Adaptive Outlier Detection over Data Stream

RUI ZHU, Shenyang Aerospace University, China
MINGYUAN JIANG, Northeastern University, China
XIAOCHUN YANG, Northeastern University, China
BAIHUA ZHENG, Singapore Management University, Singapore
BIN WANG*, Northeastern University, China
TAO QIU*, Shenyang Aerospace University, China

Continuous distance-based outlier detection in streaming data poses significant challenges and has a wide range of practical applications. Traditional threshold-based methods perform well under stable streaming conditions, where fixed parameters remain effective. However, they often struggle with dynamic data distributions and high stream speeds, leading to suboptimal performance, limited control over the number of returned outliers, and failure to meet real-time detection requirements. To address these issues, this paper introduces a novel Recall and Proportion-Aware Outlier Detection (RPA-OD) query. In RPA-OD, ρ defines a distance relaxation that enables real-time outlier detection. Specifically, objects with fewer than k neighbors within the distance threshold $r' \in (r, (1+\rho)\cdot r]$ *might* be classified as ρ-inliers. This relaxation introduces a trade-off between recall and efficiency, allowing the system to adapt under varying streaming conditions. We propose efficient algorithms to support RPA-OD in data streams, leveraging several novel data structures developed as part of this study. Extensive experiments on five real-world datasets show that RPA-OD significantly improves data throughput, provides precise control over the number of outliers detected, and consistently ensures real-time processing performance.

CCS Concepts: • **Information systems** → **Data management systems**; **Data structures**; *Data access methods*; *Proximity search*;

Additional Key Words and Phrases: Outlier Detection; Data Stream; Recall and Proportion-Aware

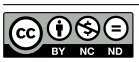
ACM Reference Format:
Rui Zhu, Mingyuan Jiang, Xiaochun Yang, Baihua Zheng, Bin Wang, and Tao Qiu. 2026. Adaptive Outlier Detection over Data Stream. *Proc. ACM Manag. Data,* 4, 3 (SIGMOD), Article 130 (June 2026), 26 pages. https://doi.org/10.1145/3802007

1 Introduction

Continuous distance-based outlier detection (CDOD) [1–7] over sliding windows is a fundamental challenge in streaming data management and has been studied extensively for nearly two decades [8–19]. It plays an essential role in real-time analytics across a wide range of applications, including financial fraud detection, traffic monitoring, network intrusion detection, and system anomaly

*Corresponding author

Authors' Contact Information: Rui Zhu, Shenyang Aerospace University, Shenyang, Liaoning, China, zhurui@sau.edu.cn; Mingyuan Jiang, Northeastern University, Shenyang, Liaoning, China, jiangmingyuan@mails.neu.edu.cn; Xiaochun Yang, Northeastern University, Shenyang, Liaoning, China, yangxc@mail.neu.edu.cn; Baihua Zheng, Singapore Management University, Singapore, bhzheng@smu.edu.sg; Bin Wang, Northeastern University, Shenyang, Liaoning, China, binwang@mail.neu.edu.cn; Tao Qiu, Shenyang Aerospace University, Shenyang, Liaoning, China, qiutao@sau.edu.cn.



ACM 2475-1421/2026/6-ART130
https://doi.org/10.1145/3802007

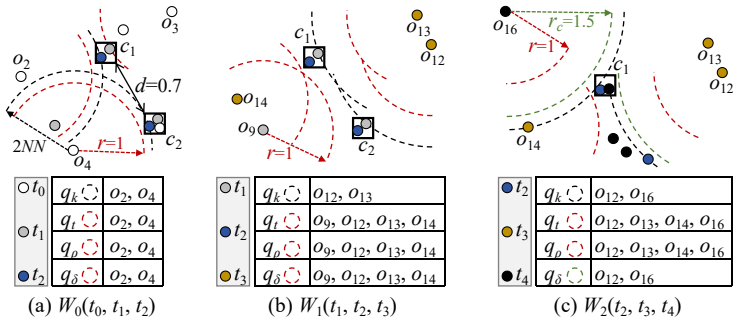


Fig. 1. Different types of outlier detection queries comparison ($n = 3$, $k = 2$, $c = 2$, $\rho = 0.25$, and $\delta = 0.2$).

diagnosis. In these settings, data arrive continuously at high speed, distributions evolve over time, and outlier detection must be performed online under strict latency constraints.

A CDOD query continuously monitors incoming objects within a sliding window $W(n)$ (hereafter referred to as the "window"). The window $W(n)$ can be defined either by count or by time, maintaining, respectively, the most recent n objects or all objects that arrived within the last n time units. As the window slides, the query detects outliers, i.e., objects that deviate significantly from the prevailing data distribution based on distance relationship among objects in the current window. In this paper, we focus on time-based windows, though our techniques readily extend to count-based windows.

Limitations of Existing CDOD Queries. Existing CDOD queries generally fall into two main categories: threshold-based and ranking-based (i.e., k_{max} or k_{avg} queries) [20]. Threshold-based queries label an object as an outlier if it has fewer than k neighbors within a fixed distance threshold r. This approach performs well when the data distribution is stable and an appropriate threshold r can be predefined. However, in streaming environments where data density and scale evolve over time, a fixed threshold often leads to highly unstable results: small distributional shifts can cause the number of detected outliers to vary drastically across consecutive windows.

Ranking-based queries address this instability by always returning a fixed number c of the most anomalous objects, ranked by their k-nearest neighbor (kNN) distances, either the largest k-th distance (k_{max}) or highest average distance (k_{avg}). While this approach provides predictable output sizes and adapts naturally to distribution changes, it incurs significantly higher computational costs, as it requires maintaining and updating kNNs for all objects in the window. This overhead makes ranking-based methods difficult to scale to high-speed data streams, especially in online settings that require continuous, low-latency processing.

Example. Consider a threshold-based query $q_t(n = 3, k = 2, r = 1)$ and a k_{max}-based query $q_k(n = 3, k = 2, c = 2)$ in Figure 1. In the initial window W_0, both return the same outliers $\{o_2, o_4\}$. As the window slides to W_1 and W_2, q_t returns four outliers in each window due to distribution changes, while q_k consistently returns exactly two outliers. Although q_k offers predictable output, monitoring every object's kNNs is computationally expensive and may violate real-time processing constraints.

Prior research has focused primarily on optimizing threshold-based queries using spatial or metric indexes (e.g., M-Tree [21] and Quad-Tree [22]) and incremental neighbor maintenance strategies. However, such algorithms rely on frequent and *costly* range queries and neighbor-list maintenance, with a worst-case complexity of $O(|O_W|)$ per object, where $|O_W|$ is the number of objects in the current window W. Moreover, dynamically adjusting parameters such as r or k to cope with evolving data distributions further increases computational complexity, potentially up to

$O(|O_W|^2)$ per window slide. As a result, existing approaches face a fundamental trade-off: either achieve computational efficiency with unstable output, or ensure output stability at the expense of scalability.

Our Approach: RPA-OD. In this paper, we propose RPA-OD (Recall and Proportion-Aware Outlier Detection), a novel CDOD query model that explicitly balances output stability, detection quality, and computational efficiency. Instead of relying on fixed distance threshold r or fixed outlier count c, RPA-OD retrieves outliers based on a self-turned distance threshold r. This threshold r is driven by a *target outlier ratio* δ_0, representing the expected fraction of outliers within each sliding window. In real implementation, the query continuously monitors the actual outlier ratio in real time based on this self-turned r, and adaptively adjusts r to keep the reported outlier proportion within a user-defined acceptable range. Here, the acceptable range directly defines the lower and upper bounds on the reported outlier proportion. This proportion-aware design ensures stable and interpretable outputs even under significant distribution shifts.

RPA-OD also incorporates recall-awareness through a controlled distance relaxation parameter ρ. When exact detection cannot be sustained due to computational constraints, the query adaptively relaxes distance conditions to ensure timely processing while preserving strong detection guarantees. In particular, all reported outliers are true outliers (100% precision), and we ensure any missed outliers have at least k neighbors within the relaxed distance threshold. This provides a clear, distance-based bound on recall loss.

Why RPA-OD is Necessary. The necessity of RPA-OD arises from a fundamental mismatch between existing CDOD query semantics and the operational requirements of real-world streaming systems. In many practical applications, the cost of outlier detection is dominated not only by accuracy, but by the volume of reported outliers, which directly determines downstream workloads such as alert handling, human inspection, and automated response. Consequently, systems require the number of reported outliers to evolve in a controlled and predictable manner over time. When outlier counts fluctuate sharply or disproportionately across consecutive windows, systems become either overloaded or ineffective, regardless of detection accuracy.

This challenge is evident in intelligent transportation systems, where vehicle density and movement patterns vary dramatically during peak hours, incidents, or adverse weather conditions. Threshold -based detectors in such environments often generate a surge of outliers during high-load periods and very few during sparse periods, overwhelming operators when attention is most constrained and under-utilizing resources otherwise. Similar issues arise in financial markets, where transaction volume and volatility change sharply across market regimes. A detector whose output size scales unpredictably with data density can flood analysts with alerts during high-activity period while suppressing meaningful anomalies during quieter regimes. These operational challenges expose a critical limitation of existing CDOD query model. Threshold-based approaches are highly sensitive to data distribution shifts, leading to unstable and unbounded variations in output size. Ranking-based methods address this instability by enforcing fixed output sizes, but do so at the cost of maintaining global neighbor rankings, which incurs substantial computational overhead and limits scalability in high-speed streaming environments. As a result, existing CDOD query models force an undesirable trade-off between output stability and real-time efficiency, with no mechanism to explicitly control outlier volume under dynamic conditions.

RPA-OD addresses this gap by introducing a proportion-aware query model that treats output controllability as a first-class objective. Rather than relying on fixed thresholds or rigid output counts, RPA-OD maintains the reported outlier volume within a user-defined range by targeting a stable outlier proportion relative to the current window. At the same time, RPA-OD incorporates recall-aware distance relaxation to gracefully handle computational constraints, enabling timely processing while providing explicit, distance-based bounds on approximation error. This integration

makes RPA-OD a practical and scalable alternative to existing CDOD queries, and a compelling foundation for real-time outlier detection in large-scale, dynamic data streams.

Technical Details. Our solution is based on a key observation: an object o is an inlier if the distance to its k-th nearest neighbor o_k, denoted $\mathsf{D}(o, o_k)$, does not exceed the distance threshold r; otherwise, o is an outlier. We extend this idea and propose a set of novel data structures and algorithms to support RPA-OD in high-speed streaming data.

We organize the streaming data using a quad-tree-like spatial index T. Each node $e \in T$ uses a hypercube as its spatial boundary and maintains a lower-bound estimate $e(l_k)$, defined as the smallest radius such that at least $k+1$ objects have their minimum distances to node e no greater than $e(l_k)$. Intuitively, $e(l_k)$ serves as a lower bound of $\mathsf{D}(o, o_k)$ for each object o within node e. We prove that if the diagonal length of node e's boundary, denoted $|e|$, satisfies $|e| \leq \frac{r \cdot \rho}{2}$, then: If $e(l_k) > r$, all objects in e can be safely classified as outliers. Otherwise, they are inliers under the relaxed distance bound. This design avoids maintaining exact neighbor lists for multitudes of individual objects, relying instead on node-level distance summaries.

To efficiently evaluate $e(l_k)$ in streaming settings, we identify a subset of *critical nodes*, termed k-nodes. Each k-node, denoted as ϵ, maintains a small set of nearby leaf nodes whose minimum distances to ϵ are bounded by $2|\epsilon|$. As the window slides, these nearby leaf nodes enable the derivation of a tight upper bound on $e(l_k)$, denoted $e(l_k^+)$, for each descendant node e of a k-node ϵ. These bounds support efficient outlier detection and dynamic adjustment of parameters r and ρ, reducing costly recomputation.

