Parallel Continuous Outlier Mining in Streaming Data

Theodoros Toliopoulos*, Anastasios Gounaris*, Kostas Tsichlas*, Apostolos Papadopoulos* and Sandra Sampaio†
*Aristotle University of Thessaloniki, Greece, Email:{tatoliop,gounaria,tsichlas,papadopo}@csd.auth.gr
†The University of Manchester, UK, Email: s.sampaio@manchester.ac.uk

***Abstract*—In this work, we focus on *distance-based outliers* in a metric space, where the status of an entity as to whether it is an outlier is based on the number of other entities in its neighborhood. In the recent years, several solutions have tackled the problem of distance-based outliers in data streams, where outliers must be mined continuously as new elements become available. An interesting research problem is to combine the streaming environment with massively parallel systems to provide scalable stream-based algorithms. However, none of the previously proposed techniques refer to a massively parallel setting. Our proposal fills this gap and studies transferring state-of-the-art techniques in Apache Flink, a modern platform for intensive streaming analytics. We thoroughly present the technical challenges encountered and the alternatives that may be applied. We show speed-ups up to 117 (resp. 2076) times over a naive parallel (resp. non-parallel) solution in Flink, by using just an ordinary 4-core machine and a real-world dataset. Our results demonstrate that oulier mining can be achieved in an efficient and scalable manner. The resulting techniques have been made publicly available in open-source.**

***Index Terms*—anomaly detection, streams, Flink**

I. Introduction

Outlier analysis forms a key mechanism in modern data science and analytics [1], aiming to detect objects that, as defined in [2], appear to be inconsistent with the remainder of the objects in the same dataset. They are used in a variety of applications, such as fraud detection, spam detection and medical diagnosis, just to name a few. One of the most commonly used definitions for an outlier is the distance-based one [3], where an object is considered an outlier if it does not have more than k neighbors in a distance up to R. Continuous outlier detection in data streams deals with the problem of keeping an updated list of all outliers after each new object arrives and/or expires. Apparently, when data grows large, this becomes a challenging task, since applying even an one-pass algorithm to all active data is prohibitively expensive. To improve efficiency and scalability, the main target of this work is to propose massively parallel solutions for continuous outlier detection in data streams.

Briefly, the relevant state-of-the-art falls into two categories. The first category contains efficient non-parallel solutions for streaming outlier detection, e.g., [4]–[7]. The second category contains parallel solutions for outlier detection, where, to date, there is a single proposal that assumes modern distributed computing platforms, such as MapReduce [8]; nevertheless, this solution does not deal with the streaming case. The novelty of our proposal is that it proposes the first solution that combines both massive parallelism and continuous distance-based outlier detection in a streaming setting. Orthogonally, techniques are classified as either exact or approximate. We target exact solutions in this work.

Devising efficient parallel solutions for this problem involves addressing a series of important issues. First, outlier detection algorithms in data streams involve windows that cannot be partitioned into non-overlapping partitions, among which no communication is required. Second, low latency is of high significance in order to deliver results in a timely manner. Third, state information needs to be kept between window slides in order to avoid unnecessary recomputations. We provide solutions to the above issues through transferring key ideas in non-parallel techniques, such as [4], [7] to the Flink[1] platform.

This work aspires to become a reference point for all future work on streaming outlier detection in massively parallel settings. The contributions of this work are summarized as follows:

(i) We explore a series of implementation alternatives, differing in the algorithmic features they employ and in the way data is partitioned.
(ii) We provide thorough experimental evaluation results along with a comparison against the study in [9].
(iii) We offer the source code as an open-source library.[2]

In summary, we show that our best performing alternative, when tested on a real-world dataset, can yield speed-ups up to 117 (resp. 2076) times over a naive parallel (resp. non-parallel) solution in Flink using just a commodity 4-core machine. Similar performance is observed for synthetically generated datasets as well.

The remainder of this work is structured as follows. Section II contains background material on parallel streaming platforms and outlier detection algorithms. Section III introduces our first parallel solution, which is extended in Sections IV and V. Performance evaluation results are offered in Section VI. We close this work with a discussion of related work and issues pertaining extensions to our techniques in Sections VII and VIII, respectively.

[1]https://flink.apache.org/

[2]The code repository may be accessed at https://github.com/tatoliop/parallel-streaming-outlier-detection

II. Fundamental Concepts and a Baseline Solution

The purpose of this section is to make the paper as self-contained as possible. We split background material into two parts, referring to the main massively parallel platform alternatives for streaming data and the distance-based outlier algorithms that inspired our solutions, respectively.

A. Parallel Frameworks and Streaming Semantics

1) Parallel Frameworks for Streaming Applications: The choices examined include the three main parallel streaming platforms from Apache: (i) Storm[3], (ii) Spark[4] and (iii) Flink.

Apache Storm is the first widely used large scale stream processing framework. Storm is able to connect with a number of queuing systems, such as Kestrel, Kafka and Amazon Kinesis, and any database system. Storm provides low latency by using a record acknowledgment architecture, where each record that is processed by an operator sends back an acknowledgment to the previous operator. Its architecture is fault-tolerant providing *at-least-once* semantics. In case of failure, the data is re-processed, but this may result in duplicate production. To offer *exactly-once* semantics, there is a high level Storm API called Trident that employs micro-batches.

Spark Streaming also enables scalable, high-throughput and fault tolerant processing of data streams. It supports many data sources such as Kafka, Flume, Twitter and TCP sockets. The processed data can be written to filesystems and databases, such as HDFS (Hadoop Distributed File System) and Cassandra. Spark divides the input streams into micro-batches. Each of these batches goes through a processing step generating another micro-batch (result stream) until it has passed through all steps and the final result is written into a data sink. Spark Streaming is fault-tolerant supporting *exactly-once* semantics with a high-throughput by using the micro-batch architecture. This architecture, however, incurs higher latency than the continuous streaming approach employed by Storm and Flink, because of the delay caused by the micro-batches.

