Adaptive Distributed Partitioning in Apache Flink

Theodoros Toliopoulos
Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
tatoliop@csd.auth.gr

Anastasios Gounaris
Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
gounaria@csd.auth.gr

***Abstract*—Dynamically adapting the workload of each worker in Flink is a challenging issue. In this work, we deal with a special case, where the data are conceptually split in contiguous overlapping regions. This scenario is encountered in several streaming applications, such as those employing nearest neighbor queries. We propose (i) an architecture for allowing such adaptations in Flink and (ii) specific data repartitioning techniques. We apply our proposal to a specific use case, namely continuous distance-based outlier detection. Our experimental evaluation provides insights into the efficiency and effectiveness of the approach.**

***Index Terms*—Flink, adaptive, partitioning, distributed**

I. Introduction

Nowadays, there are several mature engines for stateful batch and stream processing, such as Apache Spark, Flink and Storm. Such systems, internally, represent analytics jobs as directed (acyclic) graphs and aim to benefit from massively parallel principles through employing partitioned parallelism. Partitioned parallelism applies to all vertices of the execution graph and its degree defines how many worker nodes participate in the vertex computation, i.e., they receive relevant tasks.

This form of parallelism and the underlying resource manager and scheduling mechanisms are inadequate to guarantee load balancing. Let us assume that we refer to a data center setting. In such a setting, the connection speed between the worker nodes and the computation capacity of the latter may differ and a good load balancing policy needs to take into consideration these factors to attain high throughput. More importantly, these factors can change at runtime, which calls for dynamic load balancing. This holds for all applications, but especially the streaming ones. Things start to become even more complicated when the tasks are stateful, e.g., as in windowed group-bys and joins. Dynamically balancing tasks corresponding to stateful computations is orthogonal to elasticity and involves repartitioning of keys that 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, e.g., [1]–[5]. Another scenario refers to operations, such as finding the nearest neighbors in a Euclidean or metric space. These computations, when transferred to a parallel setting, partition the space into overlapping regions, e.g., [6]. Partitioning the space in overlapping contiguous regions and assigning them to workers is not the only option, e.g., as discussed in [7], but naturally lends itself to both streaming and batch applications.

The first scenario, where stateful keys are repartitioned on the fly, has been extensively investigated in the last two decades, mainly in the context of adaptive query processing. Starting from the seminal work in Flux [3], where a specific operator in the query plan becomes responsible for dynamic load balancing and fault tolerance, several works have been proposed. Briefly, [2] considers computation, memory and communication costs and, to meet performance requirements, advocates consistent hashing and lossy counting-based techniques. The main novelty compared to Flux is that the number of parallel processors can change at runtime, which is equivalent to horizontal scaling. State migration overheads are also considered; to this end, three algorithms are presented with different trade-offs between imbalance and migration costs. The proposal in [1] also targets partitioning stateful operators taking into account both load balance and cost of aggregate groups of the same key partitioned to multiple workers. It improves upon Flux-like solutions, where each key is sent to a single worker, round-robin and solutions of Nasir et al [8] that send each key to a limited number of workers. The idea is to send only some large keys to multiple workers. Finally, [4], [5] employ control theoretical techniques to approach the same problem.

To the best of our knowledge, the above techniques have been incorporated to Flink or Spark or similar tools to a certain extent.[1]. However, the second scenario, where the boundaries of partially overlapped regions of the key space need to be updated at runtime to adapt to the evolving data distributions, has not been explored in a massively parallel (streaming) setting. In this work, we focus on the second scenario and more specifically, we aim to adapt the partitioning of contiguous partially overlapped regions in Flink. Our motivation use case is continuous distance-based outlier detection [9], which encapsulates range queries, transferred to a massively parallel setting [10]. As Flink execution plans need to be acyclic for streaming applications, the main challenge is (i) to devise a generic methodology to close the control

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code:T1EDK-01944)

[1]https://www.ververica.com/flink-forward-berlin/resources/elastic-streams-at-scale

feedback loop and allow for dynamic partitioning; and (ii) to propose specific dynamic partitioning techniques that better exploit the used resources (and being orthogonal to under- and over-provisioning elasticity issues). Our contribution is that we address both issues; the former in a generic manner and the latter in a manner suitable mostly for our use case.

