2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW) | 978-1-6654-4890-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDEW53142.2021.00028

Improving Stream Load Balance through Shedding

Nikos R. Katsipoulakis
Amazon Web Services
Palo Alto, CA, USA
Email: nkatsip@amazon.com

Alexandros Labrinidis, Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, PA, USA
Email: {labrinid, panos}@cs.pitt.edu

***Abstract*—The increasing demand for real-time processing has contributed to the rapid evolution of Stream Processing Engines (SPEs). Low operational cost and timely delivery of results are important objectives, whose achievement relies on efficient load distribution. However, given the volatile nature of data, and the limitations of existing stream *partitioning* techniques, those objectives can not be met.**

In this work we investigate the combination of load shedding with *partitioning* to improve load balancing in SPEs. We propose a novel query and load distribution model, which trades accuracy for balance load allocation in a controlled fashion. Also, we present the design of a lightweight operator that is compatible with current SPEs' architectures and is based on our model. Our operator, named ShedPart, identifies opportunities for approximating parts of the input in order to achieve balanced allocation. Our experimental evaluation indicates that ShedPart can lead to up to an order of magnitude better performance with real-world datasets.

I. Introduction

Data-intensive applications are prevalent in a plethora of social and economic sectors. Their goal is to capture trends by identifying patterns in data streams. Stream processing is considered the most appropriate processing model, since it follows a "push-based" approach that involves a static set of *continuous queries* (CQs), applied on dynamic data streams of unbounded size. Stream Processing Engines (SPEs) have been developed to cover the needs for this type of processing.

A major challenge for SPEs is their inability to provide the expected level of service during the whole lifetime of a CQ, because the exact resources needed are unknown at query submission time. Often, a CQ's resource needs are estimated based on historical information, without precise forecasting of future data characteristics (i.e., input rate, volume etc.) [1]. As a result, SPE administrators can either over-provision resources, which leads to high operational costs; or allocate insufficient resources to cover the processing needs of a CQ. Under such circumstances, either data get lost or results are delayed. Ultimately, static resource allocation is precarious and results in either loss of data or increased operational expenses.

Three major techniques have been developed to address the challenge of resource provisioning in SPEs: (i) *stream partitioning* [2], [3], (ii) *load shedding* [4], [5], [6], [7], [8], and (iii) *stream re-partitioning* [9], [10], [11]. The first balances load, the second selectively drops load, and the third reconfigures resources in an online fashion. Each one of those techniques comes with its pros and cons and is better suited for different applications and congestion circumstances.

978-1-7281-6251-5/20/$31.00 ©2020 IEEE

Often, *re-partitioning* is avoided due to the need to suspend execution while resources are adjusted [9], [11], [12]. As a result, *partitioning* and load shedding are preferred since they do not disrupt CQ execution. Albeit, *partitioning*'s effect is limited by available resources and sensitive to data characteristics. It has been shown that *partitioning* can encounter situations that do not allow for balanced load allocation and end up in skewed load distribution [2]. In addition, rapidly changing data necessitate the need to *split* keys during the lifetime of a window, which results in increased latency [3]. On the other hand, load shedding is geared towards applications that can sustain reduced accuracy in the results. By dropping data in a controlled fashion, an SPE's performance can cope with increased input data rate. In the past, load shedding has been applied in a decentralized fashion that can result in unnecessary drops in accuracy [8].

In order to overcome the shortcomings of previously proposed techniques, we propose a novel solution that combines *partitioning* with load shedding. In detail, our solution selectively trades accuracy for balance in load allocation. Our solution is based on a novel query model and a *partitioning* framework, which aims at either avoiding or delaying the need for re-partitioning. Furthermore, we present a lightweight operator based on our model, that works in tandem with a *partitioning* algorithm. This operator, named ShedPart (from **shed-partitioning**) combines stream *partitioning* and load shedding and trades results' accuracy for load balance in the finest granularity. ShedPart complements a *partitioning* algorithm and detects opportunities for approximating part of the result, given a user-defined accuracy specification. At window completion, ShedPart produces selective drops of tuples, which lead to a more balanced load allocation. Our experiments with real datasets indicate that ShedPart is able to improve performance and balance load by more than an order of magnitude.

