Scalable Distance-Based Outlier Detection over High-Volume Data Streams

Lei Cao *, Di Yang†, Qingyang Wang*, Yanwei Yu‡, Jiayuan Wang*, Elke A. Rundensteiner*

*Worcester Polytechnic Institute, Worcester, MA USA
(lcao,wangqy,jwang1,rundenst)@cs.wpi.edu
†Oracle Corporation, Nashua, NH USA
di.yang@oracle.com
‡University of Science and Technology Beijing, Beijing, China
yuyanwei0530@126.com

***Abstract*—The discovery of distance-based outliers from huge volumes of streaming data is critical for modern applications ranging from credit card fraud detection to moving object monitoring. In this work, we propose the first general framework to handle the three major classes of distance-based outliers in streaming environments, including the traditional distance-threshold based and the nearest-neighbor-based definitions. Our LEAP framework encompasses two general optimization principles applicable across all three outlier types. First, our "minimal probing" principle uses a lightweight *probing* operation to gather minimal yet sufficient evidence for outlier detection. This principle overturns the state-of-the-art methodology that requires routinely conducting expensive complete neighborhood searches to identify outliers. Second, our "lifespan-aware prioritization" principle leverages the temporal relationships among stream data points to prioritize the processing order among them during the probing process. Guided by these two principles, we design an outlier detection strategy which is proven to be optimal in CPU costs needed to determine the outlier status of any data point during its entire life. Our comprehensive experimental studies, using both synthetic as well as real streaming data, demonstrate that our methods are 3 orders of magnitude faster than state-of-the-art methods for a rich diversity of scenarios tested yet scale to high dimensional streaming data.**

I. Introduction

Motivation. In recent years, both the number of mobile devices, such as smart phones, pads, and RFID equipment, and their capabilities of generating and transmitting live data have grown rapidly. As the volume and speed of data streams advance to new levels, discovering precious knowledge hidden in this data has become more critical than ever before.

Important insights extractable from such data sources are *abnormal phenomena.* Many modern applications, including credit card fraud detection, network intrusion prevention, and stock investment tactical planning, rely on finding abnormal phenomena in data streams. The basic notion of capturing abnormal phenomena in data can be traced back to initial work by Hawkins [1], which introduced the core principle still deployed for characterizing *outliers* within a set of data points. That is, the greater the distance of a data point to its neighbors, the more likely it is an outlier. Based on this foundation three main variations of distance-based outlier definitions have emerged in the literature:

- $O_{thres}^{(k,R)}$ outlier: Outliers are data points with fewer than k neighbors in the database, where a neighbor is a data point that is within a distance R [2][1].
- $O_{kmax}^{(k,n)}$ outlier: Outliers are the n data points with the highest distance values to their respective kth nearest neighbor among all data points in the database [3].
- $O_{kavg}^{(k,n)}$ outlier: Outliers are the n data points with the highest average distance to their respective k nearest neighbors [4].

Each of these three definitions has been shown to have its own scope of applicability, effectively supporting certain classes of applications [3], [5], [4]. In general, $O_{thres}^{(k,R)}$ outlier performs well in applications in which it is known apriori that an explicit behavior difference R (or a distance measure R) is critical, while $O_{kmax}^{(k,n)}$ and $O_{kavg}^{(k,n)}$ work better in situations when such a fixed threshold is either not available or is changing over time.

For example, when seeking short-term investment opportunities in the stock market, investors may look for the outlier stocks whose behavior significantly differ from that of the majority of their peer stocks. Such abnormal stocks typically are either the *hot spots* or the *forgotten treasure* in the market. Both of them may correspond to potentially excellent investment opportunities. More specifically, given two stocks s1 and s2 an investor may define a distance function to measure their difference, e.g., considering both their stock price change percentage and their company profit change percentage. Any two stocks that have the similar company profit performance but 200% difference in price change percentage will have a difference score $d(s1, s2) = |(s1.price\_gain\% - s2.price\_gain\%)$ - $(s1.profit\_gain\% - s2.profit\_gain\%)| = 2$. For this application, if the investor is confident that 2 is a good *threshold* to indicate that two stocks behave significantly different enough, she can use the $O_{thres}^{(k,R)}$ definition to find outlier stocks that are ***abnormal enough*** with 2 as the distance threshold R for neighbor search. However, if such a threshold is not known, alternatively she could utilize the $O_{kmax}^{(k,n)}$ or $O_{kavg}^{(k,n)}$ outlier definitions to find the ***top abnormal*** stocks.

In this work we thus set out to design a framework that can handle the general problem of distance-based outlier detection in streaming environments, while delivering highly scalable solutions for all three major outlier types.

[1]In the original definition introduced by [2] k represents the percentage of data points. In this work we follow the definition adopted by all streaming outlier work.

Limitations of the State-of-the-Art. The problem of detecting distance-based outliers in static datasets has been extensively studied in the literature [2], [3], [4], [5]. More recently, researchers started to look at the problem in streaming environments [6], [7]. Specifically [6] proposed a solution for detecting $O_{thres}^{(k,R)}$ outliers in count-based sliding windows. [7] improves upon this solution [6] by now supporting $O_{threh}^{(k,R)}$ outliers in time-based sliding windows. Both solutions leverage the overlap of sliding windows and thus avoid huge overhead wasted on recomputing-from-scratch at each window.

However, these existing techniques [6], [7] didn't explore the optimization opportunities enabled by the *two critical insights* below. First, they didn't exploit the fact that outliers by nature only constitute a small portion of the general stream data population (otherwise they wouldn't be called outliers after all). Thus, the outlier detection algorithms should ideally concentrate their resource utilization on strictly serving these minority outlier candidates, rather than on computing and recording neighborhoods for the general and much larger stream population. Second, the existing techniques do not take advantage of the temporal relationships among stream data points, i.e., who will survive longer in the future windows. As our experiments demonstrate, processing the data points in an intelligent time-aware order helps us to minimize the computation required for acquiring new evidence for outlier detection, achieving several orders of magnitude speed up.

Without these important optimization opportunities, the existing techniques [6], [7] cannot handle high-speed streams in real-time, say 1M tuples per second as our experiments will confirm. Yet such huge volume streams are increasingly common in modern streaming applications. As an example the US stocks market continuously receives around 1M transaction requests per second [8]. Also, existing techniques [6], [7] focused exclusively on the first simpler outlier type, namely $O_{thres}^{(k,R)}$ outlier. No existing work provides a general solution for all major types of distance-based outliers for data streams.

Proposed Solution. In this work, we present a general framework for optimizing distance-based outlier detection in high volume data streams covering all three major distance-based outlier definitions, $O_{threh}^{(k,R)}$, $O_{kmax}^{(k,n)}$ and $O_{kavg}^{(k,n)}$. We investigate the optimization opportunities missed by the state-of-the-art strategies and derive two core optimization principles.

First, we present the *minimal probing* principle that takes advantage of the *rarity* property of outliers. Unlike existing techniques [6], [7], [9], which rely on routinely conducting expensive range queries to search through the *complete neighborhoods* of all stream data points, our strategy is to minimize both the *frequency* of the *neighbor probing operation* as well as the actual *computation cycles* consumed by each search. More specifically, we only initiate the neighbor probing process for a data point when it is absolutely necessary. Second, our *probing operation* stops as soon as it has acquired the *minimally needed evidence* to identify outliers. This principle frees our proposed algorithms from having to conduct the rather expensive complete neighbor searches, such as a full range query search. Thus it saves extensive system resources otherwise dedicated to identifying and maintaining non-essential neighbor relationships among stream data points.

Second we propose the *lifespan-aware prioritization* principle. This principle utilizes the insight that the data points that arrived later in the window are guaranteed to have a more *decisive* impact on the outlier detection process compared to earlier ones. This is so because the *younger* a data point p_i is, the longer its neighbor relationships (if any) with other points will persist into the future. Since the key task for the outlier detection process is to eliminate any guaranteed *inliers*, namely those with sufficient neighbors, identifying enough *longer lasting* neighbor relationships is likely to eliminate the need for further probing for those *shorter lasting* ones. As we will show this principle guarantees that we always find the most useful neighbor relationships for outlier detection.

By exploiting these two core principles, we have designed a highly scalable outlier detection framework called LEAP. Our theoretical analysis proves that LEAP is optimal in CPU resource consumption to determine the outlier status of any data point during its entire life.

As our experimental results reveal (Sec. VII), we succeed to drive down the CPU costs by over three orders of magnitude, making the outlier detection algorithms fast enough to become practical in the truly high speed realm for exploring big data streams with high dimensions.

Contributions. Our contributions include:

1) We present the first result on efficiently supporting the major distance-based outlier classes. In particular neither the $O_{kmax}^{(k,n)}$ nor the $O_{kavg}^{(k,n)}$ outliers had been handled in the streaming outlier detection literature before.

2) We propose the *minimal probing* optimization principle, which frees detection algorithms from the burden experienced by the state-of-the-art methodologies of having to routinely conduct range query searches [6], [9], [7].