The remainder of the paper is structured as follows. We give the background with regards to Flink and the use case in Sec. II. Sec. III and IV present the Flink architecture and the repartitioning techniques, respectively. Representative experiments are presented in Sec. V. The technical limitations and roadmap are discussed next, followed by the remainder of the related work in Sec. VII. We conclude in VIII.

II. Background

We split the background section into two parts, one for the necessary Flink descriptions, and one for the use case we target.

A. *Apache Flink System Model Basics*

In this part, we present some Flink details to render our paper self-contained; the interested reader is referred to [11] for full details. Flink execution plan is denoted by a directed graph, with vertices representing compute tasks and edges data subscriptions between such tasks. Flink supports both batch and streaming applications. To allow for iterative applications, e.g., as needed in many machine learning applications, there can be cycles in the graph but only in the batch mode. 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 elasticity actions need to be coupled with scheduling mechanisms, such as dynamic slot allocation ones, which are already implemented in Spark[2] and are under development in Flink[3].

In this context, our work faces the following two challenges. 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.

B. *The continuous distance-based outlier-detection use case*

Distance-based outlier detection requires (i) a distance function $dist$, which assigns a non-negative value to each non-ordered pair of points (very commonly, the euclidean distance is used); and (ii) two parameters R and k. The neighbors of each point is 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. In a streaming setting, the set of points continuously evolve, and the outliers are reported periodically or even after every point insertion or deletion; for this problem, several techniques have been proposed [9].

[2]https://spark.apache.org/docs/latest/job-scheduling.html

[3]https://cwiki.apache.org/confluence/display/FLINK/FLIP-56\%3A+Dynamic+Slot+Allocation

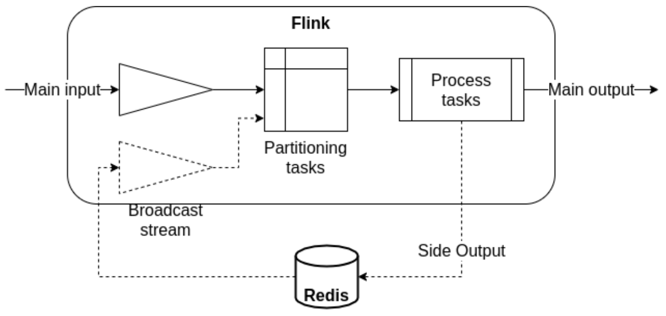

Fig. 1. Architecture overview

This use case involves range-queries, where the typical approach is to split the space into cells, i.e., contiguous regions [12], [13]. Each such region is assigned to a key, with the total number of keys being different from the number of workers in the generic case. Re-engineering some of the state-of- the-art continuous outlier detection algorithms in Flink has appeared in [10], while an extensible engine, called PROUD, is proposed in [14].[4]

In the PROUD framework, there are two separate types of Flink tasks for data partitioning and outlier detection, respectively. Their aggregate degree of parallelism should be equal to or larger than the number of workers, whereas the number of keys is configured by the user. 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 can be 60 secs, meaning that only the data from the last minute are kept and, $S = 3$ secs, meaning that the window contents are updated every 3 secs and a specific point is alive for 20 slides.

III. A Feedback Loop Architecture

As mentioned above, the main concern of the architecture is to create a feedback loop that can transfer information from the downstream Flink tasks to the upstream ones. Due to Flink's streaming architecture that allows only directed acyclic graphs, this loop has to be implemented outside the Flink engine; to this end, we employ an auxiliary data storage framework. The storage framework has two main requirements in order not to impede high throughput in real-time applications, namely fast read and writes.