Key Contributions. Our main contributions are as follows:

- We propose RPA-OD query model, which integrates the strengths of threshold-based and ranking-based CDOD queries. By introducing a distance-relaxation parameter ρ and a target outlier proportion δ_0, RPA-OD enables adaptive control of both the distance threshold r and the distance relaxation ρ in response to dynamic streams.
- We design a quad-tree-like spatial index with a novel k-node structure. This framework efficiently estimates the numbers of neighbors an object has via node-level distance bounds $e(l_k)$ and $e(l_k^+)$, significantly reducing neighbor-search costs and enabling scalable outlier detection under high-speed streaming conditions.
- We develop a self-adaptive framework that dynamically adjusts ρ and r according to real-time streaming conditions and computational constraints. The system automatically tunes ρ to balance recall and efficiency, and updates r to maintain a stable outlier proportion. The incremental cost is bounded by $O(k + \log |O_W|)$ per object at moderate-to-high stream speeds, significantly lower than the $O(|O_W|^2)$ cost per slide under naive approaches.

2 Preliminary

In this section, we first review the related work and then present the problem statement.

2.1 Related Work

Outlier detection [23–35] in data streams has been extensively researched due to its critical importance. In the following, we first discuss both exact and approximate algorithms used for this purpose. Next, we briefly explain efforts on node-based pruning.

2.1.1 Outlier Detection Algorithms in Data Streams. Many exact outlier detection algorithms have been developed for data streams. Their core mechanism monitors neighbors of each object o and classifies them as *succeeding* (arriving after object o) or *preceding* (arriving before o). Let $o(s_n)$ and $o(p_n)$ denote the numbers of succeeding and preceding neighbors of object o, respectively. The outlier status of o is then determined as follows: (i) if $o(s_n) \geq k$, o is a *safe inlier*; (ii) if $o(s_n) < k$ and $o(s_n) + o(p_n) \geq k$, o is an *unsafe inlier*; and (iii) if $o(s_n) + o(p_n) < k$, o is an *outlier*. To determine

outlier status, these algorithms maintain $k - o(s_n)$ preceding neighbors for each unsafe inlier o, and all preceding neighbors for each outlier.

To reduce the neighbor maintenance cost, the Abstract-C algorithm [36] lowers neighbor-list maintenance cost by observing that objects arriving in the same slide share identical timestamps. The Thresh-LEAP algorithm [20] employs a multi-index structure, where each index corresponds to a slide. To find an object's neighbors, indexes are scanned in reverse order, issuing range queries until k neighbors are accumulated. However, this approach incurs high time and space overhead when the number of slides is large.

The MCOD algorithm [37] leverages the property that objects within a circle of radius $r/2$ are mutual neighbors. It identifies dense regions where neighbor counts exceed a threshold, avoiding explicit neighbor-list maintenance. MMCOD extends this by inserting each object into all leaf nodes containing it, thereby discovering more dense regions. Similarly, NETS [38] employs a grid with cell diagonal r; objects within the same cell are neighbors, simplifying maintenance for cells with $\geq k + 1$ objects. However, in higher dimensions, dense cells become scarce, forcing many unsafe objects to retain neighbor lists. The CPOD algorithm [39] supports outlier detection by designating certain objects as "core objects", spaced at least r apart. Each core object o is associated with several sets of objects located at different distances, facilitating efficient neighbor searches. However, CPOD is sensitive to parameters r and data distribution, impacting its effectiveness in various scenarios.

The Approx-storm algorithm [40] uses probabilities to determine outlier status. The GAAOD framework [41] uses a kNN-based method to evaluate whether an object o is an outlier. The KAGO algorithm [42] employs a kernel density estimation approach for approximate outlier detection. Mahsa et al [30] proposed a memory efficient incremental local outlier (MiLOF) detection algorithm to support outlier detection within a fixed memory bound. Compared with our approach, existing algorithms incur high computational cost when retrieving outliers. Moreover, they cannot adaptively relax distance thresholds to balance detection quality with computational constraints.

2.1.2 Node-based Pruning Methods. Many efforts apply node-based pruning to speed up the query processing. Brin [43] employed the GNAT metric tree and exploited inter-node distance-interval bounds to prune node–node combinations during near-neighbor search. Gao et al. [44] developed pruning rules such as mindist, maxdist, and emindist between pairs of internal metric-tree nodes to eliminate unnecessary subtree combinations in k-closest-pair queries. Corral et al. [45] leveraged R^*-tree indices and refined node–node geometric reasoning by introducing MINMINDIST and MINMAXDIST over pairs of MBRs, enabling strong pruning for closest-pair search. Chen et al. [46] introduced the SPB-tree, which prunes node–node pairs via MBB overlap tests and lower-bound distances (e.g., MIND) to accelerate range and kNN queries. Compared with these efforts, our approach introduces a novel *density-aware node-level distance lower bound* $e(l_k)$, which is not only a geometric property between nodes, but also captures the data distribution. This allows us to perform efficient outlier detection, as well as guide us adjust parameters ρ and r, offering a more fundamental reduction in computational cost for continuous outlier detection over streams.

2.2 Problem Statement

To clearly explain our proposed RPA-OD, we begin with a simplified variant R-OD (*Recall-Aware Outlier Detection*). Let $q_\rho(n, k, r, \rho)$ represent an R-OD query that continuously monitors objects within a time-based sliding window covering the most recent n time units. At each slide, it identifies objects as outliers if they have fewer than k ρ-neighbors (i.e., approximate neighbors) defined based on a relaxed distance condition, as formally defined below.

Definition 1. **ρ-neighbor**. *Given an object o located in a cube c and another object o' located in cube c', o' is considered a ρ-neighbor of o if: (1) their pairwise distance satisfies $D(o', o) \leq r$; or (2) the*

minimum and maximum distances between c and c′ (or between o and c′) are within r and the range $(1+\rho)\cdot r$ *respectively.*

Consider an R-OD $q_\rho(n=3, k=2, r=1, \rho=0.25)$ over window W_0 (Fig. 1). Let cubes c_1 and c_2 have diagonal length 0.25 ($|c_1| = |c_2| = 0.25$). Their minimum distance $\mathrm{D}_{min}(c_1, c_2)$ is 0.7 ($\leq r$), and their maximum distance $\mathrm{D}_{max}(c_1, c_2) \leq \mathrm{D}_{min}(c_1, c_2) + |c_1| + |c_2| = 0.7 + 0.25 + 0.25 \leq (1+\rho)\cdot r$, making objects in c_1 and c_2 mutual ρ-neighbors. Similarly, $\mathrm{D}_{min}(c_1, o_3) \leq r$ implies $\mathrm{D}_{max}(c_1, o_3) \leq 1.25 \leq (1+\rho)\cdot r$, so all objects in c_1 are ρ-neighbors of o_3.

By checking neighbor relationships at the cube level - considering only cubes whose minimum distance to a target object is within r, R-OD drastically reduces the computational cost of neighbor search while preserving meaningful accuracy. Introducing a distance relaxation parameter ρ may cause R-OD to miss some borderline outliers, classifying them as ρ-inliers. However, any missed outliers are guaranteed to have at least k neighbors within the relaxed distance threshold. In addition, all identified outliers are true outliers, ensuring 100% precision.

Intuitively, smaller ρ values improve recall by detecting more true outliers ($\rho = 0$ corresponds to exact search with 100% recall), while larger ρ values improve efficiency. In practice, selecting a fixed ρ is difficult because stream speed, data distribution, and computation budgets vary over time. Moreover, R-OD cannot control the number of detected outliers, which may fluctuate significantly depending on the threshold r and the underlying data distribution.

To overcome these limitations, we introduce RPA-OD, an adaptive query model formalized as $q_\delta(n, k, r, \delta_0)$, where δ_0 specifies the target outlier proportion. The initial threshold r is set to identify approximately $\delta_0 \cdot |O_W|$ outliers. At each window slide, RPA-OD outputs a pair (O_R, ρ), where O_R is the set of detected outliers, and ρ represents the minimal distance relaxation required to meet real-time constraints. A value of $\rho = 0$ indicates that exact detection with 100% recall is feasible. RPA-OD continuously monitors the actual outlier proportion $\delta = |O_R|/|O_W|$. If δ remains within a user-defined range (e.g., $[0.5\delta_0, 2\delta_0]$), r remains unchanged. Otherwise, r is adjusted to restore the desired proportion. In addition, ρ is no longer a fixed input. It is adaptively tuned during query execution to ensure responsiveness under current computational constraints. This design makes RPA-OD both proportion-aware (by tracking and adapting to outlier proportions) and resource-efficient (by adjusting ρ to match available processing capacity).

Figure 1 illustrates an RPA-OD query $q_\delta(3, 2, r, 0.2)$ (targeting 20% outliers) across windows W_0–W_2, with $\rho = 0.25$ fixed for simplicity and initial $r = 1$. In W_0, 2 of 10 objects are detected as outlier (ratio 0.2), within the range $[0.1, 0.4]$, so r remains unchanged. In W_1, 4 of 8 objects (ratio 0.5) exceed the range, prompting r to increase to 1.5. In W_2, with $r = 1.5$, 2 of 9 objects (ratio ≈ 0.22) fall back within range. Using the original $r = 1$ would have reported 4 outliers in W_2, highlighting the advantage of dynamic threshold adjustment.

3 The KNode-based Framework

In this section, we introduce a novel k-node-based framework built on a quad-tree-like index T to support RPA-OD over streaming data. We begin by presenting a key insight derived from the kNN based algorithm. Next, we define the concept of k-nodes and explain their roles in supporting outlier retrieval.

3.1 Inspiration from the kNN based Algorithms

Our approach builds on a fundamental observation from kNN search: for a threshold query $q_t(n, k, r)$, if $\mathrm{D}(o, o_k)$ from an object o to its k-th nearest neighbor o_k exceeds the distance threshold r, fewer than k objects exist within a radius r from o, implying that o is an outlier. Conversely, if $\mathrm{D}(o, o_k)$

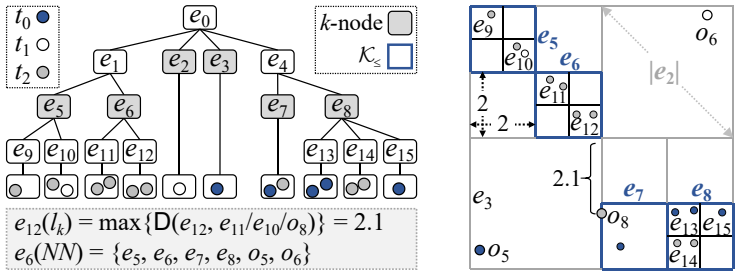


Fig. 2. The k-node-based algorithms ($\rho = 0.632$, $r = 5$, $k = 6$).

$\leq r$, then at least k objects are within distances r to o, making o an inlier. By comparing r and $\mathsf{D}(o, o_k)$ for each object o, we can effectively identify outliers.

Building on this foundation, we introduce a node-level optimization: the *node distance lower bound*, denoted as $e(l_k)$, which estimates the minimum possible $\mathsf{D}(o, o_k)$ for any object o within a node e, as formally presented in Definition 2.