Contributions In summary, our contributions are:

- A query and *partitioning* model that offers the ability to approximate results in order to achieve more balanced load allocations.
- The design of ShedPart as a lightweight operator, compatible with current SPEs. Our proposed design is optimized to add insignificant overhead during execution, and is

```
Rides(time, route, fare)
Q1 = Rides
   .time(x -> x.time)
   .windowSize(15, MINUTES)
   .windowSlide(5, MINUTES)
   .average(x -> x.fare)
   .group(x -> x.route)
   .parallelism(4);

   -- ShedPart parameter API --
   .error(10%)
   .confidence(95%)
```

Figure 1: Example CQ with accurary specification (for ShedPart.

enhanced with mechanisms that prevent deterioration of performance.

- An experimental analysis of ShedPart with real datasets.

The rest of the paper is organized as follows: Sec. II presents background information on stream processing and load shedding. Sec. III demonstrates our proposed model for ShedPart, and Sec. IV presents the details of ShedPart. Sec. V demonstrates the experimental results and Sec. VI concludes our work.

II. Prerequisites

We focus on the current generation of SPEs, which adopt a modular design targeted for modern cloud infrastructures. Examples of such SPEs include Flink, Kafka, Spark, and Storm. A user submits a CQ $\mathcal{Q}$ in either declarative or functional form. $\mathcal{Q}$ consists of data sources and transformations, which are applied to produce the expected result. Input streams are represented by S_i, where $1 \leq i \leq N$ (N is the number of input streams). Each S_i is a sequence of tuples $e_{\mathcal{X}_i}$, with a predefined schema $\mathcal{X}_i$. Fig. 1 (above the ShedPart comment) presents an example CQ in functional notation. In this CQ, there is a single stream source (S_1) named *Rides*, which consists of tuples with three fields: a $time$ used to order tuples in a monotonically increasing timeline, a $route$, and a $fare$.

Transformations applied by a CQ are represented by operators, which are categorized in *stateless* and *stateful*, based on their need to maintain state. We focus on *stateful* operators as they can be a challenge for current SPEs. Due to the unbounded size streams, a *stateful* operator's state is constrained to a subset of past tuples, which is called a *window*. In the CQ of Fig. 1, a *stateless* operator is *time*, which annotates each tuple with a timestamp using the $time$ field. The combination of the window and grouped average operators are *stateful* operators, since they buffer tuples until a window is complete for processing.

The notion of a window $\mathcal{W}_{r,s}$ complements *stateful* operators, which are applied to a group of tuples when $\mathcal{W}_{r,s}$'s conditions are met. A logical window definition can be represented as $\mathcal{W}_{r,s} : S_i \rightarrow \{S_i^1, \ldots, S_i^w\}$, where $w \rightarrow \infty$. Each S_i^w represents the tuples of S_i that belong to window w according to $\mathcal{W}$ The *range* r indicates a window's size, either in terms of tuples or duration. The *slide* s denotes the progression step. When $s = r$ a window is called *tumbling* and there is no overlap between consecutive windows. As a result, each tuple is assigned to a single window, In contrast, when $s < r$ windows are called *sliding*, and consecutive windows overlap (each tuple is assigned to $\lceil \frac{r}{s} \rceil$ windows). In the CQ of Fig. 1, the sliding window is defined by the functions *windowSize* and *windowSlide*. The former sets the window's range r to 15 minutes, and the latter sets the slide s to five minutes.

A. Execution Plan

A CQ's execution is a *Directed Acyclic Graph* (DAG). Each vertice denotes an operator of a CQ, and each edge a data transfer. The left-most node of the execution plan represents a data source, and the right-most node represents the last operator of a CQ. Intermediate nodes are operators imposed on flowing tuples. SPEs scale processing by utilizing multiple workers. Depending on the available resources, an SPE produces a physical execution plan. The parallelism of an operator relies on the number of workers ($\mathcal{V}$) assigned to it, where each worker can be a machine, a process, and/or a thread, depending on the parallelization granularity. With the use of multiple workers for each transformation step, data transfer operators are introduced in the DAG. The most popular include data shuffling, partitioning, broadcasting, and gathering. We focus on data *partitioning*, which is used to scale *stateful* operators [2], [3].

