




Grid-Ordering for Outlier Detection in Massive Data Streams

Braulio V. Sánchez Vínces   [Universidade de São Paulo | braulio.sanchez@usp.br]

Robson L. F. Cordeiro  [Carnegie Mellon University | robsonc@andrew.cmu.edu]

 *Institute of Mathematics and Computer Science - ICMC, Trabalhador São-Carlense Avenue, 400, São Carlos, SP, 13566-590, Brazil.*

Received: 8 February 2024 • Published: 13 January 2025

Abstract Outlier detection is critical in data mining, encompassing the revelation of hidden insights or identification of potentially disruptive anomalies. While numerous strategies have been proposed for serial-processing outlier detection, the ever-expanding realm of big data applications demands efficient distributed computing solutions. This paper addresses the challenge of real-time outlier detection in multidimensional data streams with high-frequency arrivals, by presenting GOOST. This novel algorithm employs neighborhood analysis by leveraging grid-based data sorting. GOOST efficiently detects distance-based outliers, ensuring accurate detection in distributed environments within a competitive and much more stable processing time than previous solutions. We perform experiments on 6 real and synthetic data sets with up to 1.2M events, and up to 55 dimensions. We demonstrate that GOOST outperforms 3 state-of-the-art methods in terms of quality of results (30% more accurate) within competitive (and 45% more stable) processing times for real-time analysis of multidimensional data streams and high event frequency, thus offering a promising solution for various scientific and commercial domains.

Keywords: Distance-based Outlier Detection, Distributed Computing, Data Streams, Apache Spark

1 Introduction

An outlier is a data point that diverges significantly from the other points in a dataset [Aggarwal, 2017]. It occurs unpredictably and may cause negative results, but it can also reveal hidden insights or the emergence of a new pattern [Yang *et al.*, 2009]. As a consequence, the detection of outliers is widely studied in data mining, being critical for a large number of applications [Chandola *et al.*, 2009]. Examples are the identification of fraud, spam and network intrusion – just to name a few. It is also common to remove outliers from a dataset as a cleaning, preprocessing step before the use of any of the many mining algorithms that are strongly sensitive to the presence of outliers.

In the last few decades, numerous studies have proposed strategies to detect outliers [Jiang *et al.*, 2022; Liu *et al.*, 2008; Breunig *et al.*, 2000; Angiulli and Fasseti, 2007; Cao *et al.*, 2014; Kontaki *et al.*, 2011; Sadik and Gruenwald, 2010; Yang *et al.*, 2009]. Most of them follow a centralized, single machine setting. However, centralized solutions can no longer meet the strict response time requirements of emerging big data applications. Big data is expanding in various scientific and commercial areas like environmental monitoring, weather analysis, medicine, IT infrastructure, cloud computing, and social networking. Technological advances have increased the volume, velocity, and variety of data, leading to new computational challenges [Rajaraman and Ullman, 2020], particularly the need for efficient distributed computing to extract knowledge from this data [Cordeiro *et al.*, 2011].

Our work focuses on real-time analysis of data streams, as the potential value lies in their freshness. We introduce GOOST, a novel algorithm capable of spotting outliers in multidimensional data streams with large frequencies of new events, highlighting the importance of real-time analysis in

this context. The well-known distance-based approach to detect outliers is employed efficiently in a parallel and distributed fashion, thus achieving results of high quality in a scalable manner. Our main contributions are:

1. **Novel algorithm:** GOOST is a novel algorithm for detecting distance-based outliers in massive data streams. It uses a new neighborhood analysis based on grid-based data ordering, parallelism, and Apache Spark¹'s distributed capabilities to efficiently identify sparse rare events, thus ensuring high performance and accurate detection in distributed environments.
2. **Accurate Solution:** Our algorithm has obtained results that are up to 30% more accurate than 3 state-of-the-art competitors on 6 multi-domain data streams. Designed with scalability in mind, GOOST seamlessly handles large data volumes without compromising accuracy.
3. **Real-time Processing:** Our algorithm achieves competitive runtime and is up to 45% more stable than the other methods evaluated. It also scales near-linearly w.r.t the volume and dimensionality of the stream.

Figure 1 showcases some of the results of our experimental evaluation. It reports the quality of the outliers detected 1(a) and the corresponding runtime 1(b) obtained by GOOST and 3 state-of-the-art competitors in real and synthetic datasets considering a wide variety of parameter configurations. As it can be seen, our GOOST offers considerably better quality of results and a very competitive runtime in nearly all cases. It also provides much more stable results, i.e., small variance, regarding both quality and runtime.

The rest of the paper follows a traditional organization, with preliminary concepts in Section 2, related work

¹<https://spark.apache.org>

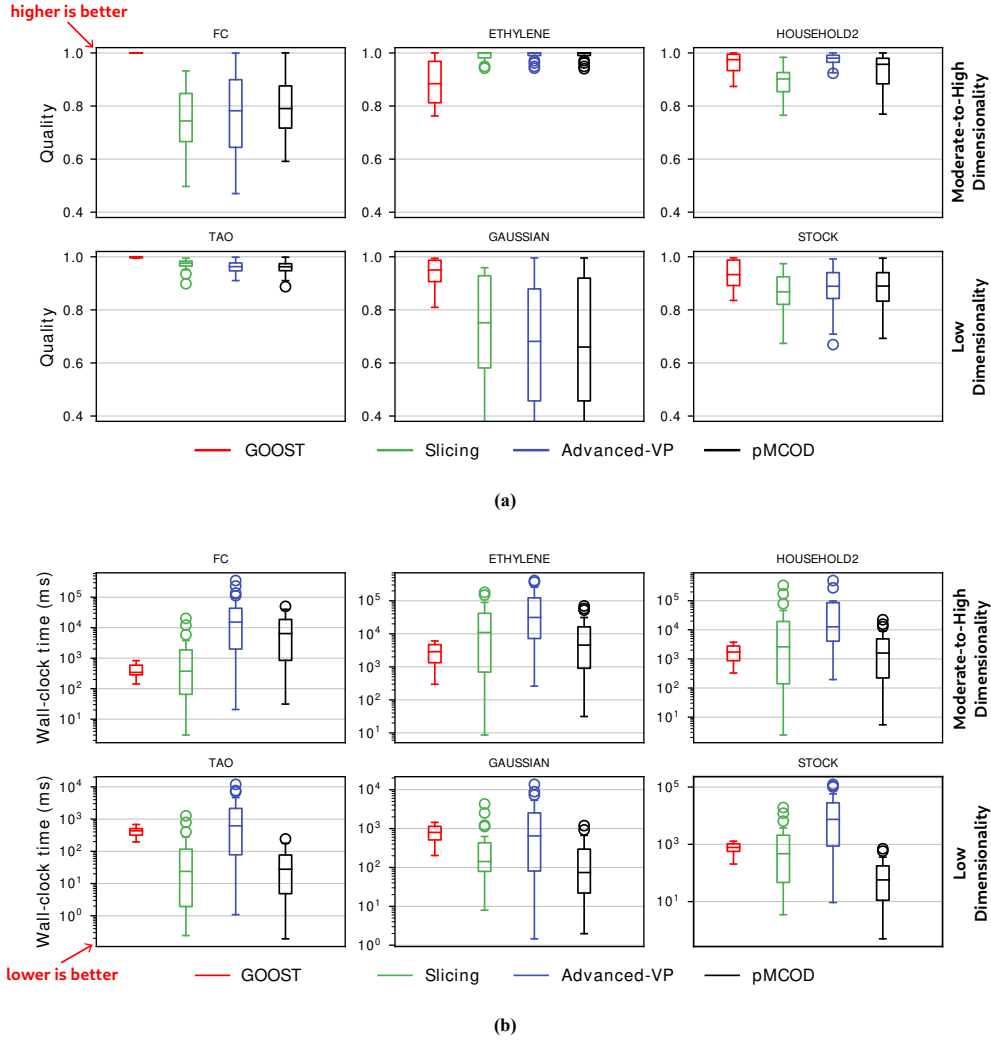


Figure 1. GOOST is accurate, fast and stable. Quality of results (a) and runtime (b) in datasets FC, ETHYLENE, HOUSEHOLD2, TAO, GAUSSIAN and STOCK considering a wide variety of parameter configurations. As it can be seen, our GOOST offers considerably better quality of results and a very competitive runtime in nearly all cases, when compared with the state-of-the-art competitors Slicing, Advanced-VP and pMCOD. Our GOOST also provides much more stable results, i.e., small variance, regarding both quality and runtime. See Section 5 for details. (best viewed in color)

overview in Section 3, proposed method in Section 4, experiments in Sections 5 and 6, and conclusion in Section 8.

