Load Balancing for Skewed Streams on Heterogeneous Cluster

Muhammad Anis Uddin Nasir[$#], Hiroshi Horii[$], Marco Serafini[*], Nicolas Kourtellis[‡]
Rudy Raymond[$], Sarunas Girdzijauskas[#], Takayuki Osogami[$],

[#]Royal Institute of Technology, Sweden [$]IBM Research Tokyo, Japan [*]Qatar Computing Research Institute [‡]Telefonica Research
anisu@kth.se, horii@jp.ibm.com, mserafini@qf.org.qa, nicolas.kourtellis@telefonica.com, rudyhar@jp.ibm.com
sarunasg@kth.se, osogami@jp.ibm.com

ABSTRACT

Primitive partitioning strategies for streaming applications operate efficiently under two very strict assumptions: the resources are homogeneous and the messages are drawn from a uniform key distribution. These assumptions are often not true for the real-world use cases. Dealing with heterogeneity and non-uniform workload requires inferring the resource capacities and input distribution at run time. However, gathering these statistics and finding an optimal placement often become a challenge when microsecond latency is desired. In this paper, we address the load balancing problem for streaming engines running on a heterogeneous cluster and processing skewed workload. In doing so, we propose a novel partitioning strategy called *Consistent Grouping* (CG) that is inspired by traditional consistent hashing. CG is a lightweight distributed strategy that enables each processing element instance (PEI) to process the workload according to its capacity. The main idea behind CG is the notion of equal-sized virtual workers at the sources, which are assigned to workers based on their capacities. We provide a theoretical analysis of the proposed algorithm and show via extensive empirical evaluation that the proposed scheme outperforms the state-of-the-art approaches. In particular, CG achieves 3.44x superior performance in terms of latency compared to key grouping, which is the state-of-the-art grouping strategy for stateful streaming applications.

1 INTRODUCTION

Distributed stream processing engines (DSPEs) have recently gained much attention due to their ability to process huge volumes of data with very low latency on clusters of commodity hardware. DSPEs enable processing information that is produced at a very fast rate in a variety of contexts, such as IoT applications, software logs, and social networks. For example, Twitter users generate more than 380 million tweets per day[1] and Facebook users upload more than 300 million photos per day[2].

Streaming applications are represented by directed acyclic graphs (DAGs), where vertices are called *processing elements* (PEs) and represent operators, and edges are called *streams* and represent the data flowing from one PE to the next. For scalability, streams are partitioned into sub-streams and processed in parallel on replicas of PEs called *processing element instances* (PEI).

Streaming applications perform light-weight operations such as filtering, aggregating, or joining, on the incoming data streams, to analyze the information in real time. For example, a typical application is the detection of trending hashtags in a stream of tweets. In this case, the PEIs responsible for counting the occurrences of the hashtags trending hashtags also receive a predominant share of the messages in the stream. The same behavior can be observed in other domains such as classification (by grouping on classes and attributes), statistical language models (grouping on words), and streaming graph processing (grouping on vertices).

Applications of DSPEs, especially in data mining and machine learning, typically require accumulating state across the stream by grouping the data on common fields [4, 5]. Akin to MapReduce, this grouping in DSPEs is usually called *key grouping* (KG) and is implemented using hashing [32]. KG allows each source PEI to route each message solely via its key, without needing to keep any state or to coordinate among PEIs. However, KG is unaware of the underlying skewness in the input streams [25], which causes a few PEIs to sustain a significantly higher load than others, as demonstrated in Figure 1 with a toy example. This sub-optimal load balancing leads to poor resource utilization and inefficiency.

The problem is further complicated when the underlying resources are heterogeneous [23, 38] or changing over time [42, 48]. For various commercial enterprises, the resources available for stream mining consist of dedicated machines, private clouds, bare metal, virtualized data centers and commodity hardware. For streaming applications, the heterogeneity is often invisible to the upstream PEIs and requires inferring the resource capacities in order to generate a fair assignment of the tasks to the downstream PEIs. However, gathering statistics and finding optimal placement often leads to bottlenecks, while at the same time microsecond latencies are desired [19].

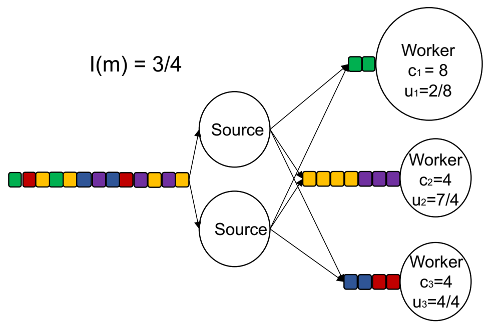

FIGURE 1: **Example showing that key grouping generates imbalance in the presence of a heterogeneous cluster. The capacity and the resource utilization of the i-th worker is represented by c_i and u_i respectively. Each key ($j \in \mathcal{K}$) is represented with different color box. Imbalance $I(m)$ is the difference between the maximum and the average resource utilization (see section 3 for details).**

[1]http://www.internetlivestats.com/twitter-statistics/
[2]http://www.businessinsider.com/facebook-350-million-photos-each-day-2013-9

Conference'17, Washington, DC, USA
2017. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Alternatively, stateless streaming applications, like interaction with external data sources, employ *shuffle grouping* (SG) to break down the stream load equally to each of the PEIs, i.e., by sending a message to a new PEI in cyclic order, irrespective of its key. SG allows each source PEI to send equal number of messages to each downstream PEI, without the need to keep any state or to coordinate among PEIs. However, similarly to KG, SG is unaware of the heterogeneity in the cluster, which can cause some PEIs to sustain unpredictably higher load than others. Further, SG typically requires more memory to express stateful computations [32].

In this present work, we study the load balancing problem for a streaming engine running on a heterogeneous cluster and processing non-uniform workload. We envision a light-weight and fair key grouping strategy for both stateless and stateful streaming applications. Moreover, this strategy must limit the number of workers processing each key, which is analogous to reducing the memory footprint and aggregation cost for the stateful computation [32]. Towards this goal, we propose a novel grouping strategy called *Consistent Grouping* (CG), which handles both the potential skewness in input data distribution, as well as the heterogeneity in resources in DSPEs. CG borrows the concept of virtual workers from the traditional consistent hashing [13, 14] and employs rebalancing to achieve fair assignment, similar to [3, 7, 12, 40, 42].

In summary, our work makes the following contributions:

- We study the load balancing problem for DSPEs running on heterogeneous cluster and processing skewed workload.
- We propose a novel grouping scheme called Consistent Grouping that provides fair assignment of messages to downstream operators by adapting to traditional consistent hashing.
- We provide a theoretical analysis of the proposed scheme and show the effectiveness of the proposed scheme via extensive empirical evaluation on synthetic and real-world datasets.
- We measure the impact of CG 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 2x, reduces the latency by 3.44x.

2 OVERVIEW OF THE APPROACH

Consistent grouping relies on the concept of virtual workers and allows variable number of *virtual workers* for each PEI. The main idea behind CG is to assign the input stream to the virtual workers in a way that each virtual worker has almost the same number of messages. Later, these virtual workers are assigned to the actual workers[3] based on their capacity. The same idea has been considered in the past in the context of distributed hash tables [13, 14]. CG allows an assignment of tasks to PEIs based on the capacity of the PEIs. Thus, the powerful PEIs are assigned more work compared to less powerful PEIs. The adaptation of CG in the streaming context requires answering several challenging questions: 1) How do we divide the messages into equal-sized virtual workers? , 2) How do we identify the imbalance in load on workers due to heterogeneity of their resources? and 3) How do we plan the migration?

First, we propose a novel strategy called *power of random choices* (PORC), which assigns the incoming messages to a set of equal sized virtual workers. The basic idea behind this scheme is to introduce

[3]We refer to downstream PEIs as workers and to upstream PEIs as sources throughout the paper

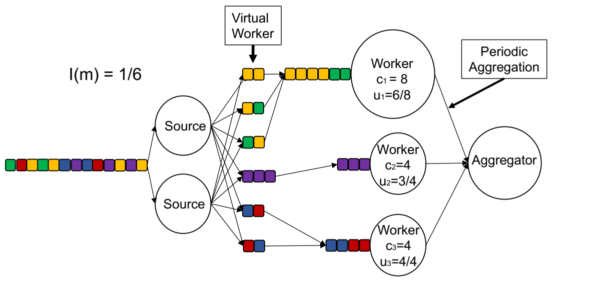

FIGURE 2: **Example showing that consistent grouping improves the imbalance in the presence of heterogeneous cluster, compared to key grouping. The capacity and the resource utilization of the i-th worker is represented by c_i and u_i respectively. Also, the each key ($j \in \mathcal{K}$) is represented with different color box. Imbalance $I(m)$ is the difference between the maximum and the average resource utilization.**

the notion of capacity for the virtual workers. In particular, we set the capacity of each virtual worker to the ceiling of the average load $\times$ (1+ϵ), for some parameter ϵ. Note that the capacity is calculated at run time using the average load. Given infinitely many hash functions that produce fixed set of choices for a assignment of a message to a virtual worker, PORC maps a key to the first virtual worker with a spare capacity. PORC allows the heavy keys to spread across the other virtual workers, thus reducing the memory footprint and the aggregation cost. The ϵ parameter in the algorithm provides the trade off between the imbalance and memory footprint.

