Noname manuscript No.
(will be inserted by the editor)

Adaptivity in Continuous Massively Parallel Distance-based Outlier Detection

Theodoros Toliopoulos · Anastasios Gounaris

Received: date / Accepted: date

Abstract We deal with the problem of dynamically allocating the workload to multiple workers in massively parallel continuous distance-based outlier detection, where the workload is conceptually split in contiguous overlapping regions. The main challenges stem from the fact that modern streaming processing frameworks, such as Apache Flink and Spark Streaming, do not support feedback loops, the process is stateful while the adaptations do not result in key redistribution but in modifying the region boundaries associated with each key. These challenges correspond to overlooked issues, which call for novel solutions that we provide in our work. More specifically, firstly, we propose an architecture for allowing such adaptations in Flink. Secondly, we propose specific techniques for adaptive region definition that are applicable to any distance metric. Finally, we conduct thorough experimental evaluation and our results show that our proposal is both efficient and effective even in small finite streams. In addition, our proposal is shown to be insensitive to the exact continuous outlier detection algorithm and outlier query parameters.

Keywords adaptive outlier detection · dynamic partitioning · massively parallel processing · Flink

1 Introduction

Distance-based outlier detection in data streams constitutes a prominent approach in runtime data analytics and is used in many applications, such as spam detection, medical diagnosis and fraud detection to name a few. It refers

T. Toliopoulos
Aristotle University of Thessaloniki, Greece,
E-mail: tatoliop@csd.auth.gr

A. Gounaris
Aristotle University of Thessaloniki, Greece,
E-mail: gounaria@csd.auth.gr

to a setting where a sliding window is applied on the data stream, new objects arrive continuously, and an object in the current window is reported as an anomaly if the number of its neighbors is below a given threshold [18]. This is still attracting significant interest, also due to the practical value that has been identified since many years [25], and several techniques have been proposed to perform the involved computations in an efficient incremental manner, e.g., [3,33,9,19,7,32,8,37,36,31]. In addition to these single-node proposals, there are efforts to parallelize distance-based outlier detection through employing partitioned parallelism on top of emerging massively parallel data processing platforms, such as Spark and Flink [30].

Traditionally, streaming data processing benefits from adaptive techniques [12], since in a streaming setting, both the data characteristics and the environmental conditions are subject to potentially frequent changes. The motivation of our work is the current lack of proposals for handling adaptivity in continuous massively parallel distance-based outlier detection. In this form of streaming application the keys correspond to overlapping regions the boundaries of which need to change dynamically without redistributing the keys.

More specifically, state management is a key element in modern big data processing systems [27,20]; when the state is in the abstract form of key-value pairs, then repartitioning of keys should be accompanied by the movement of the internal state from one worker to another potentially over the network; otherwise the result is not correct. This setting has been significantly explored in the last two decades mainly in the context of adaptive query processing [17, 14,22,16]. However, partitioning in streaming distance-based outlier detection does not conform to the afore-mentioned more explored field. In our context, partitioning is value-based and corresponds to overlapping contiguous regions, e.g. [30,11]. An example is shown in Fig. 1, where there are 12 workers and each worker is assigned one region. In the figure, the red points (numbered as 2 and 3) are replicated to the top-left partition, whereas point 1 is replicated to all three partitions that are adjacent to the top-left one. The rest of the data points belong solely to the top left cell and are not replicated. The replication due to the overlapping is essential in order for each worker to report accurate local results without any further (global) pre-processing. The need for dynamic load-balancing and adaptive partitioning stems from the fact that in a streaming scenario, the data distribution changes and evolves frequently, while it is characterized by skews during different time periods. A partitioning technique based on a sample of the data cannot cope with this change and thus a practical solution of self-adapting partitioning techniques is necessary.

The main contribution of this work is threefold. Firstly, it presents a methodology to render continuous massively parallel outlier detection in Flink adaptive and proposes specific techniques to dynamically redistribute the workload among Flink workers. The methodology is based on the development of a feedback loop architecture in Flink, which requires execution plans to be acyclic for streaming applications (something common to similar streaming platforms). Secondly, it proposes specific repartitioning techniques that go beyond the simpler value-based scenario (see also Fig. 1) to account for any

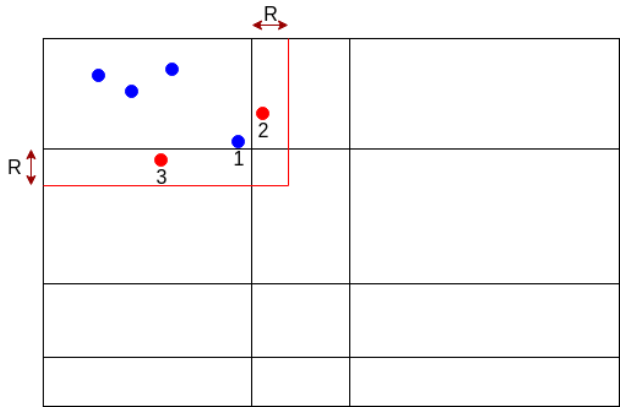

Fig. 1 Value-based grid partitioning of a 2d space

metric space rather than Euclidean only. Thirdly, we thoroughly evaluate our proposal using both numerical and text datasets, providing strong insights into the effectiveness (capability to yield lower execution times in realistic settings) and efficiency (negligible overhead) of our proposal, while investigating aspects such as sensitivity to parameters and independence from any specific outlier detection algorithm. The complete open-source implementation is provided.[1]

The remainder of the paper is structured as follows. We give the background in Sec. 2. Sec. 3 presents the repartitioning techniques. Our thorough experiments are presented in Sec. 4, followed by the remainder of the related work in Sec. 5. We conclude in Sec. 6.

2 Background

Flink execution plan is denoted by a directed graph, with vertices representing compute tasks and edges data subscriptions between such tasks. For streaming applications, the execution graph is acyclic. Regarding state, Flink distinguishes between `keyed-state` and `operator-state`. The former deals with cases where the data is either explicitly or implicitly grouped by a key value, and more commonly, the number of keys is much larger than the number of task slots. The latter holds provenance and repartitioning metadata to support elasticity actions. These elasticity actions relate to under- and over-provisioning moving data partitions to new tasks. The interested reader is referred to [10] for full details. In this context, our work faces the following two challenges.

[1] An early short version of this work has appeared in [29], which introduced the technique that is termed as *naive* in this work and was tailored for the Euclidean space and evaluated using only single-dimensional numerical datasets. We significantly extend and improve upon this early work through proposing and experimenting with more eager and sophisticated techniques, while supporting arbitrary metric distances and evaluating using both numeric and text datasets with a high number of dimensions.

Firstly, our adaptive methodology contains a feedback loop, which is not inherently supported by directed acyclic graph execution plans. Secondly, we do not deal with elasticity but with using more efficiently the already assigned resources without re-assigning keys between workers. By contrast, we modify on-the-fly the state corresponding to predefined keys, as will be explained shortly.

Distance-based outlier detection requires (i) a distance function *dist*, which assigns a non-negative value to each non-ordered pair of points; and (ii) two parameters R and k. The neighbors of each point are the set of other points the distance to which according to the *dist* function is less than R. If there are less than k neighbors, the point is reported as a distance-based outlier [18]. In a streaming setting, the set of points continuously evolve, and the outliers are reported periodically or even after every point insertion or deletion, whereas, through adopting a sliding window, old points expire after some period.

The problem of continuous distance-based outlier detection is defined formally as follows. Given a set of objects $\mathbb{O}$ and the threshold parameters R and k, for each window slide S, report all the objects $o_i \in \mathbb{O}$ for which the number of neighbors $o_i.nn < k$, i.e., the number of objects $o_j,\ j \neq i$ for which $dist(o_i, o_j) \leq R$ is less than k.