2 Background Concepts

Outlier detection refers to the problem of identifying points that do not fit into the normal behavior within a dataset. These anomalous points occur unpredictably, and may cause very negative results, but they can also reveal hidden insights or the emergence of new patterns [Yang *et al.*, 2009]. For example, outliers may indicate frauds, spam, extreme weather events and system failures – just to name a few. Hence, the detection of outliers has been studied in the literature for a long time. Notably, it is still very popular because no one knows the true nature of outliers, and their detection is highly valuable and application dependent [Aggarwal, 2017].

There are many definitions for outliers in the literature [Aggarwal, 2017]. The most common one is that of *distance-based* outlier [Knorr and Ng, 1998], as shown in Definition 1.

Definition 1 (Distance-based Outlier) Given a dataset $P = \{x_1, \dots, x_n\}$ with n d -dimensional points $x_i \in \mathbb{R}^d$, a point $x_j \in P$ is an outlier iff x_j has less than k neighbors in P . Otherwise, x_j is an inlier. The neighbors of x_j are

the points in $\{x_i \in P \setminus \{x_j\} : \delta(x_i, x_j) \leq R\}$, where $\delta : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$ is a distance function. The maximum number of neighbors $k \in \mathbb{Z}_{>0}$ and the radius $R \in \mathbb{R}_{>0}$ are user-defined parameters.

The aforementioned definition is very intuitive and does not require prior knowledge about the data’s distribution, making it suitable for fully unsupervised detection of outliers. For example, consider the 2-dimensional data shown in Figure 2, where we highlight the neighborhood of two points for a radius R . The point at the left falls into a dense region with 10 neighbors, while the point at the right falls into a sparser region with only 3 neighbors. The most normal point can be easily distinguished from the less normal one by selecting a value for k such that $3 < k \leq 10$. Additionally, distance-based outliers can be computed efficiently and have robust theoretical foundations by being a generalization of different statistical tests [Chandola *et al.*, 2009; Ramaswamy *et al.*, 2000]. As an interesting characteristic, Definition 1 is compatible with the notion of *rare events* in statistics [Knorr and Ng, 1998]. For example, if a set of 1-dimensional points follows a Gaussian distribution, any point is a rare event if it deviates from the mean of the underlying distribution by 10

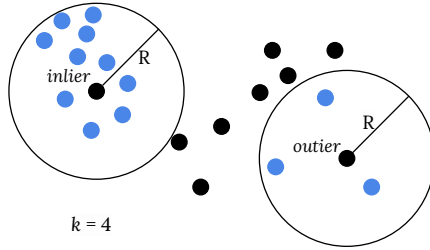


Figure 2. Distance-based outlier detection in static data.

times the standard deviation. Such points can be captured as outliers by setting R and k appropriately. Due to the aforementioned reasons, the distance-based definition for outliers is largely employed in the literature.

2.1 Outliers in Data Streams

The detection of outliers in data streams is a relatively new task when compared to the long history of outlier detection in the areas of database, data mining and statistics. Traditionally, a technique identifies patterns that reveal outliers regarding the entire set of points of a dataset. This is known in the literature as the *store-and-process* paradigm [Aggarwal, 2007]. Unfortunately, the store-and-process paradigm is not suitable for a stream because the entire set of events is *never* available due to its theoretically unbounded nature [Aggarwal, 2007].

A data stream is a collection of events ordered by their arrival time in a potentially infinite series, as it is shown in Definition 2 [Tran et al., 2016]. Hence, its analysis must be online and incremental [Stonebraker et al., 2005]. At any particular time, only a subset of the complete dataset is available. Also, old points tend to become irrelevant with the arrival of many new ones. Due to these reasons, the stream is generally processed considering only the W most recent points. The *sliding window* strategy analyzes points within a window of size W , always at the end of the stream, as per Definition 3.

Definition 2 (Data Stream) A data stream $D = (o_1, o_2, \dots)$ is a potentially infinite sequence of points $o_i \in \mathbb{R}^d$ of dimensionality d ordered according to their arrival time. For any point o_j that arrived after a point o_i , it must be true that $j > i$.

Definition 3 (Window) A window D_i of a data stream $D = (o_1, o_2, \dots, o_i, \dots)$ is the sequence containing only the W most recent points of the stream, where $W \in \mathbb{Z}_{>0}$ is an user-defined parameter such that $W \leq i$, and o_i is the most recent point known in D . Therefore, $D_i = (o_{i-W+1}, o_{i-W+2}, \dots, o_i)$.

As in several other works [Angiulli and Fasseti, 2007; Kontaki et al., 2011; Cao et al., 2014], we assume that the events are equidistant in time, consequently the subscripts i and j in Definition 2 indicate the arrival time of the event in the data stream. Thus, in Definition 3, D_i indicates the subset of events that ends with event o_i and begins with event o_{i-W+1} , since there is a temporal order and the window accommodates W events.

The window size W characterizes the volume of data to be analyzed at a time from the stream. Also, a value $S \in \mathbb{Z}_{>0}$ such that $S \leq W$ defines the length of the slide applied to the window. When S new points arrive, the window slides

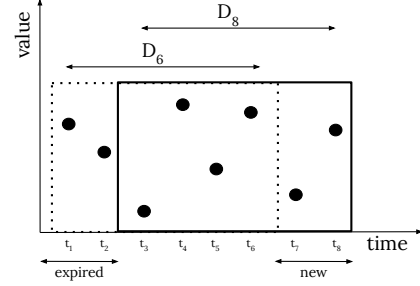


Figure 3. Two consecutive windows in a toy stream of 1-dimensional points; we consider $W = 6$ and $S = 2$.

to incorporate them; and, the S oldest points are discarded because they expired. Figure 3 illustrates these concepts by presenting two consecutive windows in a toy stream of 1-dimensional points; we consider $W = 6$ and $S = 2$. The value of each point is shown in the vertical axis of the illustration, while the horizontal axis has the corresponding arrival time of the point. As it can be seen, once $S = 2$ new points arrive, the window slides to accommodate them; meanwhile, the $S = 2$ oldest points are discarded.

The detection of outliers in a data stream usually follows Definition 1 by considering the analysis of one window at a time. This task is well-known in the literature as the problem of Distance-based Outlier Detection in Data Streams (DODDS) [Tran et al., 2016]. The corresponding formal definition is presented in Problem 1.

Problem 1 (DODDS) Given a data stream $D = (o_1, o_2, \dots)$ with points $o_i \in \mathbb{R}^d$ of dimensionality d , detect outliers in every window $D_W, D_{W+S}, D_{W+2S}, D_{W+3S}, \dots$ that is generated from stream D . For each window D_j , a point $o_i \in D_j$ is an outlier iff o_i has less than k neighbors in D_j . Otherwise, point o_i is an inlier. The neighbors of o_i are the points in $\{o_l \in D_j \setminus \{o_i\} : \delta(o_l, o_i) \leq R\}$, where $\delta : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$ is a distance function. The maximum number of neighbors $k \in \mathbb{Z}_{>0}$, the radius $R \in \mathbb{R}_{>0}$, the window size $W \in \mathbb{Z}_{>0}$ and the length of the slide $S \in \mathbb{Z}_{>0}$ are user-defined parameters.