Second, CG takes a radically different approach towards load balancing and delegates the problem to the PEIs by allowing them to decide their workload based on their capacities. We call this component as *worker delegation*. Each worker monitors its workload and sends a binary signal (increase or decrease workload) to the upstream operator(s) in case it experiences excessive workload. This simple modification changes the distributed load balancing problem to a local decision problem, where each PEI can choose its share of workload based on its current capacity. Moreover, worker delegation provides the flexibility to implement various application-specific requirements at each PEI. The upstream PEIs react to the signals by moving virtual workers from one PEI to another. Such deployments might negatively impact the performance of a streaming application, as it requires one-to-many broadcast messages across the network. To overcome this challenge, we relax the consistency constraint in the DAG and allow operators to be eventually consistent. Specifically, we propose *piggybacking* that allows encoding the binary signals along with the acknowledgment message to avoid the overhead.

Lastly, CG ensures that each message is processed in a consistent manner by discarding the message migration phase. When the upstream PEI receives a request to change (increase or decrease) the workload, CG relocates virtual workers assigned to the overloaded worker, thus, only affecting the future routing of the messages. Concretely, each worker processes the messages that are assigned to it; changes in the routing only affect the messages that arrive in a later time. CG follows the same programming primitive as PKG for stream partitioning; supporting both stateless and stateful map-reduce like applications [32]. We propose *periodic aggregation* to support such operators, which leverages the existing DAG and imposes a very low-overhead in the stream application.

Figure 2 provides an example using CG for the DAG in Figure 1. CG achieves the fair assignment of tasks by moving a virtual worker from the low capacity worker to the high capacity worker. Experiments show that CG outperforms other approaches in terms of throughput and latency while providing significant improvement in terms of imbalance.

3 PRELIMINARIES & PROBLEM DEFINITION

This section introduces the preliminaries that are used in the rest of the paper. We consider a DSPE running on a cluster of machines that communicate by exchanging messages following the flow of a DAG. For scalability, streams are partitioned into sub-streams and processed in parallel on a replica of the PE called *processing element instance* (PEI). 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 $|\mathcal{W}| = n$.

Each machine $w \in \mathcal{W}$ has a limited capacity, which is represented by $c_w \in C$. For simplicity, we assume that there is a single important resource on which nodes are constrained, such as storage, processing. Moreover, each worker ($w \in \mathcal{W}$) has an unbounded input queue (Q_w).

The input to the engine is a sequence of messages $z = \langle i, j, v, t \rangle$ where i is the identifier, $j \in \mathcal{K}, |\mathcal{K}| = m$ is the message key, v is the value, and t is the timestamp at which the message is received. The messages are presented to the engine in ascending order by timestamp. Upon receiving a message with key $j \in \mathcal{K}$, we need to decide its placement among the workers. We assume one message arrives per unit of time.

We employ queuing theory as the cost model to define the delay and the overhead at each worker. In the model, a sequence of messages arrives at a worker $w \in \mathcal{W}$. If the new message finds the worker occupied, it waits in the queue until its turn to be served comes. After the message is processed, it leaves the system. We represent the finish time for a message i using ϕ_i. The difference between the arrival time and the ϕ_i represents the message execute latency.

We define a *partitioning* function $\mathcal{H} : \mathcal{K} \rightarrow \mathcal{W}$, which maps each message of one of the PEIs. This number identifies the PEI responsible for processing the message. Each PEI is associated to one or more keys. The goal of the partitioning function is to generate an assignment of messages to the set of workers in a way that average waiting time is minimized.

In this paper, we focus on providing a generalized framework for load balancing that is capable for adapting to any definition of load imbalance. We define the *queue length* of a worker using the number of messages that are pending in the queue. At time t, the *queue length* of a worker w is defined by:

$$L_w(t) = |\{i : \mathcal{H} = w \wedge \phi_i > t\}|, \text{ for } w \in \mathcal{W}$$

Also, we define *utilization* at time t as ratio between the *queue length* and the capacity of the worker.

$$\mathcal{U}_w(t) = \frac{L_w(t)}{c_w}$$

We use a definition of *imbalance* similar to others in the literature (e.g., Flux [40] and PKG [32]). We define *imbalance* at time t as

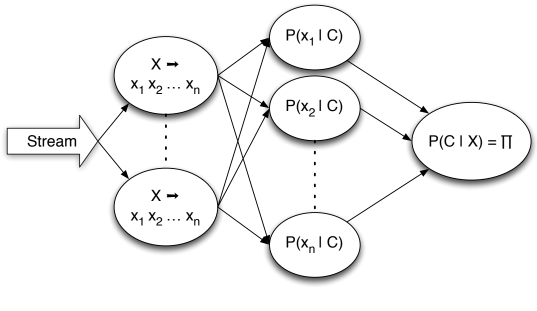

FIGURE 3: **Naïve Bayes implemented via key grouping (KG).**

the difference between the maximum and the average resource utilization:

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

Problem. Given the definition of imbalance, we consider the following problem in this paper.

PROBLEM 3.1. *Given a stream of messages drawn from a heavy-tailed distribution $\mathcal{K}$ and and set of workers $w \in \mathcal{W}$ with capacities $c_w \in C$, find a partitioning function $\mathcal{H}$ that minimizes the imbalance ($I(t)$) at any time instance t.*

Memory Cost. One simple solution to address problem 3 is to employ round robin assignment, which provides an imbalance of at most one. This load balance comes at the cost of memory as message with the same key might end up on all the workers. In our work, we would like to bound the memory by limiting the number of workers processing each key.

Example. To make the discussion more concrete, we introduce a simple application that will be our running example: the *naïve Bayes classifier.* A naïve Bayes classifier is a probabilistic model that assumes independence of features in the data (hence the naïve). It estimates the probability of a class C given a feature vector X by using Bayes' theorem:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}.$$

The answer given by the classifier is then the class with maximum likelihood

$$C^* = \arg\max_C P(C|X).$$

Given that features are assumed independent, the joint probability of the features is the product of the probability of each feature. Also, we are only interested in the class that maximizes the likelihood, so we can omit $P(X)$ from the maximization as it is constant. The class probability is proportional to the product

$$P(C|X) \propto \prod_{x_i \in X} P(x_i|C)P(C),$$

which reduces the problem to estimating the probability of each feature value x_i given a class C, and a prior for each class C.

In practice, the classifier estimates the probabilities by counting the frequency of co-occurrence of each feature and class value. Therefore, it can be implemented by a set of counters, one for each pair of feature value and class value. A MapReduce implementation is straightforward, and available in Apache Mahout.[4]

[4]https://mahout.apache.org/users/classification/bayesian.html

4 BACKGROUND

In this section, we provide the brief summary of the state-of-the-art streaming solutions and discuss other possible solutions for our problem.

4.1 Existing Stream Partitioning Functions

Messages are 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 three main primitives of interest: *key grouping*, *partial key grouping* and *shuffle grouping*.

Key Grouping (KG). KG partitioning ensures the messages with the same key are handled by the same PEI (analogous to MapReduce). It is usually implemented through hashing. KG is the perfect choice for *stateful* operators. It allows each source PEI to route each message solely via its key, without needing to keep any state or to coordinate among PEIs. However, KG is unaware of the underlying skewness in the input distribution, which causes a few PEIs to sustain a significantly higher load than others. This sub optimal load balancing leads to poor resource utilization and inefficiency.

Partial Key Grouping (PKG). PKG [31–33] adapts to the traditional power of two choices for load balancing in map-reduce like streaming operators. PKG guarantees nearly perfect load balance in the presence of skew using two novel schemes: key splitting and local load estimation. The local load estimation enables each upstream PEI to predict the load of downstream PEIs leveraging the past history. However, similar to KG, PKG assumes that each downstream PEI has same resources and the service time for the messages follows a uniform distribution, which is a strong assumption of many real-world use cases.

Shuffle Grouping (SG). SG partitioning forwards messages independently, typically in a round-robin fashion. It 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. It 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.