3) We introduce the *lifespan-aware prioritization* principle, which guides the outlier detection algorithms to probe neighbors for stream data points in a time-aware manner to minimize the frequency of probing operation.

4) We integrate these two principles into a general framework called LEAP, which is proven to be optimal in terms of the CPU costs for determining the outlier status of each point.

5) Our experimental studies based on real and synthetic data show that our proposed algorithms achieve three orders of magnitude performance gain compared to the state-of-the-art techniques in a rich variety of scenarios.

II. Related Work

Distance-based Outliers on Static Data. The $O_{thre}^{(k,R)}$ definition of distance-based outliers was first introduced by Knorr and Ng [2] for static datasets. They describe two detection algorithms. The cell-based algorithm, exponential in the number of data dimensions, is not scalable for high dimensional datasets. The index-based algorithm (using an R-tree or k-d tree) is shown to be non-competitive for three dimensional datasets and up if index building costs are considered. This implies that such relatively expensive indexing would not fit well in our streaming data scenario, because worst yet the index would have to be continuously re-built.

The kNN-based outlier definition was first introduced for static data in [3]. As they show for three dimensional datasets, their index-based (R*-tree) algorithm already performs worse than their partition-based algorithm even after excluding the index building costs. [5] proposes the Orca algorithm which outperforms the predecessor partition-based algorithm [3] with randomization and a simple pruning strategy. Orca scales well to high dimensional dataset. For this reason in this work we now adapt Orca to the streaming context and then use it as baseline to compare our framework against.

Density-Based Outliers on Static Data. Like distance-based outliers density-based outlier detection is a particular category of neighbor-based outlier detection techniques. They assign an outlier score to any given point by measuring the density relative to its local neighborhood restricted by a pre-defined threshold [10], [11]. Therefore density-based outliers are regarded as "local outliers". However distance-based outlier detection instead takes a global view of dataset and marks each point as either outlier or inlier with respect to some user defined global parameters. Furthermore, both [10] and [11] only handle static datasets without taking the potential data update into account. Therefore the techniques proposed in [10] and [11] cannot be applied to solve our problem, namely detecting distance-based outliers on streaming data.

Distance-based Outliers on Streaming Data. With the emergence of digital devices generating data streams, outliers on streaming data have recently been studied [6], [7], [9]. However existing work [6], [7], [9] only considers the simpler distance-threshold variation of distance-based outliers. The processing of the more popular kNN-based variants [3] remains unsolved in the streaming context. Next we further elaborate on the existing results on this first outlier type.

In [9], given a data point p_i, it pre-computes the number of neighbors of p_i for *each future window* that p_i will participate in. It improves CPU performance at the expense of a huge memory overhead by pre-discounting the effect of expired data points for each and every future window in advance. Our work not only improves the CPU efficiency by three orders of magnitude, but also reduces the memory consumption.

[6] analyzes the expiration time of all neighbors of a point gathered by a range query. Then they use the expiration time of the neighbors to locate *safe inliers*, namely any point p_i with more than k neighbors which have arrived after p_i.

[7] further outperforms [6] and [9] by integrating the *safe inlier* concept of [6] into an event queue, so that it can efficiently schedule the necessary checks that have to be made when points expire. However it still relies on full range query searches to process newly arriving points. Therefore it fails to respond in real time when applied to high velocity streaming data targeted by our effort. In our work by exploiting the *minimal probing* and *lifespan-aware prioritization* principles, we succeed to avoid the full range query searches, thereby satisfying the performance requirements of modern streaming applications. Furthermore the above algorithms *ignore* indexing, while in our work we also investigate whether streaming outlier detection can benefit from indexing.

Outliers on Sensor Data. In [12] an interesting online technique is proposed to detect outliers in streaming sensor data. First, it utilizes a kernel density estimator to model the distribution of the sensor data. Then given a point p_i, the number of its neighbors is estimated by the density distribution function $f(p_i)$. Therefore [12] is able to quickly approximate whether p_i is a $O^{(k,R)}_{thres}$ outlier. However the *approximation* nature determines that it cannot be directly applied to our context of computing *exact* distance-based outliers. Furthermore [12] only considers the $O^{(k,R)}_{thres}$ definition of distance-based outlier. The more popular kNN based definitions are not discussed.

Stream Clustering. The clustering definition most closely related to distance-based outliers is density-based clustering [9]. It puts adjacent points that have *enough* neighbors into the same cluster. This problem has been shown to be more expensive than distance-based outlier detection [9], because due to the inter-dependence among the data points the cluster structure is more complex to detect and update than the individual outlier points.

Most other clustering or summarization methods [13] instead focus on discovering accumulative statistical features of the stream. They do not specifically identify neighbor relationships among individual points, which is the key for distance-based outlier detection. Thus they are not directly applicable to our problem of distance-based outliers.

Yet in principle the general idea of micro-clusters or summaries [13] could potentially be exploited to eliminate points from dense areas that cannot be outliers. Clearly one could only eliminate points in dense areas as outlier candidates if the cell (micro-cluster) is small enough such that all points in the cell are neighbors with each other. However having such small cells tends to be not practical in streaming data with high dimensions, potentially requiring us to dynamically maintain too many cells (exponentially increasing with dimensions) and thus causing overwhelming costs.

III. Problem Formalization

A. Definitions of Distance-Based Outliers

Below we formally define the three major distance-based outlier variations. We use the term "data point" or "point" to refer to a multi-dimensional tuple in the data stream. The function $d(p_i, p_j)$ denotes the distance between a pair of points p_i and p_j.

Definition 1: Given a dataset D, a distance threshold R ($R \geq 0$), and a count threshold k ($k \geq 1$), a **distance-threshold outlier** denoted by $O^{k,R}_{thres}$ in D is a data point p_i if there exist fewer than k data points whose distance to p_i is no larger than R in D.

Next both $O^{(k,n)}_{kmax}$ and $O^{(k,n)}_{kavg}$ outliers are defined based on the well-known notion of "k-nearest neighbors (kNN)". Given a data point p_i and its *kth-nearest neighbor* p_j, $d(p_i, p_j)$ is called the kNN maximum distance of p_i denoted as $D^{kmax}(p_i)$, while the average distance to all its *k-nearest neighbors* is called the kNN average distance of p_i denoted as $D^{kavg}(p_i)$.

Definition 2: Given input parameters k ($k \geq 1$) and n ($n \geq 1$), a point p_i is a **kNN maximum distance outlier** denoted by $O^{(k,n)}_{kmax}$ in D if at most n-1 other points p_j exist with $1 \leq j \leq n-1$ in D such that $D^{kmax}(p_j) > D^{kmax}(p_i)$.

Definition 3: Given input parameters k ($k \geq 1$) and n ($n \geq 1$), a point p_i is a **kNN average distance outlier** denoted by $O^{(k,n)}_{kavg}s$ in D if at most n-1 other points p_j exist with $1 \leq j \leq n-1$ in D such that $D^{kavg}(p_j) > D^{kavg}(p_i)$.

B. Distance-Based Outlier Detection in Sliding Windows

We work with periodic sliding window semantics as proposed by CQL [14] for defining the substream of interest from the otherwise infinite data stream. Such semantics can be either time or count-based. Each query Q has a fixed window size

$Q.win$ and slide $Q.slide$. For time-based windows each window W_c of Q has a starting time $W_c.T_{start}$ and an ending time $W_c.T_{end}$= $W_c.T_{start}$+$Q.win$. Periodically the current window W_c slides, causing $W_c.T_{start}$ and $W_c.T_{end}$ to increase by $Q.slide$. For count-based windows, a fixed number (count) of data points corresponds to the window size $Q.win$. The window slides after the arrival of $Q.slide$ new data points.

Outliers will be generated based on the points that fall into the current window W_c, namely the population of W_c. A point p_i in W_c might have different outlier status (outlier or inlier) in the next window W_{c+1} if it is still alive in W_{c+1}, since each window has a different population. Now we define the stream outlier detection problem we tackle.

Definition 4: **Distance-Based Outlier Detection In Sliding Windows:** Given a stream S, a streaming distance-based outlier detection query Q with $O^{(k,R)}_{thres}$, $O^{(k,n)}_{kmax}$, or $O^{(k,n)}_{kavg}$ definition defined in Def. 1, 2, or 3, with window size as $Q.win$ and slide size as $Q.slide$, Q continuously detects and outputs the outliers in the current window W_c when the window slides.

Symbol	Description
p_i	the i-th data point
$p_i.ts$	the timestamp of p_i
$p_i.life$	the lifespan of p_i
W_c	the current window of a stream
$O^{(k,R)}_{thres}$	Distance-Threshold Outliers
$O^{(k,n)}_{kmax}$	kNN Maximum Distance Outliers
$O^{(k,n)}_{kavg}$	kNN Average Distance Outliers
D^{kmax}	kNN Maximum Distance
D^{kavg}	kNN Average Distance
MESI	Minimal Evidence Set for Inlier
LEAP	Lifespan-Aware Probing Operation
$p_i.evi[\,]$	Lifespan-Aware Evidence structure of p_i

TABLE I: Frequently Used Symbols.