The DODDS problem is the main focus of our work. There are two principal challenges in tackling it. The first challenge regards dealing with the changing nature of a data stream due to its effects on the outlierness of each point as newer points arrive. As the stream can change over time, a point that is an inlier regarding one window may become an outlier in another window. Figure 4 illustrates this fact by presenting two consecutive windows within a toy stream of 1-dimensional points; we consider $k = 4$, $W = 7$ and $S = 3$. Note that the point highlighted in blue is an inlier regarding window D_7 . In spite of this fact, it is considered to be an outlier in the succeeding window D_{10} . Cases like this one usually indicate the occurrence of *concept drifts* in the stream; thus, their identification is extremely desirable in many applications [Gama et al., 2014]. However, this changing nature in the outlierness of the points creates challenges that do not exist when analyzing static data, especially regarding computational efficiency [Iwashita and Papa, 2019]. Due to this reason, an algorithm with an incremental approach is usually the best option to detect outliers in streams [Yang et al., 2009].

The second challenge regards processing data streams with very large frequencies of new arriving points. For ex-

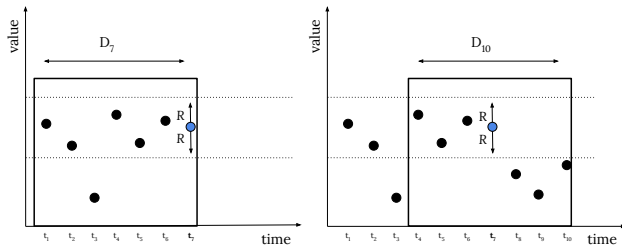


Figure 4. Two consecutive windows with $k = 4$, $W = 7$, and $S = 3$. Illustration adapted from Angiulli and Fasseti [2007].

ample, one of such streams is created from the search for exotic particles performed on the Large Hadron Collider, which produces on average approximately 300 Higgs bosons per hour for every 10^{11} collisions [Baldi et al., 2014]. Therefore, an average of 3×10^{13} new points arrive per hour in this stream, *continuously*. The potential value of events in a stream depends on their freshness, necessitating **real-time** analysis. However, a very large frequency of new arriving events demands very efficient processing of each window. Specifically, the W events of the current window must be processed before S new events arrive. Otherwise, real time processing is not a possibility.

Our work introduces a novel DODDS solution using an inherently parallel algorithm that can be implemented in any distributed computing framework, including Apache Spark², Flink³, and Storm⁴, which is ideal for handling massive data streams with thousands or millions of events per second.

3 Related Works

This section describes the most relevant algorithms to detect outliers in big data. We begin by briefly introducing techniques that can only analyze static datasets; then, we move on to our target setting of streaming data. There are many works focused on the detection of outliers from large amounts of static data in a distributed environment. For instance, the work in [Cao et al., 2017] presents an approach to reuse serial-processing outlier detectors in MapReduce. It starts by dividing the data into small partitions, and then uses an heuristic based on theoretical cost models to choose the most appropriate detector to process each partition independently. Another proposal is found in [Zhu et al., 2014]. It presents a grid-based algorithm implemented in MapReduce to spot outliers, which is automatically tuned with the help of a single-layer perceptron network. More specifically, the algorithm is threefold: i) it starts in a training phase, where the neural network is used to calibrate a threshold that determines if a point is an outlier; ii) then, a map-reduce stage preprocesses the data in parallel, and; iii) a final map-reduce stage is used to report all the outliers detected. DILOF [Yan et al., 2017] implements the well-known outlier detector LOF [Breunig et al., 2000] in a distributed manner. Its main contributions are a data-driven partitioning strategy that avoids data replication, and an early termination mechanism that reduces the network communication. DBOZ [Wang et al., 2015] is a distributed algorithm for distance-based outlier detection. It leverages a new index structure named Z-curve hierarchical tree (ZH-tree) to efficiently identify the neighbors of each

point of interest; also, a hierarchical structure is used to support space pruning. Other methods allow parallel processing, such as those in [Angiulli et al., 2012; Bhaduri et al., 2011], but, unfortunately, they do not work in the distributed, multi-machine environments that are required to process big data.

The aforementioned works are not applicable to streams due to their lack of incremental data processing. A naive solution to the DODDS problem involves storing the complete set of neighbors of each event in memory, which has a quadratic space complexity of $O(W^2)$. This is unfeasible in many scenarios, especially for big data where very large values of W are often required. As a result, some existing works investigated this problem; they are discussed as follows.

Very few studies focus on the detection of outliers in big data streams. The existing approaches are generally clever adaptations that allow reusing in a distributed environment algorithms that were either originally designed for a centralized, non-parallel setting, or; those that can only process static data. The state-of-the-art solutions in the big data stream scenario are the algorithms Advanced-VP, Slicing, and pMCOD [Toliopoulos et al., 2020]. Advanced-VP and pMCOD are modified versions of exact-STORM [Angiulli and Fasseti, 2007] and MCODE [Kontaki et al., 2011], respectively, including adaptations for distributed environment processing. Slicing is an improvement on Advanced-VP; it performs the time-slicing strategy to speed up the process based on the principle of minimal probing [Cao et al., 2014]. All of them use some kind of index structure to store the events to be processed per window. The tree algorithms follow a standard, two-steps strategy: (i) a stream handler procedure maps and replicates the new incoming events to their respective partitions in the cluster of machines, using either a tree-based, strategy or a grid-based one; (ii) then, a sliding window procedure independently processes the set of events within the corresponding partitions in each slide. Distinctly from the competitors, our proposed GOOST does not depend on indexing structures. It allows us to deal more efficiently with data partitioning and replication without impacting the accuracy of the results and also reducing the cost of communication between cluster nodes.

4 Proposed Method

Here we present the new algorithm **Grid-Ordering for Outlier Detection in Streaming Data (GOOST)**. Our solution is based on an equidistant grid with R -length cells applied to the data space. Events are sorted lexicographically according to their cells within the grid, employing a simple yet efficient strategy that reduces the cost of neighbor search by limiting the search space. The sort operation and neighborhood search utilize the MapReduce model, leveraging the Apache Spark cluster for resilient and distributed analysis of hundreds of thousands or millions of events.

Our proposal detects outliers over time by identifying each event's smallest number of neighbors in overlapping windows before it expires. Each window (D_n) contains W events ordered for neighborhood search, which is repeated for subsequent windows. We use a map-like structure to store key-value tuples for each active event (o) in the format (o_t, o_{nn}) , where o_t represents the *arrival time* of o and o_{nn} represents the *smallest number of neighbors* that o had before

²<https://spark.apache.org/>

³<https://flink.apache.org/>

⁴<https://storm.apache.org/>

expiring. When an event o expires, we check its corresponding tuple (o_t, o_{nn}) in the map. It is an outlier if $o_{nn} < k$. Expired event tuples are removed from the map. We do not store additional information per active event or maintain any advanced indexing structure for each window.

4.1 GOOST

Algorithm 1 provides the complete pseudo-code of GOOST. It processes the input stream by leveraging our grid ordering (Section 4.2) and neighborhood search (Section 4.3) procedures. The algorithm performs distributed and sequential tasks. Line 3 involves a sort operation based on a specific criterion from Section 4.2. Spark handles this as a distributed processing job with a time complexity of $O(W \log W)$.