4.2 Consistent Hashing.

Consistent Hashing (CH) is a special form of a hash function that requires minimal changes as the range of the function changes[21]. CH solves the assignment problem by substantially producing a random allocation. It relies on a standard hash function that maps both messages and workers unit-size circular ID space, i.e., $[0, 1) \subseteq \mathbb{R}$. Further, each task is assigned to the first worker that is encountered moving in the clockwise direction on the unit circle. For implementation, the hash value for all the workers are stored in a binary search tree, and the clockwise successor can be found via single search. CH provides load balancing guarantees across the set of workers. Given that the load on a node is proportional to the size of the interval it owns, no worker owns more than $O\left(\frac{\log n}{n}\right)$ of the interval (to which each task is mapped) [21].

One common solution to improve the load balance is to introduce *virtual workers*, which are copies of workers points in the circle. Whenever, a new worker is added, a fixed number of copies of the worker are also created in the circle. As each worker is responsible for an interval on the unit circle, creating virtual copies of a worker spread the workload for each worker across the unit circle. Thus, *virtual workers* enables CH to achieve better load balancing across the set of workers. Similar to other stream partitioning functions, CH is unaware of both the heterogeneity in the cluster and skewness in the input stream, which restricts its immediate applicability in the streaming context.

Hash Space Adjustment. One possible solution to deal both heterogeneity and skewness is to employ hash space adjustment for consistent hashing [18]. Such schemes require global knowledge of the tasks assignment to each worker to adjust the hash space for the workers, i.e., movement of tasks from the overloaded worker to the least loaded worker. Even though such schemes provide efficient results in terms of load balance, their applicability in stream context incurs an additional overhead due to many-to-many communication across workers. Also, if implemented without global information, these scheme may produce unpredictable imbalance due to random task movement across workers.

Consistent Hashing with Bounded Load. Independent from our work, Mirrokni et al. [28] proposed a novel version of consistent hashing scheme that provides a constant bound of the load of the maximum loaded worker. The basic idea behind their scheme is to introduce the notion of capacity for each worker. In particular, set the capacity of each bin to either floor or ceiling of the average load times $(1+\epsilon)$, for some parameter ϵ. Further, the tasks are assigned to a worker in the clockwise direction with spare capacity. CH guarantees that the load of the maximum loaded bin is at most $(1+\epsilon)$ factor of the average load.

4.3 Other Approaches

Power of a Two Choice (POTC). POTC enables achieving the load balance by first selecting two bins uniformly at random and later assigning the message to the least loaded of the two bins. For POTC, the load of each bin solely based on the number of messages. Using POTC, each key might be assigned to any of the workers. Therefore, the memory requirement in worst case is proportional to the number of workers, i.e., every key appearing on all the workers.

Greedy Scheduling. Sparrow [34] is a stateless distributed job scheduler that exploits a variant of the power of two choices [35]. It employs batch probing, along with late binding, to assign m messages of a job to the least loaded of $d \times m$ randomly selected workers ($d \geq 1$). The applicability of such schemes in the context of streaming is not clear as both probing and late binding can significantly affect the latency per message.

Rebalancing. Another way to achieve fair assignment is to leverage *rebalancing* [3, 7, 11, 40, 42]. Once a situation of load imbalance is detected, the system activates a rebalancing routine that moves part of the messages, and the state associated with them, away from an overloaded worker. While this solution is easy to understand, it applicability in the streaming context requires answering several challenging questions: How to identify the imbalance and how to plan the migration. The answer to these questions 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. For these reasons, rebalancing creates a difficult engineering challenge, which we address in our paper.

5 SOLUTION

In this section, we discuss our solution and its various components. Given the set of sources and the set of workers, the goal is to design a grouping strategy that is capable of assigning the messages to the workers proportional to their capacity.

Overview. In our work, we propose a novel grouping scheme called consistent grouping (cg). Our scheme borrows the concept of virtual workers from the traditional consistent hashing [13, 14] and employs rebalancing to achieve fair assignment, similar to [3, 7, 12, 40, 42]. cg allows variable number of *virtual workers* for each pei. The main idea behind cg is to assign the input stream to the virtual workers in a way that each virtual worker has almost the same number of messages. Later, these virtual workers are assigned to the workers based on their capacity. One of the challenges is to bound the load of each virtual worker as it implies that moving a virtual from one worker to another actually increases and decreases the workload of corresponding worker. In doing so, we propose a novel grouping strategy called *power of random choices* that is capable of providing bounded imbalance while keeping the memory cost low. Further, we propose three efficient schemes: *worker delegation*, *piggybacking* and *periodic aggregation*, which enable efficient integration of our proposed scheme into dspes.

5.1 Power of Random Choices

In this section, we propose a novel grouping strategy called *power of both choices* (porc). porc is a hybrid scheme between potc and pkg. It assigns the incoming messages to the set of virtual workers in a way that the imbalance is bounded and the overall memory footprint of the keys on the virtual workers is low. The basic idea behind this scheme is to introduce the notion of capacity for each virtual worker. In particular, we set the capacity of each virtual worker to the ceiling of the average load times $(1+\epsilon)$, for some parameter ϵ. Note that the definition of capacity is based on the average load, rather than a hard constraint. Given infinitely many hash functions $\mathcal{H}_1, \mathcal{H}_2, \ldots$ that produce fixed set of choices for a assignment of a message to a virtual worker, the algorithm maps a key to the first virtual worker with the spare capacity. The infinitely many choices for a key can be produced by using a single hash function and concatenating the *salt* in the key to produce a new assignment[5]. We refer to the virtual worker produced by the first hash function $\mathcal{H}_1$ as the *principal virtual worker*. The rational behind this approach is that the heavy keys in the skewed input distribution overload their principal worker. Therefore, we allow the heavy keys to spread across the other virtual workers, which reduces the memory footprint compared to other schemes, e.g., round robin. The ϵ parameter in the algorithm provides the trade off between the imbalance and memory footprint. porc provides an efficient and generalized solution for the fundamental problem of load balancing for the skewed stream in streaming settings while minimizing the memory footprint [32, 33]. In our work, we adapt porc for fair load balancing for streaming applications, which shows its effectiveness and applicability.

Discussion. To show the effectiveness of porc, we compare its performance with kg, pkg, potc, sg, and ch in terms of imbalance and memory footprint (see section 4 for the description of other schemes) . We leverage the zipf dataset with different skews for this experiment (see section 7 for the description of the dataset).

[5]https://datarus.wordpress.com/2015/05/04/fighting-the-skew-in-spark/

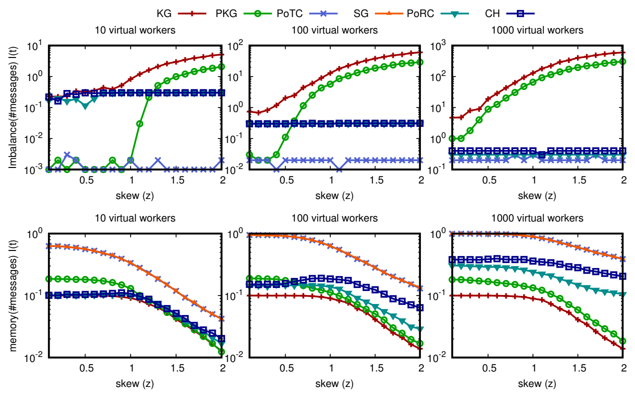

FIGURE 4: **Experiment reporting the normalized imbalance and the memory overhead for Hashing (H), Partial Key Grouping (PKG), Power of Two Choices (PoTC), Power of a Random Choices (PoRC), Consistent Hashing (CH) and Shuffle Grouping (SG) for zipf distribution with different skew and number of virtual workers.**

TABLE 1: **Normalized imbalance when varying the number of virtual workers for the Wikipedia (WP) and Twitter (TW) datasets. We set the value of $\epsilon = 0.3$ for potc and ch.**

Dataset	WP				TW			
W	10^1	10^2	10^3	10^4	10^1	10^2	10^3	10^4
kg	0.8	9.15	93	931	2.2	25	246	2469
pkg	8e-7	3.67	45.59	464	1.52	11.34	22	1233
potc	8e-7	8e-6	9e-5	1e-3	9.5e-8	2e-6	2e-5	3e-4
sg	4e-7	3e-6	4e-6	2.8e-4	0	2e-7	6e-6	4e-5
porc	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
ch	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3

Figure 4 reports the imbalance for different schemes for different number of virtual workers, i.e., 10, 100, and 1000. Results show that both hashing and partial key grouping generate high imbalance as the skew and the number of virtual workers increase. However, the other schemes perform fairly well in terms of imbalance. Similarly, Table 1 shows the imbalance for the Wikipedia and Twitter dataset when varying the number of virtual workers. Additionally, we report the memory overhead for all the schemes in Figure 4. The memory cost is calculated using the total number of unique keys that appear at each virtual worker. Results verify our claim that load balance is achieved at the cost of memory.

