Distance-based Outlier Detection in Data Streams

Luan Tran
Computer Science Dept.
Univ. of Southern California
luantran@usc.edu

Liyue Fan
Integrated Media Systems Center
Univ. of Southern California
liyuefan@usc.edu

Cyrus Shahabi
Integrated Media Systems Center
Univ. of Southern California
shahabi@usc.edu

ABSTRACT

Continuous outlier detection in data streams has important applications in fraud detection, network security, and public health. The arrival and departure of data objects in a streaming manner impose new challenges for outlier detection algorithms, especially in time and space efficiency. In the past decade, several studies have been performed to address the problem of *distance-based* outlier detection in data streams (DODDS), which adopts an unsupervised definition and does not have any distributional assumptions on data values. Our work is motivated by the lack of comparative evaluation among the state-of-the-art algorithms using the same datasets on the same platform. We systematically evaluate the most recent algorithms for DODDS under various stream settings and outlier rates. Our extensive results show that in most settings, the `MCOD` algorithm offers the superior performance among all the algorithms, including the most recent algorithm `Thresh_LEAP`.

1. INTRODUCTION

Outlier detection in data streams [2] is an important task in several domains such as fraud detection, computer network security, medical and public health anomaly detection, etc. A data object is considered an outlier if it does not conform to the expected behavior, which corresponds to either noise or anomaly. Our focus is to detect *distance-based* outliers, which was first studied for static datasets [9]. By the *unsupervised* definition, a data object o in a generic metric space is an outlier, if there are less than k objects located within distance R from o. Several variants of distance-based outlier definition have been proposed in [4, 11, 12], by considering a fixed number of outliers present in the dataset [11], a probability density function over data values [12], or the sum of distances from the k nearest neighbors [4].

With data streams [2], as the dataset size is potentially unbounded, outlier detection is performed over a *sliding window*, i.e., a number of active data objects, to ensure computation efficiency and outlier detection in a local context.

Proceedings of the VLDB Endowment, Vol. 9, No. 12

A number of studies [3, 5, 6, 8, 10, 12, 14, 15] have been performed for Distance-based Outlier Detection in Data Streams (DODDS). Among them, Subramaniam et al. [14] studied outlier detection in a distributed setting, while the rest assumed a centralized setting. Several demonstrations of the proposed algorithms have been built [5, 8]. Finally, *exact* and *approximate* algorithms have been discussed in [3].

The most recent exact algorithms for DODDS are `exact-Storm` [3], `Abstract-C` [15], `LUE` [10], `DUE` [10], `COD` [10], `MCOD` [10], and `Thresh_LEAP` [6] in chronological order. Although each paper independently evaluated its proposed approach and offered some comparative results, a comprehensive evaluation has not been conducted. For instance, four algorithms, namely `exact-Storm`, `Abstract-C`, `COD`, and `MCOD`, were integrated into the MOA tool by Georgiadis et al. [8], but their performance evaluation was not reported. Cao et al. [6] did not compare `Thresh_LEAP` with `MCOD` because "`MCOD` does not show clear advantage over `DUE` in most cases". However, in our evaluation, we observed that `MCOD` is superior to all the other algorithms in most settings. Furthermore, the most recent approximate DODDS solution, i.e., `approx-Storm` [3], has not been compared with the most efficient exact algorithms.

Moreover, a thorough evaluation of all the algorithms using the same datasets and on the same platform is necessary to fairly compare their performances. For instance, the authors of [3, 6, 15] considered both synthetic and real-world datasets while in [10] only real-world datasets were considered. `Thresh_LEAP` was implemented on the HP CHAOS Stream Engine in [6] while `Abstract-C` in [15] was implemented in C++. Moreover, by systematically setting the parameters shared by all the algorithms, we gain insights on their performances given the data stream characteristics and the expected outlier rate. For instance, the default R and k values for each dataset affect the outlier rate and in return the algorithm performance. So far, this effect has not been considered except in [10] only for `COD`, `MCOD`.

The contributions of our study are summarized as follows:
(1) We provide a comparative evaluation of five exact and one approximate DODDS algorithms, including `exact-Storm`, `Abstract-C`, `DUE`, `MCOD`, `Thresh_LEAP`, and `approx-Storm`. Note that we did not include the results for `LUE` or `COD` as their performances are superseded by their optimized counterparts `DUE` and `MCOD` in [10], correspondingly.
(2) We implement the algorithms systematically and make our source code [1] available to the research community. The performance metrics we use are CPU time and peak memory requirement, which are consistent with the original studies.

(3) We adopt representative real-world and synthetic datasets with different characteristics in our evaluation (available on [1]). We carefully choose default parameter values to control the variables in each experiment and study each algorithm thoroughly by changing data stream characteristics, such as speed and volume, and outlier parameters.

This paper is structured as follows. In Section 2, we present definitions and notions that are used in DODDS. In Section 3, we briefly review and distinguish the six algorithms under evaluation. In Section 4, we provide our detailed evaluation results. Finally, we conclude the paper with discussions and future research directions in Section 5.

2. PRELIMINARIES

2.1 Problem Definition

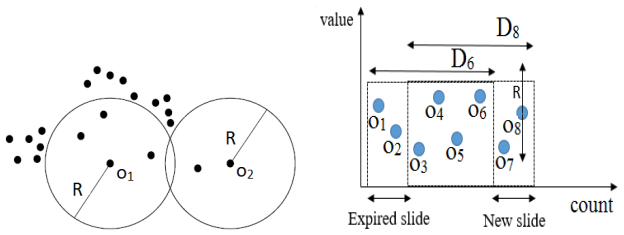
In this section, the formal definitions that are used in DODDS are presented. We first define neighbors and outliers in a static dataset.

DEFINITION 1 (NEIGHBOR). *Given a distance threshold R ($R > 0$), a data point o is a neighbor of data point o' if the distance between o and o' is not greater than R. A data point is not considered a neighbor of itself.*

We assume that the distance function between two data points is defined in the metric space.

DEFINITION 2 (DISTANCE-BASED OUTLIER). *Given a dataset D, a count threshold k ($k > 0$) and a distance threshold R ($R > 0$), a distance-based outlier in D is a data point that has less than k neighbors in D.*

A data point that has at least k neighbors is called an inlier. Figure 1(a) shows an example of a dataset from [10] that has two outliers with $k = 4$. o_1, o_2 are outliers since they have 3 and 1 neighbors, respectively.



(a) Static Dataset (b) Data Stream

Figure 1: Static and Streaming Outlier Detection

Below we define the general notions used for data streams and the DODDS problem.

DEFINITION 3 (DATA STREAM). *A data stream is a possible infinite series of data points ..., o_{n-2}, o_{n-1}, o_n, ..., where data point o_n is received at time $o_n.t$.*

In this definition, a data point o is associated with a time stamp $o.t$ at which it arrives and the stream is ordered by the arrival time. As new data points arrive continuously, data streams are typically processed in a *sliding window*, i.e., a set of active data points. There are two window models in data streams: *count-based window* and *time-based window* which are defined as follows.

DEFINITION 4 (COUNT-BASED WINDOW). *Given data point o_n and a fixed window size W, the count-based window D_n is the set of W data points: $o_{n-W+1}, o_{n-W+2}, ..., o_n$.*

DEFINITION 5 (TIME-BASED WINDOW). *Given data point o_n and a time period T, the time-based window $D(n, T)$ is the set of W_n data points: $o_{n'}, o_{n'+1}, ..., o_n$ with $W_n = n - n' + 1$ and $o_n.t - o_{n'}.t = T$.*

In this paper, we adopt the count-based window model as in the previous works [3, 6, 10, 15]. It also enables us to gain better control over the empirical evaluation, such as the window size for *scalability*. We note that it is not challenging to adapt the DODDS algorithms to the time-based window model. An extension of our source code for the time-based window model can be found on our online repository [1]. In the rest of the paper, we use the term *window* to refer to the *count-based window*. The window size W characterizes the volume of the data stream. When new data points arrive, the window slides to incorporate S new data points in the stream. As a result, the oldest S data points will be discarded from the current window. S denotes the *slide size* which characterizes the speed of the data stream. Figure 1(b) shows an example of two consecutive windows with $W = 8$ and $S = 2$. The x-axis reports the arrival time of data points and the y-axis reports the data values. When the new slide with two data points $\{o_7, o_8\}$ arrives, the window D_6 slides, resulting the expiration of two data points, i.e., $\{o_1, o_2\}$.

