Dynamic Key Grouping: A Load Balancing Algorithm for Distributed Stream Processing Engines

Orhun Dalabasmaz, Ahmet Burak Can

Abstract—Load balancing is important more than ever in distributed world, especially with Stream Processing. The load should be equally balanced among the servers in order to achieve lower latencies. We study the problem of load balancing in distributed stream processing engines in the presence of data skewness. We examined the current approaches and solutions then introduced Dynamic Key Grouping (DKG), an improved stream partitioning schema based on Partial Key Grouping (PKG) technique which also adapts the classical "power of two choices" approach. DKG is a smart stream partitioning algorithm that can detect the skewness and share the load among the servers regardless of the data content, thus reduces the latency and increases throughput. In contrast to PKG, DKG achieves this by distributing data to more than two servers depending on the amount of the load.
We test DKG alongside of KG, SG and PKG on several large datasets, both real-world and synthetic on a Storm topology. Compared to PKG, DKG improves the latency up to 7% and throughput up to %8 when the skewness is 30%. Moreover, DKG improves the latency up to 48% and throughput up to 93% when the skewness is higher than 80%. However, we conclude that PKG is acceptable and preferable with the skewness under 30%.

Index Terms—Distributed stream processing, load balancing, power of multiple choices, key grouping, stream partitioning.

1 Introduction

With the technological age we are in, technological devices are an indispensable part of our life. Every day, the devices used and the applications and users of these devices are also increasing. All these increases cause a huge grow of data produced and so the variety of data. The volume and the variety of the produced data is so increased that it is no longer possible for single machine to stand alone. On the other hand, requirements force us to process data in real-time. Therefore, cluster of machines is used for high efficiency, fault-tolerant and robust systems. By using cluster, we aim to process all data as soon as possible by distributing the data to all nodes in cluster. In order to achieve this, the data or the load should be distributed to the machines as equally as possible. Unbalanced distribution of the load means that a number of machines will work more intensively than others, and thus each machine will not be used efficiently.

Systems that process real-time data have two main problems to consider: i) the data processing speed should be higher than the speed of data arrival, and ii) the data should be distributed evenly among all the machines in the cluster. Besides the first fact, it is a necessity for all real-time data processing applications, and it is also related to the algorithms to be run directly on the data. For this reason, it is not possible to offer a general solution in this regard.

Distributing the data evenly among the machines, on the other hand, can be improved by accepted approaches, depending on the content of the data. In the ideal scenario, the data can be distributed equally to the machines so that we can obtain the lowest latency. At this point, the determinative one for us is whether there is a relationship between the data to be processed. If the data are independent of each other, the data can be distributed equally, as mentioned. However, if there is a relationship between the data, thus the data alone is insignificant or incomplete, then we need to design a structure that aggregates the stateful data and generates a valuable output.

Conventional approaches suggest two different solutions: i) each related data is distinguished by a key value should be send to the same machine, ii) all the data should be distributed equally to the machines, but afterwards an extra machine should perform the task of aggregating the data in order to have valuable result. In the first approach, depending on the content of the incoming data, if the data is so skew, some machines will be overloaded while others have very little load. This causes the data processing speed to be reduced considerably and the machines not to be used efficiently. In the second approach, although the load is always guaranteed to be distributed in a balanced manner, the fact that the data related to each other is distributed to all the machines causes the data to be insignificant and to be aggregated again on another machine. This brings additional cost to the system. Both approaches can produce different results depending on the type and content of the data. Both methods are not practicable, as they are real-time data processing, and we cannot know

- *Orhun Dalabasmaz is with Hacettepe University, Ankara, Turkey. E-mail: odalabasmaz@gmail.com.*
- *Ahmet Burak Can is with Hacettepe University, Ankara, Turkey. E-mail: abc@cs.hacettepe.edu.tr.*

the kind and the content of the upcoming data next.

In this paper, the problem of balanced distribution of data load between machines has been studied and load balancing approaches such as Key Grouping (KG), Shuffle Grouping (SG) and Partial Key Grouping (PKG) have been examined. The current approaches and the proposed new approach, Dynamic Key Grouping (DKG), have been thoroughly compared and tested with several datasets. The datasets are determined that each approach can and cannot be applied. Furthermore, we prove that the proposed approach, Dynamic Key Grouping, is particularly successful in skew datasets.

The rest of the paper is structured as follows: In Section 2, we briefly explain Stream Processing systems and then explain Storm in detail which is the Stream Processing Engine we used in our experiments. In Section 3, we examine the load balancing methods and existing methods such as Key Grouping (KG) and Shuffle Grouping (SG) in the Storm library. We also examine and elaborate the proposed new methods for the cases where the mentioned methods are inadequate in this context. In Section 4, we propose a new method against these non-performing methods in the existence of skew datasets and call it as Dynamic Key Grouping (DKG). Furthermore, we also give the details of the proposal and the architecture of the solution. In Section 5, we give the resources and datasets used, and provide information about setting up environments and related software components. In Section 6, we give details about the execution of the application. We provide simulation results demonstrating the benefits of the proposed method with existence of skew data and evaluate the results. We also mention the problems encountered in this context. In Section 7, we outline the results, discuss the method, and the future work.

2 STREAM PROCESSING METHODS

In this section, we will go through the stream processing methods and take a glance at the data processing structures which are visualized in *Figure-1*.

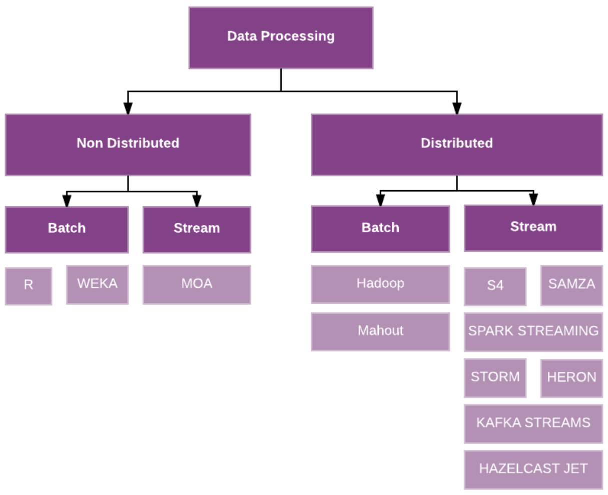


Fig 1. Data processing systems

2.1 Data Processing Types

Data processing methods are structurally divided into two groups as Distributed and Non-Distributed and in terms of processing of the data, they are divided into two groups as Batch Data Processing and Stream Data Processing.

2.1.1 Non-Distributed Data Processing

Data is processed in a single machine. Because of the system works on one physical machine, architecture does not support horizontal growth, only vertical growth, and CPU and RAM resources are possible to increase. Since the operations are performed on a single machine, the application is dominant and there is no need to receive information from other machines.

2.1.2 Distributed Data Processing

Data is distributed to several machines to be processed. In distributed structures, there are computer clusters. That is, the system can operate simultaneously on more than one physical machine. The architecture is suitable for growing horizontally, and all applications that will work should plan operations by predicting that more than one machine will work. Since the data is distributed to more than one machine, the coordination between the machines must be managed and the information in the other machines must be taken in order to be able to produce meaningful results and be combined into a single point.

2.1.3 Batch Data Processing

Batch Data Processing refers to the processing of a large volume of data independently of one another. So, there is no relation between the arrival time and the processing time of the data. For a while the incoming data are collected in a place and processed collectively at certain time intervals. In this method, when the data is being processed, the entire data is under control and its size is well known.

2.1.4 Stream Data Processing

Unlike Batch Data Processing, Stream Data Processing involves continuity and motion. Instead of collecting and processing the data, the data is turned into streams and processed one by one. Stream Data Processing methods have the ability to perform event-driven continuous processing. However, there are no time constraints to process the data and produce results.

2.1.5 Real-time Data Processing

Streaming Data Processing is often confused with Real-Time Data Processing because they have very close meanings. For this reason, it is necessary to understand the difference. Real Time Data Processing applications have to work continuously, that is, continuous data input and output is done through the system. In addition, there is time constraint compared to Stream Data Processing. In other words, the data entering the system should be processed in the shortest time to produce output. We can exemplify this in our home TVs. Televisions must process the stream of video streaming from the satellite as soon as possible and display it on the screen, otherwise we may watch a live broadcast delayed for a few minutes.

Real-Time Data Processing methods are basically Stream Data Processing methods which have to work continuously with time constraints. So, the more efficient and fast the Stream Data Processing algorithms work, the more real-time results we get. Processing of some data may take some time. In those cases where data can be accepted according to the nature of the study and the work, we will have Near Real-Time results.

2.1.6 Hybrid Data Processing

Hybrid Data Processing, which is also called as Lambda architecture, provides the combined use of both Batch Data Processing and Stream Data Processing methods for large-scale data processing. In this way, Batch Data Processing provides efficiency in terms of delay time, productivity, error resilience, and instantaneous tracking of changes with Stream Data Processing.