5.2 Consistent Grouping

We propose a novel grouping strategy called *Consistent Grouping* (cg) that is inspired by consistent hashing. cg borrows the concept of virtual workers from traditional consistent hashing and allows variable number of virtual workers for each pei [13, 14]. It is a dynamic grouping strategy that is capable of handling both the heterogeneity in the resources and the variability in the input stream at runtime. cg achieves its goal by allowing the powerful workers to acquire additional virtual workers, which leads to stealing work from the other workers. Moreover, it allows overloaded workers to gracefully revoke some of its existing virtual workers, which is equivalent to giving up on some of the allocated work.

CG is a lightweight and distributed scheme that allows assignment of messages to the workers in a streaming fashion. Moreover, it leverages PORC for assignment of keys to each virtual worker in a balanced manner, which allows it to bound the load of each virtual worker. CG is able to balance the load across workers based on their capacities, which allows the DSPEs to operate under adverse scenarios like, heterogeneous clusters and variable workloads.

Time Slot. Before moving the discussion further, we introduce the notion of *time slot* (t_0), which represents the minimum monitoring time period for a PEIS. t_o is an administrative preference that can be determined based on workload traffic patterns. If workloads are expected to change on an hourly basis, setting t_0 on the order of minutes will typically suffice. For slower changing workloads t_0 can be set to an hour. Time slot guarantees that downstream PEIS have enough sample of the input stream to predict its workload.

5.2.1 Load Reduction. Similar to consistent hashing, CG initializes with the same number of virtual workers for each worker, i.e., $O(\log n)$. CG manages a unit-size circular ID space, i.e., $[0, 1) \subseteq \mathbb{R}$ and maps the virtual workers and keys on the unit-size ID space. The main idea is to assign lower number of virtual workers for low capacity workers and higher number of virtual workers for high capacity workers. However, it is not clear how CG can reassign the keys or move the virtual workers ensuring that the load of each worker is proportional to its capacity. Ideally, we would like a scheme that is capable of monitoring the load at each worker throughout the lifetime of a streaming application and adjust the load according to the available capacity of the workers. In doing so, we introduce a novel scheme called *pairing virtual workers*:

Pairing virtual workers. The load of a worker equals the sum of load of the assigned virtual workers. Further, the load of each virtual worker equals to the load that is induced by the mapped messages. Ideally, we would like to assign one of the virtual workers from the overloaded worker to one of the idle workers. However, it is not trivial until this point on how one can achieve such an assignment. To enable such an assignment, we propose to maintain two FCFS queue: *idle* and *busy*. These queues maintain the list of idle and busy workers in the DSPE and allow CG to pair any removal and addition with the opposite to keep the number of virtual workers same throughout the execution. For instance, when a worker is overloaded, it sends a message to upstream operator. Further, the upstream operator only removes the virtual workers of the corresponding worker if it is able to pair it with an addition on another idle worker. This simple scheme ensures that the number of virtual workers in the system are same throughout the execution and the load of each virtual worker is bounded, which enables CG to perform fair assignment. Note that mapping the virtual workers with similar keys to the same worker might reduce the memory footprint. However, this requires maintaining all the unique keys in each virtual worker and each workers. Therefore, we opt for FCFS mapping of virtual workers to workers.

5.3 Integration in a DSPE

While consistent grouping is easy to understand, its applicability is case of a real world stream processing engines is not clear. In particular, we need to answer two questions: 1) How to identify the imbalance (what are the metrics for imbalance) and 2) how to plan the migration (how to keep ownership of the work). To answer these questions, we package CG with few efficient strategies that enable its applicability in variety of DSPEs.

Worker Delegation. We answer the first question by proposing an efficient scheme called *worker delegation*. This scheme pushes the load balancing problem to the downstream operators and allows them to decide their workload based on their capacity. Each PEI requires monitoring its workload and needs to take the decision based on their current workload and the available capacity. The decision can either be to increase the workload or to decrease the workload. The intuition behind this approach is that it is often the case that the cluster consists of a large number of workers and collecting the statistics periodically from the workers creates an additional overhead for streaming application.

The worker delegation scheme allows the downstream PEIS to interact with upstream PEIS by sending binary signals: (1) increase the workload and (2) decrease the workload. Each worker monitors its workload and tries to bound the workload under some threshold, i.e., if the workload exceeds the threshold, the worker sends a decrease signal to upstream operators and if the workload is below the threshold the worker sends the increase signal to the upstream operators. This simple modification comes along with the benefit that it gives the flexibility to the workers to easily adapt to the complex application-specific requirements, i.e., processing, storage, service time and queue length.

Piggybacking. Each downstream PEI requires updating all the upstream PEIS in case of experiencing undesirable workload. Such deployments might negatively impact the performance of a streaming application, as it will require one-to-many broadcast messages across the network. To overcome this challenge, we propose to relax the consistency constraint in the DAG and allow operators to be eventually consistent. We propose to *encode* the binary signals from the downstream PEIS along with the acknowledgement messages. During the execution, the upstream operators only receive the signal from the downstream operator as a response to its messages. This means that PEI might continue receiving the messages with the same key even after triggering the decision.

Periodic Aggregation. When the upstream PEI receives a request to increase the workload, it moves one of the virtual worker from the overloaded worker to a idle worker. During the change of routing, we need to ensure that the messages that are pending in the queue of the workers must be processed in a consistent manner.

CG ensures that each message is processed in a consistent manner by discarding the message migration phase. Concretely, each worker processes the messages that are assigned to it and any change in the routing only affect the messages that arrive in later time.

As a message with the same key might be forwarded to different PEIS, CG performs *periodic aggregation* of partial results from the downstream operators to ensure that the state per key is consistent. Periodic aggregation leverages the same DAG (as shown in Figure 2) and imposes a very low-overhead in the stream application. Particularly, CG follows the same programming primitive as PKG for stream partitioning; supporting both stateless and stateful map-reduce like applications.

6 ANALYSIS

We proceed to analyze the conditions under which CG achieves good load balance. Recall from Section 3 that we have a set $\mathcal{W}$ of n workers at our disposal. Each machine $w \in \mathcal{W}$ has a limited capacity, which is represented by $c_w \in C$. Capacities are normalized so that the average capacity is $\frac{1}{n}$; that is $\sum_{w \in \mathcal{W}} c_w = 1$. We assume them ordered by decreasing capacities, i.e., $c_1 \geq c_2 \geq c_3 \ldots \geq c_n$. For simplicity, we assume that there is a single important resource on which workers are constrained, such as storage, and processing. Moreover, each worker ($w \in \mathcal{W}$) has an unbounded input queue (Q_w).

The input to the engine is a sequence of messages $z = \langle i, j, v, t \rangle$ where i is the identifier, $j \in \mathcal{K}, |\mathcal{K}| = m$ is the message key, v is the value, and t is the timestamp at which the message is received. Upon receiving a message with value $j \in \mathcal{K}$, we need to decide its placement among the workers. We assume one message arrives per unit of time. The message arrive in ascending order by timestamp.

Key distribution. We assume the existence of an underlying discrete distribution $\mathcal{D}$ supported on $\mathcal{K}$ from which keys are drawn, i.e., $k_1, \ldots, k_m$ is a sequence of m independent samples from $\mathcal{D}$ ($m \gg n$). Without loss of generality, we identify the set $\mathcal{K}$ of keys with $\mathbb{N}^+$ or, if $\mathcal{K}$ is finite of cardinality $m = |\mathcal{K}|$, with $[m] = \{1, \ldots, m\}$. We represent the average arrival rate of messages as p_j. We assume them ordered by decreasing average arrival rate: if p_j is the probability of drawing key j from $\mathcal{D}$, then $p_1 \geq p_2 \ldots p_m$ and $\sum_{j \in \mathcal{K}} p_j = 1$. We model the load distribution as a zipf distribution with values of z between 0 and 2.0. The pdf of the zipf distribution with skew z, rank of each key r and total number of elements m is:

$$f(r, m, z) = \frac{1/r^z}{\sum_{x=1}^{m} (1/x^z)}.$$

Supermarket Model. We employ queuing theory to study the problem and model the problem leveraging the supermarket model [29]. In this model, a sequence of messages arrives at a worker $w \in \mathcal{W}$. We model the arrivals of messages using a zipf distribution. Further, the service time required for each messages is fixed and deterministic. Each tuple is assigned to one of the n workers for processing. If the new tuple finds the worker occupied, it waits in the queue until its turn to be served comes. After the tuple is processed, it leaves the system.