Our adaptive solution builds upon the non-adaptive parallel techniques in [30] in the context of an extensible engine, called PROUD, which is described in [28].[2] In the PROUD framework, there are two separate types of Flink tasks for data partitioning and outlier detection, respectively. Outlier detection leverages the Flink functionality for sliding windows (as opposed to tumbling windows), and the contents of each window partition along with all metadata needed are stored as `keyed-state`. Finally, there are also tasks to read and transform initial data, if needed. The window (resp. slide) size is denoted as W (resp. S). E.g., $W = 60$ seconds and $S = 3$ means that only the data from the last minute are kept and the window contents are updated every 3 seconds; as such, a specific point is alive for 20 slides.

The work in [30,28] employs two partitioning techniques, namely a *grid-based* and a *tree-based* one. For both techniques, a data point that falls close to region boundaries of multiple keys is replicated and distributed to every neighbor region, so that each worker can compute the number of neighbors of each point in parallel. The *tree-based* one, uses a VP-tree [35] in order to create a binary tree in a metric space based on the *dist* distance function. This kind of tree splits each node based on the distance of the points it contains from a data point that is chosen as the vantage point and a threshold. Its advantage is that it is more suitable for multi-dimensional datasets. A weak point of grid-based partitioning is that, during adaptations in multiple dimensions, arbitrary cell shapes may be produced that heavily impact on the effectiveness of adaptivity. Contrary to [29], in this work we employ tree-based partitioning exclusively, as the underlying structure to support repartitioning.

[2] The implementation of the whole framework along with the techniques in this work is publicly available from `https://github.com/tatoliop/PROUD-PaRallel-OUtlier-Detection-for-streams/tree/adaptive_partitioning`

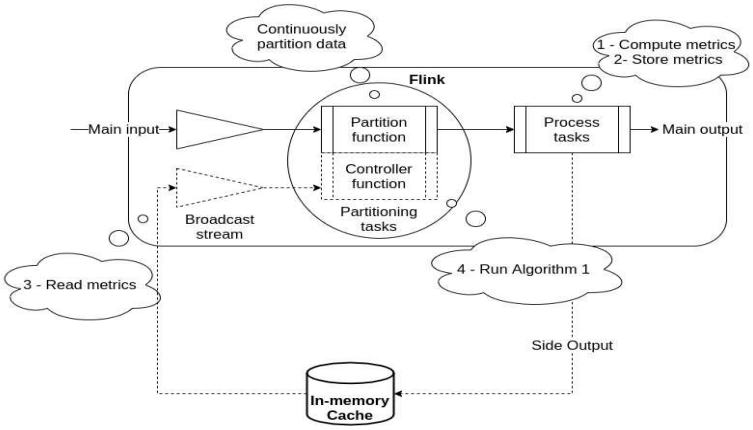

Fig. 2 Pipeline and architecture overview

3 Adaptive Repartitioning

Adapting the partitioning scheme implies changing the region boundaries of at least one key. The partitioning tasks are responsible for taking the decision whether an adaptation is required or not. The decision is the outcome of a cost function that processes metrics from the forward processing tasks. We will provide details for all these shortly, but, for the moment, the important notice is that the need for the partitioning tasks to have access to the metadata of the processing tasks arises, which corresponds to a feedback loop. This is not supported by the Flink engine, therefore it has to be created outside of it. To this end, as a first step, we employ an auxiliary data storage framework that provides in-memory cache options and does not impede high throughput in real-time applications; this implies the requirement for fast read and writes. In our current implementation, we have chosen Redis. Each Flink job writes metadata regarding data workload and processing times, while also, it reads metadata from all workers to decide the repartitioning policy.

The architecture is summarized in Fig. 2 and conforms to the PROUD framework. The solid arrows correspond to the normal flow, whereas the feedback loop is shown with the dashed lines and runs in a decentralized manner. It comprises two additional Flink tasks. The first one is a broadcasted input stream while the second one is a side output[3] that writes the metadata into the in-memory cache. Thanks to the latter, the cache is updated after each window slide. The broadcast stream continuously reads data from the cache and connects with all the partitioning tasks. Partitioning tasks implement two functions: (i) to partition new points to keys and (ii) upon arrival of new metadata from the in-memory storage, to run the *controller function* in Alg. 1, which may trigger adaptivity actions. All partitioning tasks execute the

[3] `https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/side_output.html`

Algorithm 1 The controller function

```
procedure CONTROLLER
    for each new broadcast metadata do
        run assessment policy
        if adaptation is required then
            wait β slides after the slide triggered the adaptation
            shift region boundaries by αR (see Alg. 2)
            enter transient period for W/S slides
            discard any new metadata for sleep slides
```

same deterministic code to update the distribution policy, therefore there is no need for centralized control.

The assessment policies in the controller function form an extensibility point in our solution. If an adaptation is decided, the region boundaries of the affected keys are shifted by αR, where α is a small configurable constant (the actual process may be more complex and is described in Sec. 3.3). This decision is taken by all partitioning tasks in parallel and is enacted after a predefined number β of slides; this buffer period enforces synchronization as it ensures that all partitioning tasks are ready to apply the new data distribution policy at the same window slide. The new repartitioning is enforced in the *transient* period, during which any new data points arriving are distributed according to both the old and the new keys depending on their values; the transient period lasts for $\frac{W}{S}$ slides. Then, a *stabilization* period follows, where no new adaptations are allowed for the next *sleep* slides. The implementation details of Alg. 1 are discussed below. We have currently implemented two flavors of the algorithm, termed as *Naive* and *Advanced*, respectively.

3.1 Task Metrics

Adaptations are triggered whenever an imbalance is detected according to the exact controller policies to be presented later in this section. However, imbalance can be detected in three manners, based on the exact metrics each task outputs to the in-memory cache after each slide. Overall, if there are l keys in which the incoming data are partitioning, the controller function receives a vector $< m_1, m_2, \ldots, m_l >$ of l measurements. These measurements can be of the following three types:

The first metric *M1* corresponds to the number of data points that belong to each specific task. Essentially, each task outputs the number of data points without taking into account the replicated points that were sent because they are close to the boundaries of that task. This metric represents the workload in its most traditional form, and a relatively high number indicates a skewed data distribution, which may result in the corresponding task acting as a bottleneck. The second metric *M2* is motivated by the observation that, in outlier detection, more data points to a specific key do not necessarily imply higher running times, e.g., if there are no outliers and all points belong to a dense cluster. Therefore, this metric directly computes the running time

needed to process the slide, i.e., insert the new slide's data points, compute the necessary distance functions, remove old points' stored metadata and (re-)assess each point's outlierness. The third metric *M3* aims to combine the two metrics above with a view to taking into account both the data distribution and the processing time required to make the necessary computations. To avoid any normalization issues between M1 and M2, we use their product, i.e., M3=M1· M2. Note that only the latest metrics are used, so the main-memory metadata database does not grow infinitely.

3.2 Controller function

The controller function shown in Alg. 1 is responsible for the adaptation process. This function is part of every partitioning task, meaning that each time new metadata come from the broadcast stream, every partitioning task runs the controller function to check if an adaptation is required. Each task takes the same decision at approximately the same time, but all tasks start the adaptation process at a specific future window slide. This pipeline avoids possible problems that might occur from a single controller task. In our work, it is implemented into two flavors, which differ in (1) the assessment policy, i.e., how imbalances are detected and (2) the scope of adaptations, where *Naive* modifies the boundaries of a single key (and its neighboring partitions), while *Advanced* tries to solve all imbalances detected in a single adaptivity step.

Naive policy. The *Naive* assessment policy compares each key's reported metric to the average one; the average is updated whenever new metadata are read from the broadcast stream and corresponds to the workload of each task in a perfectly balanced system. More specifically, the policy starts by finding the key with the maximum cost and compares it with the average one. If the maximum metric is greater than the average by more than ζ_{over}, then an adaptation action is triggered and the VP-tree is modified so that the region corresponding to the overloaded key shrinks. The main weak points of this policy is that it does not consider imbalances due to keys being assigned too few points and adapts only a single key each time.