Apache Flink is another massively parallel platform for continuous stream processing providing low-latency, high-throughput and fault tolerance. Flink supports a number of data connectors including Kafka, Amazon Kinesis and Twitter along with data sinks such as HDFS, Cassandra and ElasticSearch. Flink is a framework that provides *exactly-once* semantics without resorting to micro-batches. Using a snapshot algorithm, it periodically generates state snapshots of a running stream topology, storing them in persistent storage, such as HDFS. In case of failure, Flink restores the latest snapshot from the storage and rewinds the stream source to the point where the last snapshot was taken. This approach combines the *exactly-once* semantics with low latency stream processing. Flink also provides a high level API, facilitating the partitioning of a stream into windows and the development of processing operators.

[3]https://storm.apache.org/

[4]https://spark.apache.org/

In summary, Storm provides low latency, similar to Flink, but it incorporates the *at-least-once* semantics, which allows duplicates to pass through the process in case of failures. Spark, like Storm Trident, provides *exactly-once* semantics with the use of micro-batches. The main drawback of micro-batches is the higher latency compared to the continuous processing model of Storm and Flink. Finally, Flink combines the advantages of Storm and Spark, namely low latency regarding the continuous record processing, coupled with the *exactly-once* semantics. In addition, Flink naturally supports both time- and count-based windows, which is not the case for Spark, since micro-batches essentially correspond to time-based sliding windows (see discussion below).

2) Streaming Semantics: A continuous stream is an infinite sequence of data points. Each data point o is annotated with its arrival time, $o.t$. The analysis of such infinite streaming data requires different techniques than those for finite datasets. A common approach is to adopt the notion of *window*, which refers to the most recent data items. Windows are typically small enough so that they can be stored in main memory, either of a single machine or of a parallel cluster. Windowing essentially splits the data stream into either overlapping (*sliding windows*) or non-overlapping (*tumbling windows*) finite sets of data points. Orthogonally, the splitting can be based either on the time of arrival of the data points (*time-based windows*) or on the number of data points (*count-based windows*). In the former case, the window size W is measured in time units, while in the latter case the size corresponds to the number of the most recent data items held. In time-based windows, W is defined by the minimum and maximum timestamps for data items in order to be included in the window, denoted as *W.start* and *W.end*, respectively. More specifically, W=*W.end*-*W.start*.

Figure 1 shows a stream discretized in 3 windows based on time. The windows are non-overlapping with $W = 2$ time units. In a time-based window, the amount of data in the window varies through time and the contents of consecutive windows are disjoint sets. Tumbling windows conceptually divide a stream into non-overlapping partitions.

In this work, we focus on sliding windows, which generalize the tumbling ones, and our techniques support both time- and count-based windows. Therefore, without any loss of generality, whenever we use the term window, we will be referring to a sliding time-based window. Figure 2 shows examples of such windows. In sliding windows, the magnitude of each slide is denoted as S. Every time the window moves by S, *W.start* and *W.end* are increased by S as well. For example, in Figure 1, S=2 time units, and in Figure 2, S=1 time unit. In each slide, some points may expire, i.e., they are dropped, while new points are included in the current window. Table I summarizes the notation used throughout the paper.

B. Problem Definition and Non-Parallel Solutions

The problem of continuous distance-based outlier detection is defined formally as follows.

Definition 1: Given a set of objects $\mathbb{O}$ and the threshold parameters R and k, for each window slide S, report all the

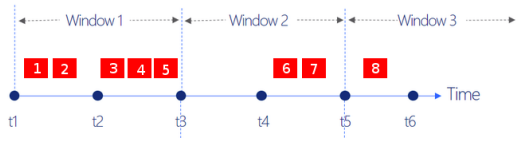

Fig. 1. Tumbling Windows

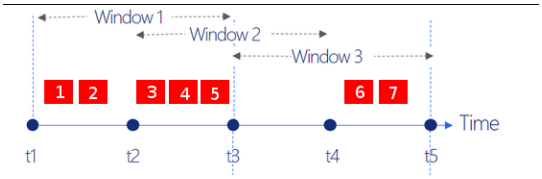

Fig. 2. Sliding Windows

objects o_i 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.

The main challenges stem from the fact that all active objects need to be continuously assessed during their lifetime, since an object may change its status as many times as the number of slides during which it remains within the window.

We exclusively focus on exact solutions. There are several exact algorithms for continuous outlier detection in data streams, such as exact-Storm [7], Abstract-C [5], LUE [4], [10], DUE [4], [10], COD [4], [10], MCOD [4], [10] and Thresh_LEAP [6][5]. All these algorithms assume a centralized environment. In a recent impartial comparison presented in [9], these algorithms were compared in terms of their memory consumption and CPU time. The comparisons were made using four datasets with varying dimensionality and settings of window length *W*, window slide *S*, number of neighbors *k* and neighbor range *R*. This study showed that MCOD is superior across multiple datasets in most stream settings. In addition, Thresh_LEAP and MCOD displayed the lowest memory consumption and CPU times, while exact-Storm, Abstract-C and DUE are the slowest and most memory consuming algorithms. Based on these findings, MCOD has served as our main inspiration of our solution to the problem of parallelization of continuous distance-based outlier detection algorithms. However, we start with a simpler and easier to parallelize algorithm, namely exact-Storm, which contains several key elements in common with MCOD, such as index structures for range queries and safe inliers, and so represents a preliminary step towards our proposed solution. The key details of these two algorithms are discussed in the following.

1) Exact-Storm: A key operation in distance-based outlier detection is the distance computation between objects and the need to continuously examine the neighborhood of each object. To avoid a quadratic number of comparisons in each slide, appropriate indices are required. To this end, exact-Storm uses a data structure called ISB to store the data points in nodes.

[5]Reference implementations are provided in the MOA tool [11] http://moa.cs.waikato.ac.nz/ and https://infolab.usc.edu/Luan/Outlier/

TABLE I
FREQUENTLY USED SYMBOLS AND INTERPRETATION

Symbol	Short description
W	The size of the stream window
S	The slide of the stream
$W.start$	The starting timestamp of the window
$W.end$	The ending timestamp of the window
$\mathbb{O}$	The set of data objects (or points) in the stream
$o_i \in \mathbb{O}$	The i^{th} data object in the stream
$o.id$	The object identifier o (either i for o_i or any other identifier)
$o.value$	The value of o
$o.t$	The arrival time of o
$o.count\_after$	The number of succeeding neighbors of o
$o.nn\_before$	A list with the arrival time of the preceding neighbors of o
$o.nn$	The count of neighbors of o
$\mathbb{PO}$	A list containing data points that are potentially outliers
R	The distance threshold in the outlier definition
k	The neighbor count threshold in the outlier definition
$dist(o_i, o_j)$	The distance function between objects o_i and o_j
P	set of Flink partitions, each handled by a separate Flink node

A node is a record containing the data point *o*, the arrival time of the point *o.t*, the number of succeeding neighbors *o.count_after* and a list *o.nn_before* of size *k* containing the arrival time of the preceding neighbors of *o* (i.e., each node contains a different data stream object along with some metadata). This data structure is a pivot-based index that provides support for fast range query search in any metric space. The range query, given a data point *o* and the range *R*, returns the nodes in the ISB whose distance to *o* is less than or equal to *R*.