The data in the stream originates from distributed sources like HDFS, Apache Kafka, etc., and is initially placed in the memory of workers across the cluster. In Line 4, we explicitly partition the data within a window into p blocks to determine the level of parallelism and data redundancy for the subsequent tasks. A custom partitioner is used to achieve balanced workloads, ensuring each block contains an equal number of events, thereby balancing the worker load. Our strategy accomplishes this task with $O(W)$ time complexity.

Algorithm 1: Grid-Ordering for Outlier Detection in Streaming Data (GOOST)

Input: *Stream*, W , S , R , k , p

```

1 Initialize minimal counting map  $C$ 
2 for each  $D_n \in \text{Stream}$  do
3   GridOrdering( $D_n$ ) // Alg. 2
4    $\text{blocks} \leftarrow \text{split } D_n \text{ into } p \text{ blocks}$ 
5    $\text{pairOfBlocks} \leftarrow \text{cartesian product of blocks}$ 
    and  $\text{blocks}$  //  $|\text{pairOfBlocks}| = p \text{ times } p$ 
6   for each pair  $\in \text{pairOfBlocks}$  do
7      $l \leftarrow 1$ 
8      $\text{block}_a, \text{block}_b \leftarrow \text{pair}_a, \text{pair}_b$ 
9     NeighborhoodSearch( $\text{block}_a, \text{block}_b, l, R, k$ ) // Alg. 3
10  end for
11  Map each event  $o$  contained in each
     $\text{pair}_a \in \text{pairOfBlocks}$  and emit a tuple
     $(o_t, o_{nn})$ 
12  Reduce each tuple  $(o_t, [o_{nn1}, o_{nn2}, \dots, o_{nnp}])$  to
    the form  $(o_t, \sum_{i=1}^p o_{nni})$  and store in  $D'_n$ 
13  for each tuple  $\in D'_n$  do
14     $C[\text{tuple}_{left}] \leftarrow \min(C[\text{tuple}_{left}], \text{tuple}_{right})$ 
15  end for
16  for each event  $o \in D'_n$  do
17    if  $o$  expired then
18      if  $o_{nn} < k$  then
19        Report  $o$  as an outlier
20      end if
21      Remove its entry from  $C$ 
22    end if
23  end for
24 end for

```

Since the data is physically distributed throughout the cluster, it is necessary to ensure that each block has sufficient re-

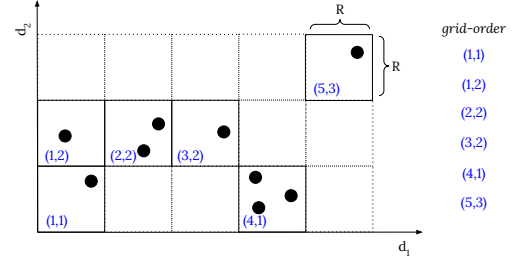


Figure 5. Illustration of the Grid-Ordering procedure onto a window. Cell coordinates in blue.

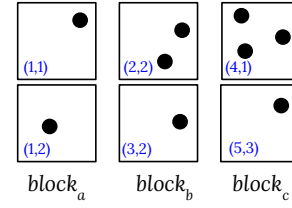


Figure 6. Partitioning a window into 3 blocks.

dundancy to guarantee an exact neighbor search result. This redundancy is done in Line 5, where we constructed a list of block pairs with the result of the Cartesian product of all blocks by themselves. The goal is to have all possible combinations of block pairs and to perform the neighborhood search in the window D_n as a whole. For example, in Figure 6 the result of the Cartesian product of that list of blocks will be $\{(block_a, block_a), (block_a, block_b), (block_a, block_c), (block_b, block_a), (block_b, block_b), (block_b, block_c), (block_c, block_a), (block_c, block_b), (block_c, block_c)\}$. This implies that each event in the window will have a redundancy factor of p times. This task has a time complexity of $O(p^2)$, but since $p \lll W$, its execution time is negligible.

In Line 9 we have a sequential task that is performed by each worker in the cluster concurrently, as explained later in Section 4.3. Since this is a neighborhood search strategy, we can estimate a worst case complexity of $O(|block_a| * |block_b|)$. However, by ensuring that all blocks have the same size W/p , we achieve a complexity of $O(\frac{W^2}{p^2})$. If we consider that we have at least p processing cores in the whole cluster, the complexity is reduced to $O(\frac{W^2}{p^3})$. In addition, we also introduced a pruning strategy in the quadratic neighbor search operation; if an event reaches k neighbors, then we avoid the other $(m - k)$ distance computations, which amortizes the overall complexity of this task. This inevitably leads to the determination of the time complexity of our solution. Considering that this analysis is based on the worst-case scenario, we experimentally confirm that the results observed have a much lower upper bound on the estimated $O(\frac{W^2}{p^3})$; see the upcoming Section 6.3.

The MapReduce task (Lines 11 and 12) removes redundancy from events and aggregates them with neighbors' sums, resulting in a $O(Wp)$ complexity. It adjusts parallelism and data redundancy using a single parameter, p . The more blocks, the more redundancy the cluster cores can compensate for.

Our two main procedures are describe in the following.

4.2 Grid-Ordering

Relational databases speed up partition-based processing by sorting tuples to group the ones with the same value in a set of attributes [Aggarwal and Yu, 2001]. Similarly, we design our proposal using grid-based ordering. For this ordering, a virtual grid is placed over the data space, anchored at the origin, and with cells of size R . As shown in Algorithm 2, we *never materialize* the grid; we only compute coordinates for each event in an “imaginary” grid. Lexicographical order is defined in the grid cells, i.e., the first dimension d_1 has the highest weight; for two grid cells with the same coordinate in d_1 , the next dimension d_2 is considered, and so on. Every event falls in exactly one cell; the event stores the grid coordinates of the cell in which it is spatially located according to R . That is, for an event o with $o_v = (v_1, v_2, \dots, v_d)$ its grid coordinates are $o_{cell} = (\lfloor v_1/R \rfloor, \lfloor v_2/R \rfloor, \dots, \lfloor v_d/R \rfloor)$ – see Line 3. Lastly, in Line 6, the events are sorted using this lexicographic order as the sorting key.

Algorithm 2: GridOrdering()

Input: D_n, R
Output: D_n with ordered events

```

1 for each event  $o \in D_n$  do
2   for  $i \leftarrow 0$  to  $d$  do
3      $o_{cell}[i] \leftarrow \lfloor o_v[i]/R \rfloor$ 
4   end for
5 end for
6 Sort events  $o \in D_n$  by its  $o_{cell}$  values as sorting keys

```

Figure 5 shows the grid’s discretization of space, arranging non-empty cells in a sequence based on their coordinates. This helps identify regions for neighborhood search and discard others. Events contained in immediately contiguous cells (see Definition 4) can be neighbors of each other. For instance, the event inside the cell (3, 2) can have neighbors only in the cells (2, 2) and (4, 1), discarding the others. On the other hand, the event located in cell (5, 3) has no neighbors since its cell has no other immediately contiguous cells. This strategy allows us to speed up the neighbor search without using advanced indexing structures. It is also easy to implement since the sorting operation is primitive within any current distributed processing platform.

Definition 4 (Immediately Contiguous Cells) *Given two non-empty cells a and b with coordinates $(a[1], a[2], \dots, a[d])$ and $(b[1], b[2], \dots, b[d])$, respectively; a and b are immediately contiguous cells if and only if $|a[j] - b[j]| \leq 1, \forall j \leq 1 \leq d$.*

After this sorting, we partition the window D_n into p subsets or *blocks* of events. It should be noted that each block is not a cell – see Figure 6. The goal is to physically distribute data throughout the cluster to ensure that each worker can perform the following operation in a distributed manner.