Advanced policy. This policy aims to directly tackle the two main weaknesses of the *Naive* one. More specifically, in the same manner *Naive* treats the most overloaded key, *Advanced* treats all keys that exceed the ζ_{over} threshold. In addition, it is examined if a key's workload is lower than the average by ζ_{under}. If so, adaptations are triggered to move workload to such tasks as well. All of the keys are considered for both types of imbalance. In addition, the boundary change rate may differ, as detailed in Sec. 3.3. Further to these three differences, the *Advanced* policy stores the latest q metrics and uses their average (i.e., the metadata from the latest q slides) instead of relying on the metrics from the latest slide only. This improves the robustness of the policy and prevents from adapting to spikes and skews lasting for a small number of slides.

3.2.1 Adaptation periods

When an adaptation is enacted, the controller function creates three logical time periods. The first one is the buffer period β which is used to synchronize all workers in order for the adaptation to start at the same time. This helps to tackle network and latency problems in the communication between the workers. More specifically, the main-memory database may not be distributed since its workload is low, i.e., holding only the cost for each key for a small history of slides. Any option, i.e., deploying in-memory cache on either a single machine or (part of) the whole cluster, may create an overhead based on the network speed connecting the machines of the cluster. Due to this overhead, partitioners may not be ready to enact an adaptation at the same time. This is addressed by the *buffer period*. In our experiments and deployment, we have adopted a non-aggressive approach, where this period is as long as the transient period.

The second period is the transient one during which the modified keys use both the new and old tree nodes; this is required to produce the correct results. Essentially, due to the transient period, the adaptations are enforced in a gradual manner so that there is no loss of information or false results, given that the process tasks have already populated their states based on the previous keys. More specifically, during this period, when a new data point is ingested, we distinguish between the three cases below:

1. If a data point is assigned to a key that is not affected by the adaptation, the process continues without modifications.
2. If both before and after the adaptation, the point belongs to the same key that is affected by the change, then the process continues as previously with the difference that it takes into account its neighbor keys based on both the old and new regions.
3. Finally, if the point belongs to different keys due to the adaptation, then it needs to be sent to all of them with reverse flagging for the rest of the transient period. A data point that, during the transient period, is sent to a key with reverse functionality means that if it is flagged as a replicated point, it will be processed as if it is a point actually belonging to that key, and vice versa.

The final period is the stabilization one. During this period the partitioning uses only the new regions and all the reverse flagging of data points stops to be taken into account. The length of this period acts as a configuration knob to prevent from too frequent adaptations.

3.2.2 Correctness

Given that the data points in task are processed independently and the VP-tree structure, the following two lemmas hold.

Lemma 1 *For a data point p_i, all data points p_j with $dist(p_i, p_j) \leq R$ need to be partitioned to the same region/key in order to correctly assess p_i as an outlier or inlier.*

Lemma 2 *A region/key p with vantage point p_{center} and threshold $r_{threshold}$ contains all data points p_i with $dist(p_i, p_{center}) \leq r_{threshold}$. In addition, to support Lemma 1, it is assigned all data points $p_{i'}$ with $r_{threshold} < dist(p_{i'}, p_{center}) \leq r_{threshold} + R$ annotated as replicated points (at the end of each slide, only non-replicated data points have their status assessed).*

Theorem 1 *When a region's boundaries change, the partition needs to distribute new data points as explained in Section 3.2.1.*

Proof A sketch of the proof is as follows. We deal with the third case in Section 3.2.1 as the most challenging one. Without loss of generality, we assume that region's p boundaries shrink by $\alpha = 1$, $r_{old\_threshold} = r_{threshold}$ and $r_{new\_threshold} = r_{threshold} - R$. The state of p during the last slide S_x before the change took place holds all data points p_j as mentioned in Lemma 2. During the first slide after the adaptation, S_{x+1}, a new data point p_i with $r_{new\_threshold} < dist(p_i, p_{center}) \leq r_{old\_threshold}$ now belongs to the the sibling region p'. However, the task corresponding to p' does not have all older neighbors to correctly assess the status of p_i, which are contained in the task of p. Therefore, through sending p_i to p as a normal point and to p' as a replicated point, we ensure that p_i can be evaluated correctly. This holds for the whole duration of the transient period, upon the completion of which, the task of key p' contains all points necessary to assess the status of the points assigned to it as defined in Lemma 2.

3.3 Vantage Point tree runtime modifications

This part describes the implementation details of the changes that are enforced by the *advanced* repartitioning technique to correctly change the boundaries of each imbalanced region. The VP-tree holds as many leaves as the number of regions, and each node holds its vantage point and the threshold based on which the children are split; each node apart from the root of the tree is termed as either *closer* (the distance of this node's points from the parent's vantage point is less than or equal to the parent's threshold) or *further* (the distance of this node's points from the parent's vantage point is more than the parent's threshold).

When only one key is involved in the adaptation, as in the *naive* technique, we just have to increase or decrease the parent's threshold depending on whether the key is overloaded or underloaded and the node type. Combining the two types of nodes and the two types of imbalance (over- and under-loaded), we have four distinct cases:

- If a *closer* leaf is *overloaded*, then the parent's threshold needs to be decreased, which means that less data points will be partitioned to the *overloaded* leaf.

Algorithm 2 The controller function

```
1: procedure BOUNDARY CHANGE
2:     changes ← map()
3:     for each overloaded/underloaded region do
4:         if only one leaf is imbalanced then
5:             changes[parent] ← single
6:         else if both leaves are imbalanced the same way then
7:             changes[parent] ← double
8:         else
9:             changes[parent] ← propagate
10:    while changes not empty OR root reached do
11:        if changes[parent] = single then
12:            Change parent's threshold by α
13:        else if changes[parent] = double then
14:            Change parent's threshold by 2 · α
15:        else if changes[parent] = propagate then
16:            change type ← Decide the change type
17:            changes[grandparent] ← change type
18:        Remove parent from changes
```

- If a *further* leaf is *overloaded* then the parent's threshold needs to be increased, which means that the closer leaf will get more data points from the *overloaded* leaf.
- If a *closer* leaf is *underloaded* then the parent's threshold needs to be increased, which means that more data points will be partitioned to the *underloaded* leaf.
- If a *further* leaf is *underloaded* then the parent's threshold needs to be decreased, which means that the *underloaded* leaf will get more data points from the closer leaf.

On the other hand when more than one region is involved in the adaptation, as in the *advanced* technique, the process becomes more complex. When the imbalanced regions are children of different parents, the same procedure that is described for the single case can be applied for each region. The higher complexity is encountered when both children of the same parent need to be changed. To solve this problem we can distinguish between three different cases.

1. If the *closer* leaf is *overloaded* and the *further* leaf is *underloaded* then the parent's threshold needs to be decreased, which means that more new data points will be partitioned to the *underloaded* leaf than the *overloaded*. However, the change factor is increased.
2. If the *closer* leaf is *underloaded* and the *further* leaf is *overloaded* then the parent's threshold needs to be increased, which means that more new data points will be partitioned to the *underloaded* leaf than the *overloaded*. Again, the change factor is increased.
3. When both leaves are either *overloaded* or *underloaded*, then the change needs to be propagated to their grand-parent.

The explanation behind the third case is that since both leaves have the same problematic behavior, the threshold of the parent cannot be changed in a manner to reflect a more balanced distribution. In this case, the change needs to directly affect the parent, i.e., the data points that get partitioned to the parent node need to be increased/decreased if both leaves are underloaded/overloaded respectively. This type of propagation can reach the root node where it stops.

After having listed all cases, we introduce Alg. 2, which starts by annotating all regions (keys) and inserting them to a map data structure. Lines 3-9 iterate over the overloaded and underloaded regions in order to decide what type of change is needed. Afterwards, in lines 10-18, we iterate over the map contents to complete the necessary actions. The node either changes its threshold or propagates the change to its own parent. In both cases, the node is taken off the map. Lines 15-17 use both lists of cases above to decide what type of change is needed for its parent and then insert it along with the parent itself into the map structure. The process ends when the map is empty (or when it has reached the root node).