A sketch of the algorithm steps in each slide is as follows. For each new data point *o*, a node is created as described above. Then, a range query is issued on the ISB structure to find the neighbors *o'* of the new node. The result of the range query is used to initialize the values of the new node's *o.count_after* and *o.nn_before*. If the size of *o.nn_before* is more than *k* then the oldest timestamps are removed. For each *o'*, the value of *o'.count_after* is increased by 1. Finally the new node is inserted into the ISB. When a data point *o* expires, meaning that *o.t* is lower than the window's starting timestamp $W.start$, it is removed from the ISB. Its timestamp, however, is not removed from other nodes' list of preceding neighbors to mitigate overheads.

The above steps are applied in each slide. After they have been completed, ISB is scanned for outliers. If a node's sum of *count_after* and the size of *nn_before*, whose timestamps is within the window borders, is lower than *k*, then the node is an outlier. An optimization that avoids checking all objects in each slide is through the notion of *safe inliers*: if a node's *count_after* is more than *k*, then this node is a safe inlier and does not need to be checked again in future scans as it is guaranteed to have at least k neighbors in the remainder of its lifetime.

In our parallel solution, we adopt both a structure for fast range queries (using however an M-tree, not ISB) and the

notion of safe inliers.

2) MCOD: The main motivation behind MCOD is that range queries are better than brute-force (all-pairs) distance computations but are still expensive. MCOD (standing for Micro-cluster-based Continuous Outlier Detection) aims to mitigate the need for range queries. The algorithm drastically reduces the number of data points that need to be addressed during a range query through creating micro-clusters and assigning data points to them. A micro-cluster has at least *k + 1* data points all of which are neighbors to each other. Its center can be a data point or just a point in the metric space and has a radius of $\frac{R}{2}$, implying that the maximum distance between any two objects in the micro-cluster is at most R. Each data point in any micro-cluster is an inlier and does not need to be checked in outlier queries. However, a data point that does not belong to a micro-cluster can be either an inlier or an outlier. Such objects are stored in a list $\mathbb{PO} \subseteq \mathbb{O}$.

On average, MCOD stores less metadata for each object than exact-Storm. More specifically, for each o in a micro-cluster, it stores the identifier of its cluster. For each o in $\mathbb{PO}$, it stores the *o.count_after* and the expiration time of the k most recent preceding neighbors. MCOD also uses an event queue to store unsafe inliers that are not in any cluster. This event queue is a specific priority queue that keeps the time point at which a non-safe inlier should be re-checked.

A sketch of the algorithm steps is as follows. For each new data point o, if o is within *R / 2* range of a micro-cluster, it is added to it; if there are multiple such micro-clusters, the closest one is picked. Otherwise, if it has at least k neighbors in $\mathbb{PO}$ within a distance of $\frac{R}{2}$, it becomes the center of a new micro-cluster. If none of the above conditions are met, o is added to $\mathbb{PO}$ and possibly to the event queue, if it is not an outlier. At each slide, all the previous non-expired outliers are checked along with the inliers for which the check time has arrived (with the help of the event queue). When a data point o expires, it is removed from the micro-cluster or $\mathbb{PO}$ and the event queue updates the unsafe inliers. If o is removed from a micro-cluster and the points remaining in that micro-cluster are less than *k + 1*, then the cluster is destroyed and each data point of the cluster is processed as a new data point, without however updating their neighbors.

In our final parallel solution, we also adopt the notion of micro-clusters.

III. Simple Solutions

The aim of this work is to build upon the state-of-the-art non-parallel techniques and manage to parallelize the window and the associated workload efficiently. First, to explain the main engineering approach and to be capable of assessing the efficiency of the implementation of the parallel streaming solutions for distance-based outlier detection proposed in this paper, we introduce a baseline approach, which broadly corresponds to a single-partition implementation of exact-Storm in Flink. By single-partition, we mean that the logical window is physically allocated to a single Flink node as a whole; the norm in Flink is windows to be physically partitioned across multiple Flink nodes. Then, we proceed to its parallelization, where we are forced to employ the notion of a *meta-window*.

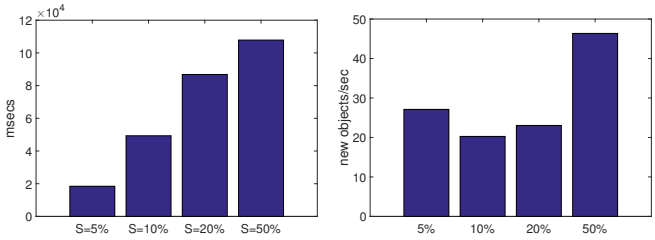

Fig. 3. Average processing time per slide (left) and input stream consumption rate (right) for the baseline approach.

A. A baseline approach in Flink

We use two modules: (i) a *stream handler*; and (ii) a *window processor*. The stream handler applies a `map` function on each stream object and sends it to the window. The window processor runs an outlier detection algorithm in each slide.

In the `map` function of the stream handler, each data point is initialized with a null *o.count_after* count and an empty list *o.nn_before*. These records are sent to the single-partitioned window. The window has its own state in which it stores the records, which is persistent across slides. In other words, changes made to a data point's metadata in a slide are kept throughout the data point's lifetime. Overall, the contents of a window at any point in time contain all the active points along with their metadata, i.e., *o.id*, *o.value*, *o.t*, *o.count_after* and *o.nn_before*.