[4]The implementation is publicly available from https://github.com/tatoliop/PROUD-PaRallel-OUtlier-Detection-for-streams

Algorithm 1 The controller function

```
procedure CONTROLLER
    for each new broadcast metadata do
        run assessment policy
        if adaptation is required then
            wait $\beta$ slides after the slide triggered the adaptation
            shift region boundaries by $\alpha R$
            enter transient period for $\frac{W}{S}$ slides
            discard any new metadata for $sleep$ slides
```

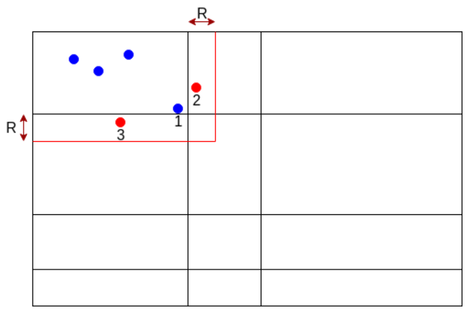

Fig. 2. Value-based grid partitioning of a 2d space in [10]

In our current implementation, we have chosen Redis[5], which is an open-source, main-memory database that is also used as a quick cache to read and write temporary data. It fills the gap in the architecture in which Flink jobs need to (i) write metadata based on data distributions and processing times and (ii) continuously read them to adapt the partitioning scheme. Redis need not be partitioned and/or replicated, despite the fact that this can be easily done, because its workload, as explained below, is not high.

The architecture is summarized in Fig. 1. In alignment to the PROUD framework introduced before, in our setting, Flink jobs comprise data source, partitioning, processing and data sink tasks. Their job is to read input data from a stream source, distribute them among the keys and process them in (count or time-based) sliding windows in order to finally output the results into a data sink. This flow is denoted by the solid arrows in the figure. The feedback loop is depicted with dashed lines and consists of the Redis external system and two additional Flink tasks. The first one is depicted as the dashed triangle and is a broadcasted input stream whilst the second one is a side output[6] that writes the metadata into Redis.

The broadcast stream is a task with degree of parallelism set to 1 that continuously reads data from Redis and connects with all the partitioning tasks. Partitioning tasks implement two functions. The first one is key assignment and is called whenever a data point from the main source arrives. The second one (hereby called *controller function*) is used whenever a data point from the broadcast stream is read and its job is to control the key distribution. Upon receipt of broadcast information, all partitioning tasks execute the same deterministic code to update the distribution policy.

More specifically, Redis is updated after each window slide, when all workers output metadata regarding the processing of the sliding that has just ended per key, e.g., number of data allocated to a specific region. This metadata are transferred back to the controller function, which applies an adaptation assessment policy. If an adaptation is decided, the region boundaries of the affected key are shifted by αR, where α is a small constant. 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.

Any adaptation results into two time periods in the controller function execution. The first is the *transient* one, during which any new data points arriving are distributed according to both the old and the new key depending on their values; details are in the next section. The transient period lasts for $\frac{W}{S}$ slides.The second period is the *stabilization* one that does not allow any new adaptation for the next $sleep$ slides.

Algorithm 1 summarizes the controller function. The assessment policy is extensible; e.g., a simple one used in the experiments is to trigger an adaptation if the number of data points in a region in the last slide is larger than ζ times the amount in a balanced execution. Also, in our experiments, setting $\beta = 2$ enforced synchronization, while $sleep$ duration is equal to the transient one. Obviously, more sophisticated policies, taking into account more metrics and spikes, can be plugged.

IV. Adaptive Repartitioning

In this section, two (re-)partitioning techniques are discussed. Both techniques distribute the data points based on their values into different keys. The first one, called *grid-based*, creates a grid with cells in the euclidean space with each cell representing a distinct key. The second one, called *tree-based*, uses a VP-tree [15] in order to create a binary tree in a metric space based on the $dist$ distance function. The tree is further processed in order to store only up to the necessary height with each leave representing one key. Both techniques require an initialization phase at the beginning; i.e., a sample of the dataset or prior knowledge is used to create the initial grid or VP-tree, respectively.

In the use case described in Sec. II-B, both partitioning techniques have two tasks. The first one is to send each incoming data point to the key it belongs to, where it will be processed and outputted as an outlier or not. The second one is to also send the same data point to neighbor keys within a range *R*; when a data point is replicated in this manner, it is annotated with a specific flag. All data points with this specific flag are processed by the outlier detection tasks but are not produced in the results. Their role is to allow the correct count of neighbor of points truly belonging to a key locally.