4 Performance evaluation

In total, 6 datasets (4 real-world and 2 synthetic) have been used to provide a wide and varying setting. We divide these datasets in three groups. The first group consists of 3 datasets with numerical values, namely *Stock*, *TAO* and *Gauss(2)*. Stock and TAO are real-world datasets with 1 and 3 dimensions, respectively, with their details described in [31], while Gauss(2) is a synthetic one that is synthesized from 2 different gaussian distributions that are applied one after the other in order to simulate a big change in the data distribution on a specific point in time. Gauss(2) provides insights on the technique's ability to cope with a sudden change that completely changes the distribution of the data, while the former two datasets encapsulate smaller but continuous evolutions of data characteristics. The second group consists of 2 text datasets, and more specifically sets of 2-grams from *Twitter* and *DBLP*, with their details described in [5]. The third group contains the second synthetic dataset, and is referred to as *Gauss(10)*, which is a one-dimensional numerical dataset synthesized from 10 gaussians being applied concurrently during the complete dataset duration. This dataset does not involve any changes in the data distribution; its role is to provide insights in the overheads that the in-memory cache and the whole adaptivity architecture incurs, when the monitoring and assessment mechanisms run while no adaptivity actions are required.

The distance function is the euclidean distance for every numerical dataset; for the text datasets, we transform each set to a binary vector of length equal to the vocabulary size and the distance is 1-J_{sim}, where J_{sim} is the Jaccard similarity between two sets. The window (W) and slide (S) size is set to 10K and 500, respectively meaning that, in every slide, 500 new data points

Table 1 R values for each dataset

Dataset	Stock	TAO	Gauss(2)	Twitter	DBLP	Gauss(10)
R	0.45	1.9	1	0.86	0.8	0.25

Table 2 Parameters for each technique

Parameter	Values	Advanced	Naive	Static
change factor α	0.25,0.5	✓	✓	-
queue size q	5,10,20	✓	-	-
overload factor ζ_{over}	130%,150%	✓	✓	-
underload factor ζ_{under}	10%,30%	✓	-	-
cost metric	M1,M2,M3	✓	✓	-

are inserted into the window and the same number of old data points expire[4]. Although this work is intended for usage in a continuous and intense streaming environment, for the sake of fair comparison, all of the datasets are resized in order to provide measurements for the same number of window slides. All measurements reported below refer to the performance after 1000 slides.

The outlier detection algorithms run with parameter k set to 50 neighbors; the R parameter differs for every dataset in order for the number of outliers to be approximately 1% of the total data points in each window. Table 1 shows the specific R value for each dataset, to allow for experiment reproducibility. In addition, all three adaptation periods last for a complete window to be totally replaced, i.e., for $\frac{W}{S}$ slides. Note that this is the minimum duration for the transient period, while, setting the buffer period length as such corresponds to a setting where there are no synchronization issues in practice at the expense of making adaptations less eager. The adaptations are made less eager also by setting the stabilization larger than 0. However, the eagerness/aggressiveness of our adaptation policies are also tuned with the help of additional parameters. More specifically, the parameters that affect the adaptation techniques are presented in Table 2 for each technique. In order to assess the efficiency of the two techniques, we have also used the default flavor of the PROUD outlier detection framework that does not include adaptation and does not use Redis for read/write operations, which is referred to as *Static* in the parameter table. Finally, the computing infrastructure of the experimental setting is as follows. A cluster of 3 heterogeneous machines connected through a 1GB Ethernet is used to run all of the experiments. The first machine is equipped with a 6-cores/12-threads CPU and 64GB of RAM. The second machine has a 8-cores/8-threads CPU with 32GB of RAM while the third one has a 8-cores/16-threads CPU with 64GB of RAM. One Flink task manager is active on each machine whereas the first machine also runs Flink's jobmanager and

[4] We have experimented with additional parameters (slide sizes of 1% up to 50% of the window size) and the results are similar to the ones to be presented; due to space constraints such experiments are omitted.

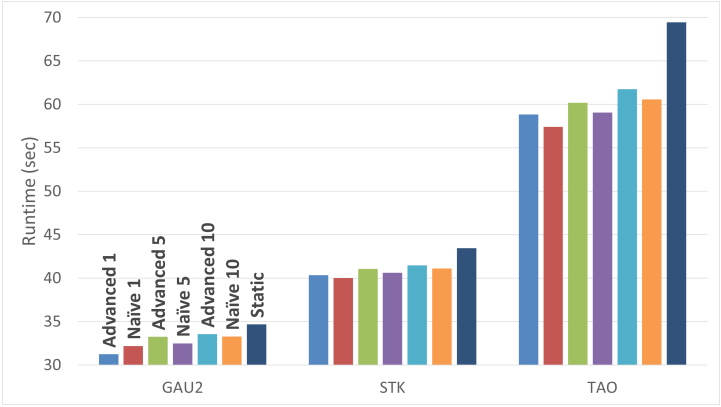

Fig. 3 Average runtimes of the Advanced, Naive and default techniques

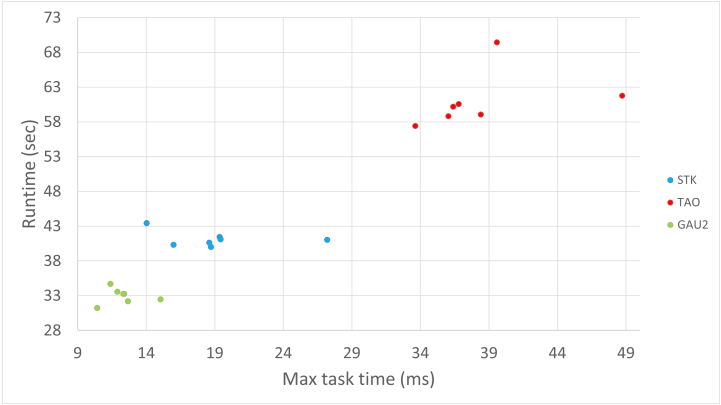

Fig. 4 Correlation between runtime and maximum processing task time

the Redis instance; due to the low load, there is no need for increased Redis parallelism. Each experiment is repeated 5 times.

In the following, we start by providing concrete evidence regarding the effectiveness of our solution and its capability to yield concrete improvements in the performance; i.e., to complete the processing of a finite stream in less time. We then thoroughly discuss the sensitivity to the configuration parameters, the outlier detection algorithm and the computation infrastructure. We conclude with experiments regarding the overhead incurred. Note here that we do not compare against the grid-based technique in [29], since it is a preliminary technique for adaptations with many limitations, e.g. works only on 1-dimensional datasets and does not support arbitrary metric spaces.

4.1 Performance experiments

Firstly, we experiment with the Stock, TAO and Gauss(2) and all 72 configuration combinations from Table 2. Fig. 3 presents the results. We are going to discuss robustness to parameters in a subsequent part of the evaluation, but here, in order to provide evidence that our proposal can indeed yield lower runtimes without sophisticated fine-tuning (which would also require examining thousands of configuration combinations instead of 72 only), we do not present just the best runtimes, but also the 5th and 10th lower average runtime for each adaptive technique (corresponding to the 93th and 86th percentile, respectively). As the figure shows, both techniques exhibit improvements on the running times by a few seconds on the 1-dimensional datasets and by a larger margin on the 3-dimensional one, while there is no clear winner between them. These results are expected since the TAO dataset involves more intense outlier detection processing in comparison to the other 2 datasets, while *Advanced* have more parameters for tuning. More specifically, *Advanced* is up to 2.95% faster than *Naive* and up to 15.27% faster than *Static*. *Naive* is up to 2.42% faster than *Advanced* and up to 17.32% faster than *Static*. Finally, as shown in the figure, despite the fact that we have only checked 72 combinations of parameters, the performance improvements hold also for the 5th and 10th best performing setting.