The outlier detection algorithm contains two steps. The first step is the update of each data point's metadata. In particular, the algorithm checks only the new arrivals. For every such data point, o, it finds the neighbors, *o'* in range R, checking all the points in the window; the range is according to the euclidean distance by default but any type of metric distance can be employed. If *o'* is in the same slide as o, then *o.count_after* is increased; otherwise the timestamp *o'.t* is added to the list *o.nn_before*. For each *o'* the value of *o'.count_after* is increased by 1. The rest of the metadata of the older data points, i.e. the *o'.nn_before* values, have already been computed when these objects were inserted in the window, and do not need to be recalculated. The second step of the algorithm is to detect the outliers. For each data point in the current window, the algorithm computes the total number of neighbors. This is done by summing the *o.count_after* and the size of the list *o.nn_before* taking into account only those values t where $t \geq W.start$, as explained previously.

Both time-based and count-based windows are naturally supported in Flink. In this work, we mainly focus on time-based windows, but it is worth pointing out that, even without explicit support of count-based windows, it is straightforward to emulate them through artificially tweaking the initial timestamps, so that a fixed number of objects arrive and expire in each slide, and thus the amount of alive objects remains stable during stream processing.

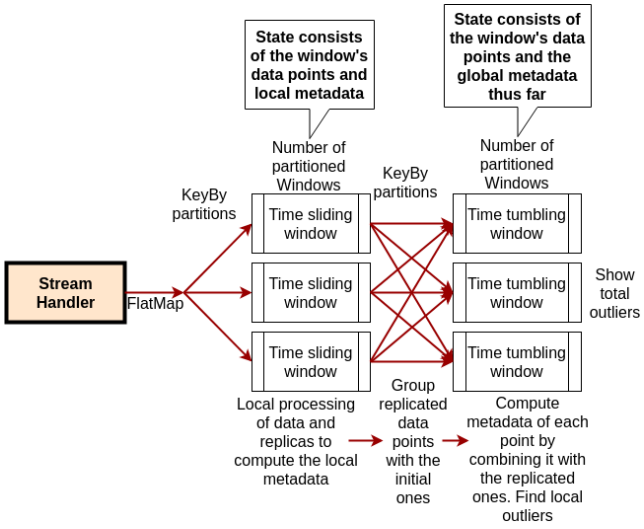

Fig. 4. Main implementation rationale with random partitioning, where the parallelized sliding window is followed by a partitioned tumbling meta-window.

To gain insights into the performance of the solution, we evaluate this baseline approach using the Stock dataset from [9]. It is a one-dimensional dataset with 1,048,575 data points[6]. Each data point, o, is assigned a unique identifier, *o.id*, and has a numeric value of type `Double`, *o.value*. We employ a machine with an Intel i7-3770K CPU at 3.5GHz, which possesses 4 cores (8 threads) and 32GB of RAM. Figure 3(left) shows the average processing time for each slide step for four values of slide magnitude S with $W = 10K$; S is given as a percentage of W. Figure 3(right) shows the corresponding input consumption rate, which is equal to the maximum throughput of stream data the baseline approach can support; in these settings, this throughput can reach 50 objects/sec. We can see that in general, the average processing time per new arrival increases for either small slides, where a few new points arrive, or relatively large ones, where most of the window contents are replaced.

B. A Naive Solution

The parallelization of the baseline approach in Section III-A yields a naive parallel solution, where the window is split into a set of multiple partitions, P. It is termed naive in the sense that it does not benefit from data structures to speed-up range queries. Notwithstanding the simplicity, the parallelization technique needs to efficiently address the challenges of (i) collaboration between physical window partitions to establish whether a point at a specific time is an outlier or not through aggregating local statistics; and (ii) keeping the window state across slides, where the state includes the object metadata.

The engineering solutions devised need to respect the principle that, in each window slide, each new object is processed only once. The latter does not allow first to compute the local aggregates for a given point and then to compute the global

[6]available from https://wrds-web.wharton.upenn.edu/wrds

Algorithm 1 Naive solution

```
procedure StreamHandler
    for each new object o do
        initialize record
        if there is no timestamp then
            add artificial timestamp
        o.partition ← o.id mod |P|
        o.flag ← 1
        for each partition p ∈ P do
            if (p == o.partition) then
                o.flag ← 0
                send o to p
                o.flag ← 1
            else
                send o to p

procedure SlidingWindowPartitionProcessor
    for each slide do
        evict expired objects and old objects with o.flag==1
        compute pairwise distances involving new objects
        update o.count_after and o.nn_before metadata
        for each object o ∈ ℙ𝕆 do
            if (o.count_after ≥ k) ) then
                ℙ𝕆 ← ℙ𝕆 \ o
        group objects in ℙ𝕆 by o.partition
        and send to a (new) tumbling window

procedure TumblingWindowPartitionProcessor
    for each slide do
        aggregate o.count_after and o.nn_before metadata
        for each object o do
            o.nn_prec ← prune o.nn_before
            if (o.count_after + |o.nn_prec| < k) ) then
                report o as an outlier
```

aggregates into the same window. Therefore, the key idea is to split the window processor into two parts, the sliding window processor and the tumbling window processor, as shown in Figure 4. The former holds the active points in its partition allowing some temporary replication, as discussed in the following, while the latter keeps the final metadata in each slide. These metadata also include the information about the outliers. Since they evolve in each slide for both the new and the old points, the window state is fully updated, and thus the tumbling window semantics apply. Essentially, the second window serves as a *meta-window*.

The implementation details are as follows. First, we extend the object record with two new fields, namely $o.flag$ and $o.partition$. The former is a binary variable, where 0 means that the object should be kept to the assigned window during its lifetime, and 1 means that the object is redundant, and should be evicted in the next slide, regardless of the $W.start$ value. The stream handler applies a `flatMap` that, apart from initializing an object's extended record, computes its $o.partition$ based on the $o.id$. Then it dispatches the point to *all* partitions, setting the $o.flag$ to 1 to all partitions different to $o.partition$. In other words, new objects are replicated across all the partitions.