IV. A Generic Outlier Detection Framework

We now introduce our scalable framework called *LEAP*, capable of continuously processing distance-based outliers with low CPU and memory resource utilization. LEAP is built on two fundamental optimization principles namely *minimal probing* and *lifespan-aware prioritization* as described below.

A. Theoretical Foundation

In all distance-based outlier definitions, points in a dataset D are classified either as outliers or inliers. Thus, the process of identifying outliers in D is equivalent to the process of eliminating inliers from it. In fact, initially, each point p_i in the dataset is a *potential outlier candidate*, until one has acquired enough evidence to show that p_i is an inlier. For example, in the process of identifying $O^{(k,R)}_{thres}$ outliers, until finding that p_i has at least k neighbors and thus qualifies as inlier, p_i cannot be safely removed from the *outlier candidate set*.

This fact leads us to an important observation. That is, to identify whether a point p_i is a distance-based outlier in a dataset D, one may not need the distance between p_i to *every* other point in D. Instead, a potentially small subset of points will be sufficient to prove that p_i is an inlier. Also due to the rarity of outliers, the majority of points in the dataset could be labeled as inliers in this way by collecting only a small amount of information. To describe the least amount of information needed to prove p_i's inlier status we define the concept of **Minimal Evidence Set for Inlier** (*MESI*).

Definition 5: Given an outlier query and a dataset D, the **MESI** set for a data point $p_i \in D$ is a dataset M such that $M \subseteq D$, if the distance set $DistSet(M, p_i)$ = $\{d(p_1, p_i), d(p_2, p_i), ..., d(p_n, p_i)|\ p_{j(1 \le j \le n)} \in M\}$ is sufficient to label p_i as an inlier, and there does not exist any $M' \subseteq D$ such that $|M'| < |M|$ and $DistSet(M', p_i)$ = $\{d(p_1, p_i), d(p_2, p_i), ..., d(p_m, p_i)|p_{j(1 \le j \le m)} \in M'\}$ is sufficient to label p_i as an inlier.

The size of *MESI* for a point p_i is usually much smaller than the size of p_i's complete neighborhood. For example, for $O^{(k,R)}_{thres}$ outlier, the *MESI* for any point p_i is composed of *any* k points that are within R distance from p_i. Thus its size is k. In general, this input parameter k is much smaller than the average number of neighbors each point may have in R distance range. Otherwise the outliers detected with fewer than k neighbors would not considered to be *abnormal phenomena* in the dataset. The cardinality of *MESI* for a point p_i in the kNN outlier definitions is also bounded by a constant value k as we will show in Sec. VI. This observation guides us to propose the **Minimal Probing** optimization principle (Sec. IV-B).

Although *MESI* is sufficient to prove a point's inlier status in the current window, unlike in static environments, locating more neighbors beyond *MESI* for a given point may be beneficial in streaming environments. These additional neighbors may help us to determine the status of this point in future windows. Thus, we now extend the concept of *MESI* in a static dataset to *MESI* in a sequence of stream windows. In particular, we define the concept of **Minimal Evidence Set for Inlier in a Window Sequence** as below.

Definition 6: Given a streaming outlier detection query Q and all points in the current window W_c, denoted by D_{W_c}, $MESI_{(W_{c,c+x})}$ for p_i in a window sequence from W_c to W_{c+x}, is a dataset M with $M \subseteq D_{W_c}$, if the distance set $DistSet(M, p_i)$=$\{d(p_1, p_i), d(p_2, p_i), ..., d(p_n, p_i)|p_{j(1 \le j \le n)} \in M\}$ is sufficient to label p_i as an inlier in windows W_c to W_{c+x}, and there does not exist any $M' \subseteq D_{W_c}$ with $|M'| < |M|$ and $DistSet(M', p_i)$ = $\{d(p_1, p_i), d(p_2, p_i), ..., d(p_m, p_i)|p_{j(1 \le j \le m)} \in M'\}$ is sufficient to label p_i as an inlier in windows W_c to W_{c+x}.

In other words, the $MESI_{(W_{c,c+x})}$ for a point p_i is a minimal subset of the current window population D_{W_c} that provides sufficient evidence to prove that p_i is an inlier in windows W_c to W_{c+x}, regardless of the characteristics of the future incoming stream. This is possible because by analyzing the time stamp of a point p_i and the query window (the slide and window sizes), we can determine the number of windows that p_i will survive in. For example, for a point p_i that just arrived with the latest slide in the current window W_c, if we found k points within R distance from p_i that arrived when p_i did, then these k points form $MESI_{(W_{c,c+x})}$ for p_i, where W_{c+x} is the last window in which p_i will be alive. This is because these points will be accompanying p_i as its neighbors until p_i expires. We are now ready to define the concept of **Life Time Minimal Evidence Set for Inlier**.

Definition 7: $MESI_{(W_{c,c+x})}$ for p_i is a **life time MESI** of p_i, denoted as $\boldsymbol{MESI_{lt}}$, if W_{c+x} is the last window in which p_i participates before its expiration.

A $MESI_{lt}$ for p_i is an ideal evidence set because it proves the inlier identity of p_i during its entire remaining life, hence named *safe inlier*. It eliminates the need for any future maintenance effort on p_i for the potential detection of its outlier status. Acquiring the $MESI_{lt}$ with *minimal CPU costs* is the key objective for outlier detection in streaming windows. This insight inspires us to propose the **Lifespan-Aware Prioritization** optimization principle in Sec. IV-C.

B. Minimal Probing Principle

As elaborated in Sec. II, all state-of-the-art techniques [6], [9], [7] rely on complete neighborhood searches to identify outliers. In this work, we abandon this methodology and instead present an optimization principle referred to as *minimal probing*. The key idea is that we no longer conduct complete neighborhood searches, such as range query searches, but instead use a lightweight operation called *probing*.

Definition 8: Given a point p_i in the current window W_c, **probing** is an operation that evaluates the distance between p_i and other points in W_c until either the MESI for p_i in W_c is acquired or p_i's entire neighborhood has been evaluated.

The goal of probing for a point p_i is the discovery of a *MESI* for p_i in the current window rather than its complete neighbor set. Therefore probing is fundamentally more efficient compared to a complete neighborhood search, as it significantly reduces the number of data points that need to be evaluated.

Furthermore, the minimal probing principle guides us to intelligently use this lightweight *probing operation* so to maximize the system resource savings. The idea is to carefully extract and then to organize the evidence gathered during each probing process, and furthermore to reuse it whenever possible to avoid repeated probing process.

For all three outlier definitions, with the probing only applied in two situations as explained below we can guarantee the correctness of the query. First, each *new* point p_i that just arrived in the query window needs a probing to figure out its status in the current window. Second, an existing point p_i *without a valid MESI* in the new window needs a probing to re-evaluate its status.

In the first situation, for a newly arriving point p_i the probing operation has to be conducted *from scratch* to search for the needed evidence of p_i.

However this is not the case in the second situation. For a point p_i two conditions can lead to the absence of its *MESI*. First, p_i had been classified as an outlier in the previous window. Therefore no *MESI* has so far been acquired. Second, p_i lost its prior *MESI* when the stream slides to the current window W_c and expired points are removed from W_c. In both cases, the known MESI evidence about p_i which survived the stream data expiration can still contribute to simplify this probing operation. Rather than searching for a new *MESI* from scratch, the probing operation instead only acquires *enough new evidence* to prepare the *MESI* for p_i for the window W_c.

Therefore although the goal for probing is to acquire *MESI* for p_i in the current window, the collected evidence provides us with much richer information than just proving p_i's current status. The method of organizing the MESI to facilitate the fully reuse of the evidence gathered by probing is discussed in Sec. IV-D.

As conclusion, the minimal probing principle uses a lightweight probing process to replace the expensive complete neighbor search. It guides us to fully exploit all evidence gathered during the probing process and thus to minimize the costs of each probing process.

C. Lifespan-Aware Prioritization Principle

Next we propose our second optimization principle termed *Lifespan-Aware Prioritization*. By utilizing the lifespan information of data points this principle further optimizes the probing operation to always discover the *best MESI*.

Lifespan of MESI. As mentioned in Sec. IV-A, the *MESI* of a point p_i in the current window as a whole may serve as the *MESI* of p_i in a sequence of future windows. The number of windows in which a *MESI* can survive, termed the **lifespan of MESI**, relies on how many windows each point p_j in this *MESI* can survive, also termed the **lifespan** of point p_j. In the sliding window scenario, the lifespan of a point p_i can be determined as follows.

Lemma 1: Given the slide size $Q.slide$ of a query Q and the starting time of the current window $W_c.T_{start}$, the **lifespan** $p_i.life$ of a data point p_i in W_c with time stamp $p_i.ts$ is calculated by $p_i.life = \lceil \frac{p_i.ts - W_c.T_{start}}{Q.slide} \rceil$[2], indicating that p_i will participate in windows W_c to $W_{c+p_i.life-1}$.

Hence given Lemma 1 the lifespan of a *MESI* can be decided as below.