As the data points in a stream are ordered by the arrival time, it is important to distinguish between the following two concepts: *preceding neighbor* and *succeeding neighbor*. A data point o is a *preceding neighbor* of a data point o' if o is a neighbor of o' and expires before o' does. On the other hand, a data point o is a *succeeding neighbor* of a data point o' if o is a neighbor of o' and o expires in the same slide with or after o'. For example, in Figure 1(b), o_8 has one succeeding neighbor o_7 and four preceding neighbors, i.e., o_3, o_4, o_5, o_6. Note that an inlier which has at least k succeeding neighbors will never become an outlier in the future. Those inliers are thus called *safe inliers*. On the other hand, the inliers which have less than k succeeding neighbors are *unsafe inliers*, as they may become outliers when the preceding neighbors expire.

The Distance-based Outlier Detection in Data Streams (DODDS) is defined as follows.

PROBLEM 1 (DODDS). *Given the window size W, the slide size S, the count threshold k, and the distance threshold R, detect the distance-based outliers in every sliding window ..., $D_n, D_{n+S}, ...$*

The challenge of DODDS is that the neighbor set of an active data point may change when the window slides: some neighbors may expire and some new neighbors may arrive in the new slide. Figure 2 from [3] illustrates how the sliding window affects the outlierness of the data points. The two diagrams represent the evolution of a data stream of 1-dimensional data points. The x-axis reports the time of arrival of the data points and the y-axis reports the value of each data point. With $k = 3, W = 7$ and $S = 5$, two consecutive windows D_7 and D_{12} are depicted by dash rectangles. In D_7, o_7 is an inlier as it has 4 neighbors, i.e., o_2, o_3, o_4, o_5. In D_{12}, o_7 becomes an outlier because o_2, o_3, o_4, o_5 expired.

A naive solution is to store the neighbors of every data point and recompute the neighbors when the window slides, which can be computationally expensive. Another approach is to use incremental algorithms: each data point stores its neighbors that can prove itself an inlier or outlier. When the window slides, the neighbor information of those data points which have at least one expired neighbor will be updated.

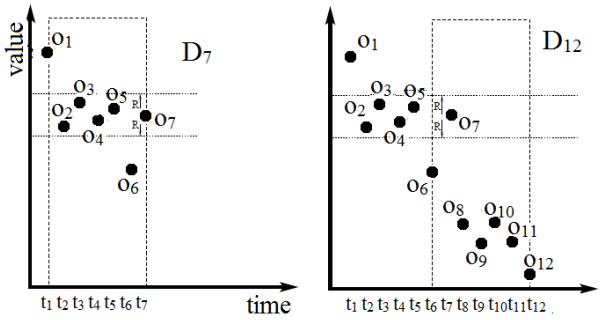


Figure 2: Example of DODDS from [3] with $k = 3$, $W = 7$, and $S = 5$

2.2 Problem Variants

Distributed vs. Centralized. In distributed systems, data points are generated at multiple nodes. These nodes often perform some computations locally and one *sink* node will aggregate the local results to detect outliers globally. To date there has not been a distributed solution for the DODDS problem. Authors of [13] and [14] studied outlier detection in distributed sensor networks. However, those studies are either inapplicable to data streams or incomplete for outlier detection. The study in [14] performed distributed density estimation in a streaming fashion and demonstrated its compatibility to multiple surveillance tasks. However, there is not guarantee for distance-based outliers detection when applied to the approximate data distributions. The study in [13] addressed distance-based outlier detection in a sensor network and aimed to reduce the communication cost between sensor nodes and the sink. However, the study assumed all data points are available at the time of computation and thus is inapplicable to data streams.

Exact vs. Approximate. While the exact DODDS solutions detect outliers accurately in every sliding window, an approximate solution is faster in the CPU time but does not guarantee the exact result. For example, `approx-Storm` [3] only searches a subset of the current window to find neighbors for each data point. As a result, it may yield false alarms for those inliers having neighbors outside the chosen subset. We will present the technical details as well as the evaluation results in the following sections.

2.3 Evaluation Metrics

CPU time and peak memory requirement are the most important utility metrics for streaming algorithms. We adopt both metrics for performance comparison, as in the original papers [3, 6, 10, 15]. The CPU time records a DODDS algorithm's processing time for each window, including the time for processing the new slide, the expired slide and outlier detection. The peak memory consumption measures the highest memory used by a DODDS algorithm for each window which includes the data storage as well as the algorithm-specific structures to incrementally maintain neighborhood information. Both metrics are studied in our evaluation by varying a range of parameters, including the window size W, the slide size S, the distance threshold R, the count threshold k, and the input data dimensionality D. In addition, we study internal algorithm-specific details relevant to their performance, such as the average length of the trigger list for `Thresh_LEAP`, and the average number of data points in micro-clusters for `MCOD`.

3. DODDS ALGORITHMS

In this section, we briefly review the five exact and one approximate DODDS algorithms. We unify the notations used in Table 1. With those notations, we summarize the time and space complexities as well as other distinctive features of each algorithm and provide a side-by-side comparison in Table 2. Although adopting different data structures and techniques, we find a common pattern among all DODDS algorithms: when the window slides, each algorithm performs three steps, i.e., processing the new slide, processing the expired slide, reporting the outliers, and the first two steps can be interchanged in order.

Symbol	Interpretation
o	A data point
$o.pn$	The list of preceding neighbors of o, equivalent to $o.nn\_before$ in [3] and $P(o)$ in [10]
$o.sn$	The number of succeeding of o, equivalent to $o.count\_after$ in [3] and $o.n_p^+$ in [10]
$o.ln\_cnt[]$	The numbers of neighbors of o in every window that o participates in, defined in [15]
$o.evil[]$	The numbers of neighbors of o in all the slides in a window, defined in [6]
PD	The list of data points that are not in micro-clusters, defined in [10]

Table 1: Frequently Used Symbols

3.1 Exact-Storm

`Exact-Storm` [3] stores data points in the current window in an index structure which supports range query search, i.e., to find neighbors within distance R of a given data point o. Furthermore, for each data point o: $o.pn$ stores up to k preceding neighbors of o and $o.sn$ stores the number of succeeding neighbors of o.

Expired slide processing. Data points in the expired slide are removed from the index but they are still stored in the preceding neighbor list of other data points.

New slide processing. For each data point o' in the new slide, a range query is issued to find its neighbors in range R. The result of the range query will be used to initialize $o'.pn$ and $o'.sn$. For each data point o in $o'.pn$, $o.sn$ is updated, i.e., $o.sn = o.sn + 1$. Then o' is inserted to the index structure.

Outlier reporting. After the previous two steps, the outliers in the current window are reported. Specifically, for each data point o, the algorithm verifies if o has less than k neighbors, including all succeeding neighbors $o.sn$ and the non-expired preceding neighbors in $o.pn$.

The advantage of `exact-Storm` is that the succeeding neighbors of a data point o are not stored since they do not expire

Algorithm	Time Complexity	Space Complexity	Neighbor Search	Neighbor Store	Potential Outlier Store
exact-Storm	$O(W \log k)$	$O(kW)$	Index per *window*	$o.pn$, $o.sn$	No
approx-Storm	$O(W)$	$O(W)$	Index per *window*	$o.frac\_before$, $o.sn$	No
Abstract-C	$O(W^2/S)$	$O(W^2/S + W)$	Index per *window*	$o.ln\_cnt[]$	No
DUE	$O(W \log W)$	$O(kW)$	Index per *window*	$o.pn$, $o.sn$	*event queue*
MCOD	$O((1-c)W \log((1-c)W) + kW \log k)$	$O(cW + (1-c)kW)$	Micro-clusters	Micro-clusters; $o.pn$, $o.sn$ if $o \notin$ clusters	*event queue*
Thresh_LEAP	$O(W^2 \log S/S)$	$O(W^2/S)$	Index per *slide*	$o.evil[]$	*Trigger list* per *slide*

Table 2: Comparison of DODDS Algorithms

before o does. However, the algorithm stores k preceding neighbors for o without considering that o might have a large number of succeeding neighbors and thus is a safe inlier. As a result, it is not optimal in memory usage. In addition, because the expired preceding neighbors are not removed from the list, retrieving the active preceding neighbors takes extra CPU time.