[5]https://redis.io/

[6]For more details about side outputs, please refer to https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/side_output.html

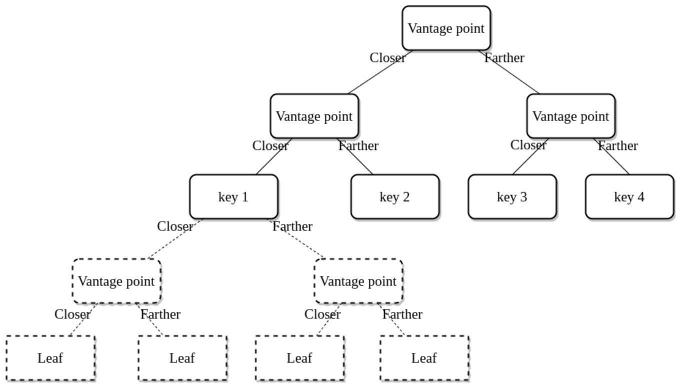

Fig. 3. Vantage Point tree

The *grid-based* partitioning technique splits the euclidean space into a number of cells and distributes each data point to one such cell and its neighbors if necessary based on its value. Fig. 2 shows an example grid. The top left cell holds all of the specified data points. The blue ones belong to the cell and are sent to the representative key without the flag. The red data points (numbers 2 and 3) belong to the neighbor cells but are within range R of the top left cell. This means that they will be also replicated with the flag and sent to the top left cell too. Similarly, point 1, will be replicated in a flagged mode to all the neighboring cells as well. The main advantage of the *grid-based* technique is that it can quickly assess where a data point belongs to by comparing its value to the cell boundaries. Also, the maximum factor of replicated data points that can be created is based on the dataset's dimensions d and is equal to 2^d. The disadvantage of the technique is that it can only be used in the euclidean space, and requires the length of each cell dimension to be at least R.

On the other hand, the *tree-based* partitioning technique can be used in any metric space. The technique starts by creating a VP-tree based on a sample from the dataset. 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. Essentially, every tree node has two children; the left one contains the data points that are within the threshold of the vantage point and the right one contains the rest of them. Finally the data points are saved in the tree's leaves leaving each node only with two variables, namely, the vantage point and the threshold. In our implementation, after the creation of the VP-tree, we save only the nodes up to a certain height along with their variables. In this way, the storage space decreases rapidly since no data points are saved and the traversal of the tree is faster due to the decreased height. The leaves of the stored tree represent distinct keys used in the partitioning tasks. When a new data point arrives, the technique starts by computing its distance to the tree's root vantage point. If the distance is within the root's threshold, then the new root becomes the left child whilst, if the distance is above the threshold, the new root is the right child. The process continues until it reaches one of the leaves; this leave is where the data point is assigned to. Meanwhile, the possible neighbor nodes are also traversed using the same process to find the neighbor keys. The *tree-based* technique has the advantage of supporting any metric space not limited to an euclidean one. The number of replicated data points that it produces is not bounded and depends on the tree initialization. Also depending on the height that is chosen, the traversal can vary in speed. Fig. 3 shows an example of such a tree with the solid lines representing the nodes chosen up to height 2.

Both techniques can adapt their keys easily by either changing the boundaries or equivalently the thresholds. We apply a cost function on the metadata to detect the most loaded key that exceeds the ζ threshold mentioned previously. In the *grid-based* technique, in order for less data points to be sent to the specified key, the only requirement is to move its boundaries inside. All cell borders are decreased by αR, as already discussed. This means that the specific region is decreased and all of its neighbors increase in space. The *tree-based* technique is a little more complicated, since it involves ancestors. In this case, a key can be either the left (closer) or the right (farther) child of a node. When a key needs to change depending on its status (closer or farther), the threshold of the parent needs to change too. In order for a closer child to receive less data, the threshold of the parent needs to be decreased. On the other hand, if the key in question is a farther child, then the threshold needs to be increased so that more data points are assigned to the closer child.