The sliding window processor is responsible for comparing

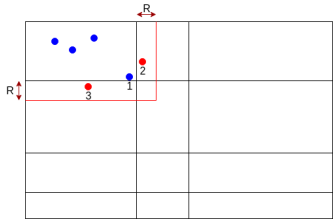

Fig. 5. Value-based partitioning

the distances between a) the new objects in the slide and b) the new objects and all its previous contents, the timestamp of which comes after $W.start$. This leads to updating the $o.count\_after$ metadata for all objects and the $o.nn\_before$ metadata for the new ones. However, the metadata for the new objects are local aggregates spread across all partitions. To produce the global aggregates, the updated objects are partitioned again according to $o.partition$ in a new tumbling window, but without replication this time. However, to save communication cost, not all objects are shuffled, but only these that are not safe inliers. The set of the safe inliers is the complement of potential outliers, $\mathbb{O} \setminus \mathbb{PO}$. In each slide, the sliding window processor, first creates the $\mathbb{PO}$ set, through checking whether $o.count\_after$ is less than k or not. The tumbling window aggregates the local aggregates of non-safe inliers, and thus derives the exact metadata needed to establish outlierness. As in [7], the list in $o.nn\_before$ may contain preceding neighbors that have expired; thus a filter is required to establish the alive ones, termed $o.nn\_prec$.

Algorithm 1 summarizes the naive approach. In summary, the naive solution requires minimal effort from the stream handler at the expense of high communication and computation cost, since new objects are sent to each partition and compared against their full contents. In terms of implementation, it requires the notion of *meta-windows* to achieve accurate results.

IV. Advanced Solution

The advanced solution extends the naive one in two complementary and orthogonal dimensions, namely through: (i) employing data structures to support fast range queries; and (ii) through performing value-based partitioning, which eliminates the need to employ a meta-window.

More specifically, the first extension involves a more advanced approach to holding state in the sliding window. Such state is stored in a M-tree [12], to which range queries are submitted. M-trees are part of each partition state, therefore each Flink partition has its own local tree. Compared to Algorithm 1, the key difference is in the sliding window processor.

The second extension is more intrusive. A limitation of the solutions thus far is that they replicate each new data point to all partitions. This is inevitable, given that the stream handler assigns points to partitions randomly and thus a new object may have neighbors in all partitions. Value-based partitioning addresses this limitation without sacrificing the accuracy of the results. Also, as will be explained below, it eliminates the need to exchange information between partitions during a slide.

We may use the M-tree also to perform value-based partitioning. However, in this work, we make an additional assumption that the space is Euclidean and therefore, it can be partitioned into grid cells. Further, we assume that there are some sample data available before execution. Based on these data, we can extract min, max and quantile information about the value distribution in each dimension, in order to construct the grid cell appropriately.

The rationale of the approach for a two-dimensional space is illustrated in Figure 5. Each cell is assigned to a single Flink node. However, an object in a cell may have neighbors in other cells as well. The key difference is that these neighbors belong to adjacent cells only; therefore the number of Flink nodes that need to be aware of the arrival of each new object is limited. More specifically, the borders of each cell are extended by a buffer zone of width equal to R. The stream handler sends a new data point (i) to the partition corresponding to its cell with flag $o.flag$ set to 0 and (ii) to all the partitions, the buffer zone of which includes the new data point with flag $o.flag$ set to 1; these partitions form the set AP in Algorithm 2. Assuming that R is much smaller than the size of a grid cell side, each data object is replicated at most 4 times if the data is 2-dimensional; this is because, when it falls near to a cell corner, it may fall into the buffer zone of three other adjacent cells. In the example in the figure, the buffer zone borders for the upper-left cell are depicted; points 1,2 and 3 are sent to the Flink node responsible for the upper-left cell along with all the other three points. For d-dimensional data, a data point can be sent to up to 2^d partitions, which grows exponentially in the number of dimensions but, by construction, is at most equal to $|P|$; i.e., the replication is never inferior to the naive solution.

According to the partitioning above, each partition has all the necessary information in order to establish object outlierness locally. Therefore, a single sliding window partition processor is required, which incorporates the responsibility of the tumbling window partition processor in the previous approaches. Algorithm 2 summarizes the advanced solution with value-based partitioning (referred to as *advanced(VP)* in the experiments). Apart from making the tumbling window processing phase obsolete, another difference to the simple advanced algorithm is that objects with $o.flag = 1$ are not dropped until they expire.

We can also extend the partitioning to metric spaces by employing the M-tree as mentioned above. In particular, instead of assuming a grid partitioning, which presupposes Euclidean space, we can partition based on a rather shallow level of the M-tree as follows. First, fix a particular level ℓ of the M-tree. Then, each Flink node is assigned to a particular node (or more) at level ℓ. Level ℓ depends on the choice of R. However, the overlap between nodes at level ℓ depends on the distribution of the elements. The investigation of such an approach is left for future work. Potentially dynamic load balancing is also left for future work; as shown in Figure 5, the

Algorithm 2 Advanced solution with value-based partitioning

```
procedure StreamHandler
    for each new object o do
        initialize record
        if there is no timestamp then
            add artificial timestamp
        o.partition ← findGridCell (o.value)
        o.flag ← 0
        send o to o.partition
        AO ← findRelevantAdjacentPartitions (o.value)
        o.flag ← 1
        for each partition p ∈ AP do
            send o to p

procedure SlidingWindowPartitionProcessor
    for each slide do
        evict expired objects from M-tree
        insert new objects in M-tree
        compute distances
        update o.count_after and o.nn_before metadata
        for each object o ∈ ℙ𝕆 do
            if (o.count_after ≥ k) ) then
                ℙ𝕆 ← ℙ𝕆 \ o
            else
                o.nn_prec ← prune o.nn_before
                if (o.count_after + |o.nn_prec| < k) ) then
                    report o as an outlier
```

Algorithm 3 pMCOD

```
procedure StreamHandler
    Same as Algorithm 2

procedure SlidingWindowPartitionProcessor
    for each slide do
        evict expired objects from M-tree
        if a micro-cluster dissolves then
            treat all points as new w/o updating their neighbors
        insert new objects in M-tree
        for each new object o' with o'.flag = 0 do
            if o' belongs to micro-cluster then
                for each object o'' ∈ ℙ𝕆 do
                    update o'' metadata due to o'
            else
                ℙ𝕆 ← ℙ𝕆 ∪ o'
                compute pairwise distances involving o'
                check if a new micro-cluster can be formed
        for each object o ∈ ℙ𝕆 do
            if (o.count_after ≥ k) ) then
                ℙ𝕆 ← ℙ𝕆 \ o
            else
                o.nn_prec ← prune o.nn_before
                if (o.count_after + |o.nn_prec| < k) ) then
                    report o as an outlier
```

grid cells are not necessarily of the same size. Here, we define cell boundaries statically, taking into consideration only the value distribution per dimension thus overlooking issues, such as actual computation and communication cost per partition.