2.2 Stream Processing Engines

Stream processing engines are used for processing data from an unbounded and lasting stream source. Stream Data Processing has also been adopted as a next generation programming approach, apart from being a Big Data Environment. Even when working on a simple array, the flow of data dynamics is beginning to be used. For such a widespread structure, many open-source coded data processing tools have been developed. In general, the problems solved by them are close to each other, but they can differ and be preferred in terms of their development and solution methods over time. Some of these are: Samza[1], S4[2], Spark[3], Storm[4], Heron[5], Kafka Streams[6], Hazelcast Jet[7] and Kinesis[8]. In this study, we used Storm to run our application on.

Storm[4] is a stream processing framework that focuses on extremely low latency and is perhaps the best option for workloads that require near real-time processing. It can handle very large quantities of data and deliver results with less latency than other solutions [1].

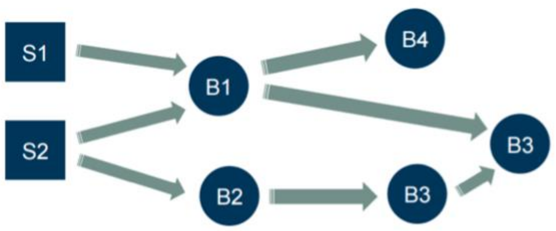


Fig 2. Storm topology

[1] Samza: http://samza.apache.org/

[2] S4: https://github.com/s4

[3] Spark: http://spark.apache.org/

[4] Storm: http://storm.apache.org/

[5] Heron: https://twitter.github.io/heron/

[6] Kafka Streams: https://kafka.apache.org/documentation/streams

[7] Hazelcast Jet: http://jet.hazelcast.org/

[8] Kinesis: https://aws.amazon.com/kinesis/

2.3 Storm Architecture

Storm architecture is very similar to Hadoop architecture. While MapReduce jobs are run in Hadoop, topologies are run in Storm. Storm is also compared to Spark. One of the biggest fundamental differences between the two is that Storm works on individual events as Samza does, and Spark Streaming works on micro-batches [2].

2.3.1 Storm Basics

Storm stream processing works by orchestrating DAGs (Directed Acyclic Graphs) which consists of spouts and bolts. This framework is called as *topology* and totally based on tuples and streams. Topologies are a process structure that includes the steps of retrieving the incoming dataset from the source, performing various operations, and generating output. A *Tuple* is the minimum data package that can be transferred between the spouts and bolts, and it also describes the data structure. *Stream*, on the other hand, refers to an unlimited number of Tuple series. The source of the Stream in topology is a Spout. *Spout* is the entrance point of the data and data may be gathered from a queue, API or any other file systems. *Bolts*, on the other hand, represents a processing step that consumes streams, applies an operation on them, and outputs the result as a stream. Bolts are connected to each of the Spouts and other Bolts as show in Fig2. to compose a topology.

2.3.2 Storm Cluster Architecture

Storm cluster architecture, as seem in Fig.3, has two types of nodes, Nimbus (master node), Supervisor (worker node) and a coordinator, Zookeeper[9]. Nimbus is a daemon similar to Job Tracker in Hadoop and responsible for running the topology. Nimbus analyses the topology, distributes the code to be run in workers, gathers the tasks to be executed and then assigns tasks to the available Supervisors (workers). A supervisor, on the other hand, may have one or more worker process and delegates the tasks to these processes.

Nimbus and Supervisor run as fail-fast and stateless. Being stateless lets the system more reliable and fault tolerant. Although the machines do not store states, Storm is not entirely stateless though. The states of the coordination and the communication between Nimbus and Supervisor should be handled separately by another system, Zookeeper. Zookeeper stores the states and helps a failed nimbus or supervisor to be restarted and made to work from where it left.

[9] Zookeeper: https://zookeeper.apache.org/

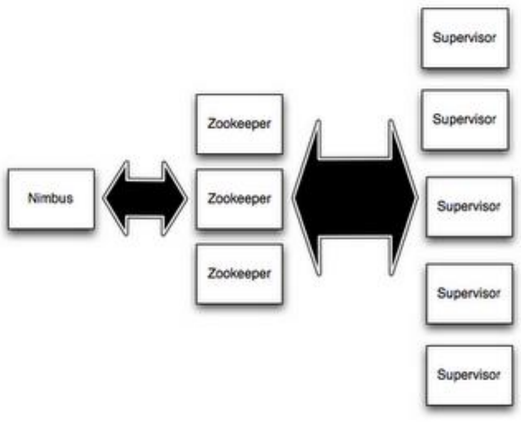


Fig 3. Storm cluster structure

2.3.3 Worker Process, Executor and Task in Topologies

In a Storm topology, there may be more than one worker processes running. Each *worker process* executes a subset of a topology and runs in its own JVM. An *executor* is a thread that is spawned by a worker process and runs within the worker's JVM. An executor may run one or more tasks for the same component (spout or bolt). A *task* performs the actual data processing and is run within its parent executor's thread of execution. Each spout or bolt that you implement in your code executes as many tasks across the cluster [3].

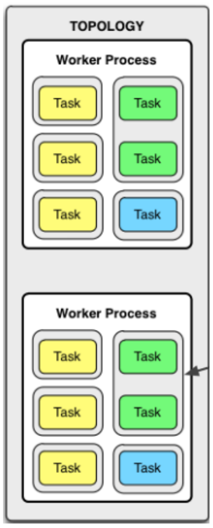


Fig 4. Storm Worker Topology

2.3.4 Grouping

A stream grouping defines how that stream should be partitioned among the bolt's tasks. There are eight built-in stream groupings in Storm, and moreover, we can implement a custom stream grouping too. Above all else, there are two base grouping methods: Shuffle Grouping and Key/Fields Grouping. In Shuffle Grouping (SG), tuples are randomly distributed across the bolt's tasks and we can basically be sure that each task gets an equal number of tuples. In Key/Fields Grouping (KG), on the other hand, tuples are distributed across the bolt's tasks by the fields specified in the grouping. These fields belong the data and we can choose which fields to be key for distribution. In case we choose user name field as key with Key Grouping, then we can guarantee that each tuple with the same user name, will always go to the same exact task, but tuples with different user names may go to different tasks. This will make it easier to do some calculations over the related data. As you may notice, contrary to Shuffle Grouping, we cannot guarantee that each task will get an equal number of tuples. Because, the distribution depends on the data and the field chosen as key and this may lead some tasks getting more tuples than the others. For such cases, the third option is to define our custom stream grouping. In custom stream grouping, we can decide which tuple will go to which task.

3 Load Balancing Methods

Distributing the data evenly may be crucial to achieve high performance which helps to increase throughput and reduce latency.

3.1 Data Distribution and Processed Data Relation

Distribution of the tuples depends directly on the content of the data. If the data is stateless, we can use Shuffle Grouping without considering the content of the data. This will help us to distribute the tuples evenly across the tasks and get high performance from the topology. With the stateful data, however, we need to gather related data into the same place in order to aggregate and obtain a single and final result. In this case, we should prefer Key Grouping because Key Grouping natively gathers related data into the same place. On the one hand, with Shuffle Grouping, we will have to gather all distributed data into the single place, and this will bring extra burden to system. On the other hand, since we cannot predict the stream data, Key Grouping may still bring extra burden to system. Because in case of skew data input, load will be gathered in a single task executor.

It's the last thing we want to have an impacted system by skew data. Because the same type of data that might come in at certain times can adversely affect the performance of the system and reduce productivity and increase latency. Such delay can be costly in systems that operate in real time and produce output. Also, because we cannot always predict future data, our system will not be stable and scalable. This means that the system will work in an unpredictable way.

Another problem is that we cannot control the entire system. If we could manage the whole distribution in the system from a single point, it would be possible for us to distribute the load in the best way as long as we dominate the entire system. However, in systems with distributed architecture, having a single machine that knows everything is not recommended, not very practical. For this reason, it is inevitable to concentrate on alternative solutions to solve the problem of load balancing in distributed systems.

3.2 Related Works

Since the need of many big data applications in science and industry have arisen, several solutions have been proposed to analyze large amounts of streamed or event data with low latency [15, 16].

Stream processing engines also gain popularity with the need of real-time results after processing large of data. [17] presents a reactive strategy to enforce latency guarantees in data flows running on scalable Stream Processing Engines (SPEs), while minimizing resource consumption. [18] shows that the custom partitioning methods, compared to default hash partitioning, save the memory space by reducing the size of aggregate states during execution of different graph processing algorithms on the resulting partitions. [19] proposes an effective partitioning strategy that uses a correlation-aware multi-route stream query optimizer (or CMR) for highly correlated data. It uses multi-route optimization which is based on the insight that tuples with similar statistical properties are likely to be best served by the same route [14]. Multi-route first divides data streams into several partitions and then creates a separate query plan for each combination of partitions. [20] proposes an adaptive input admission and management for parallel stream processing and it takes as input any available information about input stream behaviors and the requirements of the query processing layer, and adaptively decides how to adjust the entry points of streams to the system. To have a robust system at scale is also important for applications that processes large of data in a very brief amount of time like Twitter which uses Heron [21] to process stream at scale.