Lemma 2: Given a *MESI* of p_i in the current window W_c denoted as $MESI(p_i)$, the **lifespan of MESI**(p_i) $MESI(p_i).life = min\{p_j.life \mid p_j \in MESI(p_i)\}$.

By Def. 6 $MESI(p_i)$ is a $MESI_{(W_{c,c+MESI(p_i).life-1})}$ of p_i covering the window sequence from W_c to $W_{c,c+MESI(p_i).life-1}$. As introduced in Sec. IV-B, among the existing points in window W_c only those without their *MESI* covering the new window W_{c+1} must conduct probing to re-evaluate their status. Therefore, the longer a window sequence a *MESI* covers, the fewer probing processes are needed for this point. Naturally the *MESI* with largest lifespan will be the *best MESI*. Henceforth quickly deriving the best *MESI* of each point is critical for minimizing the probing frequency and in turn saving CPU resources.

Next we analyze how we can further optimize our probing process to always acquire the best *MESI*, but without sacrificing its efficiency. On the one hand, the probing process for p_i should acquire the best *MESI* of p_i. On the other hand, we want the probing process must stay lightweight, so that it stops immediately once it has gotten the *MESI* of p_i in the current window. Our solution is to leverage the lifespan theory of *MESI* in Lemma 2 to prioritize the order in which the probing operation processes the data points.

Definition 9: **Lifespan-aware Prioritization**: During the probing process of p_i, if two data points p_j and p_k have the same probability to be in the MESI of p_i for the current window, we always evaluate p_j first, if $p_j.life > p_k.life$.

Since the succeeding points p_j that arrived after p_i do not expire earlier than p_i, their influence will persist during the entire life of p_i. Therefore any such p_j contributes equally to p_i in terms of determining p_i's outlier status, although they may have different lifespans. Therefore we can treat all succeeding points of p_i as if they all had the same lifespan, namely a lifespan larger than p_i's.

[2]For count-based windows, $p_i.ts$ and $W_c.T_{start}$ are sequence numbers indicating the arrival positions of data points in a stream.

D. Lifespan-Aware Probing Operation

The above *lifespan-aware prioritization* principle together with the *minimal probing* notion implies an optimized probing operation termed *L<u>i</u>f<u>E</u>span-<u>A</u>ware <u>P</u>robing operation* or *LEAP*. LEAP represents the core operation of our framework.

Definition 10: Assume window W_c is composed of k slides denoted as S_i, (1 $\leq$ i $\leq$ k). S_i arrives earlier than S_{i+1}. Given a point p_i in W_c, **LEAP** is a probing that evaluates the status of p_i by testing other points in the S_k, S_{k-1}, ... order.

Intuitively we can see that *LEAP* is guaranteed to produce the *best MESI*. In sliding window streams the data points are naturally ordered by their arrival time and expire in a predictable order. Hence the lifespan of any point can be precisely calculated. By Lemma 1, points in a particular slide share the same lifespan, while points in different slides have distinct lifespans. Later arriving slides have longer lasting lifespans. By conducting the search with a later arriving slide first order, the points with a larger lifespan will always be tested first. Therefore given a point p_i, *LEAP* will produce a *MESI* composed of the evidence with the largest lifespan, that is the *best MESI*. Furthermore LEAP stops immediately as soon as a *MESI* is acquired. Thus it is as lightweight as an ordinary probing operation.

The information collected in the probing process of p_i needs to be carefully selected and kept to minimize the costs of the future probing for p_i (Sec. IV-B). The information shown to be valuable and termed *potential evidence*, is organized as a general *lifespan-aware evidence* structure denoted as $p_i.evi[\,]$.

Definition 11: The **lifespan-aware evidence** for a data point p_i ($p_i.evi[\,]$) represents an ordered list of *potential evidence* of p_i in the current window W_c with each entry of $p_i.evi[\,]$ corresponding to a set of data points with the same lifespan, where the ordering is determined by the lifespan.

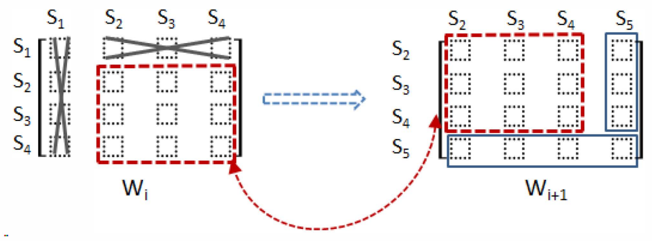


Fig. 1: Sharing of the lifetime proximity measure

As shown in Fig.1 the storage of the $evi[\,]$ structure of a particular window W_i with 4 slides can be abstracted as a two dimensional matrix M_i. The element $M_i[S_x][S_y]$ represents a linear data structure which contains the S_yth entries of all points in slide S_x. This abstract structure explicitly illustrates that our lifespan-aware evidence infrastructure is extremely conclusive to handle the stream evolution. When the window slides from W_i to W_{i+1}, by moving the elements bounded in the dash rectangles one unit up to the top left corner of M_i, it can be easily transformed into M_{i+1} of W_{i+1} only by having to conduct the computation for the elements within the new slide S_5.

Space Complexity Analysis. The storage of the $evi[\,]$ structure has a worst case space requirement $O(nr)$ with r as the ratio of the *Q.win* over *Q.slide* and n as the number of unsafe inliers and outliers. In fact this structure can be further compressed to its half size due to the observation that p_i's succeeding neighbors contribute equally to p_i in terms of determining p_i's status, even if they have different lifespan (as stated in Section IV-C). Therefore the entries representing its succeeding neighbors can be merged with the final number of entries at most being equal to its lifespan.

The precise data structure specific to each outlier type will be introduced in Sec. V and VI.

E. Optimality of LEAP

The LEAP operation, when continuously applied to determine the outlier status of a data point p_i until its expiration, is shown to be optimal in CPU resources consumed for all three outlier definitions.

Theorem 4.1: Given a point p_i in current window W_c and function *f(p_i,W_c, Pr_s)* indicating the CPU costs required by a search strategy Pr_s to evaluate the outlier status of p_i. Then $\sum_{j=c}^{c+life-1} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{c+life-1} f(p_i, W_j, Pr_s)$ with *life* denoting the lifespan of p_i.

Proof: We first establish a prerequisite. Given a data point p_i LEAP takes the same CPU cost to acquire a member of *MESI* for p_i as any other search strategy Pr_s takes. We denote this cost as $\boldsymbol{C^m}$. This prerequisite is justified as follows.

First, given a stream *S* with an unknown distribution, then each point in W_c has the equal chance to be in the *MESI* of p_i. Thus in average any Pr_s will test the same number of points, hence the same costs to acquire a member of *MESI* for p_i.

Second, LEAP is orthogonal to indexing. The both optimization principles of LEAP aim to minimize the frequency of neighbor searches, while indexing instead focuses on accelerating the search of each single neighbor by reducing the neighbor search space. Therefore LEAP is able to exploit whatever indexing methods ever invented or possibly coming up with in the future.

Then we prove Theorem 4.1 using Math Induction.

(1) First we prove $\sum_{j=c}^{c} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{c} f(p_i, W_j, Pr_s)$.

LEAP immediately stops once it acquires the complete MESI for p_i. We use $\mid MESI(p_i) \mid$ to denote the cardinality of MESI. Hence f(p_i,W_j, $LEAP$) = $\mid MESI(p_i) \mid$ * C^m. For any other probing strategy Pr_s the cost $f(p_i, W_j, Pr_s)$ = x * C^m. By Def. 5, MESI is the minimal information needed to prove p_i's status. Hence $\mid MESI(p_i) \mid \leq x$. Therefore

$$\begin{aligned}\sum_{j=c}^{c} f(p_i, W_j, LEAP) &= f(p_i, W_c, LEAP) = \mid MESI(p_i) \mid * C^m \\ &\leq x * C^m = f(p_i, W_c, Pr_s) = \sum_{j=c}^{c} f(p_i, W_j, Pr_s).\end{aligned} \tag{1}$$

(2) Then our induction step from n to n+1 is:

$$\begin{aligned}&if \sum_{j=c}^{n} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{n} f(p_i, W_j, Pr_s),\ then \\ &\sum_{j=c}^{n+1} f(p_i, W_j, LEAP) \leq \sum_{j=c}^{n+1} f(p_i, W_j, Pr_s)\ with\, n < c + life - 2\end{aligned} \tag{2}$$

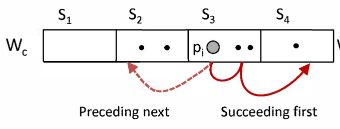


Fig. 2: W_c

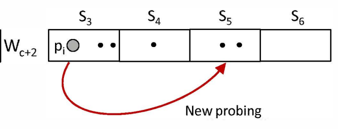


Fig. 3: W_{c+2}

Given the costs C_s LEAP reaps in savings to process p_i from W_c through W_n compared to Pr_s, we can prove Eq. 2 as follows. When the stream slides from W_n to W_{n+1}, the costs LEAP takes to ensure the status of p_i are guaranteed to be not C_s larger than the costs Pr_s takes.