3.2 Abstract-C

Abstract-C [15] keeps the neighbor count for each object in every window it participates in. It also employs an index structure for range query. Instead of storing a list of preceding neighbors or number of succeeding neighbors for each data point o, a sequence $o.lt\_cnt[]$ is used to store the number of neighbors in every window that o participates in. The intuition of Abstract-C is that the number of windows that each point participates in is a constant, i.e., W/S, if W is divisible by S. For simplicity, we only consider the case W/S is an integer. As a result, the maximum size of $o.ln\_cnt[]$ is W/S. For example, let's consider o_3 in Figure 2. If $W = 3$ and $S = 1$, o_3 participates in three windows D_3, D_4 and D_5. In D_3, o_3 has one neighbor o_2, and o_2 is still a neighbor in D_4, $o_3.lt\_cnt[] = [1, 1, 0]$. In D_4, o_3 has two neighbors because of a new neighbor o_4 and o_4 is still a neighbor in D_5, $o_3.lt\_cnt[] = [2, 1]$. And in the last window D_5, o_3 has a new neighbor o_5, o_2 expired, $o_3.lt\_cnt[] = [2]$.

Expired slide processing. In addition to removing expired data points from the index, for each active data point o, the algorithm removes the first element $o.lt\_cnt[0]$ that corresponds to the last window.

New slide processing. For each data point o' in the new slide, a range query in the index structure is issued to find neighbors for it in range of R, $o'.lt\_cnt[]$ is initialized based on the result set of the range query. For each neighbor o, $o.ln\_cnt[]$ is updated to reflect o' in those windows where both o and o' participate in. Then o' is added to the index.

Outlier reporting. After the previous two steps, the algorithm checks for every active data point o, if it has less than k neighbors in the current window, i.e., $o.ln\_cnt[0] < k$.

One advantage of Abstract-C over exact-Storm is that it does not spend time on finding active preceding neighbors for each data point. However, the memory requirement heavily depends on the input data stream, i.e., W/S. For example, the space for storing $ln\_cnt[]$ for each data point o would be very high for small slide size S.

3.3 Direct Update of Events - DUE

The intuition of DUE [10] is that not all active data points are affected by the expired slide: only those who are neighbors with the expired data points need to be updated. DUE also employs an index structure which supports range query search to store data points. A priority queue called *event queue* stores all the unsafe inliers. The data points in the event queue are sorted in increasing order of the smallest expiration time of their preceding neighbors. An outlier list is created to store all outliers in the current window.

Expired slide processing. Once the expired data points are removed from the index, the event queue is polled to update the neighbor list of those unsafe inliers whose neighbors expired. If an unsafe inlier becomes an outlier, it is removed from the event queue and added to the outlier list.

New slide processing. For each data point o in the new slide, a range query search is issued to find all neighbors in range of R. The number of succeeding neighbors $o.sn$ is initialized from the result set and only $k - o.sn$ most recent preceding neighbors are stored in $o.pn$. Then o is added to the index. If o is an unsafe inlier, it is added to the event queue. On the other hand, if o has less than k neighbors, it is added to the outlier list. For each neighbor o', the number of its succeeding neighbors $o'.sn$ is increased by 1. A neighbor o' that is an outlier previously may become an inlier in the current window as the number of its succeeding neighbors increases and o' will be removed from the outlier list then added to the event queue.

Outlier reporting. After the new slide and the expired slide are processed, the data points in the outlier list are reported.

The event queue employed by DUE has an advantage for efficient re-evaluation of the inlierness of the data points when the window slides. On the other hand, it requires extra memory and CPU time to maintain sorted.

3.4 Micro-Cluster Based Algorithm - MCOD

Range queries can be expensive, especially when carried out on large datasets and for every new data point. MCOD [10] stores the neighboring data points in micro-clusters to eliminate the need for range queries. A micro-cluster is composed of no less than $k + 1$ data points. It is centered at one data point and has a radius of $R/2$. According to the triangular inequality in the metric space, the distance between every pair of data points in a micro-cluster is no greater than R. Therefore, every data point in a micro-cluster is an inlier. Figure 3 shows an example of three micro-clusters, and data points in each cluster are represented by different symbols. Some data points may not fall into any micro-clusters. They can be either outliers or inliers, e.g., having neighbors from separate micro-clusters. Such data points are stored in a list called PD. MCOD also employs an event queue to store unsafe inliers that are not in any clusters. Let $0 \leq c \leq 1$ denote the fraction of the window stored in micro-clusters then the number of data points in PD is $(1 - c)W$.

Expired slide processing. When the current window slides, the expired data points are removed from micro-clusters and PD. The event queue is polled to update the

unsafe inliers, similarly to DUE. If a micro-cluster has less than $k+1$ data points, it is dispersed and the non-expired members are processed as new data points.

New slide processing. For each data point o, o may be added to an existing micro-cluster, become the center of its own micro-cluster, or added to PD and the event queue. If o is within distance $R/2$ to the center of a micro-cluster, o is added to the closest micro-cluster. Otherwise, MCOD searches in PD for $o's$ neighbors within distance $R/2$. If at least k neighbors are found in PD, these neighbors and o form a new micro-cluster with o as the cluster center. Otherwise, o is added to PD, and the event queue if it is an unsafe inlier.

Outlier reporting. After the new slide and expired slide are processed, the data points in PD that have less than k neighbors are reported as outliers.

One advantage of MCOD is that it effectively prunes the pair-wise distance computations for each data point's neighbor search, utilizing the micro-clusters centers. The memory requirement is also lowered as one micro-cluster can efficiently capture the neighborhood information for each data point in the same cluster.

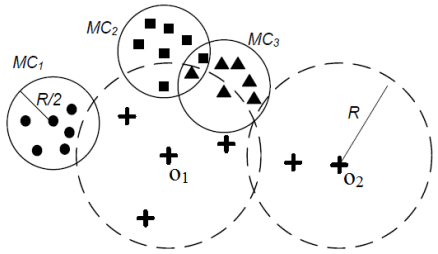


Figure 3: Example micro-clusters with $k = 4$ [10]

3.5 Thresh_LEAP

Thresh_LEAP [6] mitigates the expensive range queries with a different approach: Data points in a window are not stored in the same index structure and each slide has a separate, smaller index. As a result, this design reduces the range search cost and facilitates the *minimal probing principle*. Intuitively, the algorithm searches for the succeeding neighbors first for each data point, and subsequently the preceding neighbors per slide in a reverse chronological order. Each data point o maintains the number of neighbors in every slide in $o.evil[]$ and the number of succeeding neighbors in $o.ns$. Each slide has a *trigger list* to store data points whose outlier status can be affected by the slide's expiration.

New slide processing. For each data point o in the new slide, Thresh_LEAP adopts the *minimal probing* principle by finding o's neighbors in the same slide. If less than k neighbors are found, the *probing* process continues to the previous slide and so on, until k neighbors are found or all slides are probed. $o.evil[]$ and $o.ns$ are updated after probing and o is added to the trigger list of each probed slide. Figure 4 shows an example of probing operation when o arrives in the new slide. In order to find neighbors for data point o, the order of slides that will be probed is S_4, S_3, S_2, S_1.

Expired slide processing. When a slide S expires, the index of S is discarded and the data points in the trigger list of S are re-evaluated. For each data point o in the trigger list, the entry in $o.evil[]$ corresponding to S is removed. If it has less than k active neighbors, the algorithm probes the succeeding slides which have not been probed before. It is not needed to probe preceding slides which were skipped in the initial probing for o as those slides already expired. Figure 5 shows an example of re-probing operation for o when slide S_1, S_2 expired. The order of slides that will be probed is S_5, S_6.

Outlier reporting. After the previous two steps, each data point o is evaluated by summing up its preceding neighbors in $o.evil[]$ and succeeding neighbors in $o.sn$.

Compared to non-cluster DODDS algorithms, one advantage of Thresh_LEAP is in CPU time, thanks to the minimal probing principle and the smaller index structure per slide to carry out range queries. However, it incurs memory inefficiency when the slide size S is small, as Abstract-C.

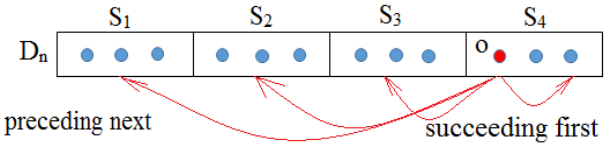


Figure 4: Probing for new data point o.