Data content is another consideration to process large amount of data to produce a single result. Although shuffle grouping in distributed stream processing engines is studied [22], grouping techniques for stateful data gain more importance than ever. While [23] describes an integrated approach for dynamic scale out and recovery of stateful operators, [24] introduces Samza which is a stateful scalable stream processing solution using at LinkedIn. This also brings auto-scaling techniques under consideration. Auto-scaling techniques allows for handling of unexpected load spikes without the need for constant overprovisioning [25, 26, 27].

Dynamic Load balancing algorithms also gain importance with using clusters to process large amount of data at scale. Distributed Stream Processing Systems (DSPS) have been widely adopted by major computing companies such as Facebook and Twitter for performing scalable event processing in streaming data. However, dynamically balancing the load of the DSPS' components can be particularly challenging due to the high volume of data, the components' state management needs, and the low latency processing requirements. Thus, [28] introduces a solution that dynamically balances the load of CEP (Complex Event Processing) engines in real-time and adapts to sudden changes in the volume of streaming data by exploiting two balancing policies. Where [29] introduces a local load balancing that does not require any global information, [30] introduces dynamic load balancing algorithm for the S4 parallel stream processing engine. [31] addresses several issues that are imperative to grid environments such as handling resource heterogeneity and sharing, communication latency, job migration from one site to other, and load balancing.

To succeed a well dynamic load balancing when processing large amount of data, the system should be able to route the traffic to the machines in the cluster evenly regarding to the content of the data. In contrast to shuffle grouping, this kind of algorithms usually called as key grouping or stream partitioning. They simply route the data by partitioning dynamically. Key partitioning techniques have been studied in MapReduce environment [32, 33, 34] as well as stream processing systems [35, 36, 37] for achieving efficient and near-optimal load balance.

Load balancing problem is also studied as content-based routing [38, 39, 40, 41, 42, 43]. Content-based routing can be performed on the actual content of a message by applying simple routing rules to the data itself by intelligent 'routing' servers. Intelligent content-based routing techniques have been proposed to achieve efficient, adaptive routing and to match up to the performance in terms of end-to-end latency and throughput.

Since an additional process is needed to produce a final result, skewness can be a headache when processing data in distributed environment. Therefore, load balancing at scale when data is skewed has also been studied [44, 45, 46, 47]. [48] presents a new key-based workload partitioning framework, with practical algorithms to support dynamic workload assignment for stateful operators and [49] proposes a novel partitioning strategy called Consistent Grouping (CG), which enables each processing element instance (PEI) to process the workload according to its capacity. The main idea behind CG is the notion of small, equal-sized virtual workers at the sources, which are assigned to workers based on their capacities.

On the other hand, similarly to our proposition, [50] proposes a novel load balancing technique that uses a heavy hitter algorithm to efficiently identify the hottest keys in the stream. These hot keys are assigned to $d \geq 2$ choices to ensure a balanced load, where d is tuned automatically to minimize the memory and computation cost of operator replication. The technique works online and does not require the use of routing tables. Morales proposes two novel techniques for this tough problem: D-Choices and W-Choices. These techniques employ a streaming algorithm to detect heavy hitter for tracking the hot keys in the stream, which constitute the head of the distribution of keys, and allows those hot keys to be processed on larger set of workers.

3.3 Load Balancing in Distributed Systems

Load balancing has become more important as the need for real-time data processing increases and the need to produce results as quickly as possible. In this context, distributed data processing systems such as S4, Storm and Samza have become even more popular. Because these systems are capable of real-time processing with very little latency on large volumes of data on clustered computers.

One solution is to migrate the processes to another machine when an overload is detected on a machine in the cluster [4, 5, 6, 7, 8 ,9]. So, the system will rebalance after a while. Even the method is so simple to understand or implement, it has some disadvantages in a distributed world. First of all, we must decide how often we need to scan the system for imbalances and how often we need to do the rebalancing. Moreover, the future messages must be directed to the new machines as well as after a migration is processed. To be able to do this, every machine in the cluster must have a number of routing tables, and the keys and target machines must be stored for each key distribution which is not feasible in distributed architectures that receive messages containing millions of keys from many sources. As the number of messages increases, the amount of memory the system needs to use is also increasing.

Flux is a fault-tolerant method that also transfers processes between the machines in order to balance the load. Flux monitors the load of each machine and ranks the machines by load. If a load imbalance is detected, it tries to rebalance the system by migrating the processes from the most loaded machine to the least loaded one, from the second loaded machine to the second least loaded one and so on [4].

Aurora* and Medusa are other methods of load balancing by transferring processed between machines [5]. Aurora can be defined as a centralized streaming data processing engine. It was developed in order to enable Aurora to work in distributed architecture and thus Aurora* and Medusa are proposed. While Aurora* supports a distribution inside the machines, Medusa builds a federation between the machines so they can rebalance the load by communicating and process transferring between them.

Borealis uses a similar approach but it also aims at reducing the correlation of load spikes among processes routed to the same server [6]. This approach is based on Aurora and builds a common infrastructure to process both sensor metrics and big server metrics. It also rebalances the system using a migration policy similar to Flux. It aims to balance the load on the global scale and this is achieved through full communication and cooperation with a brewery of machines assembled under a single point of administration.

Gedik, however, developed a new partitioning method for stateful data in distributed environments [7]. It monitors the key frequencies to control the migration cost and imbalance in the system. Even if the data is skewed, the method can keep the migration cost to minimum by balancing communication between the machines and memory usages.

Similarly, Balkesen et al. [8] proposed a method to balance the load between the machines by calculating the key frequencies of data. It aims to provide a more dynamic and efficient way of running data processing tools by controlling the system from out of the cluster.

In another study, Fernandez et al. [9] proposed to manage the states of the processes out of the system in which the stateful processes are performed. In this method, the states are stored as checkpoints and using these checkpoints, the system becomes fault-tolerant by distributing the remaining work to other machines in case of failures and also ensures that the system can be expanded horizontally.

The above-mentioned studies require either a structure to be managed by a central system or data transfer between machines. Neither is sufficient for us to be able to produce real-time results in distributed systems. Because, in such systems, the machines constantly send large amounts of data to each other, and this negatively affects the overall performance of the system and creates high network traffic. Moreover, having a centralized management leads to the whole system stopping and becoming unavailable in possible error situations. Thus, the system must be designed in a structure that is fault-tolerant and capable of operating with high performance. For this reason, we need for a non-centralized management and a solution that does not require transferring data between machines.

Azar introduced **PoTC** [10] which describes the problem in terms of balls-and-bins where the balls and the bins represent the tasks and the machines that tasks would run in respectively. In contrast to single-choice paradigm, which is the current solution used by all of DSPEs to partition a stream by using Key Grouping, PoTC uses two-choices paradigm which selects two bins uniformly at random and puts the ball into the least loaded one.

Mitzenmacher also studied PoTC problem and introduced the supermarket model et al. [11, 12, 13]. This model can be seen as a generalization of the static load balancing model studied by Azar. The model focused on defining an idealized process, corresponding to a system of infinite size, where the number of servers goes to infinity. It also demonstrated that in a simple dynamic load balancing model allowing to choose between two target machines yields an exponential improvement over distributing uniformly at random.

In our study, however, we have benefited most from the work of **The Power of Both Choices** [14], which was taken by Morales. In this work, Morales introduced Partial Key Grouping (PKG) method and we tried to develop the proposed method. Morales studied the load balancing problem on DSPEs based on the PoTC approach. In this work, it is stated that, in the case of selecting two bins, the gain is theoretically exponential compared to the only selection. Nevertheless, it is also stated that selection more than two bins will not provide an exponential gain. For this reason, two choices are made for each data within the method.

PKG differs from KG and SG by selecting two targets before determining the final target. KG and SG, on the other hand, always select only one target. Moreover, PKG method is based on two basic techniques: *key splitting* and *local load estimation*. PoTC method is used for key splitting. In this method, the system identifies two target machines for each key and directs the message to the least loaded one. However, it is hard to decide which target has less load, since we are living in a distributed world which gives us no chance to know the instant load of every target machine in a real-time manner. Thus, local load estimation technique is used to have a knowledge about the loads. In this technique, each source independently tracks the load of the downstream. Even if the system has not a global load oracle, surprisingly, it performs very well in practice and almost the same results as the traditional systems with global load oracle have been obtained.