During the transient period, the keys in both techniques save both the new and old data structures (cells and nodes respectively) in order to produce correct results. The rationale behind the transient period is to fulfill the need to incorporate the change gradually so there is no loss of information or false results, given that the process tasks have already populated their states based on the previous keys. But, this needs to happen only for the regions that are affected. There are three cases:

1) If a data point is assigned to a key that is not affected by the change, the process continues without modifications.
2) If both before and after the change, the point belongs to the same key that is affected by the adaptation, 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 change, then it needs to be sent to all of them with reverse flagging for the transient period. A data point that during the transient period is sent to a key with reverse functionality means that if the flag exists it will be processed as if it did not exist and vice versa. This is necessary in order to correctly output the outliers.

V. Experiments

In this section, we provide concrete experimental results regarding the architecture's capability to handle different dataset distributions. Due to space constraints, the experiments are limited but adequate to provide strong insights into the

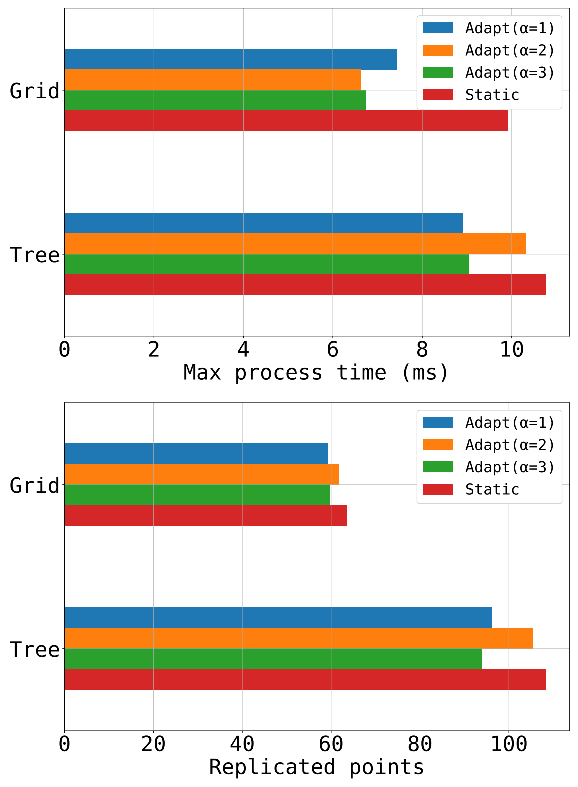

Fig. 4. Average maximum processing time per slide due to the longest running task (top) and average replicated data points per slide for the longest running task (bottom)

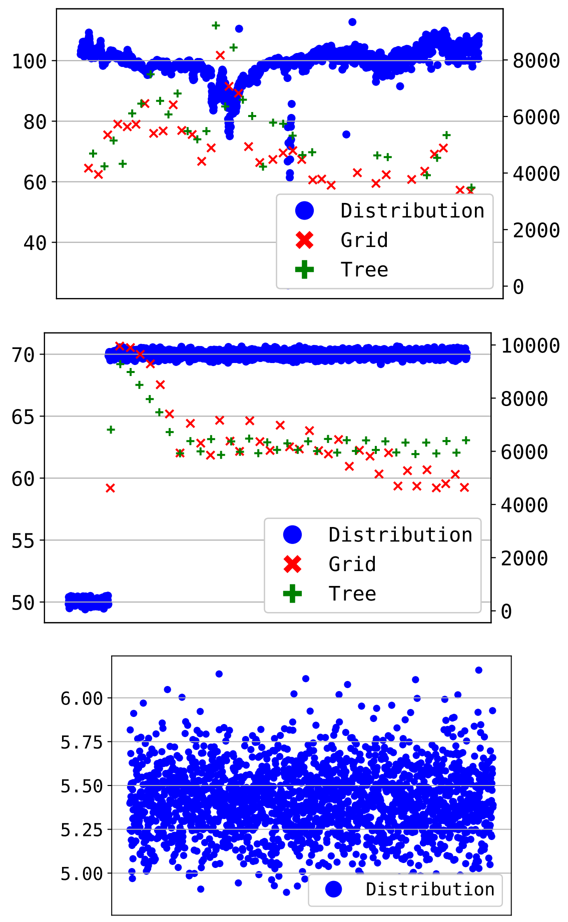

Fig. 5. Data distributions of Stock (top), Gauss2 (middle) and Gauss10 (bottom) and the respective adaptations. Left axis: dataset values. Right axis: number of points assigned to the most loaded key.