The results presented above refer to the final performance as perceived by the end user. However, the adaptivity techniques directly target task times. Fig. 4 presents the correlation between the job's runtime and the average slide processing time of the slowest outlier detection task. For the correct interpretation, first it is important to note that Flink's initial DAG creation is automated and can allocate two or more different keys on the same task slot. This means that one task/thread will be responsible for the processing of two regions simultaneously for the whole streaming process, making it the slowest task. In this plot, the runtime of the first, fifth and tenth experiments as mentioned before are presented along with the slowest processing task. In general, there is a correlation between maximum average task time and job runtime; however, relatively big differences in the maximum task times may not be reflected upon the total runtime, since, apart from other factors, internal scheduling aspects are also involved. Despite all these, as we have shown, we manage to attain tangible improvements.

The behavior of the two techniques is largely dependent on the data distributions. For example, Stock is a real-world dataset where the distribution changes very often but not drastically, which favors more limited adaptations, such as those enforced by *Naive*. Meanwhile, the Gauss(2) dataset has a stable distribution for the first period, which drastically changes after a while. In this dataset, the adaptation techniques start working after the change. The *Advanced* technique can cope with this kind of change in a better way than the *Naive* one, since it can change more than one regions concurrently; this is despite the fact that, due to the smoothing queue that it employs, *Advanced* does not start adapting as soon as the change occurs.

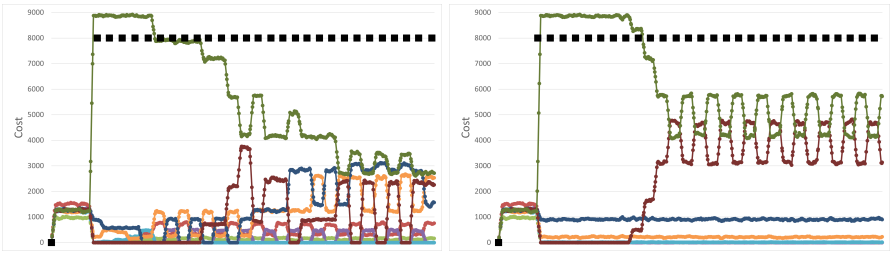

Fig. 5 Timeseries of M1 per region and adaptation points for the *Advanced* (left) and the *Naive* (right) technique when processing the Gauss(2) dataset

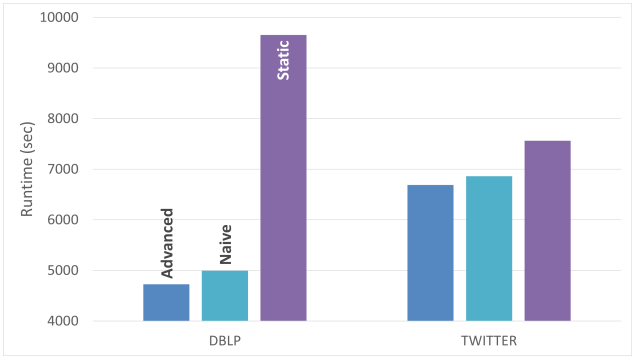

Fig. 6 Runtime difference between all techniques for the text datasets

Fig. 5 presents the M1 cost function per key on the time axis along with the timestamps (denoted as squares) where a decision that an adaptation is necessary was taken; in this setting, overall the data is split across 8 keys(regions). The two main observations are as follows. In the *Naive* technique, only two keys start converging and this stems from the fact that the technique only changes the most overloaded region. In this case, the region that starts getting every data point when the distribution changes, starts decreasing its boundaries and, at the same time, the boundaries of its neighbor are increased. This results in the second key getting most data points, which becomes the new most overloaded one. Overall, the two keys change the load between themselves. On the other hand, the *Advanced* technique affects more regions concurrently and better splits the workload between them. The figure corresponds to 2000 slides; if the stream ran infinitely, at the end, all 8 keys would have received approximately equal workload.

We now turn our attention to the text datasets. Fig. 6 presents the running times comparing both adaptation techniques with the default static flavor. The plot further strengthens the previous experiments' insights that going up in dimensions helps hide the overheads created by the adaptation techniques and improve the workload balance between all regions. The adaptive techniques improve the runtime. More specifically, *Advanced* is the dominant solution and yields lower execution times by a factor of 5.34% compared to *Naive* and

Table 3 Speedups for each metric

	Stock	TAO
M1	7.17%	15.27%
M2	7.88%	15.83%
M3	7.60%	17.32%

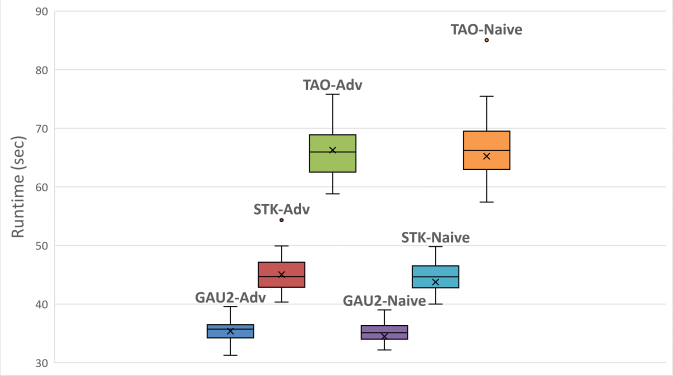

Fig. 7 Boxplots for the *Advanced* and *Naive* techniques for the numerical datasets

of 51.01% compared to *Static*. *Naive* improves upon *Static* up to 48.24%. In general, we observe small benefits of *Advanced* over *Naive*, which implies that, at least in the settings examined, lazy adaptivity solutions are competent. We have experimented with several other settings, including datasets with spikes, and still, the differences between the two adaptivity proposals were small (no detailed results are provided due to space limitations). We also observe significant differences between the two datasets: for the Twitter dataset, the improvement of both adaptive techniques are smaller. This is attributed to the fact that Twitter is more sparse than DBLP. With 37K distinct 2-grams in comparison with the 17K of DBLP, the initial partitioning yields more balanced regions. Nevertheless, both techniques manage to improve over the static flavor meaning that even in a disperse dataset, the adaptation of the boundaries can help in reducing the work of the processing tasks.

Finally, Table 3 presents the speedups for the real-world numerical datasets based on the 3 cost metrics. As expected from the previous results, the TAO dataset produced bigger speedups going up to 15.83%. All 3 cost metrics have similar speedups that are dependent on the dataset. The cost metrics are further investigated in the next experiments.

4.2 Parameter analysis and robustness

Our main claim is that our approach is not sensitive to parameters because we achieved significant improvements without resorting to fine-tuning, we presented also the 5th and 10th best performing configurations and we explored a limited number of combinations. In a sense, the benefits presented in the

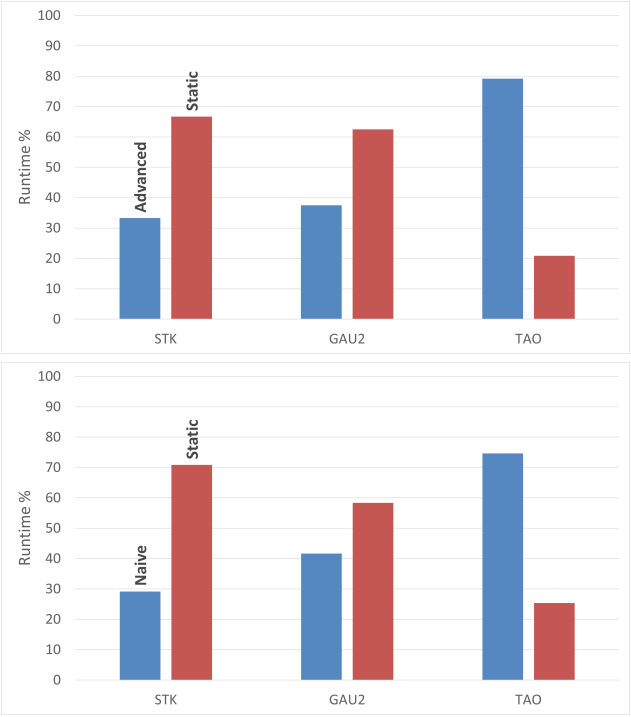

Fig. 8 Percentage of improved runtimes of the Advanced (top) and Naive (bottom) techniques compared to the static flavor

first set of experiments constitute a lower bound of the potential benefits, since exploring more configurations can lead to more significant performance improvements. Nevertheless, the purpose of this part of experimental analysis is to find possible correlations between the parameter of each technique and insights that might be of use during a fine tuning process, although fine tuning the parameters is out of the scope of this specific work that aims to introduce the adaptivity methodology.