3.4 Determining Target Machine

Determining the machine to send traffic or data is one of the basic tasks of the load balancing algorithms. These algorithms have basically two different methods: the SG method which determines the target machine regardless of the content of the data, and other KG-based methods which determine the target machine by considering the content of the data.

In the SG method, the Round-Robin method is used to distribute tasks to the target machines. Since the content of the data is not important for distribution, the data is distributed to the machines in turn. In the KG-based methods, on the other hand, the content of the data is important. Therefore, the content of the data is used to determine the target machine. In order to determine the target machine, the Hash value of the key of the data is calculated and target machine is determined by calculating the mode of this value with the number of machines present in the system as illustrated in Eq.1. The key values can be defined by using any field of the data. Defining key is important for stateful operations since relations of the data is based on the key values i.e., customer id. Moreover, with this calculation, it is guaranteed that all the data with the same key value will be sent to the exact same machine.

$$\boldsymbol{Tm} = \boldsymbol{H1}(\boldsymbol{data})\ \%\ \boldsymbol{Nm} \qquad (1)$$

In the above equation, *H1* is the first hash function, *Nm* is the number of available machines, and *Tm* is target machine index.

PKG is also a KG-based method and it also uses hashes to determine the target machine. Contrast to basic KG, PKG uses two different hash functions and these functions calculate different results for the same key. As a result, there will be two different candidates. To select the winner, minimum load of the two candidate machines will be calculated by using local load estimation as in illustrated in Eq.2.

$$\begin{aligned} \boldsymbol{M1} &= \boldsymbol{H1}(\boldsymbol{data})\ \%\ \boldsymbol{Nm} \\ \boldsymbol{M2} &= \boldsymbol{H2}(\boldsymbol{data})\ \%\ \boldsymbol{Nm} \\ \boldsymbol{Tm} &= \boldsymbol{min}(\boldsymbol{L}(\boldsymbol{M1}), \boldsymbol{L}(\boldsymbol{M2})) \end{aligned} \qquad (2)$$

In the above equation, *H1* is the first hash function, *H2* is the second hash function, *Nm* is the number of available machines, *L* is the current load of the given machine based on local load estimation, and *Tm* is target machine index. Tm is calculated by selecting the least loaded machine between the selected ones.

4 Dynamic Key Grouping (DKG)

The current key grouping methods can produce successful results for some special cases. While the SG method is the best distribution method for stateless datasets, the KG-based methods are more suitable for stateful applications. The KG method is highly efficient on homogenous datasets. The PKG method can produce good results even on skewed datasets. However, all these methods fail on highly skewed datasets. Assume that we have a skewed dataset which have a key value with an 80% proportion and there are ten machines in the cluster. The KG method will distribute 80% of the data to a single machine and 20% of the data to other nine machines. On the other hand, the PKG method will distribute 80% of the data to two machines evenly and 20% to other eight machines. This means that the system will work with nearly 10-20% efficiency under high load, and other 8-9 machines will be idle. What if our cluster has 100 machines? KG and PKG will use 1 of 100 and 2 of 100 machines, respectively. And this will end up with a 1-2% efficiency for the system. It is obvious that scaling out is not a solution when the data is highly skewed. Moreover, it will cause more cost without any benefit. For this reason, we tried to develop a method that can distribute the load in the most stable manner without making any horizontal growth and can perform successful load distribution regardless of the content of the dataset. Besides, we assumed that the number of machines to be operated in cluster is fixed. Unlike the PKG method, the number of targets is not limited to 2 and it can dynamically change regarding to the content of the data. In other words, instead of making a constant 2 selections for each data, we let the system to use more target machines for the more skewed data. Thus, the system can achieve a more balanced load distribution under high load. For non-skewed and homogenous datasets, we set 2 as the number of default targets as offered by PKG method and we named this method DKG (Dynamic Key Grouping).

DKG is a smart data distribution algorithm. It basically detects the skewness in data by measuring local load and builds a density map. This map stores the most recent hot keys, and this map is updated during the execution. The number of target machines is set as 2 by default and DKG can scale out and scale down the system by using 2 to n machines dynamically. In addition to this, in order to make the decision of scaling out, threshold values must be determined.

4.1 System Components

DKG is based on a few components. These are the components and the working principles:

4.1.1 Key Item

A Key Item component is created for every key processed in the system. It contains a list of keys, the last time they are processed, the total number of occurrences, the last time Scale Out is checked for this key, and the list of the machines that the data can be distributed. The DKG method carries out all decisions and practices through Key Item components.

4.1.2 Key Space

A component called Key Space was designed to manage Key Items and to detect data intensities or skewness. The design of this component is inspired from the JVM Heap Model as shown in Fig.5.

Objects created on the JVM are first placed in the Eden Space in Young Generation. When Eden Space is full, the Minor GC (Garbage Collecting) is activated and the surviving objects, which are still actively used, are transferred to the Survivor Space. GC refers to cleaning objects that are not used by the JVM so that they do not occupy memory, and this is essential for efficient memory use. Objects that are active in each GC cycle are preserved by moving between S0 and S1 Space, while those not being used are automatically cleared by the JVM. Objects that are still active after many GC cycles are transferred to the next stage, Old Generation. In this region, which is also referred to as Tenured, long-lived objects are held. This avoids the cost of recreating frequently used and continuously active objects. Objects in Old Generation are periodically scanned in a cycle called Major GC, and those that are not used anymore are cleaned from the memory. The Permanent Generation contains metadata required by the JVM to describe the classes and methods used in the application. Contrary to other regions, this region is not subject to any GC cycle and the information there is active throughout the application.

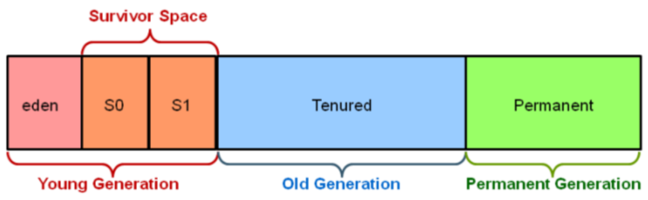


Fig 5. JVM Heap Model[10]

The DKG Key Space model is similar to the JVM Heap Model and consists of three parts: Baby Space, Teenage Space, and Old Space. These parts contain Key Items. The dimensions of these parts are dynamically determined. The number of expected different keys is estimated by the system administrator and entered into the system. 10% of this number is defined as Old Space, and 40% is defined as Teenage Space. The remaining 50% is defined as Baby Space. In the scope of this study, we set number of keys as 100 and set as Old Space 10, Teenage Space 40, and Baby Space 50 Key Items.

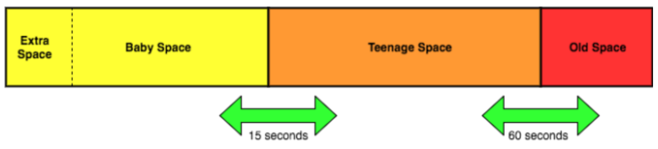


Fig 6. DKG Key Space Management

While Old Space and Teenage Space have physical boundaries, Baby Space does not have a physical boundary. Thus, Old Space and Teenage Space cannot hold more than 10 and 40 Key Items respectively. However, Baby Space can hold all Key Items without any restriction. Thus, all data entering the system are given sufficient time and space to pass to the next space.

For each key that enters the system, the Old Space, then the Teenage Space, and the last Baby Space are scanned first. If it is in the relevant space, the Key Item component is fetched. The last time and the total number of that key appeared is updated on the Key Item. If key is not found in none of these spaces, a new Key Item component is created and added to Baby Space.

4.1.3 Key Space Manager

The Key Space Manager runs in a separate Thread, which manages the transition of Key Items within the Key Space and the GC cycle. It checks the Key Space with a 15-second cycle and promotes the necessary Key Items from Baby Space to the Teenage Space. Similarly, it checks the Key Space with 60-second cycles and promotes the necessary Key Items from Teenage Space to Old Space.

Key Item change can be bi-directionally between the Spaces. When a new Key Item enters the system, Key Item densities in the space may change and the order of the Key Items becomes degraded. When the cycle time arrives, Key Items in all spaces are ordered and promotions are made. To promote Key Items, items from top to bottom of the Source Space are compared with items from bottom to up of the Destination Space and the items with higher occurrence are promoted to next Space (Baby Space to Teenage Space or Teenage Space to Old Space).

Moving Key Items from Baby Space to Teenage Space and Teenage Space to Old Space is executed based on the instructions defined in Algorithm 1, 2 and 3.