Overall, the advanced solution trades i) additional workload on the stream handler and ii) increased memory requirements on the Flink nodes for i) less communication cost between both the stream handler and the Flink nodes, and the nodes themselves, and ii) less computation cost per node. The implementation challenges are mostly related to how the value-based partitioner on the stream handler is efficiently and effectively constructed.

V. Employing Micro-clustering

The motivation behind using micro-clusters is to drastically reduce the number of range queries submitted to the M-tree, as explained in Section II-B2. *pMCOD* extends the previous algorithm with the notion of micro-clusters, as shown in Algorithm 3. In contrast to the work in [4], this version does not contain an event queue. The sliding window's state consists of the micro-clusters, the potential outliers $\mathbb{PO}$ and the M-tree. The notion of value-partitioning from section IV along with the introduction of micro-clusters means that each partition is able to fully report its outliers without the need to communicate with the other partitions and at a faster rate.

Each window slide starts with the eviction of the expired data points from the state and the dissolution of the micro-clusters with $\leq k$ elements. Each data point that belonged to a dissolved micro-cluster is treated as a new data point. For each new data point, the algorithm computes its distance to the micro-clusters and if it belongs to any of them, it proceeds to updating the $\mathbb{PO}$ metadata only. If the data point does not belong to any micro-cluster, it is inserted into the $\mathbb{PO}$ set and a range query is executed to find its neighbors. Based on the number of neighbors of a data point in $\mathbb{PO}$, a new micro-cluster may be created.

Broadly, the set of potential outliers includes only points that do not belong to a micro-cluster. Also, if a point belongs to a micro-cluster, only the metadata of points in $\mathbb{PO}$ need to be updated. After the update of each data point's metadata, the algorithm reports the outliers by checking the data points in $\mathbb{PO}$. The list $o.nn\_before$ is pruned to contain only the non-expired objects. Each data point that has $o.count\_after + |o.nn\_before| < k$ is reported as an outlier for the corresponding slide.

VI. Performance Evaluation

The experimental setting is as follows. We focus on presenting the performance as a function of the window size, the slide size, the amount of outliers and the degree of parallelism. The accuracy is always 100%, since all techniques are exact. We have employed three real and one artificial dataset. These datasets are static and finite, but are adequate to emulate a streaming setting. Unless stated otherwise, the times presented correspond to the average time per slide, aggregated over 200 slides overall. Note that, given that we are in a streaming environment, the actual full dataset size does not matter.

Initially, we focus on the Stock real-world dataset using the same machine as the one described in Section III-A. Each experiment is repeated 5 times. R and k are set to 0.45 and 50, respectively, yielding 1.02% of outliers (i.e., the setting is similar to [9]). The default degree of parallelism, i.e., the number

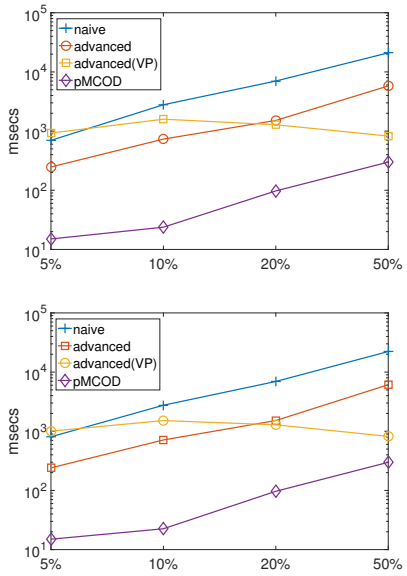

Fig. 6. Average (top) and median (bottom) processing time per slide for different slides and window of 10K objects.

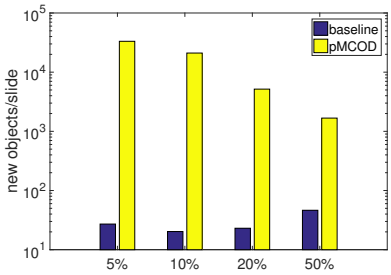

Fig. 7. Throughput comparison for $W = 10K$ and slides of 5%-50%.

of Flink partitions of the window is 16, and each Flink node runs on a single core. Stock is an one-dimensional dataset. To allow a fair comparison, the timestamps are assigned in such a way that all windows are of the same size, and the slide is given as a percentage of W, e.g., a slide of 5% means that the 5% of the window contents are new arrivals.

A. Main Results

In the first experiment, we employ a window of 10K objects, while the slide varies from 5% to 50%. The results are shown in Figure 6, where both the average and the median times are reported. From the figure, we can draw the following observations:

1) *pMCOD* improves upon the *naive* solution by two orders of magnitude; for example, it is 117X faster for slide 10%.
2) *pMCOD* improves upon the advanced solutions with both random and value-based partitioning (labeled as *advanced* and *advanced(VP)*, respectively) by an order of magnitude for slides up to 20%; e.g., for slide of 10%, it is 10X faster, and for 20%, it is 13X.
3) For slides of 50%, where half of the window points are new in each slide, *pMCOD* is faster than *advanced(VP)* by 2.74 times.
4) *Advanced* dominates *advanced(VP)* for small slides, while the latter is better for large ones, which is mostly attributed to the fact that *advanced(VP)* inherent load imbalance[7] is outweighed by the benefits of less replication and communication in large slides.
5) Despite the non-negligible standard deviation, the trends in the average values are similar to those in the median ones.

Figure 7 compares *pMCOD* against the throughput results obtained by the baseline technique, described in Section III-A. The improvements are up to 2076 times, whereas *pMCOD*'s throughput exceeds 33240 new objects per second for $S = 5\%$.