effectiveness and behavior of the approach. We first focus on comparing the processing time and number of replicated data points over Flink's tasks between the adaptive setting and a static setting using both *tree-* and *grid-based* techniques. We also check the impact of the α parameter, and finally we show the number of times that a single run of the algorithm adapts. Three different datasets are used: Stock[7], which is a real-world dataset, and 2 synthetic datasets with the first one, called Gauss(10), being synthesized using 10 different gaussian distributions concurrently, and the second one, called Gauss(2), being synthesized by 2 different gaussians one after the other. All datasets are used to create finite streams with approximately 1 million data points that are split into item-based sliding windows with size W =10000 and slide size $S = 500$. All experiments are repeated 5 times and we used a 6-core/12-thread Intel Xeon Processor E5-2620 v2 2.10GHz machine with 64GB RAM, with degree of parallelism for the Flink jobs set to 12. The key space is partitioned into 8 keys and ζ is set to 2, meaning that an adaptation is triggered if a key is assigned twice the balanced in a slide, i.e., $2\frac{10000}{8} = 2500$ data points.

Fig. 4 (top) refers to the Stock dataset and presents the average processing time per slide; the processing time per slide is defined by the longest running task. It compares the static setting against three configurations of the adaptive one for each partitioning method. We can observe that the adaptive case is more efficient even in that dataset (see its value distribution in Fig. 5 (top)). Overall, the *grid-based* technique manages to decrease the average running time per slide by 36% over the static setting. This in turn implies that the workload is getting balanced while the partitioning keys are changing and tasks with lower workload are utilized more with high-load tasks getting less data points. Fig. 4(bottom) presents the average number of replicated data points per slide for the longest running task. In general, the *grid-based* outperforms the *tree-based* one. E.g., the *tree-based* technique manages to decrease the overall execution time by 11% compared to 36% of the grid technique, due also to the higher number of replicated points. Finally, the impact of α is rather limited for the *grid-based* technique and more significant for the *tree-based* one.

The number of times a single run of the algorithm changes its partitioning boundaries depends on the dataset distribution and is shown in Fig. 5 for all datasets and both techniques ($\alpha = 1$). The Stock dataset (top plot), since it is a real world one and its distribution changes differently during its lifetime, needs less adaptations than the Gauss(2) dataset (middle plot), whose distribution changes once by a big margin. For the latter

[7]Available from https://wrds-web.wharton.upenn.edu/wrds

dataset, since our technique performs gradual adaptations, we see that these adaptations last for a long period. However, the figures also show that, at a point, a single key was allocated most of the workload, and there is a lot of space for improvements in terms of the speed of adaptation and the load balancing. Finally, in the Gauss(10) dataset (bottom plot), where there is no need for the algorithm to adapt, we see that correctly no re-partitioning takes place by our techniques.

VI. Limitations and Technical Roadmap

This work can be extended in several ways. Work in-progress includes three main directions: (i) Full incorporation into the PROUD engine in [14]. (ii) Development of additional forms of re-partitioning. Currently, the technique described is a specific form of a laze (gradual) adaptation. Several other flavors need to be investigated, including more eager ones and ones balancing multiple regions concurrently, while also ensuring result correctness. The same applies to the assessment policies, as already mentioned at the end of Sec. III. (iii) Thorough experimental investigation including additional settings, more (multi-dimensional) datasets and sensitivity analysis for all parameters not examined here, $\beta, \zeta, W, S, sleep$. Finally, the issue of Flink assignment of keys to tasks directly affects the performance and needs to be explored in depth.