Definition 2. **Node Distance Lower Bound $e(l_k)$.** *For a node e in the index T, its node distance lower bound $e(l_k)$ is the minimal distance such that at least $(k+1)$ objects (including those in e) have their minimum distance to e bounded by this value.*

This bound is computed using node-to-node distances and geometric properties of the enclosing hypercube, making it significantly cheaper than computing object-based pairwise distances. Given an object o and a node e, if there exists a node e' such that $|e'| = |e|$ and e' contains o, then the minimum distance between o and e is computed as $\mathsf{D}_{min}(e, e')$. If o is contained in a leaf node whose size is larger than $|e|$, we instead compute $\mathsf{D}_{min}(e, o)$. In Figure 2, we calculate $e_{12}(l_k)$ based on e_{12} itself, nearby nodes e_{10} and e_{11}, and object o_8 in e_7. That is: $\mathsf{D}_{min}(e_{12}, e_{12}) = 0$, $\mathsf{D}_{min}(e_{12}, e_{11}) = 0$, $\mathsf{D}_{min}(e_{12}, e_{10}) = \sqrt{2}$ and $\mathsf{D}_{min}(e_{12}, o_8) = 2.1$. Thus, $e_{12}(l_k) = 2.1$.

Efficient Outlier Identification. Our framework uses $e(l_k)$ to support efficient outlier detection in the RPA-OD setting, leveraging the following results.

Theorem 1. *Given an RPA-OD query $q_\delta(n, k, r, \delta_0)$ and a node e in the index tree T, if $|e| \leq \frac{r \cdot \rho}{2}$, then: if $e(l_k) > r$, all objects in e are outliers; if $e(l_k) \leq r$, all objects in e are ρ-inliers.*

Proof. Let O_e be the set of objects whose minimum distance to e is at most $e(l_k)$. By definition of $e(l_k)$, the set O_e contains at least $k+1$ objects. We want to prove that, if $e(l_k) \leq r$ and $|e| \leq \frac{r \cdot \rho}{2}$, every object o in O_e is a ρ-neighbor of all objects in e. We consider two cases based on whether o is contained in a node $e' \in T$ of the same size as e. Case (1): There exists a node $e' \in T$ such that $|e'| = |e| \wedge o \in e'$. Then, $\mathsf{D}_{max}(o, e)$ can be approximated by $\mathsf{D}_{min}(e, e') + |e'| + |e| \leq r + 2|e| \leq (1+\rho) \cdot r$, as $|e| \leq \frac{r \cdot \rho}{2}$. Case (2): There is no such node $e' \in T$ with $|e'| = |e| \wedge o \in e'$. Then, o must reside in a leaf node larger than e and $\mathsf{D}_{max}(o, e) = \mathsf{D}_{min}(o, e) + |e| \leq r + |e| < (1+\rho) \cdot r$. In both cases, o is a ρ-neighbor of all objects in e. Since $|O_e| \geq k+1$, every object in e has at least $|O_e| - 1 = k$ ρ-neighbors. Hence, they are ρ-inliers. If $e(l_k) > r$, each object $o \in e$ has $\mathsf{D}(o, o_k) > r$, making all of them outliers. □

Theorem 1 provides a density-aware approach to efficiently support RPA-OD. The running cost of determining outlier status varies significantly depending on the density of the regions where objects reside. In dense regions, where nodes satisfy $|e| \leq \frac{r \cdot \rho}{2}$, we can efficiently determine the outlier status of enclosed objects by computing $e(l_k)$ and comparing it with r, significantly reducing computational cost. In sparse regions, objects may reside in larger leaf nodes e_L. When

$|e_L| + 2e_L(l_k) > (1+\rho) \cdot r$ but $e_L(l_k) \leq r$, the outlier status of those objects cannot be determined based on the bounds alone. In such cases, we perform object-level comparisons to derive $\mathrm{D}(o, o_k)$, which are more computationally expensive. However, since such sparse regions typically contain significantly fewer objects, the overall cost remains manageable.

In Figure 2, since $|e_{12}| = \sqrt{2} \leq \frac{r \cdot \rho}{2} \approx 1.58$, and $e_{12}(l_k) = 2.1 < r = 5$, Theorem 1 ensures that all objects in e_{12} are ρ-inliers. Similarly, objects within nodes $\{e_9, e_{10}, e_{11}, e_{13}, e_{14}, e_{15}\}$ are also ρ-inliers.

Self-Adaptive Parameter Adjustment. Our framework allows adaptive tuning of ρ and r by analyzing the relationships among the current r, the distribution of node sizes, their associated $e(l_k)$ values, and the available computational resources. Theorem 1 implies that a smaller ρ imposes stricter inlier conditions but also increases the number of nodes (and objects) that require costly evaluation. Thus, ρ serves as a trade-off parameter between recall and computational efficiency. In addition, since the distribution of $e(l_k)$ values approximates the distribution of their distances to their k-NNs, we can refine r via $e(l_k)$ value distribution across nodes, without computing $\mathrm{D}(o, o_k)$ for individual objects. This approach yields significant savings compared to exhaustive object-level analysis.

3.2 The Concept of k-Node

$e(l_k)$ values are crucial to RPA-OD. A natural question arises: how can we efficiently compute $e(l_k)$ values for nodes? To address it, we introduce a novel concept called k-node, defined in Definition 3.

Definition 3. **k-node**. *A node ϵ in the index T is a k-node if it contains up to k objects, and its immediate parent node in T contains at least $(k+1)$ objects.*

Selecting k-nodes as pivotal nodes offers two key advantages. First, for any k-node ϵ, we have $\epsilon(l_k) = 0$, as formalized in Lemma 1. This eliminates the need to compute $\epsilon(l_k)$ for both the k-nodes and their ancestor nodes, significantly reducing the cost of re-evaluating $e(l_k)$ per window slide. Second, when ρ and r satisfy $|\epsilon| \leq \frac{(1+\rho) \cdot r}{2}$, all objects in ϵ are guaranteed to be ρ-inliers, as stated in Lemma 2. This enables immediate inlier detection without further verification. In Figure 2, e_6 contains 4 objects ($< k = 6$), while its parent node e_1 contains 7 objects ($\geq k + 1 = 7$), making e_6 a k-node. As $|e_6| = 2\sqrt{2} < \frac{(1+\rho) \cdot r}{2} = \frac{(1+0.632) \cdot 5}{2} = 4.08$, all objects in e_6 are ρ-inliers according to Lemma 2.

Lemma 1. *For any k-node ϵ in T, $\epsilon(l_k) = 0$.*

Proof. By definition, the parent node of ϵ contains at least $k+1$ objects, distributed among ϵ and its sibling nodes. Since the minimal distance of ϵ to any siblings is 0, $\epsilon(l_k) = 0$. □

Lemma 2. *Given a k-node ϵ within the index T, if $|\epsilon| \leq \frac{(1+\rho) \cdot r}{2}$, objects within ϵ are all ρ-inliers.*

Proof. As ϵ and its sibling nodes collectively contain at least $k+1$ objects, and the maximum distance between any object in ϵ and objects in its sibling nodes is bounded by $2|\epsilon|$, if $|\epsilon| \leq \frac{(1+\rho) \cdot r}{2}$, any object $o \in \epsilon$ has at least k objects within their distances to o bounded by $2|\epsilon| \leq (1+\rho) \cdot r$. Together with $\epsilon(l_k) = 0 < r$, objects within ϵ are all ρ-inliers. □

k-nodes enable efficient computation of $e(l_k)$ not only for themselves and their ancestors but also their descendant nodes. Specifically, for any descendant node e of a k-node ϵ, we can compute $e(l_k)$ efficiently using a set associated with ϵ, denoted as $\epsilon(NN)$.

$\epsilon(NN)$ contains a limited number of nodes and objects, defined as follows: (i) all nodes in T with size $|\epsilon|$ (including ϵ itself), whose minimal distances to ϵ are no larger than $2|\epsilon|$; and (ii) all objects residing in leaf nodes of size larger than $|\epsilon|$, with minimal distances to ϵ no larger than

$2|\epsilon|$. In Figure 2, e_6 is a k-node and its corresponding $e_6(NN)$ contains 4 nodes and 2 objects $\{e_5, e_6, e_7, e_8, o_5, o_6\}$.

As stated in Lemma 3, elements within $\epsilon(NN)$ suffice to compute $e(l_k)$ for any descendant node e of ϵ, without the need to search the full index, helps us reduce the computational overhead, thereby enhancing the effectiveness of our k-node-based framework.

Lemma 3. *Let e be a descendant node of a k-node ϵ within T, and let $\epsilon(NN)$ denote the set of neighbor nodes of ϵ. For any object $o \in e$, all its kNNs are contained in $\epsilon(NN)$.*

Proof. Let S_o be the set of objects enclosed by ϵ and its sibling nodes. As the maximum distance between an object $o \in e$ and objects in S_o is bounded by $2|\epsilon|$ and $|S_o - \{o\}| \geq k$ (as guaranteed by the definition of k-node), it is confirmed that $\mathrm{D}(o, o_k) \leq 2|\epsilon|$. Consequently, all kNNs of $o \in e$ must be contained in $\epsilon(NN)$. □

Above all, we propose maintaining $\epsilon(NN)$ for all non-leaf k-nodes in T to support RPA-OD, as formalized in Theorem 2. For clarity, we divide the k-node set into two subsets: $\mathcal{K}_>$ and $\mathcal{K}_\leq$, for a given RPA-OD query $q_\delta(n, k, r, \delta_0)$ according to node size. $\mathcal{K}_>$ includes all k-nodes with sizes exceeding $\frac{(1+\rho)\cdot r}{2}$ and $\mathcal{K}_\leq$ includes the remaining k-nodes. For k-nodes in $\mathcal{K}_>$, we identify outliers by evaluating their maximum descendant nodes with sizes no larger than $\frac{r\cdot\rho}{2}$, and leaf nodes with sizes larger than $\frac{r\cdot\rho}{2}$. k-nodes in $\mathcal{K}_\leq$ can be safely excluded from outlier retrieval because, as stated in Lemma 2, they do not contain any outliers.

Theorem 2. *Given a quad-tree based index T, a distance relaxation parameter ρ and distance threshold r, we can detect outliers by maintaining $\epsilon(NN)$ for all non-leaf k-nodes ϵ in T.*

Proof. Let $\mathcal{K}$ denote the set of k-nodes in T. We answer RPA-OD by evaluating each k-node. Because the evaluation depends on node sizes, we partition k-nodes into two subsets based on node sizes, i.e., $\mathcal{K}_> = \{\epsilon \in \mathcal{K} || \epsilon| > \frac{(1+\rho)\cdot r}{2}\}$ and $\mathcal{K}_\leq = \{\epsilon \in \mathcal{K} || \epsilon| \leq \frac{(1+\rho)\cdot r}{2}\}$.
Case (i): For k-nodes in $\mathcal{K}_\leq$, their enclosed objects are all ρ-inliers, as stated in Lemma 2.
Case (ii): For k-nodes ϵ in $\mathcal{K}_>$, we locate their largest descendant nodes e with sizes bounded by $\frac{r\cdot\rho}{2}$. We derive $e(l_k)$ according to the nodes/objects in $\epsilon(NN)$, as stated in Lemma 3. We thereafter compare $e(l_k)$ with r based on Theorem 1 to determine whether objects in e are outliers.
Case (iii): For k-nodes ϵ' in $\mathcal{K}_>$ having descendant leaf nodes e_L with size larger than $\frac{r\cdot\rho}{2}$, we derive $e_L(l_k)$ based on nodes/objects in $\epsilon'(NN)$. If $e_L(l_k) > r$, objects within e_L are outliers. If $e_L(l_k) \leq r$ and $e_L(l_k) + 2|e_L| \leq (1+\rho)\cdot r$, objects within e_L are ρ-inliers. Otherwise ($e_L(l_k) \leq r \wedge e_L(l_k) + 2|e_L| > (1+\rho)\cdot r$), objects within e_L may be outliers which require further evaluation. For each object $o \in e_L$, we search $\epsilon'(NN)$ for its ρ-neighbors.
Edge Case: For k-nodes ϵ_l in $\mathcal{K}_>$ that are leaf nodes themselves, we search for ρ-neighbors for objects in ϵ_l. □

In Figure 2, k-nodes e_5, e_6, e_7, and e_8 fall into the set $\mathcal{K}_\leq$ (Case (i)), and their enclosed objects are ρ-inliers by Lemma 2. Leaf-nodes e_2 and e_3 are k-nodes within $\mathcal{K}_>$ (Edge Case), and their enclosed objects require further evaluations.