Fig. 7 shows the boxplots of all 72 combinations regarding the numerical datasets; from the figure, it is clear that the runtimes can deviate significantly, and if combined with Fig. 3, it can be deduced that several configurations do not lead to performance improvements. To further elaborate on this aspect, Fig. 8 presents the percentage of the experiments for the datasets in the first group, where the runtime is improved. We can see that for TAO, the majority of cases leads to improvements, whereas, for the other two datasets, which involve more lightweight processing, this is not the case.

A main research question that may arise is as to whether we can derive in this work robust guidelines for performance tuning. To answer this question, we try to find correlations between the 5 tuning parameters in Table 2 and the resulted runtime. Fig. 9 presents the percentage of the experiments for

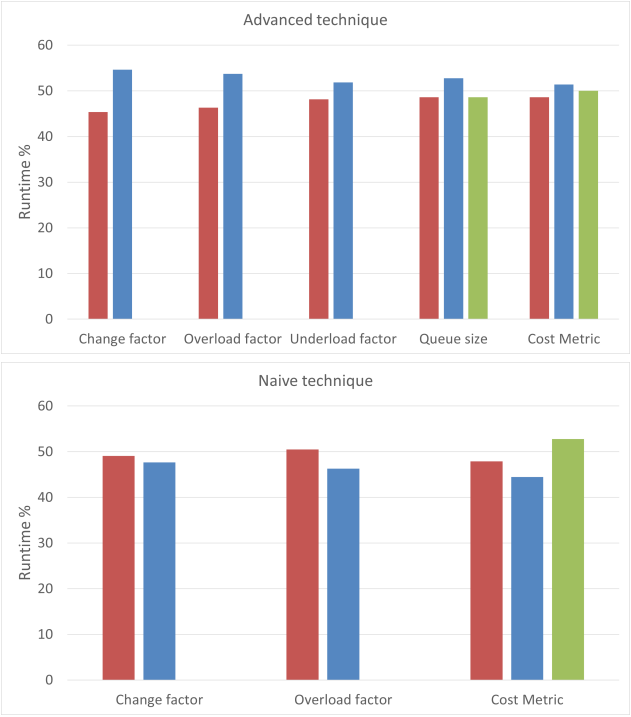

Fig. 9 Percentage of improved runtime for the Advanced (top) and Naive (bottom) techniques for each parameter value

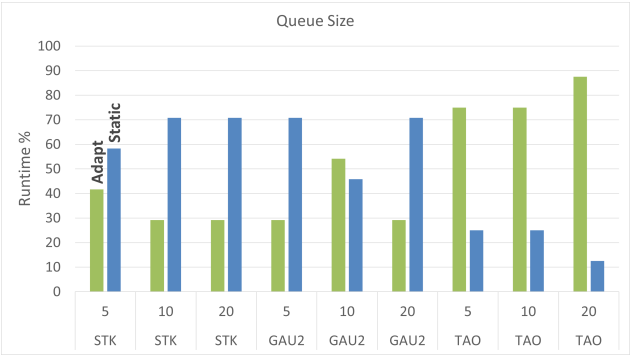

Fig. 10 Runtime difference for the Queue size parameter on the Advanced technique for each numerical dataset

each parameter value that the runtime of the system has improved. The top plot presents all 5 parameters for the *Advanced* technique and the bottom one presents the ones used for the *Naive* technique. Each column represents one of the configuration values examined. A main observation is that *Advanced* and *Naive* need to be configured in a different manner. The former technique seems

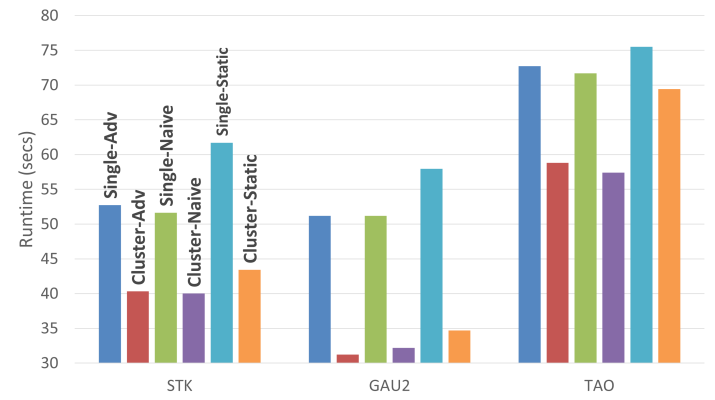

Fig. 11 Runtime comparison for each technique and dataset between the single machine and the cluster of machines

to benefit from larger values regarding the change, overload and underload factors, whereas the opposite holds for the latter technique. The cost metric also plays a more significant role in *Naive*. Finally, Fig. 10 presents the percentage of improvement cases for the *Advanced* technique for each dataset and each *Queue size* value used during the experiments. As the plot presents, there is no clear winner between the queue settings and different settings perform better in different datasets. This observation also holds for other configuration parameters.[5]

In summary, although there exist several configurations that can lead to performance improvements, the limited set of parameters chosen for the experimental analysis provides a concrete and robust baseline that improves the performance of the system when the adaptation techniques are chosen against the static one. More extensive exploration of the parameter space using specialized techniques is expected to further improve the results significantly.

4.3 Additional settings

In these experiments, we evaluate the behavior under different settings in terms of the computational infrastructure, the algorithms employed and the outlier detection queries. First, we rerun the numerical experiments using only the first machine of the cluster, where also the Flink jobmanager, taskmanager and Redis instances are deployed. Since less taskmanagers are available, the number of task slots also decreases. In the experiments the number of keys/regions is kept the same meaning that each slot will need to process more than one keys and possible other tasks, e.g. partitioning. Fig. 11 presents the comparison of the best runtimes for each technique and dataset between the single machine

[5] To further investigate any possible correlation we have run some tests using the MMPC algorithm [6] after transforming the runtime values to a binary target variable (improvement or not-improvement); this has also not yielded any concrete results.

