomes prohibitive in a DSPE that handles thousands of messages per second.

Stream processing systems. Existing load balancing techniques for DSPEs are analogous to key grouping with rebalancing [5, 8, 9, 10, 11, 12]. In our work, we consider operators that allow replication and aggregation, similar to a standard combiner in map-reduce, and show that it is sufficient to balance load among two replicas based local load estimation. We refer to Section II-A for a more extensive discussion of key grouping with rebalancing. Flux monitors the load of each operator, ranks servers by load, and migrates operators from the most loaded to the least loaded server, from the second most loaded to the second least loaded, and so on [5]. Aurora* and Medusa propose policies to migrating operators in DSPEs and federated DSPEs [8]. Borealis uses a similar approach but it also aims at reducing the correlation of load spikes among operators placed on the same server [9]. This correlation is estimated by using a finite set of load samples taken in the recent past. Gedik [10] developed a partitioning function (a hybrid between explicit mapping and consistent hashing of items to servers) for stateful data parallelism in DSPEs that leverages item frequencies to control migration cost and imbalance in the system. Similarly, Balkesen et al. [11] proposed frequency-aware hash-based partitioning to achieve load balance. Castro Fernandez et al. [12] propose integrating common operator state management techniques for both checkpointing and migration.

Other distributed systems. Several storage systems use consistent hashing to allocate data items to servers [25]. Consistent hashing substantially produces a random allocation and is designed to deal with systems where the set of servers available varies over time. In this paper, we propose replicating DSPE operators on two servers selected at random. One could use consistent hashing also to select these two replicas, using the replication technique used by Chord [26] and other systems.

Sparrow [27] is a stateless distributed job scheduler that exploits a variant of the power of two choices [24]. It employs batch probing, along with late binding, to assign m tasks of a job to the least loaded of $d \times m$ randomly selected workers ($d \geq 1$). Sparrow considers only independent tasks that can be executed by any worker. In DSPEs, a message can only be sent to the workers that are accumulating the state corresponding to the key of that message. Furthermore, DSPEs deal with messages that arrive at a much higher rate than Sparrow's fine-grained tasks, so we prefer to use local load estimation.

In the domain of graph processing, several systems have been proposed to solve the load balancing problem, e.g., Mizan [28], GPS [29], and xDGP [30]. Most of these systems perform dynamic load rebalancing at runtime via vertex migration. We have already discussed why rebalancing is impractical in our context in Section II.

Finally, SkewTune [31] solves the problem of load balancing in MapReduce-like systems by identifying and redistributing the unprocessed data from the stragglers to other workers. Techniques such as SkewTune are a good choice for batch processing systems, but cannot be directly applied to DSPEs.

VIII. Conclusion

Despite being a well-known problem in the literature, load balancing has not been exhaustively studied in the context of distributed stream processing engines. Current solutions fail to provide satisfactory load balance when faced with skewed datasets. To solve this issue, we introduced Partial Key Grouping, a new stream partitioning strategy that allows better load balance than key grouping while incurring less memory overhead than shuffle grouping. Compared to key grouping, PKG is able to reduce the imbalance by up to several orders of magnitude, thus improving throughput and latency of an example application by up to 45%.

This work gives rise to further interesting research questions. Is it possible to achieve good load balance without foregoing atomicity of processing of keys? What are the necessary conditions, and how can it be achieved? In particular, can a solution based on rebalancing be practical? And in a larger perspective, which other primitives can a DSPE offer to express algorithms effectively while making them run efficiently? While most DSPEs have settled on just a small set, the design space still remains largely unexplored.

References

[1] Y. Ben-Haim and E. Tom-Tov, "A Streaming Parallel Decision Tree Algorithm," *JMLR*, vol. 11, pp. 849–872, 2010.

[2] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM Trans. Database Syst.*, vol. 35, no. 4, pp. 1–28, 2010.

[3] G. H. Gonnet, "Expected length of the longest probe sequence in hash code searching," *J. ACM*, vol. 28, no. 2, pp. 289–304, 1981.

[4] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, 2001.

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

[6] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDM Workshops*, 2010, pp. 170–177.

[7] M. G. Noll, "Implementing Real-Time Trending Topics With a Distributed Rolling Count Algorithm in Storm," www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm, 2013.

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

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

[10] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, pp. 1–23, 2013.

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

[12] 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*, 2013, pp. 725–736.

[13] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[14] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1-2, pp. 515–529, 2010.

[15] H. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *SIGMOD*, 2007, pp. 1029–1040.

[16] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, "Parallel Randomized Load Balancing," in *STOC*, 1995, pp. 119–130.

[17] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM J. Comput.*, vol. 29, no. 1, pp. 180–200, 1999.

[18] J. Byers, J. Considine, and M. Mitzenmacher, "Geometric generalizations of the power of two choices," in *SPAA*, 2003, pp. 54–63.

[19] C. Lenzen and R. Wattenhofer, "Tight bounds for parallel randomized load balancing: Extended abstract," in *STOC*, 2011, pp. 11–20.

[20] M. Mitzenmacher, R. Sitaraman *et al.*, "The power of two random choices: A survey of techniques and results," in *Handbook of Randomized Computing*, 2001, pp. 255–312.

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

[22] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing user behavior in online social networks," in *IMC*, 2009.

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

[24] G. Park, "A Generalization of Multiple Choice Balls-into-bins," in *PODC*, 2011, pp. 297–298.

[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *STOC*, 1997, pp. 654–663.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[27] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *SOSP*, 2013, pp. 69–84.

[28] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *ECCS*, 2013, pp. 169–182.

[29] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *ICSSDM*, 2013, p. 22.

[30] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xDGP: A Dynamic Graph Processing System with Adaptive Partitioning," *arXiv*, vol. abs/1309.1049, 2013.

[31] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating Skew in MapReduce Applications," in *SIGMOD*, 2012, pp. 25–36.