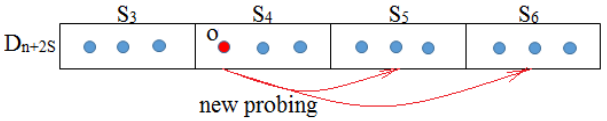


Figure 5: Re-probing as preceding neighbors expire.

3.6 Approx-Storm

Approx-Storm [3] is an approximate algorithm for DODDS. It adapts exact-Storm with two approximations. The first approximation consists in reducing the number of data points stored in each window. Up to ρW safe inliers are preserved for each window, where ρ is a predefined value $0 < \rho < 1$. The second approximation consists in reducing the space for neighbor store for each data point o, storing only a number, i.e., $o.frac\_before$, which is the ratio between the number of o's preceding neighbors which are safe inliers to the number of safe inliers in the window. The assumption is that the majority of the data points are safe inliers and they are distributed uniformly within the window.

Expired slide processing. Similar to exact-Storm, expired data points are removed from the index structure.

New slide processing. For each new data point o', a range query is issued to find its neighbors in range R. The result of the range query will be used to initialize $o'.frac\_before$ and $o'.sn$. For each o in the range query result, $o.sn$ is incremented. Then o' is inserted to the index structure. The number of safe inliers stored is controlled not to exceed ρW by randomly removing safe inliers from the index structure.

Outlier reporting. After the previous two steps, the algorithm verifies if the estimated number of neighbors for o, i.e., $o.frac\_before * (W - t + o.t) + o.sn$, is less than k.

The advantage of approx-Storm is that it does not store the preceding neighbors for each data point and only a portion of safe inliers are stored for neighbor approximation. Therefore, the time for processing the expired data points is very small compared to the other algorithms.

4. EXPERIMENTS

4.1 Experimental Methodology

Originally, the experiments in [3, 6, 10, 15] were carried out with different programming languages and on different platforms, e.g., `Thresh_LEAP` [6] was implemented on HP CHAOS Stream Engine, `MCOD` [10] and `DUE` [10] were implemented in C++. For fair evaluation and reproducibility, we implemented all the algorithms in Java and created an online repository [1] for the source code and sample datasets. Our experiments were conducted on a Linux machine with a 3.47 GHz processor and 15 GB Java heap space. We use M-Tree [7] to support range queries in all the algorithms except for `MCOD` as in [10].

Datasets. We chose the following real-world and synthetic datasets for our evaluation. **Forest Cover** (FC) contains 581,012 records with 55 attributes. It is available at the UCI KDD Archive[1] and was also used in [10]. **TAO** contains 575,648 records with 3 attributes. It is available at Tropical Atmosphere Ocean project[2]. A smaller TAO dataset was used in [3, 10]. **Stock** contains 1,048,575 records with 1 attribute. It is available at UPenn Wharton Research Data Services[3]. A similar stock trading dataset was used in [6]. **Gauss** contains 1 million records with 1 attribute. It is synthetically generated by mixing three Gaussian distributions. A similar dataset was used in [3, 15]. **HPC** contains 1 million records with 7 attributes, extracted from the Household Electric Power Consumption dataset at the UCI KDD Archive. **EM** contains 1 million records with 16 attributes, extracted from Gas Sensor Array dataset at the UCI KDD Archive. We summarize our observations across multiple datasets in the experiments below and refer readers to our technical report [1] for detailed figures with the last two datasets.

Dataset	Size	Dim.	W	S	Outlier Rate
FC	581,012	55	10,000	500	1.00%
TAO	575,648	3	10,000	500	0.98%
Stock	1,048,575	1	100,000	5,000	1.02%
Gauss	1,000,000	1	100,000	5,000	0.96%

Table 3: Default Parameter Setting

Default Parameter Settings. There are four parameters: the window size W, the slide size S, the distance threshold R, and the neighbor count threshold k. W and S determine the volume and the speed of data streams. They are the major factors that affect the performance of the algorithms. The default values of W and S are set accordingly for two smaller datasets and two larger datasets, provided in Table 3. The values of k and R determine the outlier rate, which also affect the algorithm performance. For example, memory consumption is related to k as all the algorithms store information regarding k neighbors of each data point. For fairness of measurement, the default value of k is set to 50 for all datasets. To derive comparable outlier rate across datasets as suggested in [3, 10], the default value of R is set to 525 for *FC*, 1.9 for *TAO*, 0.45 for *Stock*, and 0.028 for *Gauss*. Unless specified otherwise, all the parameters take on their default values in our experiments. Besides those parameters, we also vary the dimensionality of input data streams for *FC* dataset. Because *FC* contains attributes with different range of values, we randomly select a number of attributes for each experiment and average the results from 10 runs.

Performance Measurement. We measured the CPU time of all the algorithms for processing each window with ThreadMXBean in Java and created a separate thread to monitor the Java Virtual Machine memory. Measurements averaged over all windows were reported in the results.

4.2 Varying Window Size

We first evaluate the performance of all the algorithms by varying the window size W. Figure 6 and 9 depict the resulting CPU time and memory, respectively. When W increases, the CPU time and the memory consumption are expected to increase as well.

CPU Time. As shown in Figure 6, when W increases, the CPU time for each algorithm increases as well, due to a larger number of data points to process in every window, with an exception of `MCOD` with *Gauss* data. `Exact-Storm`, `Abstract-C`, and `DUE` have similar performances across different datasets, while `Thresh_LEAP` and `MCOD` are shown to be consistently more efficient.

`MCOD` incurs the lowest CPU time among all the algorithms, as a large portion of data, i.e., inliers, can be stored in micro-clusters. Adding and removing data points from micro-clusters are very efficient as well as carrying out range queries, compared to index structures used by other algorithms. We also observe that the CPU time of `MCOD` decreases when increasing W for *Gauss* dataset as in Figure 6(d) and it's much higher compared to other datasets. The reason is the data points in *Gauss* are sequentially, independently generated and hence tend to have fewer neighbors when W is small, e.g., when $W = 10K$. As can be seen in Figure 7, when $W = 10K$ the majority of *Gauss* data points of each window (over 99%) do not participate in any micro-cluster. In that case, `MCOD`'s CPU time is dominated by maintaining data points in the event queue and linear neighbor search for each data point due to lack of micro-clusters. Though `DUE` suffers from the event queue processing as well, it shows a slight advantage over `MCOD` when $W = 10K$ in Figure 6(d) as M-Tree is used for neighbor search. The advantage of micro-clusters can be clearly observed when W increases to $50K$ and higher. As more data points participate in micro-clusters, CPU time for `MCOD` is reduced, enlarging the performance gap between `MCOD` and other algorithms.

On the other hand, `Thresh_LEAP` stores each slide in a M-Tree and probes the most recent slides first for each incoming data point. By leveraging a number of smaller trees, it incurs less CPU time than `Abstract-C`, `exact-Storm`, and `DUE`. We observe in Figure 6 that the CPU time of `Thresh_LEAP` increases simultaneously with W for *FC* and *Gauss* but stays stable for *TAO* and *Stock* datasets. The reason is that for each slide `Thresh_LEAP` maintains a *trigger list* containing data points whose inlier status needs to be re-evaluated upon the slide's expiration. As shown in Figure 8, the average length of trigger list per slide doesn't grow with *TAO* and *Stock* as much as it does with *FC* and *Gauss*, resulting higher re-evaluation cost for the latter two datasets. We can also conclude from Figure 8 that *TAO* and *Stock* exhibit high local continuity as on average each data point has

[1] http://kdd.ics.uci.edu
[2] http://www.pmel.noaa.gov
[3] https://wrds-web.wharton.upenn.edu/wrds/