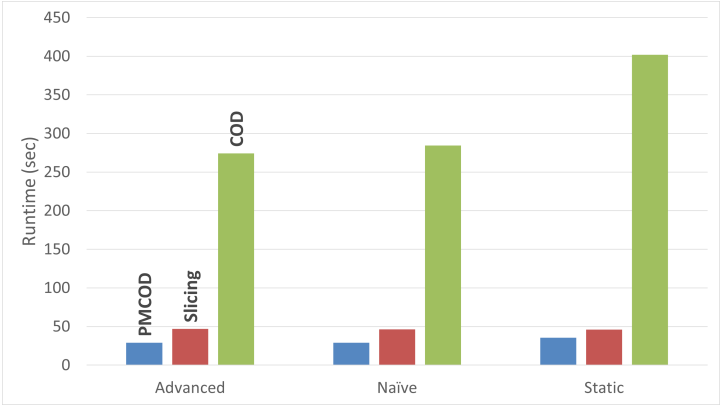

Fig. 12 Runtime difference for the different algorithms in combination with the adaptive techniques and the default flavor for the Stock dataset

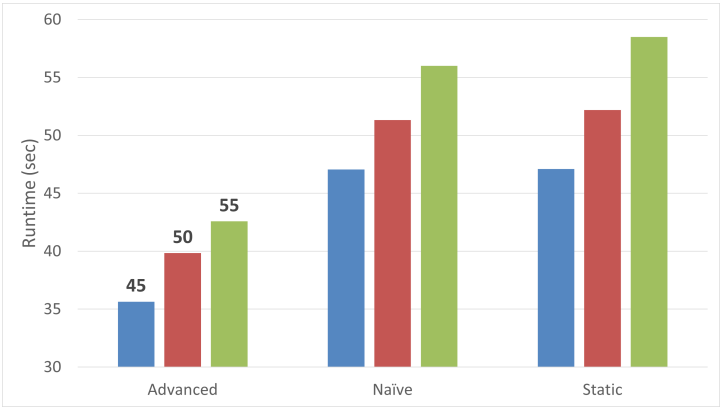

Fig. 13 Runtime difference for the different k values in combination with the adaptive techniques and the default flavor

setting and the cluster setting. As expected, the running times on the cluster are always faster than the single machine and this stems from the fact that there are more task slots and more computation power overall. Also in most of the cases, the difference between cluster and single machine is more evident in the default static algorithm.

Second, we assess the impact of different outlier detection algorithms in combination with the adaptation techniques. We aim to prove that the benefits of our methodology are independent of the actual continuous outlier detection algorithm employed. Thus far, all results were presented using the *PMCOD* and *Slicing* parallel algorithms from [30], for the numerical and text datasets, respectively. In this new experiment, we also employ the *COD* algorithm from [19] after transferring it to a distributed environment.

Fig. 12 presents the difference in runtimes between all 3 algorithms for both *Advanced* and *Naive* techniques along with the static flavor for the Stock

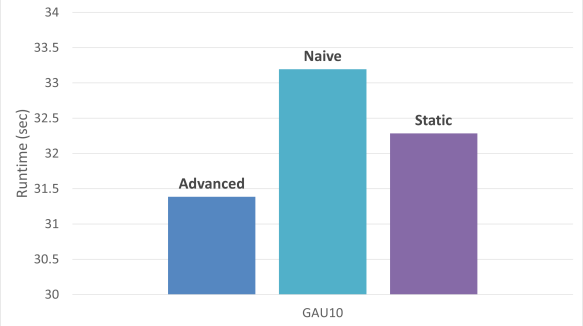

Fig. 14 Runtime difference for the Gauss(10) dataset on all techniques

dataset. As expected, the impact of the algorithm does not affect the rebalancing process. Overall, the *PMCOD* algorithm is the fastest one followed by the *Slicing* one. The *COD* algorithm is the slowest one by a big margin as expected based on the results of [32]. As mentioned previously, we have not performed fine-tuning of the adaptive techniques, but in general, the same trend applies to all continuous outlier detection algorithms

Finally, to investigate the impact of the outlier detection parameters on the adaptation techniques, another experiment is introduced. Changing either the k or R parameters for these algorithms, the percentage of outliers that are detected by the system changes as well, while this change also impacts on the performance, since each algorithm creates different data structures and has different ways of processing the data points. In this experiment, we used the *PMCOD* algorithm and present 3 different values for the k parameter using each adaptation technique, while keeping the R parameter static based on Table 1. Fig. 13 presents the runtimes for each k value and adaptation technique. As shown in the figure, the high-level behavior of each technique stays the same when changing the parameter values. As expected, a lower k value provides more inliers and decreased processing times whilst a bigger k value correspond to an increased number of outliers and increased runtime.

Overall, the adaptivity techniques can be further improved from fine tuning their parameters, as shown in the previous subsection, but are independent from additional important aspects, such as exact outlier detection algorithm employed, computational setting and outlier query parameters, as shown in this set of experiments.

4.4 Overheads

One of the main concerns described in Section 3 is the overhead that could be induced by using an external system as a main-memory cache. The problem is the communication cost that might create delays to the whole system, especially when the cache is only deployed on a single physical node. To investigate this possible delay, we used the Gauss(10) dataset which has the same distri-

bution throughout its lifetime. This means that the rebalancing techniques, while computing the cost metrics and reading/writing data from Redis, do not change the boundaries for the whole experiment. This is independent from the parameters used since all workload is already balanced. A note of attention here is that the Gauss(10) dataset is a 1-dimensional numerical dataset, which implies that the outlier tasks have minimal computational requirements. In the case where more processing time was needed for these tasks, the communication cost could have been hidden due to the distributed environment and the parallelism, as is evident in the TAO experiments in the previous experiments. Fig. 14 presents the runtimes for the Gauss(10) dataset using both rebalancing techniques as well as the default static flavor. As the figure shows, the runtime difference between the techniques and the default algorithm is less than 1 second, with *Advanced* technique being faster and the *Naive* one being slower. This means that the communication cost and the Redis overhead in general is indeed hidden behind the parallelism and the processing tasks even in one of the most lightweight scenarios possible. The difference in the runtimes is attributed to the initial sampling and creation of the VP tree.

4.5 Final Discussion and Directions for Future Work

The experiments above provide strong evidence that our technique is 1) effective, since it can yield performance improvements, 2) efficient, since it is characterized by negligible overhead and 3) robust to its parameters, since it does not rely on fine tuning. It is important to note that the magnitude of its efficiency has not been explored in depth. On the contrary, we have shown results using small finite streams of 1000 slides and we have explored very few tuning configurations that nevertheless show important gains. Even in such a limited setting, performance improvements can be up to 51.01%.

We have already identified the need to delve into fine tuning techniques for parameter optimization. We believe that, due to the dependence of the behavior on data characteristics, such techniques should be adaptive as well thus leading to a more sophisticated self-balancing and self-tuning methodology. This is left for future work. A second issue that is worth investigating is whether higher benefits can be produced if we develop external mechanisms to assign keys to Flink workers rather than relying on automated mapping. Finally, investigating more eager adaptations remains an open issue.

5 Additional Related Work

In addition to the related work discussed in the introduction, there are several other proposals that, in summary, address the problem of adaptive partitioning and load balancing in complementary settings and/or manners that are not applicable to our problem. The most relevant proposals include [2, 26], which perform adaptive partitioning over a static dataset, with the adaptations being driven by the range query workload. However, these works do not

employ partially overlapped regions as we do. An early technique for adaptive stream processing has appeared in [4], but this proposal, apart from not fitting well into the Flink framework, it assumes a single node for partial result aggregation, which creates a bottleneck. The same limitation appears in [24].

[1] presents a new technique for partitioning data in map-reduce stream processing systems that use micro-batches. The technique partitions the data of each micro-batch for both the map and the reduce phase based on different metrics such as block size whilst changing the number of keys on the fly for better resource utilization. Transferring this technique to our setting corresponds to using far more regions in the partitioning phase, which aggravates replication issues. [21] presents a Flink implementation of lazy partitioning tailored to distributed joins, which injects delays to account for transient network skew. This technique operates at a lower-level than ours and does not fit in a streaming scenario. An interesting approach appears in [23] that advocates examining random partitioning instead of partitioning in contiguous regions whilst [34] uses a k-d tree to partition the data into overlapping regions depending on the distribution. Both these proposals refer to the DBSCAN clustering technique. In addition, the work in [13] partitions spatio-temporal graphs in different workers by replicating the nodes and some of the edges. The partitioning is based on the supported queries and a single coordinator is used for meta-data storage and query processing. The work in [15] compares the different techniques for graph partitioning on large clusters where nodes need to be replicated wherever their edges are partitioned to. All these proposals contain interesting ideas regarding partitioning but do not deal with revising partitioning decisions on the fly.