LEAP will be more expensive than Pr_s only if more elements expire in $L^m(p_i)$ (the MESI for p_i produced by LEAP) than in $Pr_s^m(p_i)$ (the evidence produced by Pr_s). Suppose r more elements expire in $L^m(p_i)$ than in $Pr_s^m(p_i)$. This means that in $Pr_s^m(p_i)$ of W_n, there are at least r members younger than the oldest member of $L^m(p_i)$. However in the first window W_c, the oldest member of $Pr_s^m(p_i)$ is at least as old as the oldest member of $L^m(p_i)$. To achieve this, Pr_s must have acquired *at least r more* MESI members than *LEAP*, because LEAP always tests the points with larger lifespans first. However to re-establish the MESI of p_i in W_{n+1}, LEAP only has to acquire *exactly r* more MESI members than Pr_s.

(3) By steps (1) and (2), Theorem 4.1 is proven. ■

V. Strategies for Distance-Threshold Outliers

We now apply our framework to distance-threshold outliers.

MESI and Lifespan-Aware Evidence. By Def. 1 once acquiring k neighbors, a point p_i can be safely declared as an inlier. Therefore the *MESI* for p_i is a data set that contains exactly k neighbors. As the window slides, all other data points examined so far besides its unexpired *MESI* members have no chance to ever be in the *MESIs* of p_i. Therefore only keeping the k neighbors in the MESI is sufficient to avoid any distance re-computation for p_i. Furthermore to determine the status of p_i, we only need the number of its neighbors rather than who its *exact* neighbors are. Therefore the *Lifespan-Aware Evidence* structure of p_i ($p_i.evi[\,]$) for distance-threshold outliers is simply a list of counts, each list entry corresponding to the number of MESI members (neighbors) of p_i in a particular slide.

Thresh_LEAP. Based on the above MESI and *Lifespan-Aware Evidence* structure we present a customized algorithm *Thresh_LEAP* (Alg. 1) for $O_{thres}^{(k,R)}$ outlier detection. When a new window W_c arrives, Thresh_LEAP starts by evaluating each new arrival p_i that had not been in W_{c-1} by simply calling the LEAP operation (Alg. 2). Here we explain step by step using an example how *LEAP* works.

Algorithm 1 Thresh_LEAP(W_c)

```
1: for each $p_i \in W_c.S^{new}$ do
2:     LEAP($p_i$,$W_c$);
3: end for
4: for each $p_i \in W_c.S^{exp}$.triggered do
5:     expireEvidence($p_i$);
6:     LEAP($p_i$, $p_i$.skippedPoints($W_c$));
7: end for
```

Algorithm 2 LEAP(p_i,W_c)

Input: Data point p_i, Dataset W_c //Data points in the current window
Output: Bool isOutlier //Outlier status of p_i

```
1: Bool IsOutlier = false;
2: if (NULL == $p_i$.evi[ ]) then
3:     buildSuccEvidence($p_i$);
4: end if
5: for each q $\in$ $p_i$.succPoint($W_c$) do
6:     if (true == $p_i$.isInNeighborhood(q)) then
7:         $p_i$.updateSuccEvidence();
8:         if (true == $p_i$.isMESIAcquired()) then
9:             $p_i$.isSafe = true;
10:            return isOutlier;
11:        end if
12:    end if
13: end for
14: while $p_i$.precSlides $\neq$ NULL do
15:     slide = getSlideWithLargestLifespan($p_i$.precSlides($W_c$));
16:     $p_i$.buildPrecEvidence(slide);
17:     for each q $\in$ slide do
18:         if (true == $p_i$.isInNeighborhood(q)) then
19:             $p_i$.updatePrecEvidence(slide);
20:             if (true == $p_i$.isMESIAcquired()) then
21:                 slide.updateTriggeredList($p_i$);
22:                 return isOutlier;
23:             end if
24:         end if
25:     end for
26: end while
27: isOutlier = true;
28: return isOutlier;
```

*Example 1: We use an example query Q with $k = 5$ and a fixed R with the ratio of $Q.win$ over $Q.slide$ as 4 to explain how LEAP handles the new data points. As shown in Fig.2, window W_c is divided into four slides. Given a new data point p_i LEAP first tests its succeeding data points (Line 5). At the same time the first entry of $p_i.evi[\,]$ is established as (S_{succ}:0) which represents the number of p_i's succeeding neighbors (Line 3). Once a neighbor is acquired, we update the succeeding entry of $p_i.evi[\,]$ (Line 7), and check whether its MESI has been achieved (Line 8). By testing all its succeeding data points in this window, p_i finds three neighbors. However, it still did not acquire its MESI. Then it has to turn back and proceed to probe its preceding slides (Line 14). The slide with the largest lifespan is tested first (Line 15). In this case it is S_2. Correspondingly a new entry (S_2:0) is created and appended to $p_i.evi[\,]$ (Line 16). The search is terminated after p_i gets its fifth neighbor which completes the MESI for p_i (Line 20). p_i is labeled as **unsafe inlier**. S_2 is being remembered as the **triggering slide** of p_i, meaning that the expiration of S_2 might lead to a status transformation of p_i. To indicate this check p_i is inserted into the triggered outlier candidate list of S_2, namely $S_2.triggered$ (Line 21). The $p_i.evi[\,]$ at this point is $<(S_2:2),(S_{succ}:3)>$.*

After the new arrivals have been all processed, Thresh_LEAP proceeds to process the unexpired points from W_{c-1} that remain in W_c. Clearly the $evi[\,]$ has already been previously established for them. However not all unexpired points need to be re-evaluated. As shown in Alg.1 (Line 4), only the points in $S^{exp}.triggered$ list are re-examined by the LEAP operation with S^{exp} denoting the most recently expired slide. For example, when the stream evolves from W_c to W_{c+1} the expiration of S_1 would not trigger the examination of p_i, because p_i is not in $S_1.triggered$. Only the departure of S_2 will trigger the process of checking the status migration of p_i (Fig. 3).

Example 2: We still use p_i of Example 1 to explain the above re-evaluation procedure. For W_{c+2}, Thresh_LEAP first updates the $p_i.evi[\,]$ to (S_{succ}:3) by removing entry (S_2:2)

(Line 5, Alg. 1). Then the LEAP operation is activated again on the new slide S_5 which was skipped while S_1 expired (Line 6, Alg. 1). The MESI is filled up again after finding the two newly arriving neighbors in S_5. Its $p_i.evi[\ ]$ is updated to (S_{succ}:5). Now p_i has five MESI members which did not arrive earlier than p_i. Therefore p_i achieves its life time MESI $MESI_{lt}$. Now p_i is guaranteed to never become an outlier again and thus is marked as safe inlier (Line 9, Alg. 2). Thus at this point the $p_i.evi[\ ]$ can be safely purged altogether.

As shown in this example it is extremely efficient to determine the status of p_i with the assistance of $p_i.evi[\ ]$ structure. When the window slides from W_{c+1} to W_{c+2}, its leftmost most side S_2 entry will be pruned from $p_i.evi[\ ]$. Then by summing up the alive entries (in this case this would be only one entry S_{succ}), the LEAP operation continues to be aware of the current status of p_i. To acquire the new status of p_i, it proceeds to test the new data points from S_5 until p_i's *MESI* is again established.

VI. Strategies for kNN Outliers

MESI for kNN Outliers. We now demonstrate how we apply our framework to detect kNN outliers. We use Alg.3 to introduce the *MESI* for kNN outliers. By Def. 2 of $O^{(k,n)}_{kmax}$ outliers, Alg. 3 outputs the top-n outliers in a window W_c. Such a set called *outliersSet* in Line 1 is maintained during the search process. Let D^{kmax}_{min} be the shortest distance between any data point in *outliersSet* seen so far and its kth nearest neighbor (Line 2). Assume that for a given point p_i we are processing its distance to its kth-nearest neighbor ($D^{kmax}(p_i)$) (Lines 4 to 6). Since $D^{kmax}(p_i)$ monotonically decreases as we process more points, the current value is an upper-bound on its eventual value. If the current value becomes smaller than D^{kmax}_{min}, then p_i cannot be an outlier (Lines 7 to 9). Therefore the *MESI* for p_i is acquired, which is its kNN in the data points seen so far (neighbors(p_i)). These points are so-called the temporary kNN of p_i.

If $D^{kmax}(p_i)$ is larger than the cutoff threshold D^{kmax}_{min}, p_i will be an outlier candidate. Both the *outliersSet* and D^{kmax}_{min} will be updated (Lines 12-16). As more points are processed, more extreme outliers will be found. The top-n outliers will be finalized after all data points have been processed.