4.3 Neighborhood Search

Böhm et al. [2001] capitalize on grid ordering to speed up join operations in databases. Similarly, we design our proposal using grid ordering to search for neighbors. This procedure is performed on two blocks of events, e.g., $block_a$ and $block_b$. As shown in Algorithm 3, the purpose is to find all

Algorithm 3: NeighborhoodSearch()

Input: Block $block_a$, Block $block_b$, l, R, k

```

1  $low_a, high_a \leftarrow block_a[1]_{cell}, block_a[|block_a|]_{cell}$ 
2  $low_b, high_b \leftarrow block_b[1]_{cell}, block_b[|block_b|]_{cell}$ 
3 for  $i \leftarrow l$  to  $d$  do
4   if  $low_a[i] > high_b + 1$  or  $low_b[i] > high_a + 1$  then
5     return //  $block_a$  has no neighbors in  $block_b$ 
6   end if
7   if  $low_a[i] < high_a$  and  $low_b[i] < high_b$  then
8      $l \leftarrow i$ 
9     break
10  end if
11 end for
12 if  $|block_a| < m$  and  $|block_b| < m$  then
13   Perform neighborhood search of  $block_a$  in  $block_b$ 
14 end if
15 if  $|block_a| \geq m$  and  $|block_b| < m$  then
16    $block_{a1}, block_{a2} \leftarrow \text{split } block_a$ 
17   NeighborhoodSearch( $block_{a1}, block_b, l, R, k$ )
18   NeighborhoodSearch( $block_{a2}, block_b, l, R, k$ )
19 end if
20 if  $|block_a| < m$  and  $|block_b| \geq m$  then
21    $block_{b1}, block_{b2} \leftarrow \text{split } block_b$ 
22   NeighborhoodSearch( $block_a, block_{b1}, l, R, k$ )
23   NeighborhoodSearch( $block_a, block_{b2}, l, R, k$ )
24 end if
25 if  $|block_a| \geq m$  and  $|block_b| \geq m$  then
26    $block_{a1}, block_{a2} \leftarrow \text{split } block_a$ 
27    $block_{b1}, block_{b2} \leftarrow \text{split } block_b$ 
28   NeighborhoodSearch( $block_{a1}, block_{b1}, l, R, k$ )
29   NeighborhoodSearch( $block_{a1}, block_{b2}, l, R, k$ )
30   NeighborhoodSearch( $block_{a2}, block_{b1}, l, R, k$ )
31   NeighborhoodSearch( $block_{a2}, block_{b2}, l, R, k$ )
32 end if

```

neighbors of the events of $block_a$ in $block_b$, in a *divide-and-conquer* way. Based on Definition 4 it is only necessary to consider those events in $block_b$ whose cell coordinates are immediately contiguous to the cell coordinates of the events in $block_a$. If we consider the cell coordinates of the first and last events of each block it is possible to abstract a bounding rectangle. For example, $block_a$ has a bounding rectangle defined by $\{low_a = block_a[1]_{cell}, high_a = block_a[|block_a|]_{cell}\}$. This conceptual bounding rectangle – *never materialized* – speeds up the neighbor search procedure since we can iterate incrementally over its d dimensions to try to find one in which the intervals $\{low_a, high_a\}$ and $\{low_b, high_b\}$ are separated by at least 1. If that is the case, the neighborhood search is avoided in both blocks.

Otherwise, we check if in the current dimension l the values of the coordinates are different and, consequently, are limited by the values of the first and last cell respectively – see Lines 3 to 11. This dimension l is used to define the first dimension from which the following recursive calls will iterate to discriminate the search space between the two subsequent blocks. Each recursion divides $block_a$ and (or) $block_b$ into halves, until a sufficiently small size m is reached to perform the neighborhood search of the $block_a$ in the $block_b$

- see Line 13. The focus is not on the number of neighbors of an event, but on the total number of neighbors it has; the number of neighbors of an event o is stored in o_{nn} . Each worker in the cluster performs this task sequentially, but concurrently with other workers.

5 Proposed Evaluation

In this section we describe the proposed methodology used to evaluate our GOOST and 3 state-of-the-art methods considering the following aspects: (i) quality of results, (ii) runtime, (iii) scalability, and (iv) parameter sensitivity. It includes the materials used and details about the systematic evaluation carried out to answer the following questions:

- Q1 How accurate is our method?
- Q2 How fast is our method?
- Q3 How does our method scale-up?
- Q4 How sensitive to choosing R and k is our method?

For our evaluation, we use one synthetic and 5 real-world popular benchmark datasets coming from different application domains. Table 1 summarizes the datasets used.

5.1 Evaluation

To ensure impartiality, all algorithms evaluated were developed to work over the Java Virtual Machine using the same execution hardware. We used a cluster with three worker nodes (12 cores and 36GB of RAM each) and a master node.

The outlierness degree is binary: 0 for inlier and 1 for outlier. A base algorithm is used as *ground truth* to evaluate results – the exact-STORM algorithm. It offers a simple and accurate solution for the DODDS problem, employing an index structure for range queries and temporal considerations (Section 2). We evaluated the accuracy of the methods using the measure *F1-score* that represents the harmonic mean of the ratio of true outliers identified (*Recall*) and the ratio of which outliers are true among those identified (*Precision*).

Parameters R and k influence the outlier rate, impacting algorithm performance. We used the same default values according to Tran *et al.* [2016] for R and k per dataset (see Table 2) to set $\sim 1\%$ to $\sim 3\%$ of events identified as outliers. To simulate different densities and arrival frequencies, we set the following two scenarios: (i) set 5 incremental sizes of W for each dataset, and (ii) configure S as a percentage of the window size, i.e., 1% of W , 10% of W , 20% of W , 50% of W and 100% of W . W vary per dataset; FC and TAO with 1K, 5K, 10K, 15K, and 20K. All other datasets with values 10x larger. For runtime evaluation, we ran each experiment 5 times and report average and standard deviation results.

The first scenario shows increased processing time due to larger data. Distinctly, the second scenario shows increased incoming and expiring events and decreased active windows per event. Experiments exceeding a 10-minute execution per window time-out were interrupted due to being considered too long for real-time solutions.

Table 1. Summary of datasets.

Dataset	# Points	# Dimensions	Domain
FC [Kontaki <i>et al.</i> , 2011]	0.58M	55	Geography
Ethylene [Tran <i>et al.</i> , 2016]	1M	16	Physics
Household2 [Tran <i>et al.</i> , 2016]	1.29M	7	Energy
TAO [Angiulli and Fasseti, 2007]	0.57M	3	Climate
Gaussian [Kontaki <i>et al.</i> , 2011]	1M	1	Synthetic
Stock [Cao <i>et al.</i> , 2014]	1.05M	1	Finance

Table 2. Default R and k values per dataset.

Dataset	R	k
FC	525	50
Ethylene	115	50
Household2	6.5	50
TAO	1.9	50
Gaussian	0.028	50
Stock	0.45	50

5.2 Algorithm settings

All the algorithms evaluated use the same base parameters for DODDS: W , S , R , and k . The degree of parallelism was set to 16 for all, i.e., p (GOOST) = *parallelism* (competitors) = 16. pMCOD uses grid-based partitioning. In contrast, Advanced-VP and Slicing use the tree-based (VP-tree). To construct the VP-tree, it is necessary to specify the sample size ($VPcount$) from the streaming dataset, and we set $VPcount = 10000$ according to Toliopoulos *et al.* [2020].

6 Experimental Results

The experiments involved 25 test cases per dataset, resulting in 150 executions for quality of results and 750 for runtime evaluation. To sum up all the results obtained, we use box plots, so that we can observe the average values and their standard deviation.

6.1 Quality of results

Here we answer Question Q1 of Section 5. Figure 1a reports that GOOST – shown in red – presents qualitatively superior results by being up to 30% more accurate than the competitors for the vast majority of the scenarios evaluated, expressed in F1-scores values. Additionally, the results of our GOOST are much more stable. It is evident that for some datasets, e.g., FC and Gaussian, the competitors present large standard deviation in their results; distinctly, our GOOST presents considerably lower standard deviations.