Our goal is to design an algorithm to solve the Problem 3. Before starting the discussion on CG, assume that t_0 represents the time slot which corresponds to the minimum time period that each worker waits after sending a signal to the upstream PEIS. Also, as we are not considering elasticity, we assume that the system is well provisioned, i.e., $\frac{\sum_{j \in \mathcal{K}} p_j}{\sum_{w \in \mathcal{W}} c_w} < 1$

6.1 Imbalance with Consistent Grouping

For simplification, we divide the analysis of CG into two parts: dividing the workload into small equal-sized virtual workers and assigning the virtual workers to workers based on their capacities. Assume that $\alpha > 1$ represents the number of virtual workers assigned to each worker at initial time. Then, for n heterogeneous workers, we have $\alpha \times n$ homogeneous virtual workers. Each virtual worker has the same capacity (hence, homogeneous) and the capacity is guaranteed to be at most the capacity of the worker with the lowest capacity. The sources (which distribute keys to virtual workers) do not know the capacity of each worker. But, since all virtual workers are homogeneous, the sources can balance the loads of each worker by assigning equal number of messages to each virtual worker, and by keeping the number of virtual workers assigned to each worker is proportional to its capacity.

6.1.1 Chromatic Balls and Bins. We model the first problem in the framework of balls and bins processes, where keys correspond to colors, messages to colored balls, and virtual workers to bins. Choose d independent hash functions $\mathcal{H}_1, \ldots, \mathcal{H}_d \colon \mathcal{K} \to [\alpha n]$ 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., $P_t(k_t) = \operatorname{argmin}_{i \in \{\mathcal{H}_1(k_t), \ldots, \mathcal{H}_d(k_t)\}} L_i(t)$.

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.

Note that in case of shuffle grouping, balls with same color might be assigned to different workers. Such an assignment requires extra memory to gather partial states and requires an additional aggregate phase in case of stateful operators, e.g., aggregate, max, min. We express this behavior in terms of memory and aggregation cost. Further, observe that in case of key grouping, each ball is assigned to a single worker. Therefore, key grouping does not require any additional memory and the aggregation phase.

Next, we analyze PKG [32]. When using the PKG, we have $d = 2$, which is same as having two hash functions $\mathcal{H}_1(j)$ and $\mathcal{H}_2(j)$. The algorithm maps each key to the sub-stream assigned to the least loaded worker between the two possible choices, that is: $P_t(j) = \operatorname{argmin}_i(L_i(t) : \mathcal{H}_1(j) = i \lor \mathcal{H}_2(j) = i)$.

LEMMA 6.1. *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 is* $\mathrm{O}\left(\frac{\ln \ln n}{\ln d}\right)$, *with high probability [32].*

Observe that the imbalance in case of PKG is only guaranteed for the case when $p_1 \leq \frac{1}{5n}$. However, in the case when $p_1 > \frac{1}{5n}$, the imbalance grows proportional with the frequency of the most frequent key and number of workers.

Next, we analyze POWER OF TWO CHOICES, which was introduced by Azar et al. [2]. When using the POTC, we have two random numbers $\mathcal{R}1(m)$ and $\mathcal{R}2(m)$. The algorithm maps each message m to the sub-stream assigned to the least loaded worker between the two possible choices, that is: $P_t(k) = \operatorname{argmin}_i(L_i(t) : \mathcal{R}1(m) = i \lor \mathcal{R}2(m) = i)$. The above random numbers can be generated by using hash functions with messages as arguments. In this case, note that the POTC is different from the PKG in the sense that two hashes are applied to the messages, rather than the keys. The procedure is identical to the standard GREEDY-d process of Azar et al. [2], therefore the following bounds hold.

LEMMA 6.2. *Suppose we use n bins and let* $m \geq n^2$. *Then, the imbalance after m steps of the Greedy-d process is* $\mathrm{O}\left(\frac{\ln \ln n}{\ln d}\right)$, *with high probability [2].*

Note that these bounds can be generalized to the infinite process in which n balls leave the system in each time unit (one from each worker) and the number of balls entering the system are less than n. In such cases, the relative load remains the same, therefore the bound holds. Both CH [28] and PORC generate imbalance that is bounded by the factor ϵ , i.e., $I(m) \leq \epsilon \cdot (\frac{m}{n})$.

6.1.2 Fair Bin Assignment. Given that m messages are assigned to set of n workers using PORC, our goal is to show that consistent grouping is able to perform fair assignment to messages to the workers over time. We achieve our goal by showing that consistent grouping reduces the imbalance $I(t)$ (if it exists) over time. To make the discussion more concrete, we define the notion of busy worker using a threshold $\theta_b > 1$. In particular, we say that a worker w is busy if the load $L_w \geq \theta_b \cdot c_w$. Similarly, we define the notion of idle worker using the threshold $\theta_i < 1$. We say that a worker w is idle if its load $L_w \leq \theta_i \cdot c_w$.

Assume that α represents the average number of virtual workers per worker, i.e., the total number of virtual workers equal $\alpha \times n$. Also, assume that α^*_w represents the optimal number of virtual workers for w-th worker, namely, $\alpha^*_w = c_w n\alpha$. Clearly, $\frac{\theta_i \cdot c_w}{\alpha^*_w} \leq \frac{1}{n.\alpha} \leq \frac{\theta_b \cdot c_w}{\alpha^*_w}$.

Thanks to the load balancing mechanisms, such as, PKG or POTC, each virtual bin is guaranteed to have load at most $1/(\alpha n) + \gamma$ with high probability, where γ denotes the imbalance factor of the load balancing mechanism used. For PKG and POTC, the value of γ is at most $(\ln \ln \alpha n/(m \ln d))$ as implied by by Lemma 6.1 and 6.2 (notice that the denominator m is due to the normalization of the capacity in this paper). Therefore, the expected load of a worker w having α_w virtual workers is bounded above by

$$\mathbf{E}[L_w] \leq \alpha_w \cdot (\frac{1}{n\alpha} + \gamma)$$

Now, consider that the worker w is overloaded, i.e., $\mathbf{E}[L_w] \geq \theta_b \cdot c_w$. This implies:

$$\alpha_w \cdot (\frac{1}{n\alpha} + \gamma) \geq \theta_b \cdot c_w$$

We can rearrange the above equation to have:

$$\gamma \geq \frac{\theta_b \cdot c_w}{\alpha_w} - \frac{1}{n\alpha},$$

which implies that when the worker is overloaded, it must have an imbalance that is lower bounded by the above equation. However, such an imbalance is guaranteed to be small ϵ by the load balancing mechanism used, i.e., $\epsilon \leq (\ln \ln \alpha n/(m \ln d)) \ll 1/(\alpha n)$ for POTC and PKG when $m \geq n^2$.

Therefore, we know that for an overloaded worker, it must hold that:

$$\epsilon \geq \gamma \geq \frac{\theta_b \cdot c_w}{\alpha_w} - \frac{1}{n\alpha}$$

Now, by solving for α_w, we get:

$$\alpha_w \geq \frac{\theta_b \cdot c_w}{\frac{1}{n\alpha} + \epsilon} \geq \theta_b \cdot c_w n\alpha(1 - \epsilon n\alpha),$$

where we use the Bernoulli's inequality $(1 + \epsilon n\alpha)^{-1} \geq (1 - \epsilon n\alpha)$ to obtain the above second inequality.

Notice that the above inequality gives the lower bound on the number of virtual workers assigned to an overloaded worker. Since its optimal number of virtual workers is $\alpha^*_w = c_w n\alpha$, we can see that $\alpha_w/\alpha^*_w \geq \theta_b(1 - \epsilon n\alpha)$, which is close to 1 since $\epsilon \ll 1/(\alpha n)$. This gives an interesting property that once we know a worker is overloaded, we can be sure that its number of virtual workers is close to the optimal allocation. Thus, the upstream operators can probe the capacity of workers by assigning virtual workers (taken from overloaded workers) to workers that have not reported becoming overloaded, or if there is no such one, to those that reported becoming overloaded least recently. Also notice that by letting $\theta_b = (1 + \epsilon n\alpha)$, we can guarantee that the overloaded workers are having at least the optimal number of virtual workers they shoud have. However, when ϵ is large (due to bad load balancing mechanisms), or when αn is large (due to having many small virtual workers), $\theta_b > 1$ will become large. This will burden the overloaded workers because they can only broadcast the overloaded cases when the threshold $\theta_b \cdot c_w$ is surpassed. This illustrates the tradeoff of load balancing mechanisms, with small imbalance factor ϵ, and the right number of virtual workers (too many is not good) in our consistent grouping strategy.