Challenges. Our approach is efficient, but several challenges remain. First, in high-density regions, $\epsilon(NN)$ set tends to be large. Specifically, there can be up to $(3^d - 1)$ $|\epsilon|$-sized nodes with zero minimal distances to ϵ, resulting in expensive searches when deriving $e(l_k)$ for descendant nodes e. Second, under streaming conditions, repeatedly calculating $e(l_k)$ for the descendants e of k-nodes incurs high processing costs. Third, we need to find ways to better leverage the properties of k-nodes and $e(l_k)$ values of their descendants to guide the selection of reasonable r and ρ values. We will address these challenges in the following section.

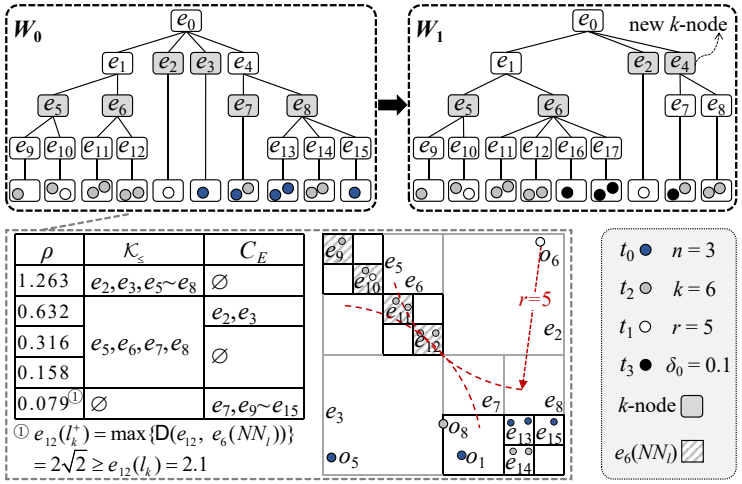


Fig. 3. Recall and proportion-aware outlier retrieval.

4 Support RPA-OD Under Data Streams

When the window slides, new objects are inserted into the index T, while expired objects are removed. After updating T, k-nodes are updated accordingly (see Section 4.1). Next, outliers are retrieved. We describe how to retrieve outliers from a single node and the overall retrieval process in Sections 4.2 and 4.3. Finally, we explain how to adjust r, if needed, in Section 4.4, and present cost analysis in Section 4.5.

4.1 Maintenance of $\epsilon(NN_l)$

In this paper, we propose maintaining a structure denoted as $\epsilon(NN_l)$, a refined version of $\epsilon(NN)$, for each non-leaf k-node ϵ, to address the first challenge. $\epsilon(NN_l)$ retains up to $2k$ leaf nodes, selected from those enclosed by the nodes in $\epsilon(NN)$. This design enables $\epsilon(NN_l)$ to dynamically adjust to local object density. In high-density areas, fewer leaf nodes are needed to maintain a tight upper bound for the $e(l_k)$ of descendant nodes. In low-density areas, more leaf nodes are included in $\epsilon(NN_l)$. As shown in Figure 3, $e_6(NN_l)$ contains 4 leaf nodes $\{e_9, e_{10}, e_{11}, e_{12}\}$, covering 7 objects. In contrast, the set $e_6(NN)$ contains 4 nodes and 2 objects $\{e_5, e_6, e_7, e_8, o_5, o_6\}$, covering a total of 16 objects. This shows that $e_6(NN_l)$ spans a smaller search space and incurs a lower storage overhead.

Upon receiving a RPA-OD, we visit the index T to form $\epsilon(NN_l)$ for each non-leaf k-node ϵ. This set is initialized to include a few leaf nodes that have the smallest minimum distances to ϵ, such that the total number of enclosed objects is at least $k+1$. As the window moves, we outline the detailed maintenance of $\epsilon(NN_l)$ under two different scenarios.

Scenario i): ϵ becomes an ancestor of new k-nodes. When the object count in ϵ increases to exceed k, ϵ no longer satisfies the criteria of being a valid k-node according to Definition 3. Despite this, we employ a lazy update strategy that continues to treat ϵ as a k-node. This lazy update strategy remains valid, as ϵ covers larger spaces and encloses more objects than its descendant nodes, ensuring that $\epsilon(l_k) \leq e_d(l_k)$, where e_d is any descendant of ϵ. Furthermore, it effectively handles cases where the number of objects in ϵ fluctuates. Once the object count in ϵ exceeds $2k$, we search k-nodes among ϵ's descendants. For each newly identified k-node ϵ_d, we form $\epsilon_d(NN_l)$ by including all elements from $\epsilon(NN_l)$ whose minimum distance to ϵ_d is bounded by $2|\epsilon_d|$. At that point, ϵ is no longer treated as a k-node, and we stop maintaining $\epsilon(NN_l)$.

Scenario ii): ϵ becomes a descendant of another k-node ϵ_a. When the number of objects within ϵ's parent node ϵ_a decreases, ϵ becomes a descendant of a newly promoted k-node ϵ_a. Given the fact that $|\epsilon_a| \geq 2|\epsilon|$, all leaf nodes in $\epsilon(NN_l)$ have their minimum distances to ϵ_a bounded by $2|\epsilon_a|$ too. Therefore, when initializing $\epsilon_a(NN_l)$, we randomly select an existing k-node ϵ' from ϵ_a's children and set $\epsilon_a(NN_l) = \epsilon'(NN_l)$. This method uses the spatial properties inherited from the previous k-node structure. ϵ now becomes a descendant of k-node.

In Figure 3, e_7 and e_8 are k-nodes in W_0. Under W_1, 5 blue objects expire from the window, while 4 black objects arrive. Node e_4 (parent of e_7 and e_8) becomes a k-node. We then initialize $e_4(NN_l)$ based on either $e_7(NN_l)$ or $e_8(NN_l)$. In contrast, k-node e_6 includes 7 objects in W_1, exceeding the threshold k (= 6) but not $2k$. Therefore, e_6 is still treated as a k-node under the lazy update.

4.2 $e(l_k^+, t, f, S)$ Guided Outlier Evaluation

We support RPA-OD based on Theorem 2. In the following, we explain how to perform the detailed evaluations, including node-based evaluation and object-based evaluation, for k-nodes ϵ belonging to Case (ii), Case (iii), and Edge Case of Theorem 2.

4.2.1 Node-based evaluation. We propose using $e(l_k^+)$ to replace $e(l_k)$ to address the second challenge. We introduce a tuple (l_k^+, t, f, S) for each descendant node e of a k-node ϵ. Here, $e(l_k^+)$ is an upper bound for $e(l_k)$, derived mainly from nodes in $\epsilon(NN_l)$. $e(t)$ denotes the expiration time of this bound. $e(f)$ is a flag indicating whether the bound is exact. If $e(f) = true$, then $e(l_k^+) = e(l_k)$. Otherwise, $e(l_k^+)$ is only an upper bound of $e(l_k)$. The key insight behind using $e(l_k^+)$ is that, in dense regions, $e(l_k^+)$ is often sufficient to determine the outlier status of enclosed objects. This allows us to avoid computing the exact $e(l_k)$ whenever possible, which is computationally expensive. Lastly, $e(S)$ is a cached set that *temporarily* stores the nodes accessed during the calculation of $e(l_k)$. It helps avoid redundant searches when computing $e_d(l_k)$ for e's descendant nodes e_d. Consider e_6 in Figure 3. As $e_6(NN_l) = \{e_9, e_{10}, e_{11}, e_{12}\}$, $\mathrm{D}_{min}(e_{12}, e_{12}) = 0$, $\mathrm{D}_{min}(e_{12}, e_{11}) = 0$, $\mathrm{D}_{min}(e_{12}, e_{10}) = \sqrt{2}$, and $\mathrm{D}_{min}(e_{12}, e_9) = 2\sqrt{2}$. We have $e_{12}(l_k^+) = 2\sqrt{2}$, while $e_{12}(l_k) = 2.1$.

To capitalize on this, we employ a lazy evaluation strategy. When a new descendant e of a k-node is created as the window slides, we initialize $e(l_k^+, t, f, S)$ to $e(+\infty, -1, false, \emptyset)$. This tuple is *only* updated when e is evaluated during outlier retrieval. This deferred computation significantly reduces the overhead of maintaining $e(l_k)$, improving overall efficiency.

Let e be a node that requires evaluation, under a specific ρ setting. We begin with Case (ii) of Theorem 2 where e is the largest descendant node with size bounded by $\frac{r \cdot \rho}{2}$ and its ancestor k-node ϵ belongs to $\mathcal{K}_>$ (Algorithm 1 Line 2). If a valid value of $e(l_k^+)$ is already available, we skip the following computation step. Otherwise, if node e is being evaluated for the first time, or its previously stored $e(l_k^+)$ value has expired (i.e., expiration time $e(t)$ has been reached, Algorithm 1 Line 3), we need to initiate or update $e(l_k^+)$ value. We first compute $e(l_k^+)$ using only the nodes in $\epsilon(NN_l)$, where ϵ is e's ancestor k-node. Next, we update $e(l_k^+)$ and $e(t)$, and set the flag $e(f) = false$ (Algorithm 1 Lines 4–5).

We then compare $e(l_k^+)$ with r, leading to three possible outcomes. First, if $e(l_k^+) + 2|e| \leq r$, all objects within e have at least k neighbors within radius r, and are guaranteed to be inliers, regardless of the value of ρ. Evaluation for node e can be safely terminated. Second, if $r - 2|e| < e(l_k^+) \leq r$, objects in e are ρ-inliers, as guaranteed by Theorem 1. These objects may become outliers as ρ decreases in later iterations (Algorithm 1 Lines 6–9).

Third, if $e(l_k^+) > r$, it triggers the calculation of $e(l_k)$ as this upper bound is too loose. We then compute the exact $e(l_k)$ by extending the search to index T. To calculate $e(l_k)$, we first check whether there are at least $k + 1$ objects whose lower-bound distances to e are no larger than r. This involves identifying nodes whose minimum distance to the center of e does not exceed $(r + \frac{|e|}{2})$.

Algorithm 1: Node-Based Evaluation

Input: Node e, distance threshold r, distance tolerance threshold ρ
Output: The tuple $e(l_k^+, t, f, S)$

1 $\epsilon \leftarrow$ FindKNode(e);
2 **if** $|e| \leq \frac{r \cdot \rho}{2} \wedge \epsilon \in \mathcal{K}_>$ **then** // Case (ii)
3 **if** $e(t) < W(t)$ **then**
4 Compute $e(l_k^+)$ and $e(t)$ by $\epsilon(NN_l)$;
5 $e(f) \leftarrow false$;
6 **if** $e(l_k^+) \leq r - 2|e|$ **then**
7 All objects in e are inliers, goto Line 22;
8 **else if** $e(l_k^+) \leq r$ **then**
9 All objects in e are ρ-inliers, goto Line 22;
10 E_R=RangeQuery($e, r + \frac{|e|}{2}, T$);
11 **if** $|E_R| < k + 1$ **then**
12 All objects in e are outliers, goto Line 22;
13 **else**
14 Compute $e(l_k)$ and $e(t)$ via E_R;
15 $e(l_k^+) \leftarrow e(l_k), e(f) \leftarrow true$;
16 update $\epsilon(NN_l)$ and $e(S)$;
17 Back to line 6;
18 **else if** e *is a leaf* $\wedge\ |e| > \frac{r \cdot \rho}{2} \wedge \epsilon \in \mathcal{K}_>$ **then** // Case (iii)
19 Leaf-Evaluation(e, ϵ, r, ρ);
20 **else if** e *is a leaf* $\wedge\ e = \epsilon \wedge \epsilon \in \mathcal{K}_>$ **then** // Edge Case
21 Object-Based-Evaluation(e, r, ρ);
22 return $e(l_k^+, t, f, S)$;

If fewer than $k + 1$ such objects are found, all objects in e are outliers, and the evaluation on e is safely terminated (Algorithm 1 Lines 10–12). Otherwise, we perform a kNN search on the identified objects to compute $e(l_k)$; details are omitted due to space constraints. Let m be the number of objects whose distance lower bounds to e are within $e(l_k)$. Since $m > k$ is common, we reduce the computational cost by randomly selecting $k' = \min(2k, m)$ objects. Then, we update $e(t)$ to the $(k + 1)$-th largest arrival order among these k' objects, and set $e(f)$ to true. We apply the earlier logic: if $e(l_k^+) + 2|e| \leq r$, all objects in e are inliers; if $r - 2|e| < e(l_k^+) \leq r$, objects in e are ρ-inliers. (Algorithm 1 Lines 13–17).