As presented in [2], a *stateful* operator is a three stage process, which starts with the partition of input data. *Partition* is a function that takes a S_i^w and produces another sequence of equal length, indicating the worker to which each $e_{\mathcal{X}_i}^w$ is going to be sent ($e_{\mathcal{X}_i}^w$ indicates a tuple $e_{\mathcal{X}_i}$ belonging to S_i^w): $P : S_i^w \rightarrow \{L_{S_i^w}^v, \text{for } 1 \leq v \leq \mathcal{V}\}$. Each $L_{S_i^w}^v$ denotes a subsequence of S_i^w and is the part sent to worker v for processing. $\mathcal{V}$ represents an operator's parallelism degree for processing the (partial) result for S_i^w.

To this end, an $e_{\mathcal{X}_i}$ can be represented as a triplet ($\tau_{\mathcal{X}_i}$, $k_{\mathcal{X}_i}$, $p_{\mathcal{X}_i}$), where $\tau_{\mathcal{X}_i}$ is the attribute used to assign each $e_{\mathcal{X}_i}$ to a logical window, $k_{\mathcal{X}_i} \subset \{\mathcal{X}_i - \tau_{\mathcal{X}_i}\}$ are the attributes utilized in the partition process, and $p_{\mathcal{X}_i} \subset \{\mathcal{X}_i - (\tau_{\mathcal{X}_i} + k_{\mathcal{X}_i})\}$ are the remaining attributes, which comprise $e_{\mathcal{X}_i}$'s payload.

In the example CQ of Fig. 1, the number of available workers $\mathcal{V}$ is four (function *parallelism*). Even though a *partitioning* algorithm is not directly defined, the SPEs imposes one with the use of the function *group*. As a result, tuples will be partitioned using the $route$ field of each tuple (i.e., $k_{Rides} = route$), and the payload is $p_{Rides} = fare$. Each worker buffers tuples on the window operator, and for each window calculates the group average.

B. Result Approximation in Stream Processing

Some applications do not require an *exact* result and can make use of an approximation. The accuracy of a window result is traded for performance, as long as there is a result's accuracy can be quantified. In the past, stream approximation techniques allowed a user to submit a $\mathcal{Q}$ with an accuracy

Table I: Model Symbol Overview

$\mathcal{V}$	number of workers
S_i	input streams $1 \leq i \leq N$
$\mathcal{X}_i = (\tau, k, p)$	schema of S_i
$e_{\mathcal{X}_i}$	tuple of S_i
$\mathcal{W}_{r,s} : S_i \rightarrow \{S_i^1, \ldots, S_i^w\}$	S_i's window definition with range r and slide s
$P : S_i^w \rightarrow \{L^v_{S_i^w}, \text{for } 1 \leq v \leq \mathcal{V}\}$	*partition* function for S_i^w
$L^{\mathcal{V}}_{S_i^w}$	window load of worker $\mathcal{V}$
R_w	exact result for window w
$\hat{R}_w$	approx. result for window w
$\epsilon : R_w, \hat{R}_w \rightarrow \mathbb{R}$	error between R_w and $\hat{R}_w$
α	confidence level
$I(P(S_i^w))$	Imbalance calculated when S_i^w is partitioned using P
N_g	number of tuples of a group g
s_g^2	values' variance of group g's tuples

specification [7], [8]. This specification consists of an accuracy measure ϵ and a confidence level α (i.e., the probability that the error will be less or equal to ϵ). For a *stateful* operator, we will represent the actual window result as R_w, and the approximate window result as $\hat{R}_w$.

The production of $\hat{R}_w$ entails that only a subset of S_i^w is processed. In order to measure the difference between R_w and $\hat{R}_w$, a metric $\epsilon : R_w, \hat{R}_w \rightarrow e$ is introduced by the user. Our focus is mainly on *stateful* operators that carry aggregate operations (e.g., mean, sum) and are widely used in SPEs. Those transform a window of input tuples to either a scalar value or a vector of values (in case there is a *group by* argument). First, an aggregate operation extracts a number of attributes from $e_{\mathcal{X}_i}$ and feeds them to a transformation function to produce R_w. In the event that $\mathcal{Q}$ carries a group statement, a result is produced for each distinct group of S_i^w.

In the CQ of Fig. 1 the last two lines present the accuracy specification of our work. The function *error* dictates that a relative error of up to 10% is allowed, with a confidence of $\alpha = 95\%$. Table I provides a summary of the symbols used in our formulation.