In the second experiment, we focus on *pMCOD* and we examine the impact of three parameters, namely the degree of parallelism P, k and the window size W. The results are summarized in Figure 8. Regarding the degree of parallelism, the left figure refers to a setting where W is 10K and the slide is 5%. We see that *pMCOD* scales well and the time per slide drops nearly two times when we go from two partitions to four. Since the machine is a 4-core/8-thread one, the gains are small for higher degrees of parallelism. We also see that our default configuration of $P = 16$ yields the highest performance. In the middle figure, we see that, as we increase the k value, the performance degrades. Increasing the k value implies more outliers and higher difficulty in forming micro-clusters; as such, this result is reasonable. Finally, the rightmost figure reveals that *pMCOD* scales well with the size of the window: a ten-fold increase in the window size results in similar increases in the processing time for slides of 5% and 10%, and smaller increases for larger slides. More specifically, for 20% slide magnitude, the increase in the processing time in the 100K window is less than 8 times, and for 50% slide, it is 6.62 times only.

B. Using More Datasets

First, we provide results using an artifical dataset, which is generated from a mixture of three Gaussian distributions and taken from [9]. We set $W = 10K$, $R = 0.28$ and $k = 50$. Figure 9 presents the results, where it is shown that the main observations drawn for the Stock real-world dataset hold.

In the next experiment, we employ two additional real-world datasets, namely Forest Cover (FC)[8] and TAO[9], considering 2 and 3 dimensions, respectively. We configure their parameters so that always k is 50, while W is kept to 10K. More specifically, for FC, we set $R = 34$ on the 2nd and the 5th dimensions (corresponding to 1.3% outliers - we discard the other 53 dimensions), and for TAO, we set $R = 1.9$ on all three dimensions (corresponding to 0.98% outliers).

[7]The grid cell used for partitioning is based on an initial sample and thus no guarantees can be provided as to how balanced the workload distribution is throughout stream processing.

[8]Available from http://kdd.ics.uci.edu

[9]Available from http://www.pmel.noaa.gov

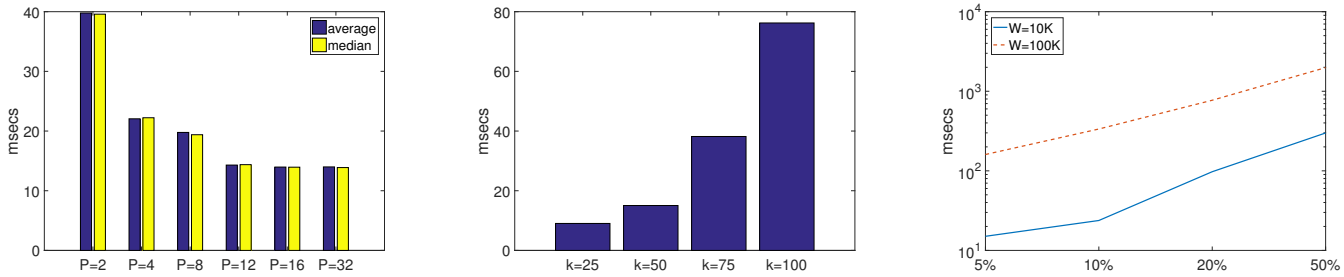

Fig. 8. The impact of the degree of parallelism P (left), k (middle) and the window size W (right - for slide sizes of 5%-50%) on pMCOD's performance.

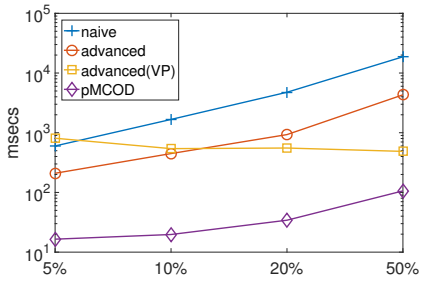

Fig. 9. Average slide processing time for slide sizes of 5%-50% for the artifical dataset

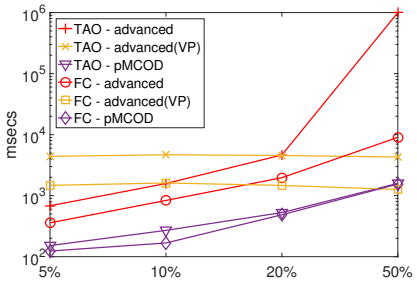

Fig. 10. Average slide processing time for slide sizes of 5%-50% for the TAO and FC datasets

The results are shown in Figure 10 (actually, for TAO and S=50%, *advanced* crashed due to memory shortage). The key observations are as follows: (i) *pMCOD* behavior is nearly the same for 2 or 3 dimensions but significantly worse than the behavior for the one-dimensional datasets; the latter is attributed to the known performance degradation of M-tree for higher dimensions and the increased replication; (ii) for the FC dataset, and due to the increased number of outliers compared to the other settings, *advanced(VP)* slightly outperforms *pMCOD* when $S = 50\%$, which means that half of the points in each window slide are new arrivals; and (iii) *advanced* and *advanced(VP)* are significantly affected by the increase in the number of dimensions considered from 2 to 3.

C. Comparison against results in [9]

Finally, we compare our results regarding the Stock dataset against those of non-parallel MCOD [4], as evaluated by third parties in [9]. The evaluation in [9] also uses a processor with clock speed at 3.5GHz, but without giving details about the number of processors. Nevertheless, our results can directly compare against those in Figures 6 and 10 in [9]. Due to the log scale used, we can only report approximate values, as shown in Table II. The speedup is between 2.66X and 6.65X, which provides strong insights into the parallelization efficiency of our solutions on a 4-core machine.

TABLE II
AVERAGE TIME TO PROCESS A SLIDE IN DIFFERENT WORKS.

W	S	MCOD in [9] (msecs)	pMCOD (msecs)
10K	5%	100 (approx.)	15.042
100K	5%	700 (approx.)	160.44
100K	10%	1420 (approx.)	335.5
100K	20%	2650 (approx.)	771.15
100K	50%	5270 (approx.)	1981.25

VII. Related Work

Outlier detection has been a topic that has attracted a lot of interest and there are several comprehensive surveys, e.g., [1], [13]–[15]. In Section II we have already discussed algorithms for outlier detection in streams. The next most related area to our work is parallel algorithms for distance-based outlier detection. A distributed outlier detection algorithm for massive datasets is proposed in [8]. The two key points of this research is the initial partitioning of the data and the different outlier detection algorithm that each partition may run. The partitioning resembles our value-based one and focuses on the workload that each partition receives. Then, in each partition, the exact outlier detection algorithm out of two candidates is chosen. However, none of these algorithms is suitable in a streaming setting. There are also some works that assume parallel infrastructures that cannot scale and do not follow the paradigm introduced by MapReduce and its modern extensions, e.g., [16], [17]. Overall, our work is the first one that combines streaming and massively parallel solutions to the problem of distance-based outlier detection. However, parallel and streaming anomaly detection has been considered for other definitions of outlierness, e.g., as in [18].