6.2 Memory with Consistent grouping

KG generates the optimal memory footprint by forwarding each key to exactly one worker. Similarly, PKG produces nearly optimal memory overhead by allowing at most two workers per key. On the other end, POTC and SG might assign each key to all the workers in the worst case producing the memory footprint proportional to the number of workers. PORC allows a trade off between imbalance and memory using the parameter ϵ. To analyze the memory footprint of PORC, we answer a very simple question: *What is the probability that a key is replicated on all the workers?* For this to happen, the load of $n-1$ workers should exceed by $(1+\epsilon)$ of the average load. Only then a key is replicated on all the workers. However, for a sufficiently large value of ϵ, i.e., $\epsilon > \frac{1}{n-1}$ this can not happen. This provides the basic intuition on why the memory overhead of PORC is lower than SG and POTC. However, we plan to consider the detailed analysis in future work.

7 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: How to tune the parameters for CG?

Q2: How does CG perform compared to other schemes?

Q3: How does CG adapt to changes in input stream and resources?

Q4: What is the overall effect of CG on applications deployed on a real DSPE?

7.1 Experimental Setup

Datasets. Table 2 summarizes the datasets used. In particular, our goal is to be able to produce the skewness in the input stream. We use two main real data streams, one from *Wikipedia* and one from *Twitter*. These datasets were chosen for their large size, different degree of skewness, and different set of applications in Web and online social network domains. The Wikipedia dataset (WP)[6] 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. We split each tweet into words, which are used as the key for the message. Lastly, we generate synthetic datasets with keys following Zipf distributions with exponent in the range $z = \{0.1, \ldots, 2.0\}$ and for different number of unique keys $K = 100k$.

Simulation and Real Deployment. We process the datasets by simulating the DAG presented in Figure 3. The stream is composed of timestamped keys that are read by multiple independent sources

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

TABLE 2: **Summary of the datasets used in experiments. Note: Percentage of messages having the most frequent key (p_1), arrival rate ($\overline{\lambda}$) for data streams.**

Stream	Symbol	Messages	Keys	p_1 (%)	$\overline{\lambda}$
Wikipedia	WP	22M	2.9M	9.32	10817
Twitter	TW	1.2G	31M	2.67	864005
Zipf	ZF	10M	1k,...,1M	$\propto \frac{1}{\sum x^{-z}}$	-

TABLE 3: **Notation for the algorithms tested.**

Symbol	Algorithm
KG	Key Grouping
CH	Consistent Hashing
PKG	Partial Key Grouping
POTC	Power of Two Choices
PORC	Power of Random Choices
CH	Consistent Hashing with Bounded Load
SG	Shuffle Grouping
CG	Consistent Grouping

TABLE 4: **Metric used for evaluation of the algorithms.**

Metric	Description
Imbalance	Difference between the maximum and the average resource utilization.
Memory Cost	Replication cost of the keys
CPU Utilization	CPU consumption (%).
Queue Length	Number of messages that are pending in the queue
Resource Utilization	Ratio between *queue length* and capacity of worker.
Execute Latency	Difference between arrival and finish time.
Throughput	Number of messages processed per second.

($\mathcal{S}$) 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. Note that for simulation, we ignore the network latency.

Algorithms. Table 3 defines the notations used for different algorithm. We use a 64-bit Murmur hash function for implementation of KG to minimize the probability of collision. Unlike the algorithms in Table 3, other related load balancing algorithms [3, 7, 10, 40, 46] 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.

Metrics. Table 4 defines the metrics used for the evaluation of the performance of different algorithms.

Resource Utilization. For CG, each worker requires monitoring its resource utilization that enables the fair message assignment. In case of simulations, we use queue length and CPU utilization as a parameter to measure the resource utilization. For the real-world experiments, we suggest using the queue length as a parameter for monitoring the resource utilization. In particular, the resource

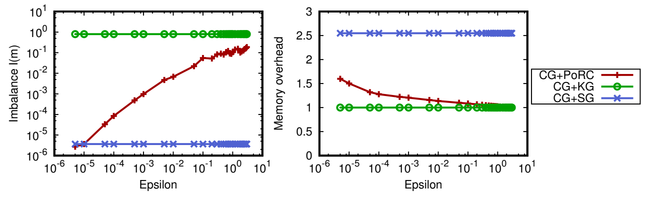

FIGURE 5: **Experiment reporting the imbalance and the memory overhead for different values of epsilon. The setup includes 10 workers with each having 10 virtual workers.**

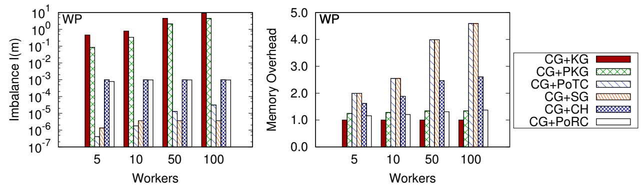

FIGURE 6: **Experiment reporting normalized imbalance and memory overhead on a homogeneous cluster with 5, 10, 50 and 100 workers using WP dataset. Each worker spawns 10 virtual workers and the ϵ equals 0.01 for CH and PORC.**

utilization is defined by:

$$\mathcal{U}_w(t) = \frac{L_w(t)(\text{\#tuples in the queue})}{c_w(\text{input queue capacity})}$$

The choice of the parameter was motivated by its availability in the standard Apache Storm distribution (ver 1.0.2).

7.2 Experimental Results

Q1: In the first experiment, we simulate the CG scheme by varying the value of ϵ by fixing the number of sources to 1 and the number of workers to 10. Each worker is homogeneous and the number of virtual workers per worker are set to 10. We select the WP dataset and simulate CG for different values of ϵ. Figure 5 reports the imbalance and the memory overhead for the experiment. The results verify our claim that epsilon provides a trade-off between imbalance and memory. In particular, CG generates low imbalance at lower values of epsilon and produces low memory footprint for higher values of epsilon. Also, the experiment shows that CG is able to interpolate well between the KG and SG schemes. Based on this experiment, we use the value of ϵ=0.01 henceforth as it provides a middle ground between memory and imbalance.

In the next experiment, we compare the imbalance of consistent grouping using different allocation strategies, i.e., KG, PKG, POTC, PORC, CH and SG. We simulate an experiment on a homogeneous cluster with 5, 10, 50 and 100 workers using the WP dataset. The number of virtual workers per worker are set to 10, i.e., equivalent to splitting the keys into 50, 100, 500 and 1000 bins. For CH and PORC, we set to value of ϵ equal to 0.01. Figure 6 shows the imbalance after the assignment of the streams. Results show that KG and PKG generate high imbalance, whereas POTC and SG generate nearly perfect load balance. Both CH and PORC bound the imbalance close to a constant factor from the value of ϵ. The imbalance in case of KG and PKG grows linearly with the increase in the number of workers. This behavior is due to the fact that both these schemes restrict a single key to a constant number of workers. CH and PORC

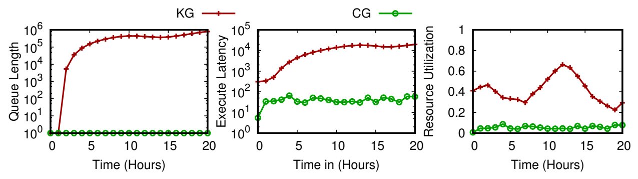

FIGURE 7: **Experiment reporting normalized imbalance and memory overhead on a homogeneous cluster with** 5, 10, 50 **and** 100 **workers using WP dataset. Each worker spawns** 10 **virtual workers and the** ϵ **equals** 0.01 **for CH and PORC.**

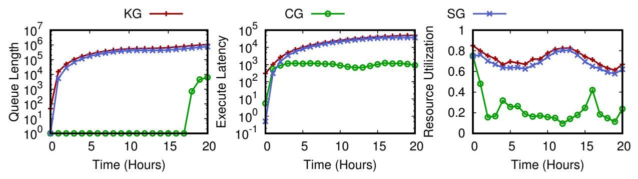

FIGURE 8: **Experiment reporting the affect on queue length, execution latency and resource utilization due to heterogeneity in the cluster for KG, CG and SG.**

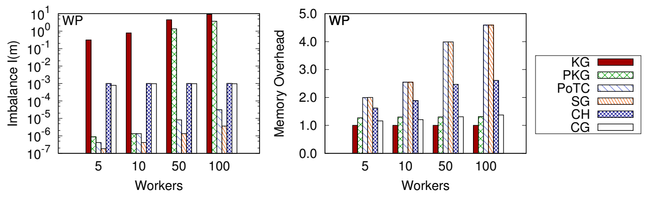

FIGURE 9: **Experiment reporting normalized imbalance and memory overhead on a homogeneous cluster with** 5, 10, 50 **and** 100 **workers using WP dataset. Each worker spawns** 10 **virtual workers and the** ϵ **equals** 0.01 **for CH and PORC.**

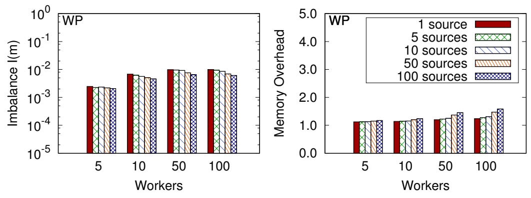

FIGURE 10: **Experiment reporting normalized imbalance and memory overhead on a homogeneous cluster with** 5, 10, 50 **and** 100 **workers with using** 1, 10, 50, **and** 100 **sources using WP dataset. Each worker spawns** 10 **virtual workers and the** ϵ **equals** 0.01 **for CG.**

bound the imbalance upto a constant factor for each bin. POTC and SG achieve linear perfect imbalance by exploiting all the possible workers. Interestingly, PORC achieves bounded imbalance while keeping the memory footprint as low as PKG, as shown in Figure 6. Henceforth, we leverage PORC for the next experiments and analyze consistent grouping.