III. Proposed Model

The effectiveness of P is measured using the *imbalance* metric [2]:

$$I(P(S_i^w)) = |\max_j(L^j_{S_i^w}) - \underset{j}{avg}(L^j_{S_i^w})|,\ j = 1, \ldots, \mathcal{V} \quad (1)$$

If S_i^w carries tuples from $\mathcal{G}$ distinct groups, then the tuples are divided in $\mathcal{G}$ groups based on their $k_{\mathcal{X}_i}$. In essence, if N_g is the set of tuples of S_i^w that belong to a group g, then $N_1 \vee \ldots \vee N_{\mathcal{G}} = S_i^w$. *Imbalance*'s first operand is the total number of tuples of the most loaded worker w, whose total load is equal to $L^w_{S_i^w} = \sum_j |N_j|$, where $\{j | e_{\mathcal{X}_i} \in S_i^w \wedge k_{\mathcal{X}_i} = \{j\}\}$ and $P(N_j) \rightarrow \{w, \ldots, w\}$. In order to reduce *imbalance*, w's load has to be reduced.

Lemma 1. *For a given S_i^w, $\mathcal{V}$ workers, and a partition algorithm P,* imbalance *$I(P(S_i^w))$ is reduced if and only if the load of the most loaded worker w is reduced.*

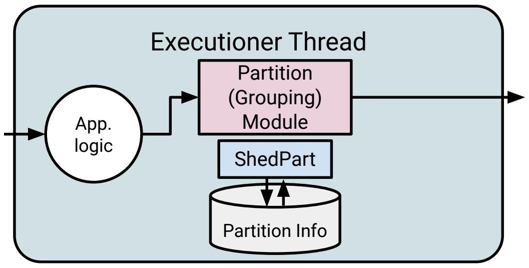

Figure 2: The ShedPart operator in an SPE worker.

We omit the proof of the Lemma due to space limitations. Lemma 1 concludes that reducing the load of w results in reducing *imbalance*. w's load is the sum of the disjoint group sets of its stored tuples. Hence, by reducing the individual sets leads to reducing $L^w_{S_i^w}$. We propose to drop tuples from each group in order to achieve the reduction of sets.

Dropping tuples leads to approximate results, which deviate from previously proposed *partitioning* techniques. Consequently, there is a need of an accuracy parameter, which consists of an error ϵ_u and a confidence α [8]. In essence, a *stateful* operation's approximate window result $\hat{R}_w$ should not deviate by more than ϵ from R_w. w's load reduction needs to take place while the accuracy of $\hat{R}_w$ is within user-defined limits. In essence, the partition model should iterate over w's assigned groups, and examine if any of them can have its aggregate result R_w be approximated. This entails that for every group g assigned to w, for which $N_g \subseteq S_i^w$, one needs to identify a simple random sample $n_g \subset N_g$ so that $\Pr(\epsilon(R_w, \hat{R}_w) \leq \epsilon_u) = \alpha$.

Our model employs normal approximation to predict the accuracy of $\hat{R}_w$. The total population's variance (S^2) is estimated, in two steps: first by creating a simple random sample of size n_1 from which the value of s^2 is gained; second, estimate n_g using s^2 as S^2's estimator. For example, if a CQ produces the arithmetic mean, s_1^2 is the variance estimated from the initial sample of size n_1 and $V = \frac{\epsilon^2}{t^2}$, where t is the inverse cumulative probability of the standard normal distribution at α. Then, $n_g = \frac{s_1^2}{V}(1 + \frac{2}{n_1})$ The distribution of the attribute values is assumed to be approximately normal, therefore n_1 needs to be fairly large. Any attempt to reduce *imbalance* given ϵ_u and α, requires the estimation of S^2 for the attribute values used to produce R_w.

IV. ShedPart

We propose **Shed**ding-**Part**itioning operator (ShedPart), which is based on our model and reduces *imbalance* by searching for groups that qualify for approximation, given (ϵ_u, α). Each SPE worker maintains a module that applies the application logic (AL) (i.e., a *stateful* operation), and a Partition Module (PM) that partitions produced tuples. Every tuple produced by AL is passed to the PM, which determines the destination of this tuple (i.e., apply P). Fig. 2 illustrates this workflow. Often, the PM maintains additional information for load-balancing purposes such as a routing index and