Algorithm 1: promoteToTeenageSpace

```
Result: Promotes Key Items from Baby Space to Teenage Space
Input: baby space BS, teenage space TS
Output: N/A
sort Key Items in BS
sort Key Items in TS
truncate BS to discard quite new items
call promoteToNextSpace(BS, TS)
wait for 15 seconds
call promoteToTeenageSpace(BS, TS)
```

Algorithm 2: promoteToOldSpace

```
Result: Promotes Key Items from Teenage Space to Old Space
Input: teenage space TS, old space OS
Output: N/A
sort Key Items in TS
sort Key Items in OS
call promoteToNextSpace(TS, OS)
wait for 60 seconds
call promoteToOldSpace(TS, OS)
```

Algorithm 3: promoteToNextSpace

```
Result: Promotes Key Items to Next Key Space
```

```
Input: source space SS, destination space DS
Output: N/A
for each item in DS do
    Fi ← first Key Item of SS
    Fio ← the number of occurrences of Fi
    Li ← the last Key Item of DS
    Lio ← the number of occurrences of Li
    if Lio >= Fio then
        break the loop (this means transition completed)
    else
        remove Fi and add to head of the DS
        remove Li and add to tail of the SS
    end
end
```

4.2 Detecting Skewness

The Keys in Old Space are considered as intensive or skew data. So, if a key comes up too often, it will be placed in the Old Space over time and the application will be able to distribute these keys to additional machines. In other words, the ability to distribute to more machines for a key is possible only if this key is in the Old Space.

4.3 Determining the Thresholds

The inclusion of a key in the Old Space is not enough to allow it to be distributed to additional machines. In addition to this, both the system needs to be working for a while (cold start) and the intensity value should reach to a certain value. Thresholds are needed to compare the values and decide.

In order to determine the threshold, we first need to determine the *ideal load* for the system. We can define the ideal load as the load evenly distributed through all machines. Since we compare the load distribution over the percentage, we can formulate the ideal load as:

$$\boldsymbol{L_i = 100/m} \qquad (3)$$

In the above equation, L_i is the ideal load and m is the number of machines. Once we calculate the ideal load we expect in the machines, we can formulate the threshold value that we determined intuitively as follows:

$$\boldsymbol{L_s = L_i + \sqrt{L_i}} \qquad (4)$$

L_s is the threshold load for scaling out. Since the threshold is based on ideal load, it depends on the number of the machines. Moreover, distribution to new machines are prevented because of the slight load increments. Also, we limit the number of machines that can be selected for a key. At this point, we are getting a more stable and more efficient system in the long run.

The table below shows the number of machines in the system, the ideal load, the threshold and the maximum number of machines that can be distributed to.

Table 1. Thresholds of ideal loads

# of Machines	Ideal Load (%)	Threshold Load (%)	# of Max Machines
5	20.00	24.47	5
10	10.00	13.16	8
20	5.00	7.24	14
50	2.00	3.41	30
100	1.00	2.00	51

4.4 Determining the Target Machine

Target machine index is determined by calculating the hash value of the key and then normalized to stay in the range. Since there are at least 2 target machines by default, index and index + 1 are the initial targets of the key. We simply normalize the index after each modification to assure that index is in the range.

$$\boldsymbol{index = hash(key)} \ \% \ \boldsymbol{N_m} \qquad (5)$$

N_m is the number of machines. Rather than keeping the machines to distribute, PKG only keeps the number of machines to distribute. Since default value is 2, system only keeps the keys distributing more than 2 machines (i.e., K1=4, K2=3). Details are given in Algorithm 4.

Algorithm 4: chooseBestTask

```
Result: Choses target machine to distribute
Input: key, OldSpace (OS), threshold to scale out (T)
Output: machine index of target machine
hashOfKey ← calculate the hash of the key
targetWorkerIndex ← calculate target machine by normalizing the
hashOfKey (Eq.5)
Nm ← number of machines that the key can be distributed (2 by
default)
LLmin ← minimum local load between the available machines
CM ← best candidate machine so far
NLL ← number of machines have less load than threshold
call findBestCandidate()
call shouldScaleOut()
call shouldScaleDown()
if shouldScaleOut then
    CMnew ← call normalize (targerWorkerIndex + Nm)
    LLnew ← current local load of CMnew
    if LLnew < LLmin then
        CM ← CMnew
        Nm ← Nm + 1
    end
else if shouldScaleDown then
    Nm ← Nm – 1
    call chooseBestTask(key)
end
return CM

Procedure normalize(index)
    return index % TotalNumberOfMachines
```

[10] JVM Heap Model: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

```
end

Procedure findBestCandidate()
  for each machine in available machines do
    cl ← current local load of machine
    if cl < T then
      NLL ← NLL + 1
    end
    if cl < LLmin then
      LLmin ← cl
      CM ← machine
    end
  end
end
```

As mentioned before, PKG uses two different hash functions to determine the target machines. The definition of the hash function is important. Since the hash of the data strictly depends on the content of the data, even two different hash functions may produce the same result. On the other hand, although the hash functions produce different results, the data might be sent to same target machine according to the Eq.1. In this case, the hash function does not help to distribute data, and load balancing cannot be achieved. Thus, it results in performance losses in the system. Contrary to what is mentioned in the PKG method, there is no guarantee that the load will be distributed to at least 2 machines. Because it depends entirely on the content of the data and the key.

In the DKG method, however, hash function is only used to determine the first machine index. There is no second hash function. After the first index is determined, the index values of the other machines are determined by incrementing the first index value by 1. In this case, it is always guaranteed that the load will always be distributed among at least 2 machines.

4.5 Scaling Out

To be able to decide on a scale out, the system needs to be working for a while which is also called as cold start. Cold start gives the system a chance to gather as much as data in order to distinguish which data is skewed or intensive. After this period, the system can check for a scale out. As mentioned before, for each key, a target machine index is calculated, and number of machines is stored which indicates the spreading width.

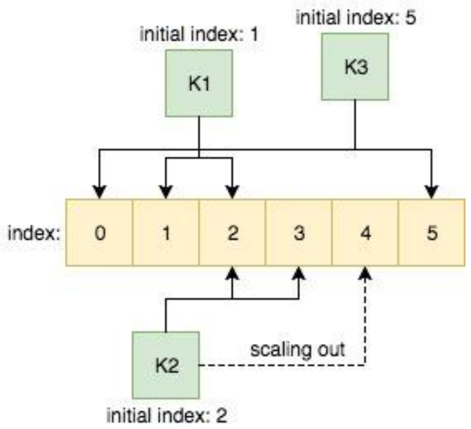


Fig 7. DKG indexes

As summarized in Algorithm 5, when a new Key Item comes up, the system checks current load of every machine that key can be distributed and the least loaded one is selected. After target machine is selected, the system compares the load of the machine with the threshold to decide if a scale out is needed. However, even exceeding the threshold is not enough to scale out, the key of the data also should be in Old Space. Since the DKG method focused on skewed and highly intensive data, Old Space helps the system to distinguish skew data and thus unnecessary scale out is prevented for non-skew data. After the above-mentioned checks, if the system decides on a scale out, a new machine is designated as a candidate by increasing the latest index of the available machines. When the new candidate target machine has been identified, the current load of the candidate is compared with the load of the least loaded machine in the present. If the load of the candidate is greater than the present, that means there is no need to scale out and the system will continue to work as present. If the load is less than the present, then number of the machines available will be increased by 1 and the system will begin to route traffic to this machine too.

Algorithm 5: shouldScaleOut

```
Result: Checks if scale out is needed
Input: key, min load L, scale out threshold T, old space OS
Output: boolean value of result
if not isWarmUp then
  return false
else if L < T then
  return false
else if key not in OS then
  return false
else
  return true
end

Procedure isWarmUp()
  RT ← elapsed time in seconds during system running
  if RT > 15 then
    return true
  else
    return false
  end
```

4.6 Scaling Down

To achieve dynamic scaling, DKG should support scaling down when the density is back to normal. Otherwise, after scaling out, the system will be routing traffic to machines even if it is not necessary. Contrary to scale out operation, the scaling down is easy. As summarized in Algorithm 6, when a new data arrives, the current loads of all available machines are considered. If the number of the available machines is greater than 2 and at least 2 of them have less load than the threshold, the system will decide on a scaling down. So, the number of machines available will be decreased by 1 and the system will immediately stop routing traffic to the latest machine in present.

Algorithm 6: shouldScaleDown

```
Result: Checks if scale down is needed
Input: WC, NLL
WC ← number of target machines for the key
NLL ← number of machines that has less load than the scale up threshold
Output: boolean value of result
if WC > 2 and NLL >= 2 then
    return true
else
    return false
end
```

4.7 Monitoring the Distribution

Monitoring is required both for instantaneous monitoring of system performance and for comparison of splitting methods. Besides the latency and the throughput, we also should measure extra metrics to be able to compare the methods. These metrics are gathered by Distribution Observer which is a completely external component of the system. It listens the workers and monitors the data coming to all workers and how they are distributed to the target machines. We can consider this as a global oracle who know everything about the topology and all metrics. This is designed to work with test purposes only to compare methods, since it is not applicable in a distributed world.