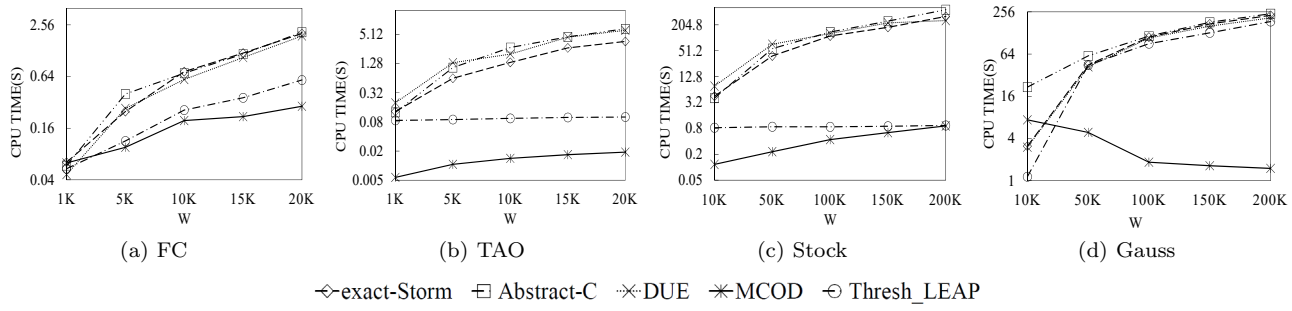


Figure 6: CPU Time - Varying Window Size W

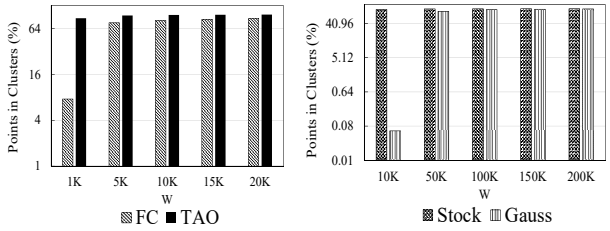


Figure 7: Average Percentage of Data Points in Micro-Clusters for `MCOD` - Varying Window Size W

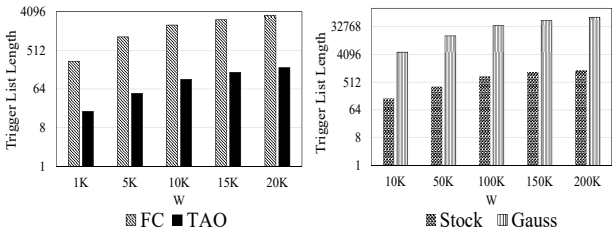


Figure 8: Average Length of Trigger List for `Thresh_LEAP` - Varying Window Size W

sufficient neighbors after probing a small number of slides.

Peak Memory. Figure 9 reports the peak memory consumption of the evaluated algorithms when varying the window size W. As every algorithm stores data points as well as their neighborhood information in the current window, the memory requirement increases with W consistently across different datasets.

We observe that `MCOD` incurs lowest memory requirement in comparison across all datasets, thanks to the memory efficient micro-clusters. The benefit of the micro-cluster structure is that it stores a set of data points that participate in a cluster of radius $R/2$ and can represent a lower bound of the R neighborhood of every member data point. That results in desirable memory saving as a large percentage of data fall into clusters, as in Figure 7.

In contrast, all other algorithms explicitly store neighborhood information for every data point, i.e., `Abstract-C`, `exact-Storm`, and `DUE`, or every slide, i.e., `Thresh_LEAP`, thus higher dependency on the window size W. `Abstract-C` has space complexity $O(W^2/S)$ as the algorithm stores for each data point the number of neighbors in every window it resides. It is clearly confirmed in Figure 9(c) that `Abstract-C` shows the fastest rate of growth in memory, quite low when $W = 10K$ and highest when $W = 200K$. Similar trend can be observed in other datasets as well. `Exact-Storm` and `DUE` have similar complexity $O(kW)$ and it can be seen that `DUE` consistently incurs lower memory cost than `exact-Storm`. The reason is `exact-Storm` stores k preceding neighbors for every data point, while `DUE` stores only $k - s_i$, where s_i represents the number of *succeeding* neighbors of data point i. We can also observe that `exact-Storm` demands more memory than `Abstract-C` when the window size is small, i.e., $W/S < k$, as in Figure 9(b). On the other hand, `Thresh_LEAP` stores a trigger list for every slide, where each data point in the list should be re-evaluated when the slide expires, yielding space complexity of $O(W^2/S)$. However, since the length of trigger list depends on the local continuity of the input stream (Figure 8), the worst case complexity doesn't always hold in practice. As can be seen in Figure 9, `Thresh_LEAP` appears to be superior to `Abstract-C` for *TAO* and *Stock* datasets, and performs similarly to `Abstract-C` for *FC* and *Gauss* datasets.

4.3 Varying Slide Size

We further examined the algorithms' performances when varying the slide size S to change the speed of the data streams, e.g., from 1% to 100% of the window size W as in [15]. When S increases, more data points arrive and expire at the same time, while the number of windows that a point participates in decreases. $S = W$ is an extreme case, in which every data point resides only in one window. All data points within the current window are removed when the window slides and none would affect the outlier status of data points in adjacent windows, i.e., no preceding neighbors. Figure 10 and 12 depict the results of CPU time and memory requirement, respectively.

CPU Time. As shown in Figure 10, `MCOD` incurs lowest CPU time in most cases, while `exact-Storm`, `DUE` and `Abstract-C` incur highest CPU time and behave similarly across all datasets; `Thresh_LEAP` shows a different trend from the other algorithms. When S increases from $1\%W$ to $50\%W$, the CPU time of all the algorithms except `Thresh_LEAP` increases as there are more new data points as well as expired data points to process when the window slides. When further increasing S from $50\%W$ to $100\%W$, we observe a drop in CPU time for `exact-Storm`, `Abstract-C`, `DUE`, and `MCOD` in most cases. That is because when $S = W$ the M-Tree for the entire window can be discarded as the window slides, instead of sequentially removing expired data points one by one as is done when $S < W$, thus reducing the processing time for expired data points. We notice that `MCOD`'s CPU time contin-

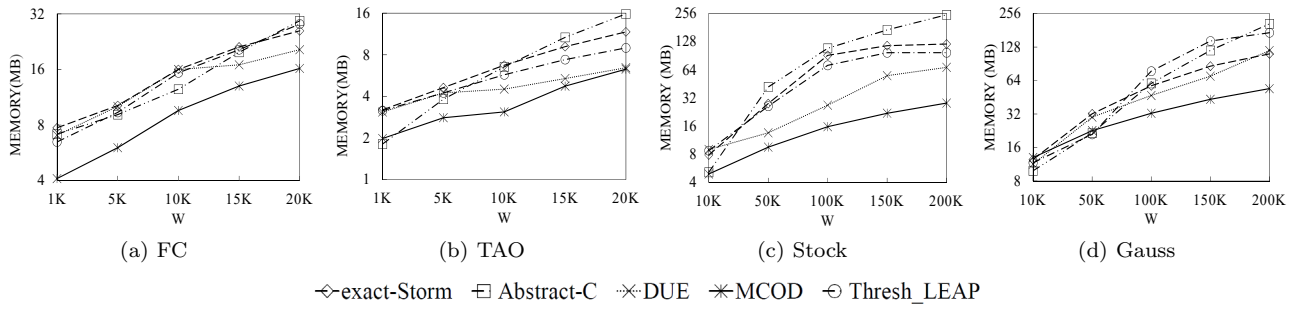


Figure 9: Peak Memory - Varying Window Size W

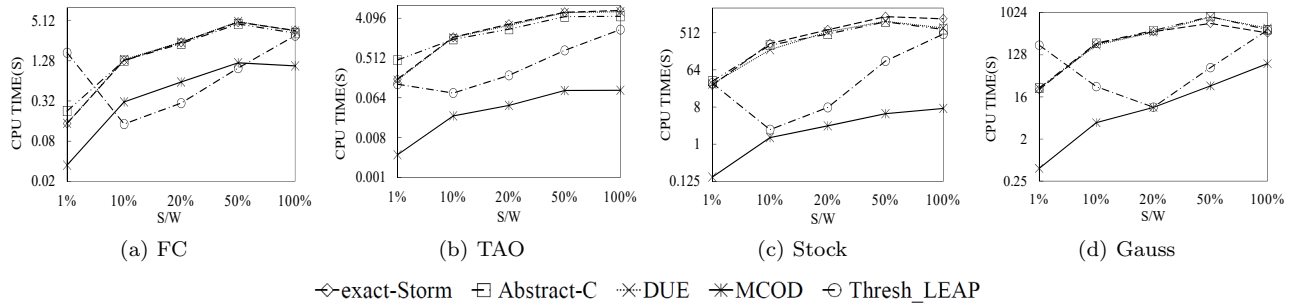


Figure 10: CPU Time - Varying slide size S

ues to grow between $S = 50\%W$ and $S = 100\%W$ for *Stock* and *Gauss* datasets. We believe that when window size is large, i.e., $W = 100K$, the CPU time needed for `MCOD` to process 50% new data points outweighs the saving from discarding expired data points. Based on the above observations, we conclude that processing arriving data points in `MCOD` does not scale as well as that of expired data points, which can be further improved in future work. On the other hand, `Thresh_LEAP` behaves differently from the other methods. For every dataset, `Thresh_LEAP`'s CPU time first decreases as S increases and starts to increase after a turning point, e.g., when $S = 10\%W$ or $20\%W$. The reason is when the slide size is small, more slides need to be probed in order to find neighbors for each new data point, resulting in high processing time for new data points as well as high re-evaluation time when each slide expires, due to longer trigger list per slide as in Figure 11. As S increases, fewer slides need to be probed and the average trigger list becomes shorter, thus reducing the overall CPU time. When S is further increased beyond $10\%W$ for *FC*, *TAO*, and *Stock*, and $20\%W$ for *Gauss*, the CPU performance of `Thresh_LEAP` shows inefficiency caused by maintaining larger M-Trees (one per slide). Eventually when $S = W$, `Thresh_LEAP` yields similar CPU time to `exact-Storm`, `Abstract-C`, and `DUE`.