Algorithm 1 Shed-Part algorithm

```
procedure ShedPart($L, R, GI$)
    conclude = False, black_list = $\emptyset$
    trim_info = INIT_MAP($\emptyset$)
    sorted_group_map = SORT_GROUPS($R, GI$)
    repeat
        conclude = True
        w= MOST_LOADED_INDEX($L$)
        $\mathcal{G}_w$=sorted_group_map.get(w)
        if $\mathcal{G}_w \neq \emptyset$ then
            for all $g \in \mathcal{G}_w$ do
                if $g \notin$ trim_info $\wedge$ $g \notin$ black_list then
                    $s_g = GI.var(g)$
                    $n_g = GI.freq(g)$
                    $s$ = SAMPLE_SIZE($n_g, \alpha, \epsilon, s_g$)
                    if $s < GI.freq(g)$ then
                        conclude=False
                        UPDATE_LOAD($w, g, L, s$)
                        trim_info.add(g,s)
                    else
                        black_list.add(g)
        else
            return trim_info
    until ¬conclude
    return trim_info
```

partition info, which include the number of tuples and the number of distinct groups sent to each worker [3].

ShedPart is part of a worker thread's memory space and it operates within the PM as a complementary load balancing mechanism. Its goal is to keep track of PM's routing decision and incrementally identify distinct groups that can be approximated. At tuple arrival, ShedPart monitors the PM's decisions, and updates information for distinct groups' sample size estimation incrementally. At watermark arrival, ShedPart explores for opportunities to limit *imbalance*, by iteratively scanning the most loaded worker's groups. By the time ShedPart concludes its exploration, it allows a window's watermark to reach the downstream operators, along with information regarding shedding of tuples for individual groups. ShedPart needs to keep track of each group's frequency and variance of values. In addition, the load of each worker needs to be available, along with the routing index. Apart from each group's attribute values' variances, the rest of the information are readily available from PM.

A. Proposed Algorithm

ShedPart's operation differs during (a) tuple and (b) watermark arrival [7].

Tuple Arrival: Every time a tuple is produced by AL and is partitioned by the PM, ShedPart keeps track of values' variance for each group g. Variance requires constant memory and can be calculated incrementally. ShedPart's overhead in CPU and memory remains low, as only a sample of the values is needed for the estimation of variance (Sec III).

Watermark Arrival: ShedPart explores S_i^w's groups for possible opportunities in reducing *imbalance*. The ShedPart algorithm is depicted in Alg. 1 and consists of a main loop (Lines 5- 23), which repeats as long as the most loaded worker w's load is being reduced. This outer loop emanates from Lemma 1, and concludes if either all of w's groups have been examined (i.e., a resulting sample size $n_g \geq N_g$), or because all of w's groups have already been selected for approximation (i.e., a sample size has been established). We have to point out that on every iteration, the most loaded worker w is likely to change.

Alg. 1 has the following input: L which is an integer array of size $\mathcal{V}$ with the total number of tuples assigned to each worker. This array is the same array (L) used by existing load balancing algorithms [3]. The next argument of ShedPart is the route index $R : g \to \{w_1, \ldots, w_m\}$, which maps each distinct group g to (a) worker(s). Most of the times, R can be the hash function(s) used for partitioning each group g. Finally, a map $GI : g \to (N_g, s_g^2)$, which returns the frequency and estimated values' variance of each group. GI is the single additional structure that ShedPart requires and has an amortized memory overhead of $O(\mathcal{G})$. This overhead can be further reduced by using sketches (e.g., CountMin). The output of Alg. 1 is a map trim_info : $g \to s_g$, which for a subset of groups returns a sample size s_g (Line 24). This value indicates that tuples that belong to group g can have their result approximated by a simple random sample of size s_g.

First, an index named sorted_group_map is created, which maps a list of of groups to each worker in descending order of appearance (Line 4). The creation of this index allows for faster execution and its creation carries a worst case time complexity of O($\mathcal{G} \log \mathcal{G}$). Another auxiliary structure is initialized, named black_list, which is a set that carries the groups that have already been addressed. At every iteration of the inner loop, w is located (Line 7). Then, w's sorted list of assigned groups are retrieved from sorted_group_map (Line 8). In the event that there are no remaining groups left, ShedPart concludes execution (Line 22). Otherwise, each group $g \in \mathcal{G}_w$ is scanned (Line 10). If a group g is neither part of trim_info nor black_list, then its sample size s is estimated (Line 14). If $s < N_g$, then w's load is updated (Line 17), a record is added in trim_info (Line 18), and the outer loop's condition is updated (Line 16). On the other hand, if $s \geq N_g$, then group g is added to black_list so that it is not revisited in the future (Line 20).