We now consider Case (iii), where a leaf node e_L satisfies $|e_L| > \frac{r \cdot \rho}{2}$ and its ancestor k-node belongs to $\mathcal{K}_>$. We execute the function Leaf-Evaluation (Algorithm 1 Lines 18–19). This function follows logic similar to Case (ii), with two key differences. First, objects in e_L are regarded as ρ-inliers if the following two conditions are satisfied: (i) $e(l_k^+) \leq r$, and (ii) $r - 2|e_L| < e_L(l_k^+) \leq (1+\rho) \cdot r - 2|e_L|$. Second, once $e(l_k)$ is computed, $e_L(f)$ is set to true. Under this case, if $(1+\rho) \cdot r - 2|e_L| < e_L(l_k^+) \leq r$, the outlier status of objects in e_L cannot be determined at the node level. Therefore, we perform object-based outlier evaluation, i.e., we invoke the function Object-Based-Evaluation. Lastly, we consider the Edge Case, where e_L itself is a k-node. In this situation, we also perform object-based outlier evaluation for objects within e_L (Algorithm 1 Lines 20–21).

Take the k-node e_6 in Figure 3 as an example. Assume that in window W_0 with $\rho = 0.079$, we need to perform node-based outlier evaluation on descendants of k-node e_6, where $e_6 \in \mathcal{K}_>$. As e_6 only has descendant leaf nodes with size greater than $\frac{r \cdot \rho}{2} (\approx 0.198)$ (Case (iii)), we need to evaluate e_{11} and e_{12}. First, we calculate $e_{11}(l_k^+)$ based on $e_6(NN_l)$. As $e_{11}(l_k^+) + 2|e_{11}| = 3\sqrt{2} \leq r (= 5)$, all objects

in e_{11} are inliers, and $e_{11}(l_k^+, t, f, S)$ is updated to $(\sqrt{2}, t_1, \text{false}, \emptyset)$. Second, we derive $e_{12}(l_k^+) = 2\sqrt{2}$, again based on $e_6(NN_l)$. Given $e_{12}(l_k^+) < r$ and $e_{12}(l_k^+) + 2|e_{12}| > (1+\rho)\cdot r$, its current bound is too loose and we derive $e_{12}(l_k) = 2.1$ by conducting kNN search on its neighbors identified among those enclosed in $e_6(NN_l)$. Since $e_{12}(l_k) + 2|e_{12}| = 2.1 + 2\sqrt{2} < 5$, its objects are also inliers, and $e_{12}(l_k^+, t, f, S)$ is updated to $(2.1, t_1, \text{true}, \emptyset)$.

We want to highlight two points. First, once $e(l_k^+)$ is set to the actual bound $e(l_k)$ (i.e., when $e(f)$ is set to *true*), we update $\epsilon(NN_l)$ for e's ancestor k-node ϵ by inserting all leaf nodes used to derive $e(l_k)$ to $\epsilon(NN_l)$. This expansion improves the accuracy of future $e'(l_k^+)$ values of ϵ's descendant nodes e'. To control its size, we limit $\epsilon(NN_l)$ to at most $2k$ leaf nodes. If the set exceeds this size constraint, we retain only the most recent $2k$ nodes entering $\epsilon(NN_l)$, storing them in a FIFO queue. Furthermore, $\epsilon(NN_l)$ is updated at most once per window to limit its update frequency.

Second, as will be discussed in Section 4.3, the value of ρ decreases gradually during outlier retrieval, requiring the evaluation of descendant nodes e_d of e. To avoid frequently searching on the whole index, once the exact $e(l_k)$ is computed, we update $e(S)$ to include the following elements: (i) nodes of size $|e|$ whose minimal distances to the center of e are within $r + \frac{|e|}{2}$, and (ii) objects from larger leaf nodes that also lie within this bound. This cached set supports efficient computation of $e_d(l_k)$ values for descendant nodes e_d of e. Specifically, if multiple ancestor nodes of e_d have computed their l_k values, we select the ancestor e with the smallest size (i.e., the one nearest to e_d in the index tree T). We then reuse the cached set $e(S)$ to compute $e_d(l_k)$ efficiently. The set $e(S)$ is deleted upon completion of outlier retrieval for the current window to optimize space usage. Note, we do not update $e_{11}(S)$ and $e_{12}(S)$ in our above example, because evaluations on e_{11} and e_{12} are completed after all their enclosed objects are identified as inliers.

4.2.2 Object-based evaluation. A leaf node e_L with $e_L(l_k) \le r$ but $e_L(l_k) + 2|e_L| > (1+\rho)\cdot r$ (Case (iii)), as well as any leaf k-node with size greater than $\frac{(1+\rho)\cdot r}{2}$ (Edge Case), requires further evaluation. As discussed in Section 4.3, the same objects in e_L may be evaluated multiple times across different iterations under different ρ values. To avoid repeated calculations, we maintain a tuple $o(l_k, u_k, t, S(\rho_0))$ for each object $o \in e_L$. Here, $o(l_k)$ and $o(u_k)$ store the lower and upper bounds of $\mathsf{D}(o, o_k)$, respectively. $o(t)$ indicates the expiration time of these bounds. $o(S(\rho_0))$ is a cached supporting set, used to avoid redundant searches during reevaluations of o under different ρ values. It maintains (i) nodes with sizes $|e| \le r\cdot\rho_0$ whose minimum distances to o are within r, and (ii) objects from leaf nodes e'_L with $|e'_L| > r\cdot\rho_0$ that are within distance r from o. ρ_0 refers to the ρ value used when the last time $S(\rho_0)$ is formed or updated under the current window. Similar to the logic discussed before, the tuple $o(l_k, u_k, t)$ is discarded once $o(t)$ is reached and rebuilt if o is re-evaluated. $o(S(\rho_0))$ is deleted upon completion of outlier retrieval to save space cost.

For each object $o \in e_L$, we calculate (or update) $o(l_k)$ under a few cases. Because of space constraints, we only explain the procedure when the tuple $o(l_k, u_k, t, S(\rho_0))$ is unavailable, where we could use the similar logic to process o under other cases. Specifically, we perform a radius-r range query around o using the cached set $e_L(S)$. The lower bound $o(l_k)$ is derived using the similar logic of $e(l_k)$ calculation presented in Section 4.2.1. Let m be the number of objects whose distance lower bounds to o are within $o(l_k)$. We randomly select $k' = \min(2k, m)$ objects. From these, we identify o's kNNs, set $o(u_k)$ to o's distance to its k-th neighbor, and set $o(t)$ accordingly. Each object $o \in e_L$ is classified as an inlier if $o(u_k) \le r$, a ρ-inlier if $o(l_k) \le r$ and $o(u_k) \le (1+\rho)\cdot r$, or an outlier otherwise. To support re-evaluation under decreasing ρ values in subsequent iterations, if o is a ρ-inlier, we temporarily retain the support set $o(S(\rho_0))$, following the logic discussed in Section 4.2.1.

Discussion. RPA-OD ensures correctness via the following logic: 1) If $e(t)$ is valid and $e(l_k^+)$ yields a conclusive decision (e.g., $e(l_k^+) > r$ or $e(l_k^+) \le r$ with $|e| \le \frac{r\cdot\rho}{2}$), we immediately determine the

status of all objects in e; 2) If $e(l_k^+)$ is expired or too loose, we recompute or tighten it using the compact neighbor set $\epsilon(NN_l)$ of e's ancestor k-node ϵ and then re-evaluate; 3) If the decision remains unclear, we compute the exact $e(l_k)$ via index-based search; correctness then follows directly from Theorem 2; 4) If the status of objects in a leaf node e_L cannot be determined from $e_L(l_k^+)$ and $|e_L|$, we perform object-based evaluation.

4.3 Outlier Retrieval

After updating T, we initialize ρ so that all k-nodes have their sizes bounded by $\frac{(1+\rho)\cdot r}{2}$ and thus belong to $\mathcal{K}_{\leq}$ initially, i.e., $\mathcal{K}_{\leq} = \mathcal{K}$ (Algorithm 2 Line 2). As shown in Figure 3, r is set to 5. Since the largest k-node e_2 has a size of $4\sqrt{2}$, we initialize ρ to $\frac{2\cdot|e_2|}{r} - 1 \approx 1.263$. Accordingly, the threshold $\frac{(1+\rho)\cdot r}{2} = 5.6575$, and all k-nodes have sizes bounded by this threshold.

Lines 3–18 of Algorithm 2 describe the core outlier retrieval procedure. Leveraging properties of k-nodes and Theorem 2, the algorithm jointly retrieves outliers and adapts ρ through four phases: *k-node-guided ρ tuning*, *candidate set update*, *candidate set evaluation*, and *running cost evaluation*.

k-node-Guided ρ Tuning. Assume k-nodes ϵ_i in set $\mathcal{K}$ are sorted in descending order of their size, i.e., $|\epsilon_1| \geq |\epsilon_2| \geq |\epsilon_3| \cdots$. By Theorem 2, only the k-nodes in $\mathcal{K}_{>}$ (those with size larger than $\frac{(1+\rho)\cdot r}{2}$) may contain outliers. We therefore initialize ρ to $(\frac{2|\epsilon_1|}{r} - 1)$, so that the size bound $\frac{(1+\rho)\cdot r}{2} = |\epsilon_1|$, ensuring all k-nodes in $\mathcal{K}$ meet this bound, i.e., $\mathcal{K}_{>} = \emptyset$ and $\mathcal{K}_{\leq} = \mathcal{K}$ (Algorithm 2 Line 2). We then form a candidate node C_E, which contains nodes that have potential outliers and is initially empty. No outliers are retrieved at this stage, and the retrieval process incurs negligible running cost. We then iteratively halve ρ. Each decrease in ρ value updates the candidate set C_E (Algorithm 2 Line 4).

Take the k-nodes in W_0 of Figure 3 as an example. The initial value of ρ is $\frac{2|e_2|}{r} - 1 \approx 1.263$. In the first iteration, ρ is halved (i.e., ≈ 0.632), so that $\frac{(1+\rho)\cdot r}{2} = 4.08$. As a result, e_2 and e_3 move out of $\mathcal{K}_{\leq}$. This operation triggers an update to C_E, followed by an evaluation of its nodes to identify outliers. Both node-based and object-based evaluations are performed, as detailed in Section 4.2.