VII. 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 [12], [13], which perform adaptive partitioning over a static dataset, with the adaptations being driven by the range query workload. These works do not employ partially overlapped regions either. An early technique for adaptive stream processing has appeared in [16], but this proposal does not fit well into the Flink framework, while it assumes a single node for partial result aggregation. The same limitation appears in [17].

[18] presents a Flink implementation of lazy partitioning tailored to distributed joins, which injects delays to account for transient network skew. An interesting approach appears in [7] that advocates examining random partitioning instead of partitioning in contiguous regions. Adaptive techniques in a massively parallel setting have also appeared in proposals, such as [19] and [20], whereas adaptive partitioning of data placement in distributed databases has been extensively investigated, e.g., [21]. All these are orthogonal aspects compared to our problem.

VIII. Summary

In this work, we investigated the problem of rendering Flink engine adaptive with a view to supporting a use case, in which the value-based partitioning of streaming data into keys needs to adapt to evolving data characteristics. We have proposed an architecture that implements a feedback loop in Flink, getting statistical metadata from tasks downstream and passing them in upstream tasks that control the key assignment. We have successfully used this technique for adaptive re-partitioning in continuous distance-based outlier detection, with the initial results being promising. We have also identified several directions to make this work more complete.

References

[1] 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.

[2] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB J.*, vol. 23, no. 4, pp. 517–539, 2014.

[3] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems." in *ICDE*, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, Eds., 2002, pp. 25–36.

[4] A. Gounaris, C. A. Yfoulis, and N. W. Paton, "Efficient load balancing in partitioned queries under random perturbations," *TAAS*, vol. 7, no. 1, pp. 5:1–5:27, 2012.

[5] C. A. Yfoulis and A. Gounaris, "Online load balancing in parallel database queries with model predictive control," in *Workshops Proc. of ICDE*, 2012, pp. 269–274.

[6] I. Cordova and T. Moh, "DBSCAN on resilient distributed datasets," in *2015 International Conference on High Performance Computing & Simulation, HPCS*, 2015, pp. 531–540.

[7] H. Song and J. Lee, "RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*, 2018, pp. 1173–1187.

[8] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *2016 IEEE 32nd Int. Conf. on Data Engineering (ICDE)*, 2016, pp. 589–600.

[9] L. Tran, L. Fan, and C. Shahabi, "Distance-based outlier detection in data streams," *PVLDB*, vol. 9, no. 12, pp. 1089–1100, 2016.

[10] T. Toliopoulos, A. Gounaris, K. Tsichlas, A. Papadopoulos, and S. Sampaio, "Parallel continuous outlier mining in streaming data," in *DSAA*. IEEE, 2018, pp. 227–236.

[11] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *PVLDB*, vol. 10, no. 12, pp. 1718–1729, 2017.

[12] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "AQWA: adaptive query-workload-aware partitioning of big spatial data," *PVLDB*, vol. 8, no. 13, pp. 2062–2073, 2015.

[13] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *PVLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.

[14] T. Toliopoulos, C. Bellas, A. Gounaris, and A. Papadopoulos, "PROUD: parallel outlier detection for streams," in *SIGMOD (demo track, to appear)*, 2020.

[15] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, no. 194, 1993, pp. 311–321.

[16] C. Balkesen and N. Tatbul, "Scalable data partitioning techniques for parallel sliding window processing over data streams," in *International Workshop on Data Management for Sensor Networks (DMSN)*, 2011.

[17] L. Su, W. Han, S. Yang, P. Zou, and Y. Jia, "Continuous adaptive outlier detection on distributed data streams," in *International Conference on High Performance Computing and Communications*, 2007, pp. 74–85.

[18] L. Rupprecht, W. Culhane, and P. R. Pietzuch, "Squirreljoin: Network-aware distributed join processing with lazy partitioning," *PVLDB*, vol. 10, no. 11, pp. 1250–1261, 2017.

[19] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*, 2019, pp. 705–722.

[20] N. Nikolaidis and A. Gounaris, "Adaptive filter ordering in spark," *CoRR*, vol. abs/1905.01349, 2019.

[21] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "Adaptdb: Adaptive partitioning for distributed joins," *PVLDB*, vol. 10, no. 5, pp. 589–600, 2017.