Overall, the algorithm's time complexity is affected by the distribution of groups per worker, and values' variance. The worst runtime complexity materializes when all distinct group values are visited. In this case the main loop will have a time and storage complexity of O($\mathcal{G}$).

B. ShedPart Optimizations

We enhanced ShedPart with two optimizations, which alleviate its worst-case time and storage complexity.

1) Storage Optimization: we expand ShedPart's internal structure to separate group tracking based on each group's frequency. ShedPart has a parameter T_g, which is a frequency threshold used to separate distinct groups of S_i^w into two categories: the *heavy hitters* (i.e., groups that $N_g \geq T_g$), and

Table II: Imbalance on DEBS.

	50%ile	75%ile	99%ile	Max
Storm	172	240	416.8	545
Shedpart	172	240	416.8	545

Table III: Imbalance on GCM.

	50%ile	75%ile	99%ile	Max
Storm	98050	150360	1571172.39	1763230
Shedpart	128	185	307.6	315

the *cold groups* (i.e., groups that $N_g < T_g$). This optimization is similar to the one proposed in [13] and in [3].

ShedPart maintains an approximate data structure of, constant size, to keep track of *cold groups'* frequencies. Currently, ShedPart uses a CountMin sketch [14], whose size is controlled by the desired level of accuracy. In addition, ShedPart maintains a hash table for each group's tuple (N_g, s_g^2) $\mathcal{H}$. At tuple arrival, $\mathcal{H}$ is checked if it contains a record of the tuple's group. If one does not exist, the estimated count for g is retrieved from the CountMin sketch, $\hat{N}_g$. If $\hat{N}_g = 0$ then g's record in the CountMin sketch is incremented. Otherwise, ShedPart checks if $\hat{N}_g \geq T_g - 1$. If the previous is true, g is promoted to the *heavy hitters* groups, and a record is created in $\mathcal{H}$. Otherwise, the CountMin sketch is updated accordingly.

In our experience, we found that setting $T_g \geq \frac{|S_i^w|}{\mathcal{G}}$ provides the best results. ShedPart aims to improve *imbalance*, which appears in skewed datasets. The former is not the case with uniform datasets, since the P manages to balance load effectively. Until a group g makes it to $\mathcal{H}$ the attribute values needed for s_g^2 will be lost. However, this does not impact the sample size estimation since it requires the variance of a subset of the population. Finally, this optimization is used to limit the time complexity of Alg. 1 as well, by considering only *heavy hitter* groups during exploration.

2) Incremental Variance Monitoring: In our experience with real datasets, we encountered situations where the overall performance improvement from ShedPart was minimal. This happened because the attribute values' did not allow for approximation. In those circumstances, executing ShedPart increases total processing time. To this end, we enhanced ShedPart to keep track of the percentage of tuples dropped incrementally:

When a *heavy hitter* group's variance is updated, its required sampling size $\hat{n}_g$ is updated as well. At watermark arrival, the percentage $\frac{|D-\hat{D}|}{|S_i^w|}100$ is calculated, where D is the sum of frequencies of all *heavy hitter* groups[1] and $\hat{D}$ is the sum of the estimated sample sizes $\hat{n}_g$. If this percentage is lower than $\mathcal{T}$, which is ShedPart's percentage parameter, then ShedPart does not execute Alg. 1. This enhancement protects ShedPart from unnecessary execution.

V. Experimental Evaluation

We document the merits of ShedPart by experimenting with real-world datasets. Specifically, we identify under which circumstances ShedPart offers performance improvements.

[1]Each one incremented by T_g.

A. Experimental Testbed

1) ShedPart Implementation Details: We have implemented ShedPart on Storm v1.2, as a windowed bolt that sits between a data generator, and a Bolt performing a *stateful* operation. When ShedPart's watermark mechanism triggers, it performs Alg. 1 and generates the sample sizes for each group for the current window. It forwards this information to the downstream bolts to down-sample for the active window accordingly, and also produces a watermark to trigger execution.