To monitor in a great manner, we introduced some new metrics besides the present ones and we hereby discussed these metrics and how to use them when comparing the methods:

- **Total Count:** Specifies the total number of keys in the data the application is processing. This information is measured directly on the data collected by the Distribution Observer.
- **Latency (ms):** Refers to the time required by the application to process all the data and it is calculated by measuring the elapsed time between the first output and the last output.
- **Throughput (rec/s):** Refers to the amount of output of the application, in other words, its productivity. It is calculated by dividing the total processed data by the total elapsed time in seconds as formulized in Eq.6.

$$Th = TotalCount \,/\, (Latency \,/\, 1000) \qquad (6)$$

- **Standard Deviation (StdDev):** Refers the quantity expressing by how much the distribution of data or the load differ from the mean of the load values for all machines. In other words, it shows how balanced the load is between the machines. The lower the standard deviation, the more balanced the load is distributed. We can calculate StdDev as formulized in Eq.7 where L_i is load of the machine, L_{avg} is average load of all machines, and N_m is the total number of machines.

$$L_{avg} = \frac{\sum_{i=0}^{N_m} L_i}{N_m}$$
$$L_{SD} = \frac{\sqrt{\sum_{i=0}^{N_m} (L_i - L_{avg})^2}}{N_m} \qquad (7)$$

- **Distribution Cost (DistCost):** Refers to how many different machines the data is distributed. The more data is distributed to the machines, the more machines will be waited to gather results from and the more aggregation will be done in order to produce the final result. For this reason, the higher values of DistCost indicates the system is running inefficiently. DistCost is calculated by dividing the total number of keys in the machines by the total number of different keys as formulized in Eq.8. where N_k is number of total keys, and N_{dk} is number of distinct keys. The lowest possible value is 1 while the highest value is the number of the worker.

$$L_{DC} = \frac{N_k}{N_{dk}} \qquad (8)$$

The SG method has the lowest StdDev since it is the most stable distributing method of the load, and it has the highest DistCost since it distributes each data to each machine. While the KG method has the lowest DistCost because of distributing the data by considering its content, it has higher StdDev according to the distribution of the data. The PKG method has a higher DistCost and lower StdDev because of distributing better than the KG method. However, the DKG method, which is introduced as an alternative to the PKG method, may show higher and lower values than the PKG method depending on the content of the data. The StdDev and DistCost that the methods have according to the data types and contents can be examined in detail in the experimental outputs in Section 5.

5 EXPERIMENTS

Experiments were performed to measure the performance of the proposed algorithm and compare the outputs with other methods' results. Within the scope of the experiment, the algorithms have been tested with several data sets and configurations. The comparisons were based on the outputs: StdDev, DistCost, Latency and Throughput.

5.1 Datasets

In the experiment, 5 real data sets and 12 synthetic data sets were used. The real data sets consist of twitter and wikipedia content. Twitter data contains ticker values[11] and tweet messages[12] collected during the 2016 US elections. Wikipedia data contains clickstream data[13] and pageview records[14,15]. The synthetic data consist of the names of the 204 countries on the globe. In order to observe how load distribution methods, behave under different amount of loads, several skewed data sets are produced at different skewness ratios. The details of the data sets are given in Table 2.

Table 2. Datasets used in experiment

Dataset	Data type	Total Keys (Million)	Skewness Ratio (%)
twitter-election	Real	5	68
twitter-ticker	Real	1,5	10
wikipedia-clickstream	Real	8.000	10
wikipedia-pageviews	Real	22	9
wikipedia-pageviews-by-lang	Real	588	27
country-skew-r0	Synthetic	10	0
country-skew-r10	Synthetic	10	10
country-skew-r20	Synthetic	10	20
country-skew-r30	Synthetic	10	30
country-skew-r40	Synthetic	10	40
country-skew-r50	Synthetic	10	50
country-skew-r60	Synthetic	10	60
country-skew-r70	Synthetic	10	70
country-skew-r80	Synthetic	10	80
country-skew-r90	Synthetic	10	90
country-skew-r100	Synthetic	10	100
country-half-skew-r80	Synthetic	10	40

5.2 Experiment Topology

Experiment topology is illustrated in Fig.8. In experiment, all methods (SG, KG, PKG, DKG) are tested with all data sets and with number of spouts of 5, 10, 15, 20 and number of workers of 10, 50 and 100. Spouts consume the data from Kafka[16] and direct to Splitters. Splitters then split the data and distribute to workers which execute the real processing tasks. After execution, results of workers are gathered in aggregators and then output is reduced to yield a single and final result. Final results are also directed to Kafka to be stored permanently. To visualize and monitor all processes instantly, Kafka Connect is used to migrate data into InfluxDB[17] from Kafka. Grafana[18] helps to visualize metrics stored in InfluxDB. Distribution Observer, however, is only attached to the system for detailed monitoring purposes and should be considered outside of the topology.

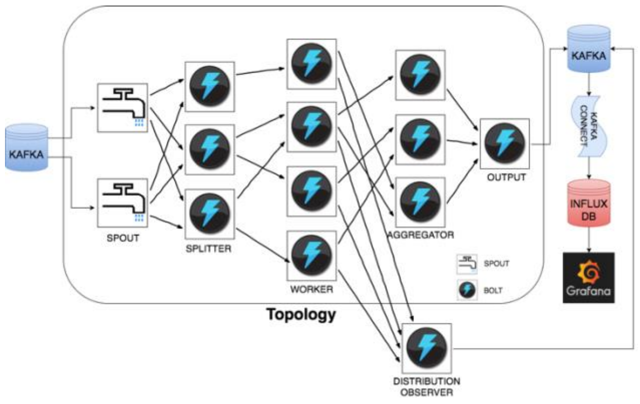


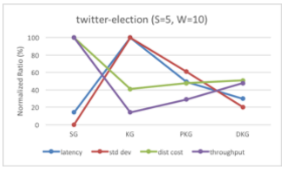
Fig 8. System topology used in the experiments

5.3 Experiments on Real Datasets

The first experiment was executed with 5 spouts and 10 workers on all the real datasets mentioned above. Since this paper is focused on high skew datasets and key based splitting methods, we put twitter-election under the spotlight which has 68% skewness. Performance metrics observed on this dataset are given in Table 3. The results for each dataset are also given in Fig.9. In the graphs, left-y axises represent the normalized ratios, which are calculated by dividing the value by the highest value observed in the experiment. Also note that we want to achieve low Latency, StdDev and DistCost but high Throughput.

Table 3. twitter-election performance metrics

METHOD	LATENCY (ms)	STD DEV	DIST COST	THROUGHPUT (rec/sec)
SG	559,329	0.0001	2.4491	9,511
KG	3,955,045	20.3202	1.0000	1,344
PKG	1,941,879	12.3513	1.1647	2,739
DKG	1,174,979	4.0972	1.2414	4,529



(a)

[11] Tweets Ticker Symbols Used in the Stock Market [14]

[12] Tweets During the 2016 Election, http://anuragprasad.com/TwitterElection.html

[13] Wikipedia Clickstream, https://figshare.com/articles/Wikipedia_Clickstream/1305770

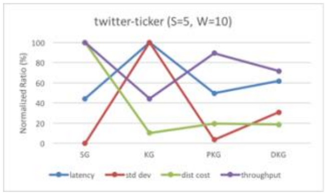
[14] Wikipedia Pageviews Data During a Day [14]

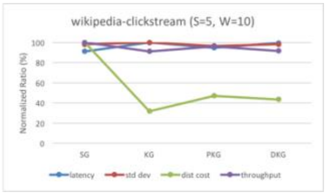
[15] Wikipedia Pageviews by Language, https://dumps.wikimedia.org/other/pageviews/

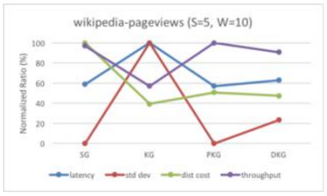
[16] Kafka: https://kafka.apache.org/

[17] InfluxDB: https://www.influxdata.com/

[18] Grafana: https://grafana.com/


(b)


(c)


(d)

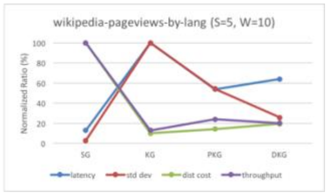

(e)

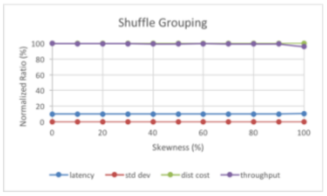
Fig 9. Performance of the tested method with real datasets

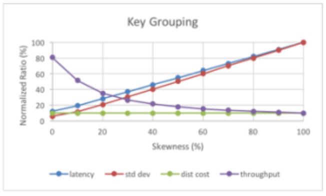
As seen in the Fig.9.a, the highest DistCost value was obtained in the SG method in spite of the highest throughput rate. On the other hand, the KG method has the optimal DistCost value but also has high Latency, high StdDev and low Throughput values. These metrics leads to poor performance in the KG method. On the other hand, lower DistCost and StdDev values were observed with PKG and DKG methods which yields more successful results. Despite the close DistCost values, the DKG method produced much more successful results than the PKG method with higher Throughput and lower StdDev and Latency values.