The process continues iteratively until one of the following two termination conditions is met. First, C_E becomes empty and all k-nodes in $\mathcal{K}_{\leq}$ have their sizes bounded by $\frac{r}{2}$. Note that k-nodes with size bounded by $\frac{r}{2}$ only contain inliers, regardless of ρ setting. This condition indicates that all outliers have been retrieved, implying exact outlier retrieval with $\rho = 0$ (Algorithm 2 Lines 5–6). Second, the current time limit is reached and the window is about to slide, i.e., EVALUATION-COST function returns $false$ (Algorithm 2 Lines 7–8). We roll back ρ to its previous value and terminate the retrieval.

Candidate Set Update. Each time ρ is reduced, all k-nodes in $\mathcal{K}_{\leq}$ that exceed the new threshold $\frac{(1+\rho)\cdot r}{2}$ are moved out of $\mathcal{K}_{\leq}$ and added to $\mathcal{K}_{>}$. This triggers an update to the candidate set C_E. Specifically, k-nodes newly added to $\mathcal{K}_{>}$ are inserted into C_E. Next, we evaluate all non-leaf nodes in C_E whose sizes are not bounded by the new threshold $\frac{r\cdot\rho}{2}$, and replace them with their largest descendant nodes that are within this bound, along with any descendant leaf nodes that exceed this threshold, if any[1].

Candidate Set Evaluation. We scan C_E to identify outliers, following the evaluation presented in Section 4.2. We first conduct node-based evaluation for each node $e \in C_E$. For nodes e whose evaluations are terminated (i.e., e contains inliers or outliers but not ρ-inliers), e is removed from C_E (Algorithm 2 Lines 11–14). Only nodes e that include ρ-inliers remain at C_E. We then conduct object-based evaluation for leaf node $e_L \in C_E$ with $e_L(l_k^+) \leq r \wedge e_L(l_k^+) + 2|e_L| > (1+\rho)\cdot r$ (Algorithm 2 Lines 15–16). We remove e from C_E if all the enclosed objects are inliers or outliers.

[1]Our implementation accelerates candidate set maintenance by introducing auxiliary structures that limit processing to k-nodes that have not been part of $\mathcal{K}_{>}$ within the last n time units. It only visits nodes with either an invalid $e(l_k^+, t, f, S)$ or $e(l_k^+) > r$.

Algorithm 2: Outlier Retrieval

Input: Newly arrived object set s_{in}, expired object set s_{out}
Output: Outlier Set O_R, distance tolerance threshold ρ

1 $O_R \leftarrow \emptyset, C_E \leftarrow \emptyset$;
2 $\mathcal{K} \leftarrow$ Update-KNodes(T); $\rho \leftarrow \frac{2 \cdot \max_{e \in \mathcal{K}} |\epsilon|}{r} - 1$, $\mathcal{K}_{\leq} \leftarrow \mathcal{K}$;
3 **while** *the termination conditions are not met* **do**
4 $\rho \leftarrow \rho / 2.0, C_E \leftarrow$ Update-CandidateSet($\rho, r, \mathcal{K}_{>}, T$);
5 **if** $\max_{\epsilon \in \mathcal{K}_{\leq}} |\epsilon| \leq \frac{r}{2} \wedge C_E = \emptyset$ **then**
6 $\rho \leftarrow 0$, break; ▷ exact search, 100% recall
7 **if** *Evaluation-Cost*(C_E, ρ) = *false* **then**
8 $\rho \leftarrow \rho \times 2.0$, break; ▷ window is about to slide
9 **for** *each element* $e \in C_E$ **do**
10 Node-Evaluation(e, r, ρ); ▷ Sec. 4.2.1
11 **if** $e(l_k^+) + 2|e| \leq r$ **then**
12 $C_E \leftarrow C_E - \{e\}$; ▷ inliers
13 **else if** $e(l_k^+) > r$ **then**
14 $O_R \leftarrow O_R \cup e(O), C_E \leftarrow C_E - \{e\}$; ▷ outliers
15 **if** *(e is a leaf node)* $\wedge\ r \geq e(l_k^+) > (1+\rho) \cdot r - 2|e|$ **then**
16 OBJ-Evaluation(e, r, O_R, ρ); ▷ Sec. 4.2.2
17 **if** *all objects in* e *are identified as inliers or outliers* **then**
18 $C_E \leftarrow C_E - \{e\}$;
19 return (O_R, ρ) ;

Returning to the example in Figure 3. Under W_0, when $\rho = 0.632$, C_E is updated to $\{e_2, e_3\}$. As both are leaf k-nodes and $e_2(S) = e_3(S) = \emptyset$, we perform radius-$r$ range searches around o_5 and o_6 in T. As both range searches return $< k$ objects, o_5 and o_6 are identified as outliers, and nodes e_2 and e_3 are removed from C_E. As shown in the table in Figure 3, ρ continues to halve its value, while C_E remains empty. Only when ρ reduces to 0.079, k-nodes $\{e_5, e_6, e_7, e_8\}$ are moved out of $\mathcal{K}_{\leq}$. For non-leaf k-nodes $\{e_5, e_6, e_8\}$, we locate their largest descendant nodes with sizes bounded by $\frac{r \cdot \rho}{2} \approx 0.198$ (i.e., $e_9 \sim e_{15}$) and insert them into C_E, together with the leaf k-node e_7. We then perform node-based evaluation (e.g., for nodes $\{e_9, e_{10}, \cdots\}$) and object-based evaluation (e.g., for objects o_1 and o_8 in e_7). After processing these nodes, both C_E and $\mathcal{K}_{\leq}$ become empty, indicating that all outliers have been found. $(O_R, \rho = 0)$ is returned to complete the retrieval process.

Running Cost Evaluation. We retrieve outliers through an iterative process. Outliers that are enclosed by larger nodes tend to be detected earlier, and their $\mathrm{D}(o, o_k)$s are generally larger. The decision to proceed to the next iteration is governed by whether the system has the ability to evaluate candidate nodes in C_E before the window slides (Algorithm 2 Line 7). In the following, we explain how to estimate the running cost required for the upcoming iteration. The candidate set C_E includes two types of nodes, those newly added, denoted as C'_E, and those expanded from nodes already in C_E, denoted as C''_E.

Nodes e in C'_E need to update their l_k^+. In the worst case, this involves a construction of the cached set $e(S)$, which includes nodes of size $|e|$ whose minimal distances to the center of e are within $r + |e|/2$, and objects from larger leaf nodes that also lie within this bound. Recall that $|e|$ is bounded by $r \cdot \rho/2$, so $r + |e|/2$ is bounded by $r + r \cdot \rho/4 = (1 + \rho/4) \cdot r$. In general, the number of nodes with size bounded by $r \cdot \rho/2$ that intersect the boundary of a hyper-sphere of radius $(1 + \rho/4) \cdot r$ is bounded by $(\frac{(1+\rho/4) \cdot r}{r \cdot \rho/2\sqrt{d}})^d = O\big((\frac{2\sqrt{d}}{\rho})^d\big)$. Accordingly, the total number of nodes in T intersecting the boundary of a hyper-sphere with radius $(1 + \rho/4) \cdot r$ is bounded by: $O((\frac{2\sqrt{d}}{\rho})^d(1 + \frac{1}{2^d} + \frac{1}{4^d} + \cdots))$

$= O((\frac{2\sqrt{d}}{\rho})^d)$. The cost of resetting $e(t)$ is $O(k)$. Each time we update $e(l_k^+)$, we insert at most $O(k)$ leaf nodes into $\epsilon(NN_l)$ with $O(1)$ per node. The overall cost of constructing or updating $e(l_k^+)$ is $O(k + (\frac{2\sqrt{d}}{\rho})^d)$, expressed as Cost-Node(ρ). The computational cost incurred by C'_E is $|C'_E| \cdot$ Cost-Node(ρ).

For each node $e \in C''_E$, its running cost can be approximated by the cardinality of $e_a(S)$, denoted as $|e_a(S)|$, where e_a is e's closest ancestor with $e_a(S)$ available. Since elements in $e_a(S)$ have at most $(|e_a|/|e|)^d \cdot |e_a(S)|$ descendant nodes, the running cost is bounded by $O(|e_a|/|e|)^d \cdot |e_a(S)|)$. The total running cost for C''_E is therefore bounded by $O(\sum_{e \in C''_E}(|e_a|/|e|)^d \cdot |e_a(S)|)$. Thus, the running cost to evaluate nodes within C_E can be approximated by Equation (1).

$$Cost_E(\rho) = |C'_E| \cdot \text{Cost-Node}(\rho) + \sum_{e \in C''_E}(|e_a|/|e|)^d \cdot |e_a(S)| \tag{1}$$

We also perform object-based evaluation. Let C_O be the set of objects that require object-based evaluation. We further partition C_O into C'_O that includes objects whose $(l_k, u_k, t, S(\rho_0))$ is unavailable under the current window, and $C''_O = C_O \setminus C'_O$ that include objects whose $(l_k, u_k, t, S(\rho_0))$ were previously computed in earlier iterations. Following the logic of node-based evaluation, this part of running cost is evaluated via Equation (2).

$$Cost_O(\rho) = \sum_{o \in C'_O} |e_L(S)| + \sum_{o' \in C''_O}(\rho_0/\rho)^d \cdot |o'(S(\rho_0))| \tag{2}$$

Last, the algorithm dynamically adjusts ρ based on the available processing time t_a before the window slides. In the real implementation, we first profile the system to obtain two throughput constants: α (index nodes accessible per ms) and β (data objects accessible per ms). When considering a reduction of ρ to improve recall, the cost model estimates the additional nodes ΔN and objects ΔO required. The expected processing time is then computed as $\Delta t_{\text{est}} = \Delta N/\alpha + \Delta O/\beta$. If $t_a \geq \Delta t_{\text{est}}$, the system proceeds with the refined ρ and evaluates the candidate sets (function Evaluation-Cost returns true); otherwise, outlier detection terminates and ρ rolls back to its previous value (Algorithm 2 Lines 7–8).

Discussion. Our proposed method is efficient. First, we compute node-level l_k^+ values instead of performing costly kNN searches for individual objects. Second, we update or calculate l_k^+ values only for a selective subset of nodes. Finally, although the current work mainly focuses on a single-threaded implementation, our method naturally supports parallel execution. In particular, the Node-Based Evaluation module processes nodes independently, making it well suited for multi-threaded execution. To validate this property, we implement a multi-threaded version of the Node-Based Evaluation module and compare it with the single-threaded implementation. The detailed experimental evaluation and performance comparison are presented in the experimental section. In addition, the r value adjustment can also be extended to run concurrently with the main detection process with minor modifications.

4.4 Adjusting Distance Threshold r

Let δ represent the percentage of identified outliers in the current query window, and δ_0 the expected percentage of outliers. If δ falls within the range $[\frac{\delta_0}{2}, 2\delta_0]$ (or another application-specific range), we do not update r. Otherwise, we adjusts r to restore the desired outlier ratio. When excessive outliers are detected ($\delta > 2\delta_0$), as increasing r is expected to bring the number of outliers back to the specified range in future windows, we relax the radius r by setting it to the $(\delta_0 \cdot |O_W|)$-th smallest $o(l_k)$ value among outliers in set O_R.

When an insufficient number of outliers is detected ($\delta < \frac{\delta_0}{2}$), we tighten r to ensure that at least $\delta_0 \cdot |O_W|$ objects satisfy $\mathrm{D}(o, o_k) > r$. The tightening procedure is as follows. First, we locate all leaf

nodes that contain the current outliers. If these leaf nodes collectively contain fewer than $\delta_0 \cdot |O_W|$ objects, we traverse the index upward from these leaf nodes, visiting their parent nodes one by one in descending order of their sizes, until accumulated set of nodes contains at least $\delta_0 \cdot |O_W|$ objects. Next, we construct the node set E by iteratively adding the largest node (in terms of size $|e|$) from the visited nodes until the total number of objects in E reaches or exceeds $\delta_0 \cdot |O_W|$. Then, we compute $o(l_k)$ for each object o within these nodes, and set r to the k-th largest $o(l_k)$ value.