2) Experimental Setup: We conducted experiments on a cluster of five Amazon EC2 r4.xlarge nodes. Each node ran on Ubuntu Linux 16.04, OpenJDK Java v1.8, and python v2.7 and had access to four virtual CPUs of an Intel Xeon E5-2686 v4 and 32GBs of RAM. One node was set up as the master, having a single-instance Zookeeper v3.4.10 server; each of the remaining nodes ran a single Storm supervisor process, which accommodated up to 4 workers. In all experiments, we enabled Storm's acknowledgment mechanism to guarantee processing of all tuples. Also, we enabled Storm's back-pressure mechanism to guarantee in-order delivery of tuples and avoid polluting measured times. We note that our setup did not experience any late or out-of-order tuples. In order to measure processing times with high precision, we used Storm's metrics API, which provides periodic reporting of runtime telemetry for each worker.

The CQ execution plan of our experiments consisted of a single source operator. This operator read data from a memory-mapped file, and pushed tuples to ShedPart. In turn, ShedPart distributed data to workers that executed a *stateful* aggregate operation. Each worker pushed its results to an output file. All of the experimental results are the arithmetic mean of seven runs, without the maximum and the minimum reported values. For all experiments, we used the relative error formula as our error metric.

3) Experimental Datasets: In our evaluation we used real-world streaming datasets. For each one of the real datasets, we conducted a grouped aggregate operation on a predefined event-time sliding window. Below, we provide additional details for each dataset:

- **ACM DEBS 2015 Challenge dataset (DEBS):** This dataset features ride information from a New York City Taxi company [15]. For this dataset we used one of the two queries that come with it. The CQ we selected features a 30 minute sliding window, with a 15 minute slide, for calculating the average fare amount for each route. Routes are created by dividing the New York City area in a 300 by 300 cell grid.

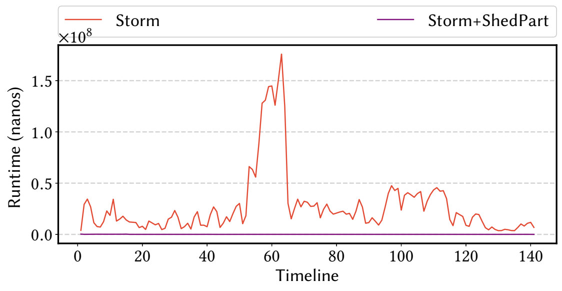

Figure 3: Max parallel step runtime on GCM.

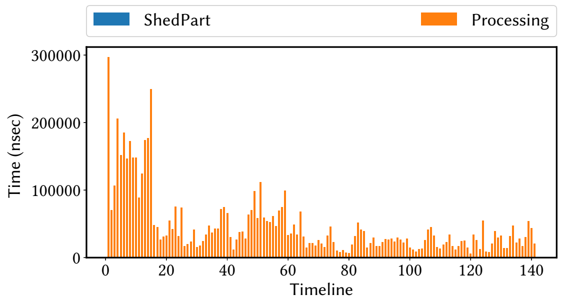

Figure 4: Breakdown of time spent on each window, in terms of ShedPart and processing (GCM).

- **Google Cluster Monitoring dataset (GCM):** For this dataset we used part of the *task-events* table and performed Query 1 from [16], which requires the average CPU time requested by different scheduling classes. In order to examine ShedPart's performance on larger windows, we set the sliding window size to 60 minutes and the slide to 30 minutes.

B. Impact on Imbalance

ShedPart's main goal is to reduce *imbalance* among workers. DEBS and GCM present significant differences in the number of groups per window (up to four for GCM and 10 thousand on average for DEBS), and in the values' variance. As a result, ShedPart is expected to face different opportunities to balance load on each dataset. For this set of experiments we set $\epsilon_u = 10\%$ and $\alpha = 95\%$. In addition, we set $T_g = 3$, and $\mathcal{T} = 15\%$, and 4 workers for calculating the grouped mean. Tables II and III depict the *imbalance* (Eq. 1) achieved with Storm's partition algorithm, and with ShedPart. We used FLD grouping for Storm, which proved to be equivalently good for those datasets. As far as DEBS is concerned, ShedPart fails to improve on *imbalance*. This happens because values' variance is high, and infrequent groups cover a significant portion of the window. As a result, ShedPart is unable to identify approximation opportunities, and *imbalance* remains the same as Storm's.

On the other hand, the GCM dataset features up to four groups per window, and values maintain lower variance. Therefore, ShedPart is able to approximate the result by using a small set of tuples. This results in significant improvement in terms of *imbalance*, as is shown in Table III. Furthermore, we measured ShedPart's impact on the most loaded worker's load. This aspect is important as it affects the time of the partial-evaluation step of a *stateful* operation. As far as DEBS is concerned, ShedPart is unable to limit the maximum load, as it was unable to reduce *imbalance*. Turning to GCM, ShedPart reduces the maximum worker load by two orders of magnitude.