In Table 3, we detail the F1-score values for all the scenarios described in Section 5.1. It is possible to observe that all algorithms reach their highest F1-score when processing the window without overlap, i.e., $S = 100\%W$, with some exceptions where the F1-score drops when processing the window with the highest value of S , e.g., Advanced-VP and pMCOD for FC at the first window size. Importantly, note that the lowest values reported by GOOST are ~ 0.76 ; higher by far than the rest of the algorithms, which were ~ 0.2 .

It is evident that GOOST managed to identify the same outliers in all datasets as exact-STORM for the $S = 100\%W$ configuration at all 5 values of W . In this configuration, exact-STORM evaluates whether a point o has less than k neighbors by considering only the number of preceding neighbors of o , and it does so by performing a range query for each event in the current window. Since there is no overlap between windows, o has no succeeding neighbors. GOOST also performs the neighbor search and reports the outliers identified in the current window, since with the next sliding all the W events analyzed will expire.

GOOST presented the best possible performance with FC for all the sizes of W and S , outperforming the other algorithms by ~ 0.3 . Another case to highlight is the one of the dataset Gaussian; while GOOST was above ~ 0.8 , the competing algorithms presented a very unstable performance

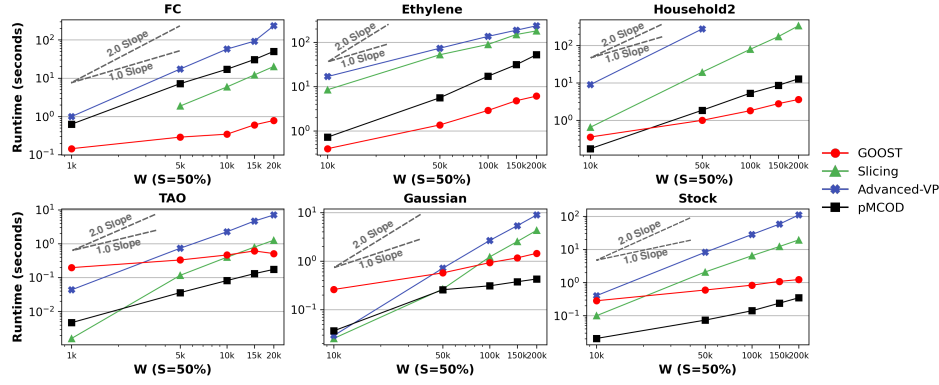


Figure 7. Runtime for GOOST, Advanced-VP, Slicing, and pMCO in seconds when $S = 50\%W$ in the 5 variations of W . Both axes are in logarithmic scale. (best viewed in color)

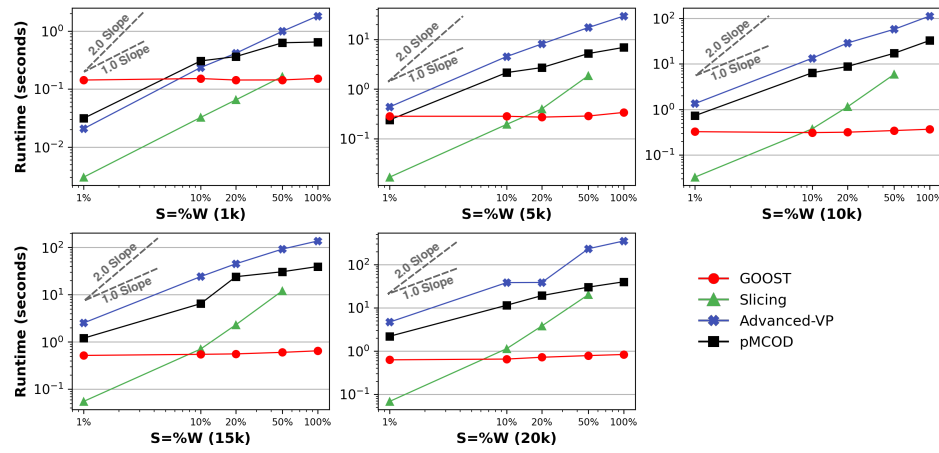
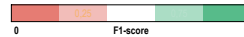


Figure 8. Runtime in seconds for dataset FC. Both axes are in logarithmic scale. (best viewed in color)

Table 3. Quality of results for each dataset. NA indicates the value was not collected due to execution errors. (best viewed in color)