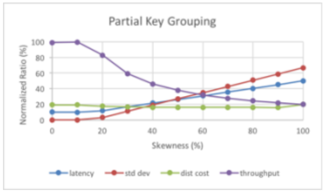
Since the skewness of other datasets are close, all other graphs above, (9.b, 9.d, 9.e), the PKG and DKG methods produced similar results. However, with wikipedia-clickstream dataset, as seen in the Fig.9.c, the KG method produced better results than other methods.

5.4 Experiment on Synthetic Datasets

The second experiment was also executed with 5 Spouts and 10 Workers. In contrast to the first experiment, synthetic datasets were used to reveal the performances of methods with highly skewed datasets.


(a)


(b)


(c)

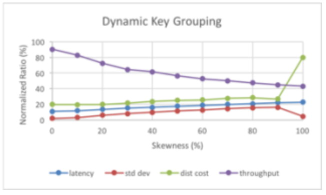
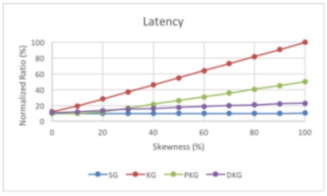
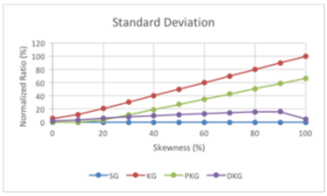
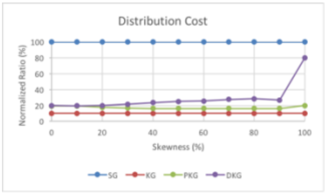

(d)

Fig 10. Performance of the tested methods according to changing skewness ratio of the synthetic datasets

Fig.10. shows the performance of each algorithm with respect to different ratios of skewness. The increase of skewness did not cause any change in the SG method (Fig.10.a) since keys are randomly distributed to workers. In the KG method, very successful results were achieved up to the 30% skewness, while performance over 30% showed a gradual decrease (Fig.10.b). In the PKG method (Fig.10.c), even though performance degradation was observed after 20-30% skewness ratio, much more successful results were obtained up to 60% skewness compared to the KG method. In the DKG method, the performance of the system is much more successful and predictable than the other methods, even with the highest skewness values (Fig.10.d). This also demonstrates that DKG is a predictable and scalable method which can run better under high loads (>60%). The DKG method yields the second highest Throughput even under quite high loads like 80-90% while maintaining small DistCost, StdDev and Latency values. The SKG method has the highest Throughput value in all test cases but also has the highest DistCost value.


(a)


(b)


(c)

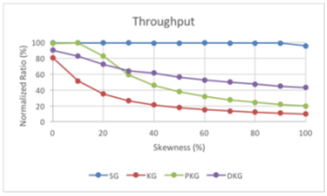

(d)

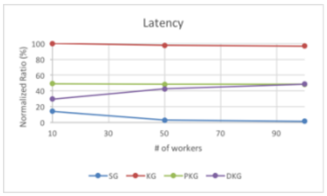
Fig 11. Performance of the tested methods on synthetic datasets in terms of performance metric

Fig.11. shows the performance metrics of different methods with changing skewness of synthetic data. As seen in Fig.11.a, DKG and SG had better Latency values since the keys are more evenly distributed on the worker nodes. KG and PKG had continuously increasing latencies with increasing skewness since load distribution is negatively affected. Similarly, in Fig.11.b, DKG and SG had clearly better StdDev values than KG and PKG with increasing skewness. DKG produced much better results after 30% skewness and yielded scalable results where PKG produced continuously increased StdDev values with the skewness.

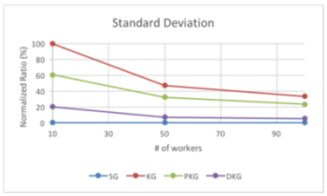
As seen in Fig.11.c, SG produced worst DistCost values. However, other methods produced very similar results even under higher skewness. DKG had a peek result under 100% skewness, which can be considered as a special case and out of this paper's scope. On the other hand, the Throughput in Fig.11.d had a decrease when the skewness was getting higher unless the SG method is used. Although the PKG method had higher Throughput values under the 30% skewness, the DKG method had better Throughput values than other methods when the skewness is higher than 30%. Although the SG method always have higher Throughput values than other methods, the DKG method provides better trade-off with lower DistCost and high Throughput values.

5.5 Experiments on Worker Set Size

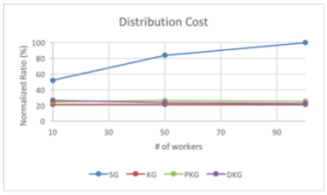
The third experiment was executed to observe the performance of the methods against increasing number of Workers in the topology. Both real and synthetic dataset were used within the experiment. Since the results were close to each other with the different skewness ratios, we only present the results of twitter-election dataset in Fig 12. The reason for this choice is that it is a real dataset and had a skewness of 68%.



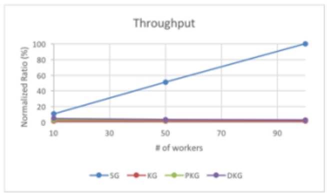
(a)



(b)



(c)



(d)

Fig 12. Performance against increasing workers

In Fig 12, all graphs basically demonstrated that the number of workers has not so much impact on the performance when the skewness is more than 30%. As seen in the Fig.12.a, the Latency values barely changed when number of workers are increasing. The Latency of DKG slightly increased, but it was still better than PKG. KG has highest, and SG has lowest Latency values as observed in the previous experiments.

As seen in Fig.12.b, StdDev values decreased for all methods. KG has highest and SG has lowest StdDev values as expected. DKG is much better than PKG with closest results to SG. In Fig.12.c, SG had the highest DistCost values as expected. Other methods, however, produced close and stable results with the increasing workers compared to SG. Similar to DistCost values, SG had the highest Throughput values as seen in Fig.12.d and all other methods produced close results compared to SG.

We conclude that increasing the number of workers is not a good fit for all cases, especially under highly skewed data load.

5.6 Experiments on Spout Set Size

The fourth experiment was executed to observe the performance of the methods against increasing number of Spouts in the topology.

We also used twitter-election dataset for this experiment and showed the performance results in Fig.13. The results also showed that increasing number of spouts had no effect to performance under highly skewed data load especially the skewness is more than 30%. The DKG method have better Latency, StdDev, and Throughput values than the KG and PKG methods. The distribution costs of KG, PKG, and DKG methods are similar.

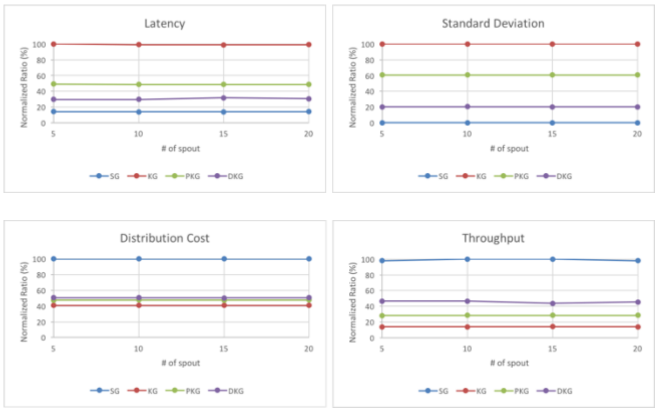


Fig 13. Performance against increasing spouts

6 Conclusions

We studied the load balancing problem in the presence of highly skewed and unbalanced stateful data in distributed stream processing engines (DSPEs). Although the load balancing is a well-known and studied problem in the literature, there is not much study with highly skewed and unbalanced datasets processing in distributed environments. PKG was introduced to solve this issue as a new stream partitioning strategy. According to the study, PKG achieved to reduce the imbalance by improving throughput and latency up to 45% in contrast to KG. However, even PKG did not consider the fact that the variety can increase instantaneously, and the system can face with a skewness of higher than 30%. In this study, we addressed to this problem and introduced DKG to achieve better results in the presence of highly skewed datasets. DKG improved the stream partitioning strategy by using a dynamic partitioning algorithm that allows the system to distribute the load more than two machines, and a skewness detection mechanism to help partitioning decision by improving local estimation technique. In contrast to PKG, we observed that DKG produced better results by improving the latency up to 7% and throughput up to 8% with a skewness ratio of 30%. Results have also shown that DKG produced much better results by improving the latency up to 48% and throughput up to 93% with a skewness ratio of higher than 80%. As a result of this study, we concluded that DKG should definitely be preferred if the skewness is higher than 30% whereas PKG can be preferred with lower skewness values.