C. Overall Performance

In order to measure ShedPart's impact on performance, we measure the total window runtime in terms of ShedPart execution and actual processing. Similar to the previous experiments on those datasets we set $\epsilon_u = 10\%$, $\alpha = 95\%$, $T_g = 3$, $\mathcal{T} = 15\%$, and the number of workers to 4. The reported times represent the sum of ShedPart's execution and the processing time for each window.

The main challenge for ShedPart on DEBS is to detect fast that approximation is infeasible. This is crucial because DEBS features tens of thousands of groups per window, and the iterations can impact performance. Also, DEBS is unable to be accelerated because of the values and the frequency of appearance of its groups. At watermark arrival, our ShedPart implementation identifies that this is infeasible. Thus, ShedPart does not run its algorithm and jumps directly to processing. In our experiments, we documented no difference in the total window runtime of Storm and ShedPart, which emanates from the optimizations we introduced in ShedPart. As a result, ShedPart spends no time exploring potential groups fit for approximation, and proceeds with processing.

ShedPart leads to significant performance improvements with GCM. Fig. 3 illustrates the total runtime for Storm and Storm with ShedPart. ShedPart reduces the load on the most loaded worker of each window. As a result, the total runtime drops significantly. As shown in Fig. 3, in the windows between 50 and 65 there is a temporary increase in the frequency of a particular group. Storm has to sustain this temporal skew in input. On the other hand, with ShedPart this temporary *imbalance* disappears. Fig. 4 presents the breakdown of Storm with ShedPart. GCM features up to four distinct groups per window, which results in insignificant time spent for ShedPart execution. As shown in Fig. 4, all time is spent in processing the result.

VI. Conclusion

In this paper, we presented ShedPart, which is a *partitioning* algorithm leveraging load shedding to balance load in SPEs. Shortcomings of existing partitioning algorithms motivate the use of ShedPart, which can mitigate load imbalance under certain circumstances.

ShedPart reduces *imbalance* by approximating part of the output. It achieves this by implementing a novel query and load distribution model, which enable trading accuracy for load balance.

Our proposed ShedPart operator is lightweight and compatible with current SPEs' architectures. Experiments with real datasets indicate that ShedPart can improve performance by more than an order of magnitude.

References

[1] B. Babcock, S. Babu, M. Datar *et al.*, "Models and issues in data stream systems," in *PODS*, 2002, pp. 1–16.

[2] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *PVLDB*, vol. 10, no. 11, pp. 1286–1297, 2017.

[3] A. Pacaci and M. T. Ozsu, "Distribution-aware stream partitioning for distributed stream processing systems," in *BeyondMR*, 2018, pp. 6:1–6:10.

[4] N. Tatbul, U. Cetintemel, S. Zdonik *et al.*, "Load shedding in a data stream manager," *PVLDB*, vol. 29, pp. 309–320, 2003.

[5] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE*, 2004, pp. 350–361.

[6] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," *PVLDB*, 2007.

[7] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "Concept-driven load shedding: Reducing size and error of voluminous and variable data streams," in *IEEE BigData*, 2018, pp. 418–427.

[8] ——, "Spear: Expediting stream processing with accuracy guarantees," in *ICDE*, 2020, pp. 1105–1116.

[9] R. Castro Fernandez, M. Migliavacca *et al.*, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013, pp. 725–736.

[10] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis, "Uninterruptible migration of continuous queries without operator state migration," *SIGMOD Record*, vol. 46, no. 3, pp. 17–22, 2017.

[11] N. R. Katsipoulakis, C. Thoma, E. Gratta *et al.*, "Ce-storm: Confidential elastic processing of data streams," in *ACM SIGMOD*, 2015, pp. 859–864.

[12] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1002–1015, 2019.

[13] Y. Zhou, T. Yang, J. Jiang, B. Cui *et al.*, "Cold filter: A meta-framework for faster and more accurate stream processing," in *SIGMOD*, 2018, pp. 741–756.

[14] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[15] Z. Jerzak and H. Ziekow, "The debs 2015 grand challenge," in *DEBS*, 2015.

[16] A. Koliousis *et al.*, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD*, 2016, pp. 555–569.