Datasets	1st size of W					2nd size of W					3rd size of W					4th size of W					5th size of W				
	S=1%W	S=10%W	S=20%W	S=50%W	S=100%W	S=1%W	S=10%W	S=20%W	S=50%W	S=100%W	S=1%W	S=10%W	S=20%W	S=50%W	S=100%W	S=1%W	S=10%W	S=20%W	S=50%W	S=100%W	S=1%W	S=10%W	S=20%W	S=50%W	S=100%W
GOOST results																									
FC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Ethylene	0.9743	0.9682	0.9675	0.9663	1	0.8843	0.8765	0.9156	0.9156	1	0.8314	0.8261	0.8261	0.8581	1	0.7857	0.7858	0.8123	0.9047	1	0.7829	0.7776	0.7626	0.781	1
Household2	0.9944	0.9944	0.995	0.9972	1	0.969	0.973	0.9748	0.9892	1	0.9334	0.9415	0.9565	0.9749	1	0.9011	0.9071	0.9236	0.9836	1	0.8338	0.8929	0.9075	0.943	1
TAO	0.9981	0.998	0.9984	0.9998	1	0.995	0.9951	0.9962	0.9995	1	0.9963	0.9962	0.9973	0.9993	1	0.996	0.9976	0.9974	0.9992	1	0.996	0.9967	0.9963	0.9991	1
Gaussian	0.9849	0.9861	0.9869	0.9898	1	0.9451	0.9502	0.9555	0.9697	1	0.9078	0.9122	0.9206	0.946	1	0.86	0.8728	0.8931	0.9593	1	0.8055	0.8315	0.8485	0.9065	1
Stock	0.9859	0.9867	0.9878	0.9905	1	0.9328	0.9367	0.9398	0.9635	1	0.8977	0.8958	0.8916	0.9282	1	0.8514	0.8474	0.8424	0.896	1	0.8402	0.8355	0.8448	0.906	1
Slicing results																									
FC	0.9325	0.8823	0.8725	0.9144	NA	0.4972	0.5456	0.7084	0.839	NA	0.7032	0.7966	0.811	0.8895	NA	0.6167	0.7542	0.7935	0.6731	NA	0.5434	0.6442	0.681	0.7334	NA
Ethylene	0.9426	0.9904	0.9962	0.9991	NA	0.9711	0.9969	0.9999	0.9998	NA	0.9814	0.9999	0.9999	0.9999	NA	0.9816	0.9996	1	1	NA	0.9493	0.9622	0.9999	0.9999	NA
Household2	0.9317	0.9662	0.9835	0.9762	NA	0.9212	0.9213	0.9077	0.8657	NA	0.8717	0.8965	0.9052	0.824	NA	0.8226	0.8733	0.7653	0.9324	NA	0.7711	0.8199	0.864	0.9243	NA
TAO	0.9356	0.9625	0.9761	0.9899	NA	0.8955	0.9511	0.9833	0.9715	NA	0.9771	0.9851	0.9904	0.9761	NA	0.9652	0.9845	0.9947	0.9731	NA	0.9759	0.9855	0.9822	0.9616	NA
Gaussian	0.9437	0.9437	0.9358	0.9364	NA	0.739	0.7641	0.7951	0.9588	NA	0.443	0.5868	0.712	0.8875	NA	0.3525	0.5429	0.6274	0.7728	NA	0.297	0.5644	0.7259	0.9261	NA
Stock	0.8947	0.9355	0.9403	0.9165	NA	0.856	0.921	0.9418	0.974	NA	0.8249	0.8448	0.9089	0.9372	NA	0.7834	0.8428	0.7676	0.8552	NA	0.7034	0.6738	0.808	0.8801	NA
Advanced-VP results																									
FC	0.6959	0.9347	0.8726	0.8785	0.8516	0.5326	0.5572	0.9091	0.8994	0.9408	0.5746	0.4702	0.7941	0.804	0.9317	0.6443	0.716	0.7368	0.7822	0.9994	0.5734	0.6442	0.681	0.7168	0.944
Ethylene	0.9426	0.9908	0.9962	0.9991	0.9999	0.9711	0.9969	0.9999	0.9998	0.9999	0.9809	0.9999	0.9999	0.9999	0.9999	0.9816	0.9996	1	0.9929	0.9923	0.9493	0.9622	0.9999	0.9999	0.9999
Household2	0.9765	0.9774	0.9835	0.9933	0.9947	0.9254	0.9235	0.9607	0.9854	0.9959	0.9717	0.9851	0.9904	0.9761	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
TAO	0.9161	0.935	0.9507	0.9629	0.9662	0.9304	0.9872	0.9103	0.9769	0.9985	0.9964	0.9458	0.9712	0.9533	0.9963	0.9382	0.9695	0.9702	0.9476	0.9956	0.947	0.9741	0.9649	0.9946	0.9943
Gaussian	0.9207	0.8726	0.9197	0.9676	0.879	0.588	0.649	0.7407	0.6085	0.9927	0.2887	0.4152	0.5286	0.6975	0.9929	0.2141	0.3726	0.457	0.7215	0.6813	0.1745	0.3964	0.5697	0.8623	0.9953
Stock	0.9105	0.9031	0.9404	0.9743	0.8664	0.8598	0.8878	0.946	0.767	0.9784	0.811	0.8891	0.9137	0.9372	0.9916	0.7835	0.8428	0.8774	0.8951	0.9898	0.7086	0.7515	0.6692	0.8801	0.9916
pMCO results																									
FC	0.8979	0.8761	0.8658	0.8602	0.848	0.5916	0.8759	0.8564	0.839	0.9408	0.6527	0.7283	0.7729	0.7905	0.9317	0.6393	0.7542	0.7863	0.7519	0.9202	0.6126	0.6442	0.681	0.7168	1
Ethylene	0.9406	0.9904	0.9962	0.9991	0.9999	0.9711	0.9969	0.9999	0.9998	0.9999	0.9814	0.9999	0.9999	0.9999	0.9999	0.9816	0.9996	1	1	1	0.9493	0.9622	0.9999	0.9999	0.9999
Household2	0.9762	0.98	0.9758	0.9933	0.9947	0.9346	0.9494	0.9607	0.9722	0.9999	0.8749	0.9007	0.9092	0.9572	1	0.8234	0.8733	0.8837	0.9617	1	0.7695	0.8127	0.864	0.9243	1
TAO	0.9101	0.9305	0.9513	0.9625	0.972	0.8874	0.9658	0.9531	0.9769	0.9985	0.939	0.9835	0.9712	0.9533	0.9101	0.926	0.9645	0.9541	0.9476	0.9956	0.9534	0.9741	0.9649	0.9945	0.9943
Gaussian	0.8783	0.9428	0.9197	0.8803	0.8668	0.6057	0.6183	0.6599	0.6085	0.9927	0.2876	0.4101	0.5528	0.7977	0.9629	0.2147	0.3726	0.457	0.7215	0.9921	0.1744	0.3985	0.5401	0.8623	0.9953
Stock	0.8984	0.8911	0.9404	0.9166	0.9844	0.857	0.8897	0.8649	0.767	0.9947	0.8249	0.8447	0.9089	0.9479	0.9916	0.7899	0.8428	0.8774	0.8951	0.9898	0.7899	0.8428	0.8774	0.8951	0.9898
<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>																									



with values ranging from ~ 0.17 to ~ 0.9 . This is an indicator of a high level of error in identifying true outliers in each window when the sliding size is small. On the other hand, it was with dataset Ethylene that GOOST slightly underper-

formed in cases where the value of S is less than $100\%W$.

The results suggest a strong similarity among the base algorithm exact-STORM and GOOST. It can be due the simplification in terms of assumptions when comparing the two al-

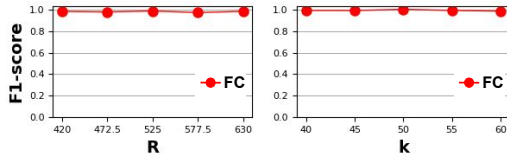


Figure 9. Sensitivity results w.r.t. R and k parameters in the FC dataset with $W = 10k$ and $S = 20\%W$.

gorithms, for example, the exact-STORM stores the number of preceding and succeeding neighbors of each point, while GOOST assumes the smallest number of neighbors of a point in all windows that it was present.

To answer question *Q1: How accurate is our method?* We conclude that our strategy of only keeping the minimum number of neighbors per event until expiration is sufficient to address the needs of an accurate solution for DODDS.

6.2 Runtime

Although GOOST does not present particular optimizations that take advantage of Apache Spark, it is possible to see in Figure 1b that it presents a competitive and very stable runtime (45% more stable) for all the evaluated scenarios.

In addition, it is worth noting that algorithm Slicing presented problems of excessive consumption of resources, both CPU and cluster memory for cases where $S = 100\%$ of W . Similarly, Advanced_VP exceeded the 10-minute execution time-out by window, in dataset Household2 for sizes $W = 100K$, $150K$ and $200K$. Thus, those results are not reported.

Answering the question *Q2: How fast is our method?* We conclude that our method is indeed a competitive and stable technique in terms of runtime.

6.3 Scale-up results

In order to answer the question *Q3* we report the scale-up results. As shown in Figure 7, by increasing W the execution time of each algorithm increases. This behavior was observed in all the evaluated algorithms. And it is evident because we are increasing the density of the stream.

To evaluate this aspect, we analyze the behavior of algorithms assessed on stress cases in real-time analysis environments of large data streams. We use the configuration $S = 50\%W$ in the 5 variations of W for each dataset, which is similar to using different random sample sizes from a static dataset and analyzing the growth of the time curve from the smallest sample to the entire dataset, but adapted to the streaming sample.

All related-work algorithms incur a near-quadratic growth w.r.t. W ; distinctly, our GOOST exhibits an evident near-linear growth, which is an extremely desirable feature in solutions that will deal with large data streams.

The same behaviour is expected when we increase S , since we are increasing the speed of data stream input. However, looking at the curves in Figure 8, the GOOST execution time shows a slight growth when processing consecutive S values. For example, when comparing the processing times for a 10K events window with slide $S = 1\%W$ and $S = 100\%$ the increase is at most $\sim 10\%$, and this is the same for any size of W . This is because GOOST processes each window as if it were a static dataset – see Section 4. The slight increase in time as S increases reflects the insertion, update, and removal tasks on the map-like structure that stores the minimum count of the number of neighbors per event; thus,

the more new events per window, the more operations there will be on the map. Compared to the other algorithms evaluated, GOOST does not store additional information per active event nor does it maintain any advanced indexing structure per window. This feature is clearly a competitive advantage over the other algorithms, and does not negatively impact the quality of results as seen in the previous section.

We observed similar patterns in each dataset; that is why only the runtime results of FC are detailed.

It is worth noting that, although the other algorithms start from lower runtimes for $S = 1\%W$, the increase in time for the next value $S = 10\%W$ makes them slower than GOOST. And the increase continues in a near-quadratic growth, which makes them unfeasible in use cases of real-time processing of high-frequency and massive data streams.