Algorithm 3 kNN_MESI(W_c)

```
1: outliersSet = $\emptyset$; //the top-n outliers set
2: $D^{kmax}_{min}$ = 0;
3: for each $p_i \in W_c$ do
4:    for each $p_j \in W_c$ - $p_i$ do
5:       neighbors($p_i$) = nearest($p_i$,neighbors($p_i$) + $p_j$, k);
6:       $D^{kmax}(p_i)$ = maxDist($p_i$, neighbors($p_i$));
7:       if ((| $neighbors(p_i)$ | == k) $\wedge$ ($D^{kmax}(p_i) \leq D^{kmax}_{min}$)) then
8:          $p_i$.outlierCandidate = false;
9:          break;
10:      end if
11:   end for
12:   if (false $\neq$ $p_i$.outlierCandidate) then
13:      outliersSet = topOutliers(outliers + $p_i$, n);
14:      if (| $outliersSet$ | == n) then
15:         $D^{kmax}_{min}$ = min($D^{kmax}(p_i)$ — $\forall$ $p_i$ in neighbors);
16:      end if
17:   end if
18: end for
```

By replacing the $D^{kmax}(p_i)$ with $D^{kavg}(p_i)$ and D^{kmax}_{min} with D^{kavg}_{min} the same rule can be applied to $O^{(k,n)}_{kavg}$ outlier.

Lifespan-Aware Evidence. Unlike the $O^{(k,R)}_{thres}$ outlier for kNN outliers, only recording *MESI* of p_i is not sufficient for avoiding the distance re-computation whenever the status evaluation is triggered. The *non-MESI* points could also contribute to the *MESIs* of future windows. For example, given a point p_i whose *MESI* members all expire, if no arriving points are close enough to p_i, the *MESI* of p_i in the next window must be formed based on the points which have been evaluated before but were not yet part of the *MESI* of p_i. In this case to avoid re-computation we would have to keep more information besides the *MESI* of p_i in the current window. However keeping all pre-computed distances is not practical.

Fortunately this is where our insight comes to the rescue. Namely keeping the kNN corresponding to each unexpired slide (or the temporary kNN for a slide not completely evaluated) is sufficient to avoid re-computation. The global kNN with respect to the unexpired data points seen so far is guaranteed to be in the union of these local kNN sets. That is, this global kNN can be easily derived by merging and sorting the local kNN sets. We need to evaluate the distance between p_i and new arrivals only if the kNN distance of this global kNN is still larger than the cutoff threshold. Otherwise this global kNN will remain to be the *MESI* of p_i in the new window. In short, with this structure the distance re-computation is completely eliminated. Therefore for kNN outliers $p_i.evi[\ ]$ is a list of data points (along with their distances to p_i) sets, each corresponding to its kNN in each unexpired slide. Since $p_i.evi[\ ]$ is compact, keeping it for each point does not introduce prohibitive memory overload.

LEAP Operation for kNN Outliers. Given a point p_i, LEAP first probes the points with larger lifespan. An entry of $p_i.evi[\ ]$ is established to represent the kNN in its succeeding points. If its *MESI* is not acquired by considering its succeeding points, then the search will need to proceed by processing the preceding points in decreasing order with respect to their lifespans. During this process an entry is created for each preceding slide. LEAP continues to evaluate the distance between p_i and other data points in W_c until either its temporary kNN distance is smaller than the cutoff threshold (*MESI* is acquired) or all points of W_c have been tested. Only in the latter case, the probing operation will return the traditional full kNN of p_i. In this case both the top-n outliers set *outliersSet* and the cutoff threshold will be updated. Due to space restriction, the pseudo code is omitted here.

kNN Outliers Detection With LEAP (kNN_LEAP). Alg. 4 shows how LEAP is utilized to detect kNN outliers in a window W_c. kNN_LEAP first resets the top-n outlier candidates set and the cutoff threshold (Lines 1 to 2). Then it starts processing the unexpired data points, namely the points that were already in window W_{c-1} (Line 3). Given a point p_i, kNN_LEAP first purges the expired entry of $p_i.evi[\ ]$. Then it re-calculates its temporary kNN (Lines 4, 5) if its MESI consists of expired data points (unsafe status). If its current kNN distance (either the previous distance for a safe point or the newly established one for an unsafe point) is larger than the cutoff threshold, the LEAP operation for p_i will be triggered again on the points skipped last time (Lines 7 to 11). Then kNN_LEAP proceeds to process the new arriving data points with the LEAP operation (Lines 13 to 15) until all new arrivals are evaluated.

Algorithm 4 kNN_LEAP(W_c)

```
$D^{kmax}_{min}$ = 0; //Reset cutoff threshold
*outliersSet* = $\emptyset$; //Reset the top-n outliers set
for each $p_i \in W_c$.unExpired do
    if (false == $p_i$.isSafe) then
        expireEvidence($p_i$);
    end if
    if (true == isStillInlier($p_i$)) then
        continue;
    else
        LEAP($p_i$,$p_i$.skippedPoints($W_c$));
    end if
end for
for each $p_i \in W_c$.newArrival do
    LEAP($p_i$,$W_c$);
end for
```

VII. Experimental Evaluation

A. Experiment Setup & Methodologies

All algorithms are implemented on the HP CHAOS Stream Engine [15]. Experiments are performed on a PC with 3.0G Hz CPU and 4GB memory, which runs Windows 7 OS.

Real Datasets. We use two real streaming datasets. The Stock Trading Traces dataset (STT) [16] has one million transaction records throughout the trading hours of a day. The high dimensional Forest Cover (FC) dataset available at the UCI KDD Archive (url:kdd.ics.uci.edu) also used by [7], contains 581,012 records with 54 quantitative attributes.

Synthetic Datasets. We deploy a data generator to produce streams with a controlled number of outliers and data distribution types. Those datasets contain Gaussian distributed data points as inlier candidates with uniform distributed noise. Both the Gaussian distributed points and noise are randomly distributed in each segment of the stream.

Metrics. We measure two metrics common for stream systems, namely CPU time and peak memory consumption. Each experiment evaluates 10,000 windows. Both metrics are averaged over all windows. Although the experiments are reported using count-based windows, time-based windows provide similar results.

Alternative Algorithms. Our experiments focus on evaluating the effectiveness of our both optimization principles, namely *minimal probing* and *lifespan-aware prioritization*, in detecting distance based outliers. For distance-threshold outliers, we compare our algorithms Thresh_MinProbe and Thresh_LEAP (Sec. V) against the state-of-the-art method DUE [7] as introduced in Sec. II. [3] The Thresh_MinProbe applies only the first principle. It essentially equals to DUE enhanced with *minimal probing*. Thresh_LEAP instead utilizes the LEAP framework which applies both principles. For kNN outliers, no existing algorithms in the literature tackle this type of outlier in the streaming context. Hence, we compare our kNN_MinProbe and kNN_LEAP algorithms against kNN_BASIC which applies the static Orca algorithm [5] to compute the top-n outliers from scratch for each window. Similar to distance-threshold outliers, kNN_MinProbe applies only Minimal Probing principle, while kNN_LEAP applies both. Since these three methods all experience only a slight difference for $O^{(k,n)}_{kmax}$ and $O^{(k,n)}_{kavg}$ outlier types, for space reasons, we present the results for $O^{(k,n)}_{kmax}$ outliers only. To evaluate the effect of indexing, similar to [17] we implement a hash-based grid index augmented with a time-aware mechanism for efficiently evicting expiring data. We carefully tune the granularity of the cells and equally apply the same best setting to all compared algorithms. We denote each algorithm xx augmented by this index by "xx-Index".

[3]In this work we chose to compare against DUE rather than the more sophisticated MCOD algorithm of [7], because in the experiments of [7], MCOD does not show clear advantage over DUE in most of the cases.

Methodology. We evaluate the performance of the proposed methods by varying the most important parameters. Specifically, our experiments cover the three major cost factors, namely stream velocity, volume, and outlier rate. We vary the velocity of a data stream by varying the slide size from 0.5k to 50k while leaving all other settings constant.[4] We also measure scalability on high volume streams by varying the window size ω from 1k to 200k. Similarly, we measure how well these methods work for different outlier rates. For the distance-threshold type, this means varying R, while for kNN types varying n as defined in Sec. III. Both control outliers from being rare (0.001%) to being common in the dataset (100%). Since this change also affects the density of neighboring area for each data point, this experiment also reflects data distribution variation in essence. We also measure the scalability of our approach over data dimensionality by varying dimensions from 2 to 40.

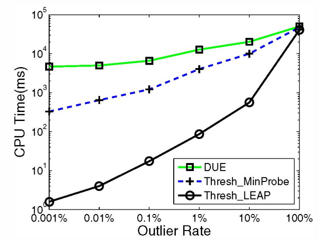
B. Evaluating Distance-Threshold Outliers

1) Varying Outlier Rates: We first analyze the effect of the outlier rate β by varying β from 0.001% to 100% with a fixed slide size of 500 and window size of 100K on synthetic data. As shown in Fig. 4, our Thresh_MinProbe and Thresh_LEAP are superior to DUE with respect to both CPU time and memory usage. In particular, as outlier rate is smaller than 0.01% which is common in real life application [4], Thresh_MinProbe shows a 10 times improvement over DUE while Thresh_LEAP gains another 100 times improvement on this basis in terms of CPU time. Thresh_MinProbe wins over DUE by applying the probing operation, which stops immediately after acquiring MESI rather than evaluates the complete neighborhood as range query search does. Thresh_LEAP further outperforms Thresh_MinProbe by applying the lifespan-aware prioritization principle which enables probing operation to always produce MESI with largest lifespan without introducing any additional cost. This minimizes the frequency of conducting probing in continuously evolving streams. The CPU time of all three methods increases as β increases, because more computation time is spent on verifying the larger number of outliers. Our methods win for all outlier rates from 0.001% to 100%. That is, even in the extreme case when all data points are outliers, the overhead introduced by our methods is still smaller than DUE.