A related yet different problem is examined in [19]. In a production distributed environment, a stream of data points may be split across multiple nodes, each holding part of the

values of a data point. These parts will eventually need to be aggregated on a core node for outlier detection, but this incurs significant communication cost. The solution proposed is based on compressing local data into a sketch. Another related problem is that of supporting multiple outlier detection queries, i.e., combinations of R and k values. Examples include [20] and [4]. The latter presents multi-query extensions to MCOD and its approach is compatible with the our parallel pMCOD solution; here, we have examined single-query solutions only.

Finally, apart from the platforms discussed, there are additional alternatives. For example, ChronoStream [21] is a prototype system for big stream processing in a distributed environment, providing low latency. However, we have decided to adopt Flink because it combines strong positive features, as discussed in Section II-A, with mature engineering and a broad user community, while we did not consider scaling issues in this work.

VIII. Concluding Remarks

This work targets streaming distance-based outlier detection and provides the first solutions to date to this problem, when examined in a massively parallel setting, such as Flink. We have proposed a series of alternative techniques, with the one termed as pMCOD being a clear winner in the experiments that we have conducted using three real-world and one synthetic dataset. The improvements upon other solutions are significant, if not impressive, reaching up to an order of magnitude compared to the second best solution and up to three orders of magnitude compared to baseline solutions. There are also good speedups, between 2.66 and 6.65 times, compared to the non-parallel solutions implemented by third parties in [9], when running on a 4-core machine. Also, our solutions have been made publicly available. The motivation behind our work is to fill a gap in the currently offered solutions in large-scale streaming big data analytics. Moreover, our solutions aspire to act as a reference point for future techniques that target both continuous reporting of distance-based outliers and a massively parallel setting; to this end, the alternative techniques are not tailored to Flink but they can be transferred to other similar frameworks.

We identify three avenues for such future extensions. First, there are several features in current non-parallel solutions the parallelization of which might yield benefits. Two such features is the event queue and the full MCOD algorithm in [4]. The event queue is a priority queue, and its efficient distribution across several nodes depends heavily on the partitioning of the stream into different nodes. In addition, in the original MCOD proposal, the expensive M-tree is less used. Both features reduce the number of range queries considerably and their efficient parallel implementation are left to the extension of this paper. Second, further research is required in order to make value-based partitioning more practical and adaptively balanced, possibly using a M-tree instead of a grid, addressing also the issue of acquiring both initial and evolving metadata to reach efficient partitioning decisions; to this end, the early results in [8] need to be transferred into a streaming environment. Third, there are additional ways, in which the logical window can be partitioned, e.g., through adopting the notion of time slicing [22], which may be more efficient when multiple distance-based outlier detection queries are active simultaneously; this notion is also employed in [6]. Finally, another line of future research is on approximate outlier mining and on additional definitions of outlierness; here, we have provided exact solutions and we considered distance-based outliers only.

References

[1] C. C. Aggarwal, *Outlier Analysis*. Springer, 2013.

[2] R. Johnson, *Applied Multivariate Statistical Analysis*. Prentice Hall, 1992.

[3] E. Knorr, R. Ng, and V. Tucakov, "Distance-based outliers: algorithms and applications," *The VLDB Journal*, vol. 8, no. 3-4, pp. 237–253, 2000.

[4] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Efficient and flexible algorithms for monitoring distance-based outliers over data streams," *Inf. Syst.*, vol. 55, pp. 37–53, 2016.

[5] D. Yang, E. Rundensteiner, and M. Ward, "Neighbor-based pattern detection for windows over streaming data," in *EDBT*, 2009, pp. 529–540.

[6] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," in *ICDE*, 2014, pp. 76–87.

[7] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," in *CIKM*, 2007, pp. 811–820.

[8] L. Cao, Y. Yan, C. Kuhlman, Q. Wang, E. A. Rundensteiner, and M. Y. Eltabakh, "Multi-tactic distance-based outlier detection," in *ICDE*, 2017, pp. 959–970.

[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] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams," in *ICDE*, 2011, pp. 135–146.

[11] ——, "Continuous outlier detection in data streams: an extensible framework and state-of-the-art algorithms," in *SIGMOD*, 2013, pp. 1061–1064.

[12] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 426–435.

[13] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han, "Outlier detection for temporal data: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 9, pp. 2250–2267, 2014.

[14] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection for discrete sequences: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 5, pp. 823–839, 2012.

[15] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data Min. Knowl. Discov.*, vol. 29, no. 3, pp. 626–688, 2015.

[16] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Distributed strategies for mining outliers in large data sets," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 7, pp. 1520–1532, 2013.

[17] K. Bhaduri, B. L. Matthews, and C. Giannella, "Algorithms for speeding up distance-based outlier detection," in *Proc. of SIGKDD*, 2011, pp. 859–867.

[18] L. Rettig, M. Khayati, P. Cudré-Mauroux, and M. Piórkowski, "Online anomaly detection over big data streams," in *Big Data*, 2015, pp. 1113–1122.

[19] Y. Yan, J. Zhang, B. Huang, X. Sun, J. Mu, Z. Zhang, and T. Moscibroda, "Distributed outlier detection using compressive sensing," in *Proc. of SIGMOD*. ACM, 2015, pp. 3–16.

[20] L. Cao, J. Wang, and E. A. Rundensteiner, "Sharing-aware outlier analytics over high-volume data streams," in *Proc. of SIGMOD*, 2016, pp. 527–540.

[21] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *ICDE*. IEEE, 2015, pp. 723–734.

[22] S. Wang and E. A. Rundensteiner, "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing," in *EDBT*, 2009, pp. 299–310.