To answer question *Q3: How does our method scale-up?* We conclude that GOOST is an excellent choice in stream scenarios with high event frequency since its empirical impact on runtime growth is near-linear, and thus considerably lower than that of previous, state-of-the-art solutions.

6.4 Sensitivity results

Here, we answer question *Q4* of Section 6. We compare the results obtained by exact-STORM and GOOST with five different values for R and k , with $W = 10k$ and $S = 20\%W$ on the FC dataset. Figure 9 shows the F1-score versus R and k , respectively. Note that both lines are nearly flat, highlighting that GOOST produces almost the same result as exact-STORM regardless of the values of R and k .

In this way, we experimentally demonstrate that our approach of upholding the minimum neighbor count of an event per window is highly correlated with the approach used by exact-STORM to report outliers, i.e., the summation of succeeding neighbors with unexpired preceding neighbors.

Answering the question *Q4: How sensitive is our method to choosing R and k ?* Our method shows little or almost no sensitivity to the chosen R and k parameters.

7 Discussion

Mining high-dimensional data poses special challenges, but such data is also known to have lower intrinsic dimensionality [Faloutsos et al., 2010; Zimek et al., 2012]. We achieved good results with FC despite the fact it has 55 dimensions, but for datasets of even higher dimensionality it would also be plausible to apply dimensionality reduction preprocessing, e.g. PCA or SVD, before analyzing an event window. Thus, the dimensionality curse problem would be mitigated.

The adjustment of parameters W and S depends on the application [Aggarwal, 2007]. For instance, climate data analysis requires a W of months, while for ECGs it is minutes. On the other hand, S determines the frequency of the analysis to be performed, i.e. how quickly the specialist needs updated results. Nevertheless, as shown in Table 3, our method tends to be the best regardless of the values chosen for W and S .

8 Conclusions

In this paper, we introduce one novel approach to dealing with large data streams (multidimensional, high frequency) to detect distance-based outliers in a parallel and distributed fashion, achieving this with high precision in a scalable and fast way. We demonstrate the effectiveness and efficiency of

our algorithm GOOST through extensive experimental evaluations. Our major contributions are:

1. **Novel algorithm:** GOOST is a novel algorithm for massive data streams that uses grid-based data ordering, parallelism, and Apache Spark's distributed capabilities to efficiently identify distance-based outliers and detect sparse events in distributed environments.
2. **Accurate solution:** GOOST consistently achieves up to 30% higher accuracy on 6 multi-domain data streams compared with 3 competitors of the state-of-the-art. It maintains precision while seamlessly managing substantial data volumes, emphasizing scalability.
3. **Real-time Processing** GOOST demonstrates competitive runtimes, exhibiting up to a 45% increase in stability compared with other evaluated methods. This enhanced stability enables nearly linear scalability concerning stream volume and dimensionality.

Finally, to our knowledge, no previous work presents a truly *efficient real-time solution* that combines massive parallelism and distance-based outlier detection in a streaming context. GOOST is agnostic, allowing easy implementation on any distributed data stream processing platform, which results in an additional contribution.

Reproducibility: Our data, code, and full set of results are at <https://github.com/BraulioSanchez/GOOST>

References

- Aggarwal, C. C. (2007). *Data streams: models and algorithms*. Springer Science & Business Media.
- Aggarwal, C. C. (2017). *Outlier Analysis(Second Edition)*. Springer International Publishing.
- Aggarwal, C. C. and Yu, P. S. (2001). Outlier detection for high dimensional data. *SIGMOD Record*, 30(2):37–46.
- Angiulli, F., Basta, S., Lodi, S., and Sartori, C. (2012). Distributed strategies for mining outliers in large data sets. *TKDE*, 25(7):1520–1532.
- Angiulli, F. and Fassetto, F. (2007). Detecting distance-based outliers in streams of data. In *CIKM*, page 811.
- Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):4308.
- Bhaduri, K., Matthews, B. L., and Giannella, C. R. (2011). Algorithms for speeding up distance-based outlier detection. In *SIGKDD*, pages 859–867.
- Böhm, C., Braunmüller, B., Krebs, F., and Kriegel, H.-p. (2001). Epsilon grid order. *SIGMOD Record*, 30(2):379–388.
- Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. (2000). Lof: Identifying density-based local outliers. *SIGMOD*, 29(2):93–104.
- Cao, L., Yan, Y., Kuhlman, C., Wang, Q., Rundensteiner, E. A., and Eltabakh, M. (2017). Multi-tactic distance-based outlier detection. In *ICDE*, pages 959–970.
- Cao, L., Yang, D., Wang, Q., Yu, Y., Wang, J., and Rundensteiner, E. A. (2014). Scalable distance-based outlier detection over high-volume data streams. *ICDE*, D:76–87.
- Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection. *ACM Computing Surveys*, 41(3):1–58.
- Cordeiro, R. L. F., Traina Jr, C., Traina, A. J. M., Hernandez, J. L., Kang, U., and Faloutsos, C. (2011). Clustering very large multi-dimensional datasets with mapreduce. In *KDD*, volume 11, pages 408–516.
- Faloutsos, C., Wu, L., Traina, A., and Traina Jr, C. (2010). Fast feature selection using fractal dimension. *JIDM*, 1(1):3–3.
- Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37.
- Iwashita, A. S. and Papa, J. P. (2019). An overview on concept drift learning. *IEEE Access*, 7(Section III):1532–1547.
- Jiang, S., Cordeiro, R. L. F., and Akoglu, L. (2022). D.MCA: outlier detection with explicit micro-cluster assignments. In *ICDM*, pages 987–992.
- Knorr, E. M. and Ng, R. T. (1998). Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403.
- Kontaki, M., Gounaris, A., Papadopoulos, A. N., Tsichlas, K., and Manolopoulos, Y. (2011). Continuous monitoring of distance-based outliers over data streams. In *ICDE*, pages 135–146.
- Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). Isolation forest. In *ICDM*, pages 413–422.
- Rajaraman, A. and Ullman, J. D. (2020). *Mining of massive datasets*. Cambridge University Press.
- Ramaswamy, S., Rastogi, R., and Shim, K. (2000). Efficient algorithms for mining outliers from large data sets. In *SIGMOD*.
- Sadik, M. S. and Gruenwald, L. (2010). Dbod-ds: Distance based outlier detection for data streams. In *DEXA, Part I 21*, pages 122–136.
- Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47.
- Toliopoulos, T., Gounaris, A., Tsichlas, K., Papadopoulos, A., and Sampaio, S. (2020). Continuous outlier mining of streaming data in flink. *Information Systems*, 93.
- Tran, L., Fan, L., and Shahabi, C. (2016). Distance-based outlier detection in data streams. *PVLDB*, 9(12):1089–1100.
- Wang, X.-T., Shen, D.-R., Bai, M., Nie, T.-Z., Kou, Y., and Yu, G. (2015). An efficient algorithm for distributed outlier detection in large multi-dimensional datasets. *JCST*, 30(6):1233–1248.
- Yan, Y., Cao, L., Kuhlman, C., and Rundensteiner, E. (2017). Distributed local outlier detection in big data. In *SIGKDD*, pages 1225–1234.
- Yang, D., Rundensteiner, E. A., and Ward, M. O. (2009). Neighbor-based pattern detection for windows over streaming data. In *EDBT*, pages 529–540.
- Zhu, S., Li, J., Huang, J., Luo, S., and Peng, W. (2014). A mapreduced-based and cell-based outlier detection algorithm. *WUJNS*, 19:199–205.
- Zimek, A., Schubert, E., and Kriegel, H.-P. (2012). A survey on unsupervised outlier detection in high-dimensional numerical data. *Stat. Anal. Data Min.*, 5(5):363–387.