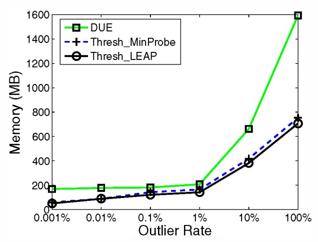
Thresh_MinProbe and Thresh_LEAP use on average 35% and 40% less memory than DUE. This is because Thresh_MinProbe and Thresh_LEAP only store the neighbor count of each slide for outlier candidates, while DUE maintains the actual neighbor relationships. Comparing against Thresh_MinProbe, Thresh_LEAP maximally accelerates the speed of discovering safe inliers. Since safe inlier introduces zero memory overhead, Thresh_LEAP consumes less memory.

2) Varying Slide Sizes: Fig. 5 depicts the performance of the three algorithms for varying slide sizes on synthetic data when the outlier rate is fixed to 0.01% and the window size

[4]Here we only present the results for distance-threshold outliers, since kNN outliers are confirmed to be not sensitive to slide size.

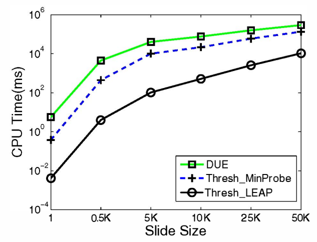


(a) CPU (log scale)

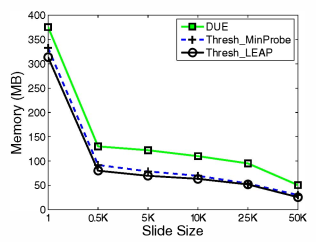


(b) Memory (linear scale)

Fig. 4: Varying Outlier Rates on Synthetic Dataset

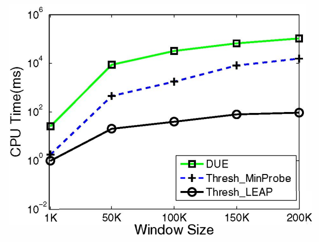


(a) CPU (log scale)

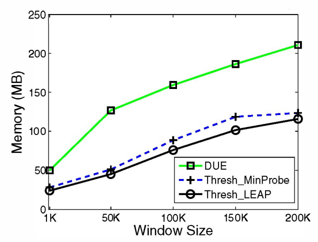


(b) Memory (linear scale)

Fig. 5: Varying Slide Sizes on Synthetic Dataset

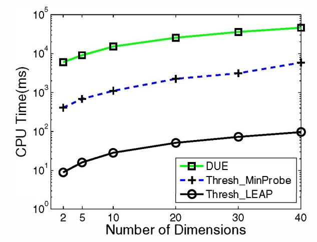


(a) CPU: STT (log scale)

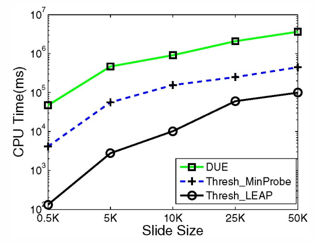


(b) Memory: STT (linear scale)

Fig. 6: Varying Window Sizes on STT Real Dataset



(a) Synthetic dataset



(b) FC cover real dataset

Fig. 7: Dimension Experiments

to 100k. Again Thresh_LEAP and Thresh_MinProbe clearly outperform DUE in CPU time, reaching up to 15 times and 1350 times improvement than DUE for small slide sizes. This is again due to the effectiveness of the LEAP operation as explained in the previous section. As the slide size increases, the processing time on each window increases accordingly. The reason is obvious. The larger slide size introduces more new data points, which in turn cost more CPU time to process. The CPU time of DUE increases by 290 seconds when varying the slide size from 0.5k to 50k, while Thresh_MinProbe and Thresh_LEAP increase only by 130 and 10 seconds.

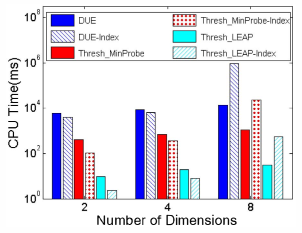
Again, our method is not only superior in CPU but also in memory consumption. As the slide size increases, the percentage of safe inliers over the whole window increases, leading to less memory consumption for all three algorithms to store information for unsafe inliers.

3) Varying Window Sizes: Next, we evaluate the effect of varying window sizes ω from 1k to 200k. We show the results on real dataset STT with fixed k as 30, the outlier rate 0.1%, and slide size 500. In Fig. 6(a)-(b), Thresh_LEAP and Thresh_MinProbe outperform DUE in terms of both CPU and memory. In all cases, the CPU time consumed by Thresh_MinProbe is up to 1 order of magnitude smaller than DUE. Thresh_LEAP further outperforms Thresh_MinProbe by 2 orders of magnitude. As the window size increases, all algorithms consume more CPU time. Thresh_LEAP and Thresh_MinProbe take more CPU resources to process the triggered outlier candidates, while for DUE, larger window takes the range query more time to search for neighbors for new arrivals. Thresh_LEAP wins more against Thresh_MinProbe in larger window. The reason is lifespan-aware prioritization enables probing operation to always acquire MESI with largest lifespan. When window size increases, the potential value of this lifespan increases, making this optimization even more effective. As the window size increases, Thresh_LEAP and Thresh_MinProbe still incur less memory consumption compared to DUE since only DUE has to store the actual neighbors.

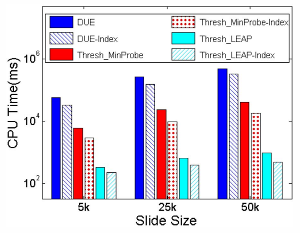
4) Varying Dimensionality of Data: We evaluate the scalability of our algorithms on high dimensional data by varying the number of dimensions from 2 to 40. We fix the window size to 100K, slide size to 5K, and outlier rate to 0.1%. As shown in Fig. 7(a), Thresh_MinProbe algorithm consistently outperforms DUE around 15 fold in terms of CPU time, while Thresh_LEAP further outperforms Thresh_MinProbe around 35 fold. This is expected, since both our optimization principles are orthogonal to the number of data dimensions. The CPU costs of all three algorithms are near linear in the data dimensionality, because the cost of the distance calculation between two points is linear in the number of dimensions, while distance calculation costs are the most significant fraction of the overall outlier detection costs. This is the base price any method has to pay.

We also evaluate the performance of our algorithm on **real life** FC cover dataset (54 dimensions) by varying the slide size. The results shown in Fig. 7(b) again confirm the effectiveness of our approach to high dimensional datasets.

5) Effectiveness of Indexing: We compare all three algorithms against their corresponding indexed versions on synthetic dataset with the number of dimensions varying from 2 to 8. Other settings remain the same as in Sec. VII-B4. As shown in Fig. 8(a), for the 2D and 4D cases, indexing improves the performance of all algorithms. This is as expected, because our both optimization principles are orthogonal to indexing as shown in Sec IV-E. In the 2D case with the help of the index, DUE reduces around 37



(a) Synthetic dataset



(b) STT real dataset

Fig. 8: Indexing Experiments

percent of their CPU costs, while Thresh_MinProbe and Thresh_LEAP reduce 64 and 76 percent respectively. This is because Thresh_LEAP and Thresh_MinProbe have different stopping criteria for the neighbor search than DUE. Given a data point p_i, Thresh_LEAP and Thresh_MinProbe first locate and probe for neighbors in the cell that p_i falls in. This cell can be located in constant time using the grid index. Potentially Thresh_LEAP and Thresh_MinProbe will acquire enough neighbors of p_i, and hence terminate after searching through this single cell. On the other hand, DUE would not stop its search until all neighbors of p_i are acquired. Therefore it locates all cells which could contain the neighbors of p_i, leading to a larger cell lookup costs compared to Thresh_LEAP and Thresh_MinProbe. Second, Thresh_MinProbe performs worse than Thresh_LEAP because it triggers probing operation more frequently and consequently introduce more expensive cell lookup operations.

However, as the number of dimensions increases, the number of the cells in the index to be examined also increases exponentially, leading to a significant increase of index maintenance overhead. This overwhelms the performance gain achieved by utilizing the grid index when the dimensions rise up to 8. In the 8D case, DUE introduces 900ms on average index maintenance costs per each slide which is much larger than the 60ms saved for distance calculation. This condition holds for all the algorithms. Thus, indexing performs well only on low dimensional datasets as had previously been observed for static data in the literature [2], [3].

As shown in Fig. 8(b), our experimental results on **real life** STT dataset with varying slide size also confirms the orthogonality of our approach to the indexing.