Q2: To answer this question, we compare the imbalance and the memory overhead of CG with KG, PKG, POTC, CH and SG. We simulate the DAG for the *naïve Bayes classifier* (see section 3) using the WP dataset and report the value of imbalance measured at the end of the simulation. The cluster consists of different number of workers, i.e., 5, 10, 50 and 100 workers. Each experiment considers a cluster of homogeneous machines. For CG and CH, we set the value of epsilon equal to 0.01. Figure 9 reports the imbalance and the memory overhead for different schemes (note the log scale). Results show that KG performs the worst in terms of the imbalance while generating the optimal memory footprint. PKG on the other hand provides nearly perfect imbalance and optimal memory footprint for smaller deployments, i.e., 5 and 10 workers. However, the imbalance grows as the number of workers increase. POTC and SG provide very similar performance, i.e., provide nearly perfect imbalance and generate higher memory footprint. CH provides bounded imbalance and reduces the memory footprint compared to POTC and SG. CG provides the bounded imbalance and improves the memory footprint compared to CH. This behavior is due to the fact that CG leverages randomness to redistribute the messages once the principal worker reaches the capacity, whereas CH always choose the next worker in the ring.

In the next experiment, we report the queue length, average execute latency and the resource utilization of the workers by setting the capacity of the workers in a way that each worker operates at 80% of the capacity using shuffle grouping. We report each metric as a difference between the maximum and minimum value. Due to space restriction, we only report the results for 10 workers. For comparison, we also simulate and report KG and CG. Note that as PKG, POTC and SG provide nearly perfect imbalance, we do not report their results. We simulate the WP dataset, set the value of ϵ equal to 0.01 and set the number of virtual worker per worker equal to 10 for CG. Figure 7 shows the results of the experiment over time. Results show that the difference between the maximum and minimum queue length and execute latency increases over time using KG, whereas CG keeps both queue length and execute latency very low. Also, the difference between the maximum and minimum resource utilization is positive, whereas CG keeps this difference to close to zero.

In the next experiment, we mimic the heterogeneity in the cluster by assuming a cluster consisting of n machines in which y machines are z times more powerful than rest of the machines. In particular, we vary the value of z between 2 to 10 and vary the value of y between 1 to $n - 1$. For instance, when for $y = 1$ and $z = 2$, a machine in a 10 machine cluster has twice the capacity than all the other nine machines. We define the notion of idle and busy worker using the $\mathcal{U}_w(t) < 0.75 \cdot c_w$ and $\mathcal{U}_w(t) > 0.85 \cdot c_w$ thresholds respectively. We simulate the KG, SG, CG for comparison and use the value of epsilon equal to 0.01. In case of CG, each worker is initialized with 10 virtual workers. We observe similar behavior in all the configurations and report only a single iteration with $y = 3$ and $z = 5$. Figure 8 reports the queue length, execution latency and resource utilization for the three approaches. Results show that queue length and execution latency grow for KG and SG. Similarly, the difference between the maximum and minimum resource utilization is pretty high for these approaches. On the other hand, CG provides the lower queue length and average latency. Also, it keeps the difference between the maximum and minimum resource utilization close to zero.

Q3: Further, we evaluate the performance of CG by increasing the number of sources. In particular, we compare the performance of different deployments using 1, 10, 50, and 100 sources. For assignment of messages to sources, we use SG. Figure 10 reports the performance of CG in terms of imbalance and memory overhead. Results show that both imbalance and memory footprint almost

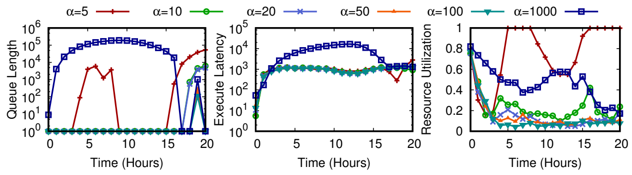

FIGURE 11: **Experiment showing the queue length, execution latency and resource utilization for different number of virtual workers.**

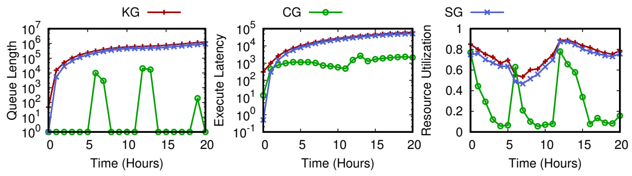

FIGURE 12: **Experiment showing the queue length, execution latency and resource utilization for when resources are changing over time. The resources change after processing 6M and 12M messages.**

remain the same on a log scale by both increasing the number of workers and number of sources. Therefore, we can conclude that CG is able to provide similar performance even under higher number of sources and workers.

In the next experiment, we study the behavior of CG on the number of virtual workers. We reuse the configuration for the experiment reported in Figure 8 and report the queue length, execute latency and resource utilization for CG, i.e., $y = 3$ and $z = 5$. We perform the experiment using number of virtual workers equals to 5, 10, 20, 50, 100 and 1000. Results show that setting the number of virtual workers to a value of 5 does not provide desired results. This is due to the fact that there are not enough virtual workers to move around the workers. Similarly, when the number of virtual workers are equal to 1000, the system takes longer time to converge, hence impacting the performance. Executions using 10 and 20 virtual workers provide similar similar performance. Lastly, the execution using 100 virtual workers generate the best results.

In the next experiment, we study the performance of CG by dynamically changing the resources over time. To initialize the resources, we reuse the configuration from the previous experiment and change the capacity of resources twice during the execution, i.e., after processing 6M and 12M messages. We execute the experiment for 100 virtual workers and change the resources in a way that the sum of resources remains the same. Also, we report the results of KG and SG for comparison. Figure 12 reports the queue length, execute latency and resource utilization of the experiment. Results show that CG adapts very efficiently to the change in resources.

Q4: Lastly, we study the effect of CG on streaming applications deployed on an Apache Storm cluster running in a private cloud. We implement and test our technique on the streaming naïve bayes classifier example, and perform experiments to compare CG, PKG, KG, and SG on the TW dataset. The parameters are selected in a way that the number of sources and workers match the number of executors in the Storm cluster. In this experiment, we use a topology configuration with 8 sources and 24 workers. We report overall throughput, end-to-end latency and memory footprint.

In the first experiment, we evaluate the performance of the algorithms in a homogeneous cluster. We emulate different levels of

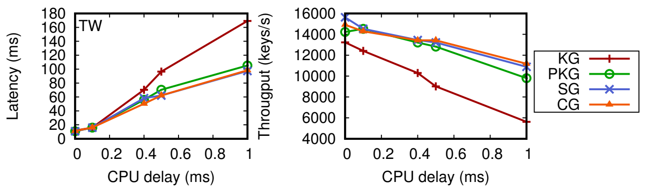

FIGURE 13: **Experiment reporting the throughput and latency for TW dataset on a homogenous storm cluster.**

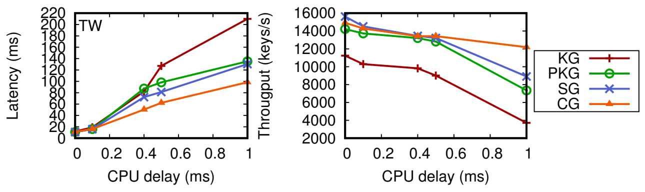

FIGURE 14: **Experiment reporting the throughput and latency for TW dataset on a heterogenous storm cluster.**

CPU consumption per key, by adding a fixed delay to the processing. We prefer this solution over implementing a specific application to control better the load on the workers. We choose a range that can 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, CG allows better resource utilization, therefore supporting the same workload on a smaller number of machines, but working on a higher overall load point each. 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.[7] Nevertheless, even more expensive tasks exist: parsing a sentence with NLP tools can take up to 500ms.[8]

Figure 13 reports the throughput and end-to-end latency for the TW dataset on the homogenous cluster. Also, during the experiment, KG was consuming 7% of memory in the cluster vs. 8.5% for PARTIAL KEY GROUPING and CG and 14% for SG. Result shows that KG provides low memory overhead but coupled with low throughput and high execution latency. Alternatively, PARTIAL KEY GROUPING, SG and CG provide superior performance in terms of throughput, latency and memory consumption.