Peak Memory. Figure 12 depicts the peak memory requirements of all the algorithms. We observe that when S increases, the memory cost of all the algorithms decreases. When $S = W$, every data point does not have any preceding neighbors and only participates in one window which is why all the algorithms show similar memory consumptions. `MCOD` continues to be superior to other algorithms in memory efficiency thanks to micro-clusters. It shows a decreasing trend as S increases, as a result of fewer preceding neighbors to store for each data point in the event queue, similar

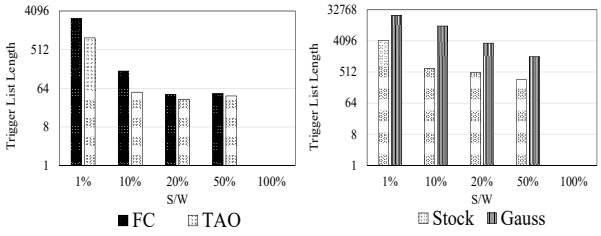
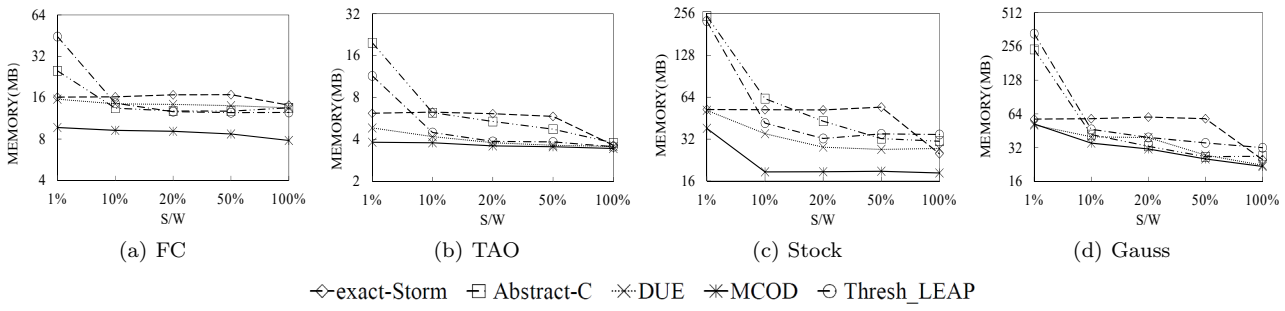


Figure 11: Average Length of Trigger List in `Thresh_LEAP` - Varying Slide Size S

to `DUE`. `Exact-Storm` shows stable memory consumption until S is increased to $100\%W$, when the number neighbors of each data point to store drops from k to 0. `Abstract-C` and `Thresh_LEAP` perform similarly, showing a high reduction in memory from $S = 1\%W$ to $10\%W$ and slower reduction as S further increases. In *FC* and *Gauss* datasets, `Thresh_LEAP` shows a slightly higher memory consumption than `Abstract-C` when $S = 1\%W$. The reason is the trigger list for each slide is long when S is small and `Thresh_LEAP` stores a neighbor map for each data point, i.e., the number of neighbors in each slide, which is proportional to W/S.

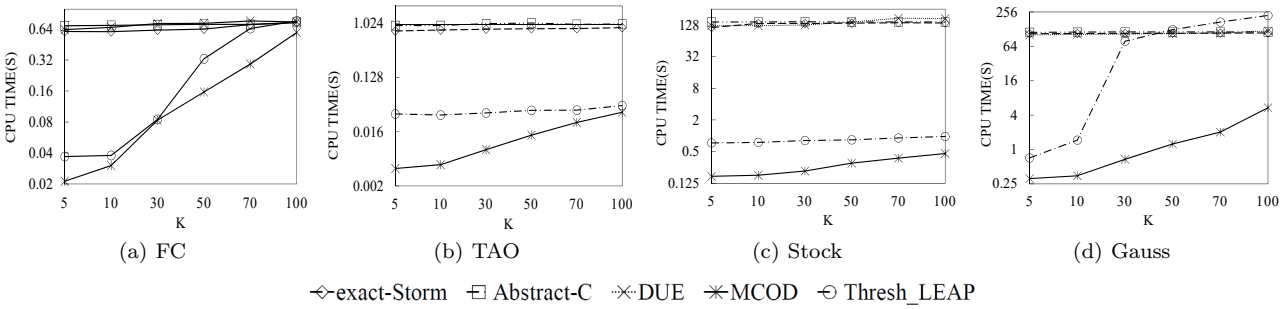
4.4 Varying k

The neighbor count threshold k is also an important parameter affecting the outlier rate as well as the space required for neighbor store. Figure 13 and 14 depict the resulting CPU time and peak memory, respectively. When k increases, the memory consumption of all the algorithms except `Abstract-C` is expected to increase as well. The CPU time and memory consumption of `Abstract-C` is expected to be stable because $o.ln\_cnt$ only depends on W and S.



(a) FC (b) TAO (c) Stock (d) Gauss

Figure 12: Peak Memory - Varying Slide Size S



(a) FC (b) TAO (c) Stock (d) Gauss

Figure 13: CPU Time - Varying K

CPU Time. As shown in Figure 13, when k increases, the CPU time of `Abstract-C`, `DUE`, and `exact-Storm` does not show much variation, as expected. As described in the previous sections, those algorithms do not heavily depend on k. `MCOD` incurs the lowest CPU time among all the algorithms consistently. The CPU time of `MCOD` increases when k increases, as fewer data points fall in micro-clusters. `Thresh_LEAP` behaves differently across the 4 datasets. Its CPU time is stable for *TAO* and *Stock* and increases for *FC* and *Gauss*. With *Gauss*, when $k \geq 70$, the CPU time of `Thresh_LEAP` is the highest among all the algorithms. As k increases, `Thresh_LEAP` needs to probe more slides to find k neighbors for each data point. With *TAO* and *Stock*, the additional probing due to the increase of k is not as significant as in *FC* and *Gauss*, since more neighbors can be found locally in these two datasets.

Peak Memory. Figure 14 depicts the peak memory requirements of all the algorithms. We observe that the peak memory of `Abstract-C` is stable as expected and the rest algorithms show increasing memory requirements as the storage for neighbors is dependent on k. `DUE` and `exact-Storm` store a longer preceding neighbor list for each data point when k increases. `MCOD` continues to be superior to all other algorithms in memory efficiency. It shows an increasing trend as k increases, due to a larger number of data points which are not in any micro-clusters. Furthermore, in `MCOD`, each data point in PD stores more preceding neighbors when k increases. When $k = 100$ with *FC*, `MCOD` requires more memory than `Abstract-C`. `Thresh_LEAP`'s memory requirement is stable with *TAO* and *Stock*, but increases with *FC* and *Gauss*. The increase in the memory requirement matches with the increase in CPU time with *FC* and *Gauss* since each data point has a longer neighbor count list.

4.5 Varying R

The distance threshold R is also an important parameter affecting the range query and the outlier rate. We vary R with respect to its default values. Table 4 shows the corresponding outlier rates for each dataset when varying R. When R increases, every data point has more neighbors, therefore the number of outliers decreases. However, the CPU time for range queries is expected to increase. Figure 15 and 16 depict the CPU time and peak memory requirement of all the algorithms.