C. Evaluating KNN Outliers

1) Varying Outlier Rates: This experiment evaluates the impact of varying outlier rates, namely varying n, on performance. We fix the window size at 10k and slide size at 1k, while varying n from 10 to 300. Most practical applications have a low outlier rate (below 1%). Here we adopt outlier rates ranging from 0.1% to 3% as done in [7].

The CPU costs of all three algorithms increase as the outlier rate increases because a major part of the computation time is spent on processing the potential outliers. As shown in Fig. 9, KNN_MinProbe and KNN_LEAP both significantly outperform the baseline method KNN_BASIC. In particular, KNN_MinProbe outperforms KNN_BASIC 2.5 fold. KNN_LEAP further outperforms KNN_MinProbe 6 fold. The reason that KNN_MinProbe wins over KNN_BASIC is that it exploits the minimal probing principle to reuse the unexpired MESI members. Similar to distance-threshold outlier, KNN_LEAP wins over KNN_MinProbe because it searches for the MESI in an intelligent time-aware order. This minimizes the probing frequency needed.

The memory consumption of KNN_MinProbe is a little more than KNN_LEAP, while KNN_BASIC consumes less. This is as expected, because the first two need to maintain a similar kNN metadata structure per slide to reuse it in the next window. KNN_LEAP consumes less memory than KNN_MinProbe since it reduces the demand for acquiring new MESI members. The memory consumption is stable even with increasing outlier rates, making this a practical compromise for the tremendous gain achieved in CPU resources.

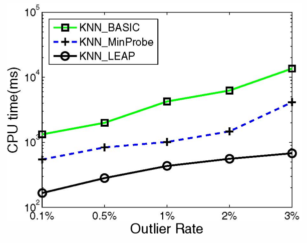
2) Varying Window Sizes: Here, we use the real dataset to evaluate the impact of varying window sizes. We fix the slide size at 200 and n at 100, while varying the window size from 1k to 40k. As depicted in Fig. 10, the CPU costs of all algorithms rise as the window size increases. Yet our best solution KNN_LEAP consistently utilizes the least CPU time and exhibits the slowest increase in CPU consumption. KNN_LEAP and KNN_MinProbe are about 8 and 2 times faster than KNN_BASIC at ω = 1k case and up to 15 and 3 times faster when ω reaches 40k. For a fixed outlier rate, a larger window size results in a larger number of inliers and a wider lifespan range. Both factors are key for our framework to outperform the full kNN query search.

The memory consumption also scales with the window size. For KNN_LEAP and KNN_MinProbe, when the window size increases 40 times, the overhead only increases by about 2 fold. The reason is that the lifespan-aware evidence structure shares more lifetime proximity as the window size increases. This helps our approaches to achieve more compact storage.

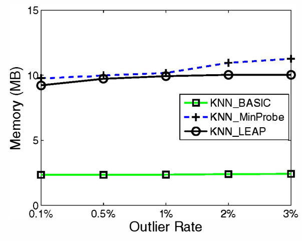
3) Varying Dimensionality of Data: Fig. 11(a) demonstrates the CPU costs of all three algorithms as the number of dimensions increases from 2 up to 40. We fix window size at 10k, outlier rate at 1%, and slide size at 500. KNN_MinProbe and KNN_LEAP outperform KNN_BASIC even more as the dimension number increases. In 2D case, the KNN_MinProbe and KNN_LEAP outperform KNN_BASIC by 2.5 and 12 times respectively, while in 40D case they outperform KNN_BASIC by 4 and 20 times. This is because minimal probing and lifespan-aware principle both minimize the frequency of when the distance calculation has to be deployed. Therefore, when the distance calculation itself constitutes an even large percentage of overall computation cost with the increasing dimensions, they perform even better. In conclusion, KNN_LEAP performs consistently well as the number of data dimensions increases.

We also run experiment on **real life** dataset FC Cover by varying slide size. The results shown in Fig. 11(b) again confirm the effectiveness of LEAP to high dimensional datasets.

4) Effectiveness of Indexing: Fig. 12(a) shows that indexing improves the CPU resource consumption of all three algorithms for low dimensional data ($<$ 4D), while it starts to negatively impact the detection efficiency in higher dimensional cases. In the 8D case, indexing for the KNN_BASIC method reduces the distance calculation cost by 3000ms, yet costs 4500ms for maintaining the grid. A similar situation of maintenance costs superseding any achievable gain holds for our proposed algorithms. Therefore, the grid index benefits

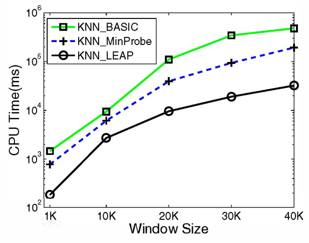


(a) CPU (log scale)

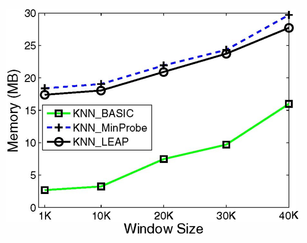


(b) Memory (linear scale)

Fig. 9: Varying Outlier Rates on Synthetic Dataset

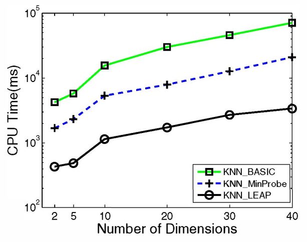


(a) CPU: STT (log scale)

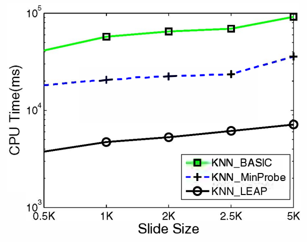


(b) Memory: STT (linear scale)

Fig. 10: Varying Window Sizes on STT Real Dataset

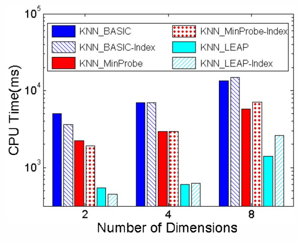


(a) Synthetic dataset

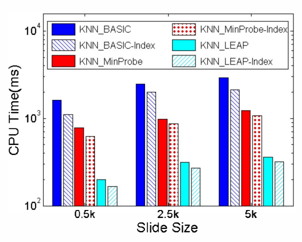


(b) FC cover real dataset

Fig. 11: Dimension Experiments



(a) Synthetic dataset



(b) STT real dataset

Fig. 12: Indexing Experiments

kNN outlier detection only when the data dimensions is rather low (in our case, < 4).

As shown in Fig. 12(b) the other experiment by varying slide size on **real life** STT data also confirms that our approach could benefit from the indexing as the data dimension is low.

VIII. Conclusion

Outlier detection for extracting abnormal phenomena from huge-volumes of streaming data is an extremely important yet difficult task. We propose two novel optimization principles, namely "minimal probing" and "lifespan-aware prioritization" to achieve scalable outlier detection. Our solution framework incorporating these principles is the first unified methodology to handle three types of distance-based outlier definition. It is proven to be optimal for determining the outlier status of data points. Our experimental evaluation with both real and synthetic datasets shows that the proposed approaches are up to 3 orders of magnitude faster than the state-of-the-art.

An interesting direction for future work is to leverage modern distributed multi-core clusters of machines for further improving the scalability of outlier detection.

Acknowledgements

This project is supported by NSF grants IIS-1018443 and IIS-0917017. Thanks also go to China government for supporting Yanwei Yu half of the cost when visiting WPI.

References

[1] D. M. Hawkins, *Identification of Outliers.* Springer, 1980.

[2] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," in *VLDB*, 1998, pp. 392–403.

[3] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *SIGMOD Conference*, 2000, pp. 427–438.

[4] F. Angiulli and C. Pizzuti, "Fast outlier detection in high dimensional spaces," in *PKDD*, 2002, pp. 15–26.

[5] S. D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule," in *KDD*, 2003, pp. 29–38.

[6] F. Angiulli and F. Fassetti, "Distance-based outlier queries in data streams: the novel task and algorithms," *Data Min. Knowl. Discov.*, vol. 20, no. 2, pp. 290–324, 2010.

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

[8] A. Nazaruk and M. Rauchman, "Big data in capital markets," in *SIGMOD Conference*, 2013, pp. 917–918.

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

[10] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," in *SIGMOD Conference*, 2000, pp. 93–104.

[11] S. Papadimitriou, H. Kitagawa, P. B. Gibbons, and C. Faloutsos, "Loci: Fast outlier detection using the local correlation integral," in *ICDE*, 2003, pp. 315–326.

[12] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos, "Online outlier detection in sensor data using non-parametric models," in *VLDB*, 2006, pp. 187–198.

[13] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, 2003, pp. 81–92.

[14] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[15] C. Gupta, S. Wang, I. Ari, M. C. Hao, U. Dayal, A. Mehta, M. Marwah, and R. K. Sharma, "Chaos: A data stream analysis architecture for enterprise applications," in *CEC*, 2009, pp. 33–40.

[16] I. INETATS., "Stock trade traces." *http://www.inetats.com/.*

[17] K. Mouratidis and D. Papadias, "Continuous nearest neighbor queries over sliding windows," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 6, pp. 789–803, 2007.