4.5 Cost Analysis

This section explains the running cost incurred in $\epsilon(NN_l)$ construction, outlier retrieval, and r adjustment.

$\epsilon(NN_l)$ **Construction and Maintenance.** We construct $\epsilon(NN_l)$ from scratch during initialization for all k-nodes ϵ. This requires traversing the index T to locate ϵ's k nearest leaf nodes, with a cost of $O((\frac{r}{|\epsilon|/\sqrt{d}})^d + k)$, which simplifies to $O(k)$ per k-node. After initialization, as objects expire and new objects arrive, updates to k-nodes are triggered. For new k-nodes ϵ, their corresponding $\epsilon(NN_l)$ is constructed using (NN_l) from either their ancestor k-node ϵ_a or descendant k-node ϵ_d. Since both $\epsilon_a(NN_l)$ and $\epsilon_d(NN_l)$ contain up to $2k$ leaf nodes, the construction cost remains $O(k)$.

Outlier Retrieval. Let ρ indicate the final distance relaxation returned, and u be the total number of nodes whose l_k^+ values are evaluated during node-based evaluation. The running cost for outlier retrieval is bounded by $O(u \cdot (k + \log |O_W|))$.

r **Adjustment.** As stated in Section 4.4, adjusting r requires computing $o(l_k)$ for affected objects. This cost is dominated by a range query on the index T, which is $O((\frac{\sqrt{d}}{r})^d + k) = O(k)$. Therefore, the total running cost is $O(\delta_0 \cdot |O_W| \cdot k)$.

Overall Incremental Cost per Update. In this paper, we use a quad-tree structure as the indexing structure when presenting our framework and techniques to support RPA-OD. Our implementation uses QC-Tree [47], a quad-tree extension designed to maintain balanced partitions for streaming and dynamic datasets. QC-Tree preserves the core properties of quad-trees, so all structural assumptions and pruning strategies in the paper still apply. Its height remains $O(\log |O_W|)$ in the worst case, ensuing efficient neighborhood search and pruning for RPA-OD. Let s_{out} and s_{in} denote the sets of expired and newly arrived objects during window sliding, respectively. The overall insertion and deletion cost is bounded by $O((|s_{out}| + |s_{in}|) \cdot \log |O_W|)$. In addition, each insertion/deletion only affects a constant number of k-nodes, so the k-node maintenance cost is bounded by $O((|s_{out}|+|s_{in}|) \cdot k)$. Via further considering *outlier retrieval* and r *adjustment* cost, the total computational complexity per window slide is bounded by $O(u \cdot (\log |O_W| + k) + \delta_0 \cdot |O_W| \cdot k + (|s_{out}| + |s_{in}|) \cdot (\log |O_W| + k))$.

Note that u depends on the number of object insertions and deletions, the number of nodes that contain outliers, and the number of outliers within the window. Given r and ρ, only a constant number of nodes of size $r \cdot \rho$ (including leaf nodes larger than $r \cdot \rho$) have their minimal distances to an object bounded by r. Thus, each insertion or deletion affects a constant number of nodes and objects. Thus, $O(u)$ is bounded by $O(|s_{out}| + |s_{in}|)$. Assuming that $\delta \cdot |O_W| \ll |s_{out}| + |s_{in}|$, which holds under moderate-to-high stream speeds, the amortized cost per update is $O(\log |O_W| + k)$.

5 Performance Evaluation

This section presents the results of our extensive experiments that are designed to evaluate the performance of our algorithms.

5.1 Experimental Settings

Data Sets. We conduct our experiments using five real datasets, as listed in Table 1. (i) STOCK consists of 1 billion stock transactions for 2,300 stocks from the Shanghai/Shenzhen Stock Exchange

Table 1. Datasets and Default Parameter Values.

Dataset	d	Data Scale	n	s	r	δ_0	ρ	k	$std(s)$
Stock	2	1.0B	20	5K	3.72	1%	0.05	50	2.50K
Tao	3	0.6M	20	0.5K	1.90	1%	0.05	50	0.25K
Hpc	7	1.0M	20	5K	6.50	1%	0.05	50	2.50K
Gas	10	1.0M	20	5K	2.75	1%	0.05	50	2.50K
Em	16	1.0M	20	5K	115.00	1%	0.05	50	2.50K

over 24 months. The dataset focuses on volume and price, sorted by transaction time. (ii) Tao [48] contains 575,468 records of sea surface temperature, relative humidity, and precipitation. (iii) Hpc [49] contains 1 million records of electric power consumption, with 7 attributes per record. (iv) Gas [49] contains 1 million records of household gas sensor data, with 10 attributes per record. (v) Em [49] contains chemical sensor data, with 16 attributes per record.

Query Parameters. We evaluate the algorithm performance using the parameters n, s, r, k, ρ and $std(s)$. Following [38, 39], the default parameter settings are listed in Table 1. Specifically, n denotes the number of slides within the window, and s represents the stream speed, defined as the number of objects entering or expiring from the window at each window slide. When the stream speed is constant, $n \times s = |O_W|$ indicates the number of objects in the window. δ_0 is the expected outlier ratio. ρ is the distance relaxation bound. $std(s)$ is the standard deviation of s when the data stream moves at varying speeds. These parameters are chosen based on domain requirements. The choice of k is grounded in statistical theory. As recommended in Robust Statistics [50], a relatively large sample (e.g., $k \geq 50$) provides reliable distance estimates for skewed distributions, which explains its adoption (i.e., k=50) in prior work [38, 39]. The parameter r is set via standard practice: after computing $\mathrm{D}(o, o_k)$ for all objects in an initial window, r is set to the $(\delta_0 \cdot |O_W|)$-th largest $\mathrm{D}(o, o_k)$ value. Note, our approach goes beyond this static initialization by automatically tuning r, whenever the observed outlier ratio δ is outside the acceptable range, we adjust r to restore the desired proportion.

Performance Metrics. We use the following four metrics. (i) *Data Throughput* measures the operational efficiency of different algorithms per window slide. (ii) *Recall Ratio* records the average proportion of outliers retrieved by different algorithms. (iii) *Incremental Running Time* measures processing time spent on each window. (iv) *Memory* records the average and peak memory used by different algorithms.

Competitors. This study presents the first attempt to support RPA-OD over data streams, denoted as Knode. In addition, we extend several SOTA algorithms designed for threshold-based queries as competitors, which are: (i) Cpod [39] supports RPA-OD by re-forming the core point set and re-organizing objects when r is updated. Specifically, if r is reduced, we re-search for neighbors of affected objects. (ii) Nets [38] uses a grid-based index for streaming data. To support RPA-OD, we update the grid-based index whenever r is changed. If r is reduced, we re-search neighbors for objects not located in cells whose diagonal length equals r. If r is enlarged, we only search neighbors for outliers to reduce cost. (iii) Leap [20] uses a set of indexes to manage objects across different slides, where each slide corresponds to an index. To support RPA-OD, if r is reduced, we re-scan indexes in reverse order for each object and re-identify its k neighbors. If r is enlarged, we only search neighbors for outliers to reduce cost. (iv) MCod [51] uses M-Tree-based indexes for streaming data. If r is reduced, we re-search neighbors for objects out of dense regions. If r is enlarged, we only search neighbors for outliers to reduce cost. For all these algorithms, we follow the logic discussed in Section 4.4 to adjust r when needed. All algorithms are implemented in C++ and run in memory. The experiments are conducted on a 622R CPU with 256GB of memory, on Win 10.

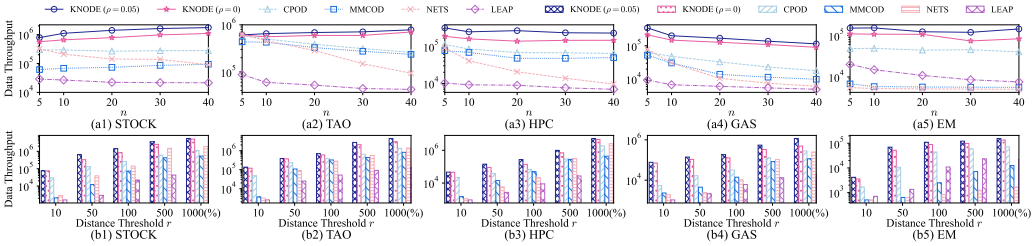


Fig. 4. The data throughput (unit: objects per second) performance of different algorithms vs. n and r.

Table 2. The Effect of ρ on KNODE (time:ms) / (**recall ratio:%**).

ρ	STOCK	TAO	HPC	GAS	EM
0.5	0.4 / **64.9**	0.1 / **74.3**	1.3 / **94.2**	10.4 / **90.8**	19.3 / **51.5**
0.3	1.0 / **84.3**	0.3 / **91.7**	5.9 / **98.9**	13.8 / **95.4**	24.7 / **93.6**
0.2	1.7 / **94.1**	0.5 / **95.6**	11.4 / **99.3**	18.9 / **97.6**	32.8 / **95.2**
0.1	2.9 / **97.2**	0.6 / **97.7**	15.6 / **99.9**	24.3 / **98.9**	37.4 / **98.5**
0.05	3.6 / **98.4**	0.7 / **98.1**	18.9 / **99.9**	29.7 / **99.6**	40.1 / **99.3**
0.01	5.3 / **98.5**	0.8 / **98.3**	32.7 / **99.9**	34.1 / **99.9**	44.5 / **99.9**
0	6.4 / **100.0**	0.8 / **100.0**	36.3 / **100.0**	39.4 / **100.0**	47.2 / **100.0**

5.2 Comparisons Under Fixed Parameters

In our first set of experiments, we compare KNODE and its competitors under fixed parameters. In this setup, the streaming data moves at a constant speed s, and the number of objects within the window and query parameters k and r are fixed. We evaluate KNODE's performance under two different values of ρ ($\rho = 0$ and $\rho = 0.05$), where $\rho = 0$ corresponds to the exact algorithm. We assume that all updates can be processed within the available time.

Overall Performance Comparison. Figures 4(a1)–(a5) report the average data throughput under varying window sizes n. Among all exact baselines, KNODE ($\rho = 0$) consistently achieves the highest throughput. On average, it outperforms CPOD, MMCOD, NETS, and LEAP by factors of 2.55×, 8.58×, 9.35×, and 18.49×, respectively. The efficiency gain stems from KNODE's ability to avoid per-object neighbor searches. Instead, it maintains only $\epsilon(NN_l)$ for k-nodes and $e(l_k^+, t, f, S)$ for a subset of nodes—quantities that are cheaper to update and far fewer in number than the objects in the window. In contrast, competitors incur higher costs: CPOD runs a range query for every new object; LEAP also searches neighbors per object; and while MMCOD and NETS skip searches in dense regions, their overhead grows when dimensionality rises or data becomes sparse, as dense regions shrink.

Relaxing the distance threshold via ρ further improves performance. Compared to exact search ($\rho = 0$), KNODE with $\rho = 0.05$ achieves an average speedup of 1.47×. The improvement arises because a positive ρ relaxes the neighbor-distance bound, allowing coarser node-level decisions that reduce fine-grained, point-wise classifications—albeit with a slight reduction in detection recall.

We report the data throughput of different algorithms across various values of r in Figures 4(b1)-(b5). The percentages on the x-axis (i.e., 10%, 50%, 100%, 500%, and 1000%) represent the ratio of tested r values to the default r settings listed in Table 1. Our results show that most algorithms' data throughput increases as r increases. This is because a larger r allows more objects to be classified as safe inliers, thereby reducing the maintenance costs. Again, KNODE ($\rho = 0$) outperforms all competitors across all datasets, and KNODE ($\rho = 0.05$) further improves the throughput by sacrificing its recall.