CPU Time. As shown in Figure 15, when R increases, due to expensive range queries, the CPU time of `Abstract-C`, `exact-Storm`, and `DUE` increases. Unlike the previous algorithms, the CPU time of `MCOD` and `Thresh_LEAP` initially increases and then decreases when R is large enough. The initial increase of `MCOD`'s CPU time is due to sorting, adding, and removing a larger number of neighbors for each data point in PD. As shown in Table 4, the outlier rates when R is less than 10% of its default value are very high. In those cases, very few data points are in micro-clusters, and the efficiency of micro-clusters is not utilized by the majority of the data points within the current windows. As a result, `MCOD`'s CPU time is even higher than the other algorithms in those cases. When R is further increased, the number of data points in micro-clusters increases. Because of the efficiency in checking and maintaining inlier status of the micro-clusters, the CPU time of `MCOD` decreases and becomes superior to all the other algorithms. When R is further increased, it takes more time to find all the neighbors of a data point. That causes the increase in the CPU time of `MCOD` with *Stock*, when R increases from 500% to 1000% of the default value. `Thresh_LEAP` behaves quite similarly to `MCOD`. When R increases initially, each data point has a larger number of preceding neighbors and each slide has a

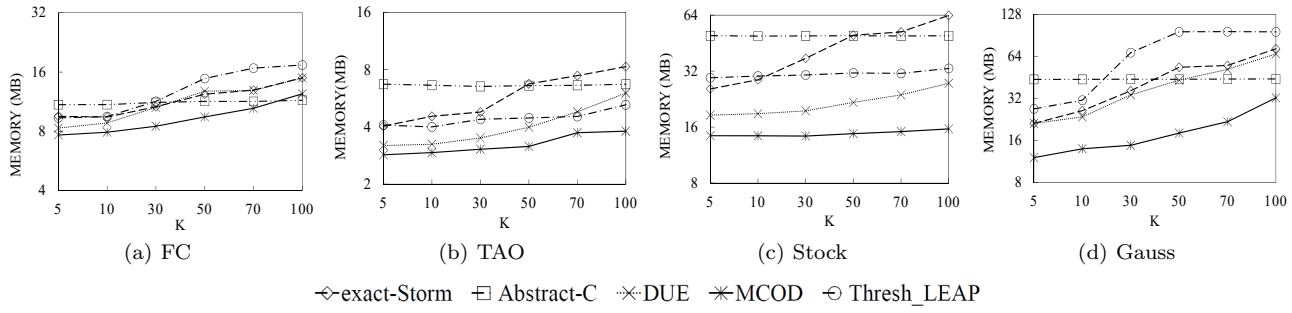


Figure 14: Peak Memory - Varying K

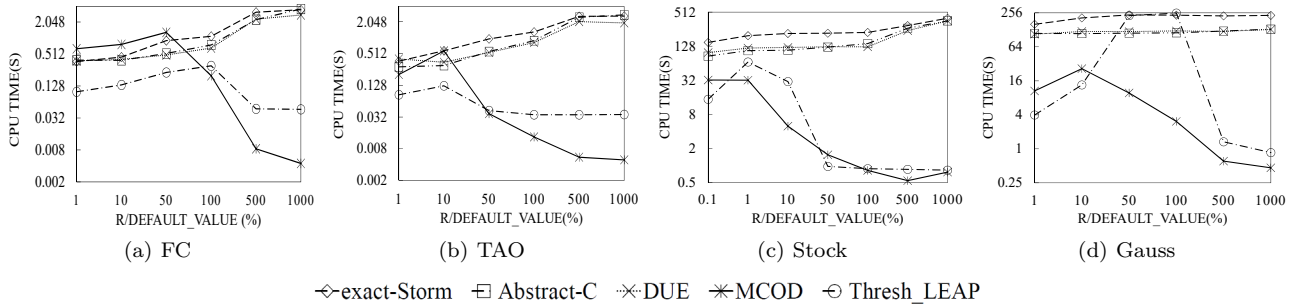


Figure 15: CPU Time - Varying R

R/Default_R (%)	FC(%)	TAO (%)	Stock(%)	Gauss(%)
1	100.00	99.60	45.95	100.00
10	100.00	54.34	6.80	39.50
50	10.79	3.44	2.09	3.18
100	1.00	0.98	1.01	0.98
500	0.00	0.19	0.17	0.29
1000	0.00	0.11	0.08	0.23

Table 4: Outlier Rate - Varying R

longer trigger list, thus incurring the higher CPU time for `Thresh_LEAP`. When R is further increased, each data point can find enough neighbors with a small number of probes, which causes the trigger lists to be shorter. Therefore, the CPU time of `Thresh_LEAP` decreases.

Peak Memory. As shown in Figure 16, in most cases, `MCOD` requires the lowest memory among all the algorithms. The memory requirement of all the algorithms first increases then becomes stable or decreases. Initially, when R increases, because each data point has more neighbors, the peak memory increases. When we further increase R, i.e., from 100% to 1000% with *FC* and *TAO*, from 50% to 1000% with *Stock* and *Gauss* of the default value, the result of range queries does not expand much, which yields the memory of `exact-Storm` and `Abstract-C` to be stable. For each data point o in `DUE`, because the number of succeeding neighbors $o.sn$ increases, the number of stored preceding neighbors in $o.pn$ decrease. Therefore the memory requirement of `DUE` decreases. Similarly, in `Thresh_LEAP`, the trigger list of each slide and the preceding neighbor counts of each data point, i.e., $o.evil[]$, are shorter, and therefore the memory requirement of `Thresh_LEAP` decreases. When R increases, each data point has a higher chance to have enough neighbors within distance of $R/2$ to form a new micro-cluster. It causes the increase in the number of data points in micro-clusters, thus incurring the decrease in memory requirement of `MCOD`.

4.6 Varying Dimensionality

In addition, we also vary the input data dimensionality D and analyze its impact on the performance of the algorithms. With *FC* dataset, D is varied from 1 to 55. We run 10 times then average the results with D attributes that are randomly selected each time. Figure 17 depicts the CPU time and memory requirement of all the algorithms. When D increases, the distance computation requires more time. As the distances between data points are larger, the outlier rate is expected to increase.

CPU Time. As shown in Figure 17(a), when D increases, the CPU time of `exact-Storm`, `Abstract-C`, `DUE` decreases, and that of `MCOD` and `Thresh_LEAP` increases. `MCOD` is still superior to all the other algorithms in CPU time in all cases. Because each data point has fewer neighbors when D increases, updating neighbor information when the window slides takes less time. The decrease in CPU time for processing neighbors overweights the increase in CPU time for distance computations for `Abstract-C`, `exact-Storm`, and `DUE`. `MCOD` has fewer data points in micro-clusters when D increases because of larger distances between the data points, and therefore its CPU time increases. Similarly, the CPU time of `Thresh_LEAP` increases when D increases. One reason is that each data point has to probe more slides to find neighbors and computing pairwise distances takes more time.

Peak Memory. As shown in Figure 17(b), when D increases, the memory consumption of all the algorithms increases. When D is small, e.g., from 1 to 4, `DUE` requires the lowest memory, and when D is further increased, `MCOD` is the most efficient in memory. When D increases, the storage for the attribute values of the data points increases

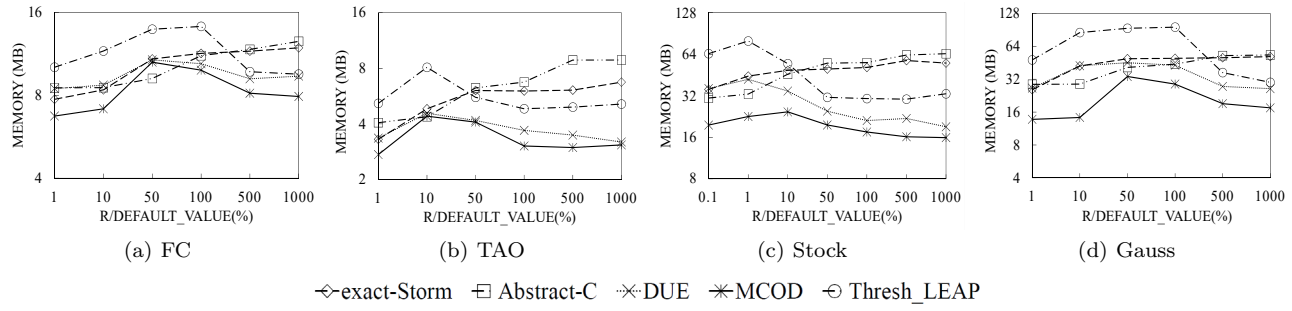


Figure 16: Peak Memory - Varying R

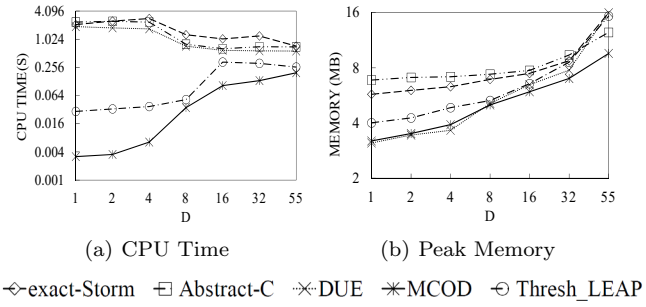


Figure 17: Varying Dimensionality D

linearly with D. In `Abstract-C` and `exact-Storm`, the memory consumption is mostly changed by the storage for attribute values, therefore it increases along with the increase in D. In `MCOD`, because the distances between the data points increase, more data points are not in micro-clusters, thus resulting larger CPU time. With D is from 1 to 4, `DUE` consumes the lowest memory because each data point has enough succeeding neighbors and has to store very few preceding neighbors. When D increases from 4 to 55, in `DUE`, each data point stores more preceding neighbors and it is not as efficient as the micro-clusters in `MCOD`.