6 Conclusions

Streaming distance-based outlier detection is a hot area with several proposals in the recent years. However, although the problem inherently refers to a volatile setting in terms of data and environment characteristics, no adaptive solutions have been proposed so far. Our proposal aims to address this limitation and proposes an adaptive architecture along with concrete techniques for repartitioning data on the fly in the Flink massively parallel stream processing platform. To this end, we address several key challenges: to devise a feedback loop architecture on top of an engine that does not allow cyclic processing logic, to account for the fact that streaming parallel outlier detection splits the space into partially overlapping regions instead of multiple keys, and to devise concrete adaptive techniques. The experiments show that our proposals are efficient and capable of yielding tangible performance benefits, even in settings where the streams are finite and small.

Acknowledgements This research work has been supported by the European Commission under the Horizon 2020 Programme, through funding of the LifeChamps project (Grant 875329).

Declarations

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Funding: European Commission under the Horizon 2020 Programme, LifeChamps project (Grant 875329).

Availability of data and material (data transparency): all datasets used are publicly available from third-part repositories.

Code availability (software application or custom code): all code is available from `https://github.com/tatoliop/PROUD-PaRallel-OUtlier-Detection-for-streams/tree/adaptive_partitioning`

References

1. Abdelhamid, A.S., Mahmood, A.R., Daghistani, A., Aref, W.G.: Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 2455–2469 (2020)
2. Aly, A.M., Mahmood, A.R., Hassan, M.S., Aref, W.G., Ouzzani, M., Elmeleegy, H., Qadah, T.: AQWA: adaptive query-workload-aware partitioning of big spatial data. PVLDB **8**(13), 2062–2073 (2015)
3. Angiulli, F., Fassetti, F.: Detecting distance-based outliers in streams of data. In: CIKM, pp. 811–820 (2007)
4. Balkesen, C., Tatbul, N.: Scalable data partitioning techniques for parallel sliding window processing over data streams. In: International Workshop on Data Management for Sensor Networks (DMSN) (2011)
5. Bellas, C., Gounaris, A.: An empirical evaluation of exact set similarity join techniques using gpus. Inf. Syst. **89**, 101485 (2020). DOI 10.1016/j.is.2019.101485. URL `https://doi.org/10.1016/j.is.2019.101485`
6. Brown, L.E., Tsamardinos, I., Aliferis, C.F.: A novel algorithm for scalable and accurate bayesian network learning. In: M. Fieschi, E.W. Coiera, J.Y. Li (eds.) MEDINFO 2004 - Proceedings of the 11th World Congress on Medical Informatics, San Francisco, California, USA, September 7-11, 2004, *Studies in Health Technology and Informatics*, vol. 107, pp. 711–715
7. Cao, L., Wang, J., Rundensteiner, E.A.: Sharing-aware outlier analytics over high-volume data streams. In: ICDM, pp. 527–540. ACM (2016)
8. Cao, L., Yan, Y., Kuhlman, C., Wang, Q., Rundensteiner, E.A., Eltabakh, M.Y.: Multi-tactic distance-based outlier detection. In: ICDE, pp. 959–970 (2017)
9. Cao, L., Yang, D., Wang, Q., Yu, Y., Wang, J., Rundensteiner, E.A.: Scalable distance-based outlier detection over high-volume data streams. In: ICDE, pp. 76–87 (2014)
10. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: Consistent stateful distributed stream processing. PVLDB **10**(12), 1718–1729 (2017)
11. Cordova, I., Moh, T.: DBSCAN on resilient distributed datasets. In: 2015 International Conference on High Performance Computing & Simulation, HPCS, pp. 531–540 (2015)
12. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007)
13. Ding, M., Chen, S.: Efficient partitioning and query processing of spatio-temporal graphs with trillion edges. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1714–1717. IEEE (2019)
14. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. VLDB J. **23**(4), 517–539 (2014)

15. Gill, G., Dathathri, R., Hoang, L., Pingali, K.: A study of partitioning policies for graph analytics on large-scale distributed platforms. Proceedings of the VLDB Endowment **12**(4), 321–334 (2018)
16. Gounaris, A., Yfoulis, C.A., Paton, N.W.: Efficient load balancing in partitioned queries under random perturbations. TAAS **7**(1), 5:1–5:27 (2012)
17. Katsipoulakis, N.R., Labrinidis, A., Chrysanthis, P.K.: A holistic view of stream partitioning costs. PVLDB **10**(11), 1286–1297 (2017)
18. Knorr, E.M., Ng, R.T., Tucakov, V.: Distance-based outliers: Algorithms and applications. The VLDB Journal **8**(3-4) (2000)
19. Kontaki, M., Gounaris, A., Papadopoulos, A.N., Tsichlas, K., Manolopoulos, Y.: Efficient and flexible algorithms for monitoring distance-based outliers over data streams. Information systems **55**, 37–53 (2016)
20. Monte, B.D., Zeuch, S., Rabl, T., Markl, V.: Rhino: Efficient management of very large distributed state for stream processing engines. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD, pp. 2471–2486 (2020)
21. Rupprecht, L., Culhane, W., Pietzuch, P.R.: Squirreljoin: Network-aware distributed join processing with lazy partitioning. PVLDB **10**(11), 1250–1261 (2017)
22. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: An adaptive partitioning operator for continuous query systems. In: U. Dayal, K. Ramamritham, T.M. Vijayaraman (eds.) ICDE, pp. 25–36 (2002)
23. Song, H., Lee, J.: RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD, pp. 1173–1187 (2018)
24. Su, L., Han, W., Yang, S., Zou, P., Jia, Y.: Continuous adaptive outlier detection on distributed data streams. In: International Conference on High Performance Computing and Communications, pp. 74–85 (2007)
25. Subramaniam, S., Palpanas, T., Papadopoulos, D., Kalogeraki, V., Gunopulos, D.: Online outlier detection in sensor data using non-parametric models. In: VLDB, pp. 187–198 (2006)
26. Tang, M., Yu, Y., Malluhi, Q.M., Ouzzani, M., Aref, W.G.: Locationspark: A distributed in-memory data management system for big spatial data. PVLDB **9**(13), 1565–1568 (2016)
27. To, Q., Soto, J., Markl, V.: A survey of state management in big data processing systems. VLDB J. **27**(6), 847–872 (2018)
28. Toliopoulos, T., Bellas, C., Gounaris, A., Papadopoulos, A.: PROUD: parallel outlier detection for streams. In: SIGMOD (demo track, to appear) (2020)
29. Toliopoulos, T., Gounaris, A.: Adaptive distributed partitioning in apache flink. In: 36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020, pp. 127–132. IEEE (2020)
30. Toliopoulos, T., Gounaris, A., Tsichlas, K., Papadopoulos, A., Sampaio, S.: Continuous outlier mining of streaming data in flink. Inf. Syst. **93**, 101569 (2020)
31. Tran, L., Fan, L., Shahabi, C.: Distance-based outlier detection in data streams. PVLDB **9**(12), 1089–1100 (2016)
32. Tran, L., Mun, M., Shahabi, C.: Real-time distance-based outlier detection in data streams. PVLDB **14**(2), 141–153 (2020)
33. Yang, D., Rundensteiner, E., Ward, M.: Neighbor-based pattern detection for windows over streaming data. In: EDBT, pp. 529–540 (2009)
34. Yang, K., Gao, Y., Ma, R., Chen, L., Wu, S., Chen, G.: Dbscan-ms: Distributed density-based clustering in metric spaces. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1346–1357. IEEE (2019)
35. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA, vol. 93, pp. 311–321 (1993)
36. Yoon, S., Lee, J., Lee, B.S.: NETS: extremely fast outlier detection from a data stream via set-based processing. PVLDB **12**(11), 1303–1315 (2019)
37. Zhao, G., Yu, Y., Song, P., Zhao, G., Ji, Z.: A parameter space framework for online outlier detection over high-volume data streams. IEEE Access **6**, 38124–38136 (2018)