Acknowledgment

I'd like to thank to Mr. Ozan Ali Kaya for giving me his professional business server to execute my experiments. I called it Monster and could not execute the experiments without it. And I'd like to thank to my family, especially to my wife, Tuğçe Dalabasmaz, for supporting me on this

long and exhausting journey.

REFERENCES

[1] Justin Ellingwood, Big Data Frameworks Compared, https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared (2017).

[2] Jim Scott, Stream Processing Everywhere, https://mapr.com/blog/stream-processing-everywhere-what-use/ (2017).

[3] Michael G. Noll, Understanding the Parallelism of a Storm Topology, http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/(June, 2017)

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

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

[13] 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.

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

[15] Bifet, Albert; Holmes, Geoff; Kirkby, Richard; Pfahringer, Bernhard (2010). "MOA: Massive online analysis". The Journal of Machine Learning Research 99: 1601–1604.

[16] Morales, Gianmarco De Francisci, and Albert Bifet. "SAMOA: scalable advanced massive online analysis." Journal of Machine Learning Research 16.1 (2015): 149-153.

[17] B. Lohrmann, P. Janacik and O. Kao, "Elastic Stream Processing with Latency Guarantees," 2015 IEEE 35th International Conference on Distributed Computing Systems, Columbus, OH, 2015, pp. 399-410.

[18] Z. Abbas, "Streaming Graph Partitioning : Degree Project in Distributed Computing at KTH Information and Communication Technology," Dissertation, 2016.

[19] Lei Cao , Elke A. Rundensteiner, High performance stream query processing with correlation-aware partitioning, Proceedings of the VLDB Endowment, v.7 n.4, p.265-276, December 2013

[20] Cagri Balkesen, Nesime Tatbul, M. Tamer Özsu, Adaptive input admission and management for parallel stream processing, Proceedings of the 7th ACM international conference on Distributed event-based systems, June 29-July 03, 2013, Arlington, Texas, USA

[21] Sanjeev Kulkarni , Nikunj Bhagat , Maosong Fu , Vikas Kedigehalli , Christopher Kellogg , Sailesh Mittal , Jignesh M. Patel , Karthik Ramasamy , Siddarth Taneja, Twitter Heron: Stream Processing at Scale, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, May 31-June 04, 2015, Melbourne, Victoria, Australia

[22] Nicoló Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, Bruno Sericola. Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems Research Paper. ACM/IFIP/USENIX Middleware 2016 , Dec 2016, Trento, Italy.

[23] Raul Castro Fernandez , Matteo Migliavacca , Evangelia Kalyvianaki , Peter Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, June 22-27, 2013, New York, New York, USA

[24] Shadi A. Noghabi , Kartik Paramasivam , Yi Pan , Navina Ramesh , Jon Bringhurst , Indranil Gupta , Roy H. Campbell, Samza: stateful scalable stream processing at LinkedIn, Proceedings of the VLDB Endowment, v.10 n.12, August 2017

[25] T. Heinze, V. Pappalardo, Z. Jerzak and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," 2014 IEEE 30th International Conference on Data Engineering Workshops, Chicago, IL, 2014, pp. 296-302.

[26] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, pp. 2351-2365, Dec. 2012.

[27] B. Gedik, S. Schneider, M. Hirzel and K. L. Wu, "Elastic Scaling for Data Stream Processing," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1447-1463, June 2014.

[28] Zacheilas N., Zygouras N., Panagiotou N., Kalogeraki V., Gunopulos D. (2016) Dynamic Load Balancing Techniques for Distributed Complex Event Processing Systems. In: Jelasity M., Kalyvianaki E. (eds) Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science, vol 9687. Springer, Cham

[29] 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 (Middleware '16). ACM, New York, NY, USA, Article 21, 14 pages.

[30] V. Gil-Costa, N. Hidalgo, E. Rosas and M. Marin, "A Dynamic Load Balance Algorithm for the S4 Parallel Stream Processing Engine," 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA, 2016, pp. 19-24.

[31] R. Shah, B. Veeravalli and M. Misra, "On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments," in IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 12, pp. 1675-1686, Dec. 2007.

[32] M. Hanif and C. Lee, "An efficient key partitioning scheme for heterogeneous MapReduce clusters," 2016 18th International Conference on Advanced Communication Technology (ICACT),

Pyeongchang, 2016, pp. 364-367.

[33] Gothai Ekambaram and Balasubramanie Palanisamy, "A Modified Key Partitioning for BigData Using MapReduce in Hadoop," Journal of Computer Science 2015, 11 (3): 490.497

[34] Ibrahim, S., Jin, H., Lu, L. et al. Peer-to-Peer Netw. Appl. (2013) 6: 409.

[35] Nicoló Rivetti , Leonardo Querzoni , Emmanuelle Anceaume , Yann Busnel , Bruno Sericola, Efficient key grouping for near-optimal load balancing in stream processing systems, Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, June 29-July 03, 2015, Oslo, Norway

[36] R. Wang and K. Chiu, "A stream partitioning approach to processing large scale distributed graph datasets," 2013 IEEE International Conference on Big Data, Silicon Valley, CA, 2013, pp. 537-542.

[37] N. Xu, B. Cui, L. Chen, Z. Huang and Y. Shao, "Heterogeneous Environment Aware Streaming Graph Partitioning," in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 6, pp. 1560-1572, June 1 2015.

[38] Pedro Bizarro , Shivnath Babu , David DeWitt , Jennifer Widom, Content-based routing: different plans for different data, Proceedings of the 31st international conference on Very large data bases, August 30-September 02, 2005, Trondheim, Norway

[39] Guoli Li , Vinod Muthusamy , Hans-Arno Jacobsen, Adaptive content-based routing in general overlay topologies, Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, December 01-05, 2008, Leuven, Belgium

[40] S. Bhowmik, M. A. Tariq, L. Hegazy and K. Rothermel, "Hybrid Content-Based Routing Using Network and Application Layer Filtering," 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, 2016, pp. 221-231.

[41] Muhammad Adnan Tariq , Boris Koldehofe , Kurt Rothermel, Efficient content-based routing with network topology inference, Proceedings of the 7th ACM international conference on Distributed event-based systems, June 29-July 03, 2013, Arlington, Texas, USA

[42] Sukanya Bhowmik , Muhammad Adnan Tariq , Jonas Grunert , Kurt Rothermel, Bandwidth-efficient content-based routing on software-defined networks, Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, June 20-24, 2016, Irvine, California

[43] Muhammad Shafique, Adaptive Content-based Routing using Subscription Subgrouping in Structured Overlays, 2016. (https://arxiv.org/abs/1604.06853)

[44] YongChul Kwon , Magdalena Balazinska , Bill Howe , Jerome Rolia, SkewTune: mitigating skew in mapreduce applications, Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, May 20-24, 2012, Scottsdale, Arizona, USA

[45] X. Zhang, H. Chen and F. Hu, "Back Propagation Grouping: Load Balancing at Global Scale When Sources Are Skewed," 2017 IEEE International Conference on Services Computing (SCC), Honolulu, HI, 2017, pp. 426-433.

[46] Y. Le, J. Liu, F. Ergün and D. Wang, "Online load balancing for MapReduce with skewed data input," IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, Toronto, ON, 2014, pp. 2004-2012.

[47] L. Cheng, S. Kotoulas, T. E. Ward and G. Theodoropoulos, "Efficiently Handling Skew in Outer Joins on Distributed Systems," 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, 2014, pp. 295-304.

[48] Junhua Fang, Rong Zhang, Tom Z.J. Fu, Zhenjie Zhang, Aoying Zhou, and Junhua Zhu. 2017. Parallel Stream Processing Against Workload Skewness and Variance. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17). ACM, New York, NY, USA, 15-26.

[49] Muhammad Anis Uddin Nasir, Hiroshi Horii, Marco Serafini, Nicolas Kourtellis, Rudy Raymond, Sarunas Girdzijauskas, Takayuki Osogami, Load Balancing for Skewed Streams on Heterogeneous Cluster, https://arxiv.org/abs/1705.09073

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

Orhun Dalabasmaz is AWS Certified Solution Architect and DevOps Engineer currently working at OpsGenie. He graduated from Department of Computer Engineering at Hacettepe University in 2010. He is interested in Big Data and Cloud Solutions like AWS, Storm, Kafka etc. He is currently researching and working on making scalable, highly available and reliable systems in the AWS environment.

Ahmet Burak Can is the Vice Head of the Department of Computer Engineering at Hacettepe University. His primary research interests concentrate on two main fields: network security and computer vision. While he is interested peer-to-peer networks and applications of cryptography in the network security field, he is also interested in medical image processing and usage of depth information in various vision problems in the computer vision field.