4.7 Approximate Solution

We also evaluate the approximate solution for DODDS by varying the parameter ρ in `approx-Storm` from 0.01 to 1, where $\rho = 1$ determines *all* safe inliers of the window are stored. As only ρW of the total safe inliers in each window are preserved and the algorithm assumes that data points distributed uniformly within the window, `approx-Storm` may *miss* or *falsely report* some outliers in each window. With *TAO* and *Gauss* datasets, we summarize the precision and recall measures, with the ground truth provided by `exact-Storm`, in Table 5. Results with other datasets are presented in our technical report [1]. As is shown, the recall for both datasets increases when increasing ρ. It is not surprising that the safe inliers are not uniformly distributed within the window in practice. The algorithm tends to overestimate the number of preceding neighbors when only a subset of the safe inliers are preserved for approximation. Similar to the findings in [3], the precision initially increases and then drops for both datasets. The reason is with larger sample sizes the algorithm underestimates the number of neighbors for those inliers. We find the dataset-specific ρ value that

ρ		0.01	0.05	0.1	0.5	1
TAO	Precision (%)	96	98	97	96	96
	Recall (%)	68	73	77	80	80
Gauss	Precision (%)	36	56	75	56	50
	Recall (%)	96	99	99	100	100

Table 5: `Approx-Storm` Precision and Recall

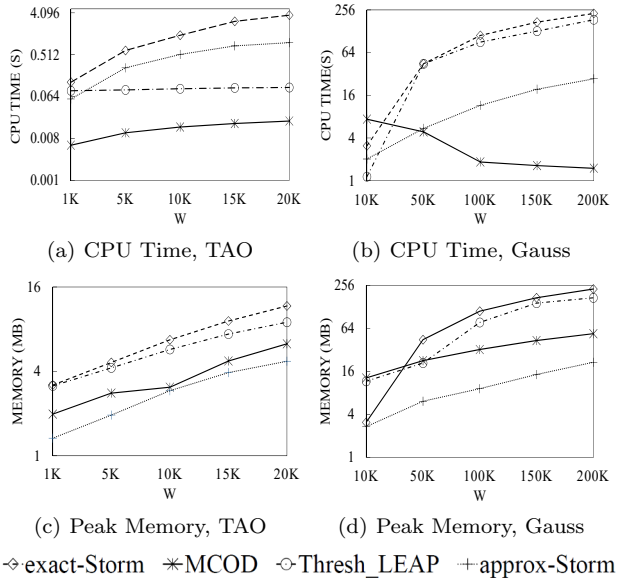


Figure 18: `Approx-Storm` vs. Exact Algorithms

achieves the optimal estimation for the precision measure, i.e., $\rho = 0.05$ for *TAO* and $\rho = 0.1$ for *Gauss*.

We further compare `approx-Storm` to the most efficient exact algorithms, i.e., `MCOD` and `Thresh_LEAP`. We set $\rho = 0.1$ for both of the datasets to reach above 75% precision and recall. As can be seen in Figure 18(a) and 18(b), the CPU time of `approx-Storm` is superior to `exact-Storm` and is quite comparable to that of `MCOD` and `Thresh_LEAP`. It is expected as `approx-Storm` does not have to update the neighbor information for each data point when the window slides. As shown in Figure 18(c) and 18(d), the peak memory requirement of `approx-Storm` is consistently lower than `MCOD` and `Thresh_LEAP` as W increases. It is also expected as for each data point o, only two numbers $o.sn$ and $o.frac\_before$ are stored for neighbor information.

5. CONCLUSION AND DISCUSSIONS

In this paper, we performed a comprehensive comparative evaluation of the state-of-the-art algorithms for distance-based outlier detection in data streams (DODDS). We reviewed the most recent exact and approximate algorithms and systematically evaluated their performances in CPU time and peak memory consumption with various datasets and stream settings. Our results showed that `MCOD` [10] provides superior performance across multiple datasets in most stream settings, outperforming the most recent algorithm `Thresh_LEAP` [6]. We found that `MCOD` benefits significantly from the micro-cluster structure, which simultaneously supports both neighbor search and neighbor store. On the contrary, all the other algorithms utilize separate structures, i.e., indices, preceding neighbor lists, etc., for different functionalities. We also discovered that `Thresh_LEAP` incurs the lowest CPU time when the input data stream exhibits strong local continuity, i.e., when each data point has many neighbors within one slide. Furthermore, our results showed that `MCOD` does not scale well for processing arriving data points when the slide size is large. Considering the performance across all datasets with different outlier rates, we concluded that `MCOD` is more feasible and scalable than `Thresh_LEAP`.

The following directions can be explored in future DODDS research: 1) Storing several consecutive slides in `Thresh_LEAP` to achieve a tradeoff between minimal probing and shorter trigger lists; 2) Considering the data expiration time in micro-clusters in `MCOD` in order to minimize the number of dispersed clusters when the window slides; 3) Designing a hybrid approach which benefits from both the micro-clusters in `MCOD` and the minimal probing in `Thresh_LEAP`; 4) Designing DODDS solutions in a decentralized setting to ensure complete outlier detection while minimizing the communication cost between nodes, e.g., adapting `MCOD` to share the local cluster centers across multiple nodes. In addition, the processing of new/expired slides needs to be optimized when the window slides.

6. ACKNOWLEDGMENTS

This research has been supported in part by NSF grants IIS-1320149, CNS-1461963, the USC Integrated Media Systems Center (IMSC), and the USC Center for Interactive Smart Oilfield Technologies (CiSoft). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of the sponsors such as NSF.

7. REFERENCES

[1] Distance-based outlier detection in data streams repository. http://infolab.usc.edu/Luan/Outlier/.

[2] C. Aggarwal, editor. *Data Streams – Models and Algorithms*. Springer, 2007.

[3] F. Angiulli and F. Fassetti. Detecting distance-based outliers in streams of data. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 811–820, New York, NY, USA, 2007. ACM.

[4] F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *Knowledge and Data Engineering, IEEE Transactions on*, 17(2):203–215, Feb 2005.

[5] L. Cao, Q. Wang, and E. A. Rundensteiner. Interactive outlier exploration in big data streams. *Proc. VLDB Endow.*, 7(13):1621–1624, Aug. 2014.

[6] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 76–87, March 2014.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435, 1997.

[8] D. Georgiadis, M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous outlier detection in data streams: An extensible framework and state-of-the-art algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1061–1064, New York, NY, USA, 2013. ACM.

[9] E. M. Knorr and R. T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 392–403, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[10] M. Kontaki, A. Gounaris, A. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 135–146, April 2011.

[11] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 427–438, New York, NY, USA, 2000. ACM.

[12] M. S. Sadik and L. Gruenwald. *Database and Expert Systems Applications: 21st International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part I*, chapter DBOD-DS: Distance Based Outlier Detection for Data Streams, pages 122–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[13] B. Sheng, Q. Li, W. Mao, and W. Jin. Outlier detection in sensor networks. In *Proceedings of the 8th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '07, pages 219–228, New York, NY, USA, 2007. ACM.

[14] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 187–198. VLDB Endowment, 2006.

[15] D. Yang, E. A. Rundensteiner, and M. O. Ward. Neighbor-based pattern detection for windows over streaming data. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 529–540, New York, NY, USA, 2009. ACM.