Further, we evaluate the performance of CG in the presence of heterogeneity in the cluster. We use the **cpulimit** application to change the resource capacity over time and monitor the behavior of different approaches in terms of throughput and end-to-end latency. In particular, we limit the cpu resources of two of the executors to 30% of the available CPU resources to mimic the heterogeneity in the cluster. During the experiment, we give the system 10 minutes grace period to reach a stable state before collecting the statistics. Figure 14 reports the throughput and the end-to-end latency of the experiment. Results show that CG outperform other approaches both in terms of throughput and end-to-end latency. In particular, and compared to KG, it provides up to 2× better end-to-end latency and 3.44× better performance in terms of throughput.

Overall, we observe that CG is a very competitive solution with respect to KG, PKG and SG, performing much better with respect to

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

throughput and end-to-end latency and imposing a small memory footprint, while at the same time tackling the problem of heterogeneity of available resources at the workers in the cluster.

8 APPLICATIONS

Consistent grouping follows the same programming primitive as PKG for stream partitioning and not every algorithm can be expressed with it. In general, all algorithms that use shuffle grouping can use CG to reduce their memory footprint. In addition, many algorithms expressed via key grouping can be rewritten to use CG 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 CG. Henceforth, we assume that each message contains a data point for the application, e.g., a feature vector in a high-dimensional space.

8.1 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 [4] 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.

8.2 Heavy Hitters and Space Saving

The heavy hitters problem consists in finding the top-k most frequent items occurring in a stream. The SpaceSaving [27] algorithm solves this problem approximately in constant time and space. Recently, Berinde et al. [5] 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.

9 RELATED WORK

Load Balancing is one of the very well-studied problems in distributed systems. Also, it is very extensively studied in theoretical computer science [29, 30]. We refer to section 4 for load balancing in stream processing systems and provide a brief overview of the load balancing problem for several large-scale distributed systems.

Graph processing systems. Load balancing in graph processing systems is often found along with balancing graph partitioning, where the goal often is to minimize edge-cut between different partitions [15, 16, 26]. Further, several systems have been proposed specifically to solve the load balancing problem, e.g., Mizan [22], GPS [37], and xDGP [44]. Most of these systems perform dynamic load rebalancing at runtime via vertex migration. Yan et al. [47] and Chen et al. [9] propose a mirroring of high-degree vertices the to achieve better load balancing in pregel-like systems.

Map-Reduce like systems. Load balancing and scheduling often appears in a similar context in map-reduce like systems, where the goal is to schedule the jobs to set of machines in order to maximize the resource utilization [17, 45]. Sparrow [34] is a stateless distributed job scheduler that exploits a variant of the power of two choices [35]. Ahmad et al. [1] improves the load balance for map-reduce in heterogenous environment by monitoring and scheduling the jobs based on communication patterns. SkewTune [24] solves the problem of load balancing in MapReduce-like systems by identifying and redistributing the unprocessed data from the stragglers to other workers.

Other distributed systems. Dynamic Load balancing in database systems is often implemented using rebalancing, similar to all the other systems [36]. Also, online load migration is effective for elasticity in the database systems [42, 43]. Serafini et al. [39] propose an online partitioning approach that relies on identification of hot tuple from the loaded partition for migration along with it's co-accessed tuples. Lastly, dynamic load balancing is considered in the context of web servers [6], GPU [8], data ware housing [20], peer-to-peer systems [41] and many others.

10 CONCLUSION

We studied the load balancing problem for streaming engines running in a heterogeneous cluster and processing varying workload. In doing so, we proposed a novel partitioning strategy called Consistent Grouping. CG leveraged two very simple, but extremely powerful ideas: *power of random choices* and *fair virtual worker assignment.* In doing so, it efficiently achieved fair load balancing for streaming applications processing skewed workloads. We provided a theoretical analysis of the proposed algorithm and showed via extensive empirical evaluation that the cg outperforms the state-of-the-art approaches. In particular, CG achieved 3.44x superior performance in terms of latency compared to key grouping.

11 ACKNOWLEDGMENT

This paper is produced during the internship of the first author at IBM Research Tokyo. We would like to thank Scott Schneider, Tatsuhiro Chiba and Takeshi Yoshimura for their constructive feedback on the paper.

REFERENCES

[1] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2012. Tarazu: optimizing MapReduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 61–74.

[2] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. 1999. Balanced allocations. *SIAM J. Comput.* 29, 1 (1999), 180–200.

[3] Cagri Balkesen, Nesime Tatbul, and M Tamer Özsu. 2013. Adaptive input admission and management for parallel stream processing. In *DEBS*. ACM, 15–26.

[4] Yael Ben-Haim and Elad Tom-Tov. 2010. A Streaming Parallel Decision Tree Algorithm. *JMLR* 11 (2010), 849–872.

[5] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J. Strauss. 2010. Space-optimal heavy hitters with strong error bounds. *ACM Trans. Database Syst.* 35, 4 (2010), 1–28.

[6] Valeria Cardellini, Michele Colajanni, and S Yu Philip. 1999. Dynamic load balancing on web-server systems. *IEEE Internet computing* 3, 3 (1999), 28.

[7] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 international conference on Management of data*. ACM, 725–736.

[8] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. 2010. Dynamic load balancing on single-and multi-GPU systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.

[10] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. 2003. Scalable Distributed

[11] F. Farhat, et al. "Stochastic modeling and optimization of stragglers." IEEE Transactions on Cloud Computing (2016).

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

[13] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. 2004. Load balancing in dynamic structured P2P systems. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, Vol. 4. IEEE, 2253–2262.

[14] P Brighten Godfrey and Ion Stoica. 2005. Heterogeneity and load balance in distributed hash tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Vol. 1. IEEE, 596–606.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.

[16] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 599–613.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*. Berkeley, CA, USA.

[18] Jinho Hwang and Timothy Wood. 2013. Adaptive Performance-Aware Distributed Memory Caching.. In *ICAC*, Vol. 13. 33–43.

[19] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing under Overload. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 541–553.

[20] Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. 2005. ETL queues for active data warehousing. In *Proceedings of the 2nd international workshop on Information quality in information systems*. ACM, 28–39.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 654–663.

[22] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 169–182.

[23] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 555–569.

[24] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 25–36.

[25] Jimmy Lin and others. 2009. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, Vol. 1.

[26] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[27] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *ICDT*. 398–412.

[28] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2016. Consistent Hashing with Bounded Loads. *arXiv preprint arXiv:1608.01350* (2016).

[29] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (2001), 1094–1104.

[30] Michael Mitzenmacher, Ramesh Sitaraman, and others. 2001. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*. 255–312.

[31] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. 2015. Partial key grouping: Load-balanced partitioning of distributed streams. *arXiv preprint arXiv:1510.07623* (2015).

[32] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*. 137–148.

[33] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. 2016. When two choices are not enough: Balancing at scale in Distributed Stream Processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 589–600.

[34] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *SOSP*. 69–84.

[35] Gahyun Park. 2011. A Generalization of Multiple Choice Balls-into-bins. In *PODC*. 297–298.

[36] Erhard Rahm and Robert Marek. 1995. Dynamic multi-resource load balancing in parallel database systems. In *VLDB*, Vol. 95. Citeseer, 11–15.

[37] Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 22.

[38] Scott Schneider, Joel Wolf, Kirsten Hildrum, Rohit Khandekar, and Kun-Lung Wu. 2016. Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems. In *Proceedings of the 17th International Middleware Conference*. ACM, 21.

[39] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[40] Mehul A Shah, Joseph M Hellerstein, Sirish Chandrasekaran, and Michael J Franklin. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 25–36.

[41] Sonesh Surana, Brighten Godfrey, Karthik Lakshminarayanan, Richard Karp, and Ion Stoica. 2006. Load balancing in dynamic structured peer-to-peer systems. *Performance Evaluation* 63, 3 (2006), 217–240.

[42] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 513–527.

[43] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 245–256. DOI: https://doi.org/10.14778/2735508.2735514

[44] L Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2013. xDGP: A Dynamic Graph Processing System with Adaptive Partitioning. *arXiv* abs/1309.1049 (2013).

[45] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and others. 2013. Apache hadoop yarn: Yet another resource negotiator. In *SCC*. 5.

[46] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic load distribution in the borealis stream processor. In *ICDE*. 791–802.

[47] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1307–1317.

[48] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Osdi*, Vol. 8. 7.