We also observe that the advantage of our KNODE approach over CPOD becomes smaller on the EM dataset. This is because EM has a relatively high dimensionality (16). KNODE is built on quad-tree-like

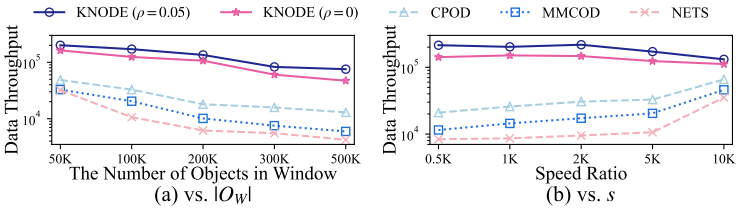


Fig. 5. Data throughput vs. window size and slide speed on count-based window (Gas dataset).

Table 3. Running Time Comparison of Knode (unit: ms).

Data Structures Used	Stock	Tao	Hpc	Gas	Em
$\epsilon(NN) + e(l_k)$	16.7	2.8	127.8	133.4	148.1
$\epsilon(NN_l) + e(l_k)$	11.2	1.6	71.2	74.3	92.7
$\epsilon(NN_l) + e(l_k^+, t, f, S)$	6.4	0.8	36.3	39.4	47.2

Table 4. Node-Based Evaluation: Single-thread vs. Multi-thread Execution Time (unit: ms).

Dataset	Stock	Tao	Hpc	Gas	Em
Single-thread	42.9	25.8	42.9	41.4	101.6
Multi-thread	**23.8**	**14.7**	**10.5**	**11.0**	**20.3**

structure, which is more effective in low-dimensional spaces, while Cpod uses an M-tree, which is specifically designed to handle higher-dimensional data more efficiently. Nevertheless, even in higher-dimensional spaces like EM, Knode still outperforms Cpod, demonstrating its robustness and adaptability. In addition, Leap consistently exhibits lower data throughput compared to the other algorithms, primarily due to its need to search through all neighbors for each newly arrived object. Consequently, we exclude Leap from the subsequent experiments for enhanced clarity.

The Impact of ρ. We further analyze ρ's impact on kNode's performance shown in Table 2. We find that when ρ is not excessively large (e.g., < 0.2), kNode can effectively retrieve 97% or more outliers. Moreover, while a smaller ρ helps detect more outliers, it also increases the computational cost. Conversely, a larger ρ reduces the running time but leads to more missed outliers. Therefore, the choice of ρ affects both performance and the recall, consistent with our expectations.

The Effect of $\epsilon(NN_l)$ and $e(l_k^+, t, f, S)$. To demonstrate the effectiveness of $\epsilon(NN_l)$, we report the running time required to support RPA-OD using different data structures in Table 3, with all parameters set to their defaults. The results show that using $\epsilon(NN_l)$ and $e(l_k^+, t, f, S)$ significantly reduces the running time across all datasets. The main reason is that $e(l_k^+)$ is sufficient to determine the outlier status of enclosed objects for many nodes and hence avoid the expensive computation of $e(l_k)$.

Comparisons under Count-based Window. Although this paper primarily focuses on time-based window, our techniques also support count-based windows. We evaluate the impact of window size and slide speed (i.e., slide ratio) on algorithm performance under a count-based window, using the Gas dataset for comparison. Experiments are conducted under two values of ρ ($\rho = 0.05$ and $\rho = 0$). To evaluate the impact of window size n on data throughput, we vary n from 50K to 500K while fixing all other parameters at their default. To evaluate the impact of the slide speed s, we vary s from 500 to 10K, again keeping all other parameters at default. As shown in Figure 5, Knode consistently achieves the best performance across all settings, under both $\rho = 0.05$ and $\rho = 0$. As discussed earlier, Knode avoids performing neighbor searches for each newly arrived object and

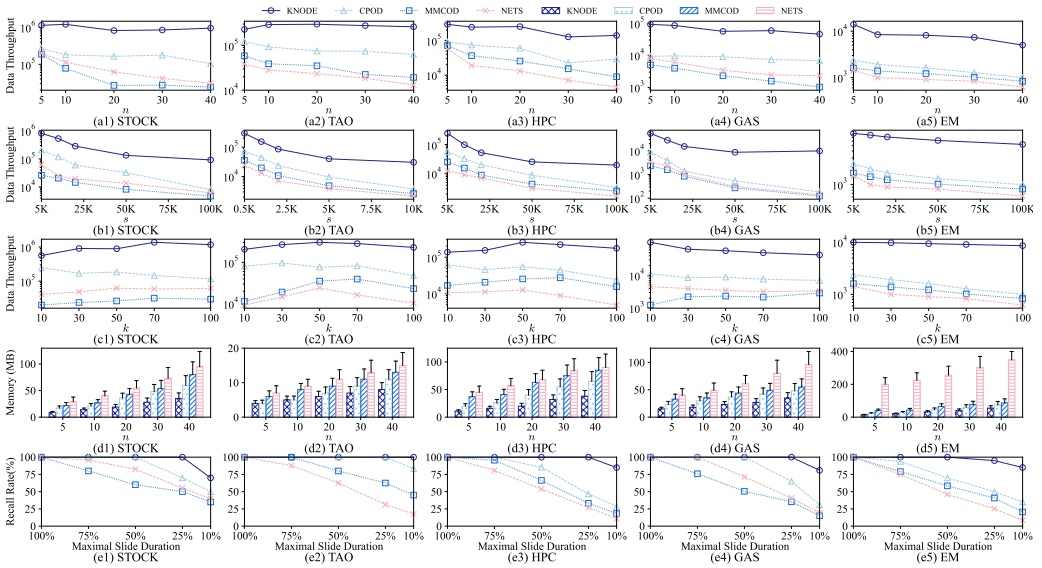


Fig. 6. The performance of different algorithms under dynamic parameters.

can identify inliers through node-level distance estimates. Consequently, it achieves both high efficiency and stable throughput across a wide range of window sizes and slide ratios.

Parallel execution of Node-Based Evaluation. To evaluate the parallelization capability of our framework, we implement a multi-threaded version of the Node-Based Evaluation module and compare it with the single-threaded implementation across five datasets. Table 4 reports the execution time of the two implementations. The results show that the multi-threaded implementation consistently reduces the processing time across different datasets, demonstrating that our method can effectively benefit from parallel computation.

5.3 Comparisons Under Dynamic Parameters

In our second set of experiments, we assess the performance of algorithms with dynamic parameters. In this setup, the streaming data moves at varying speeds, with s and $std(s)$ in Table 1 representing the average speed and standard deviation of speed per slide respectively. We monitor the proportion of outliers within the window and adjust r when the proportion falls outside the range $[\frac{\delta_0}{2}, 2\delta_0]$, where $\delta_0 = 1\%$. Since none of the competitors supports dynamic adjustment of r, we apply the algorithm presented in Section 4.4 to all the competitors to reset r when needed. Again, we assume that all updates can be processed within the available time (i.e., Knode performs exact search with $\rho = 0$), unless stated otherwise.

Data Throughput Comparison under Dynamic Parameters. We first examine the impact of n on algorithm performance. As shown in Figures 6(a1)-(a5), Knode significantly outperforms all other algorithms with substantial advantages. Compared to the results in Figure 4, the data throughput gap between Knode and its competitors widens. This is because frequent adjustments to the parameter r are necessary to keep the proportion of outliers stable. When r is reduced, Knode's competitors often need to re-search neighbors for many or even all objects, leading to significantly higher running costs. In addition, algorithms such as Cpod and Net have to re-form the index.

Next, we analyze the impact of s on algorithm performance, shown in Figures 6(b1)-(b5). Note, s indicates the average speed of streaming datasets in this setup, while the actual speed fluctuates dynamically. Once again, Knode demonstrates superior performance and it is more stable. At higher

Table 5. The average count and costs of adjusting r per 20 slides.

Dataset	Stock	Tao	Hpc	Gas	Em
Adjustment Count	0.25	1.00	0.10	2.00	1.50
Running Time Portion (%)	3.00	8.00	2.00	18.00	12.00

Table 6. Outlier ratios comparison in Gas (others/**Knode**, $\delta_0 = 1.0\%$).

Window	W_0	W_{10}	W_{20}	W_{30}	W_{40}	W_{50}	W_{60}	W_{70}
r value	2.7/**2.7**	2.7/**1.1**	2.7/**1.1**	2.7/**0.8**	2.7/**0.8**	2.7/**0.4**	2.7/**0.4**	2.7/**0.5**
Outlier(%)	1.00/**1.00**	0.35/**1.03**	0.11/**0.86**	0.07/**1.00**	0.03/**1.13**	0.05/**1.04**	0.00/**0.92**	0.05/**1.01**
Window	W_{80}	W_{90}	W_{100}	W_{110}	W_{120}	W_{130}	W_{140}	W_{150}
r value	2.7/**1.7**	2.7/**2.8**	2.7/**0.8**	2.7/**0.8**	2.7/**1.7**	2.7/**2.6**	2.7/**1.3**	2.7/**0.1**
Outlier(%)	0.43/**1.04**	2.05/**1.55**	0.46/**1.05**	0.41/**0.95**	0.11/**1.08**	0.98/**1.00**	0.16/**0.89**	0.08/**1.02**

speeds, when more objects are within the window, Knode can compute a tighter $e(l_k^+)$ using $\epsilon(NN_l)$ of the corresponding k-nodes, demonstrating that Knode is suitable for supporting RPA-OD under high speed stream.

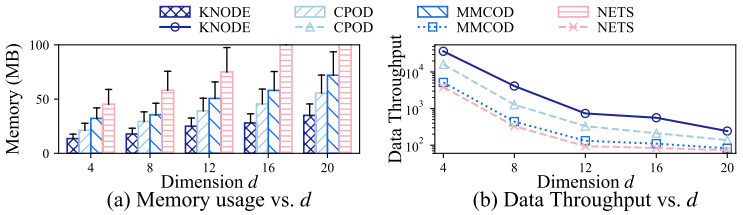
We then evaluate the impact of k on algorithm performance in Figures 6(c1)-(c5). Knode consistently demonstrates its superior results. For kNode, a larger k results in higher running costs to maintain $e(l_k^+, t, f, S)$; thus, as k increases, the data throughput decreases. However, kNode mitigates this by avoiding the need to search for ρ-neighbors for many objects, making it less sensitive to changes in k. In contrast, for Nets and MMcod, a larger k leads to fewer dense regions, requiring more neighbor searches and maintenance. For Cpod, an increase in k necessitates searching through more objects to identify inliers/outliers.

Memory Cost Comparison. Figures 6(d1)-(d5) report the average and peak memory usages of different algorithms under varying window sizes. We observe that Knode consistently consumes less memory, both in terms of average and peak usages, because it avoids maintaining per-object ρ-neighbor lists. Although Knode uses auxiliary structures, such as $\epsilon(NN_l)$ for k-nodes, $e(l_k^+, t, f, S)$ for selected descendant nodes, and $o(l_k, u_k, t, S(\rho_0))$ for a subset of objects, the number of these entries remains small, keeping overall space overhead low.

In contrast, all other methods must maintain neighbor lists for partial unsafe inliers and all outliers, which leads to significantly higher memory costs. In addition, Knode avoids re-constructing the index when updating r, further limiting its peak memory usage compared to competing approaches.

Recall ratio under various slide durations. By default, we assume that all updates can be processed within the available time, meaning that the duration of each slide is sufficient to handle all updates (i.e., $\rho = 0$ for RPA-OD and all algorithms achieve 100% recall). Under this setting, all algorithms have enough processing time to complete the entire workload within each window slide, which corresponds to a scenario similar to strong-scaling evaluation where sufficient computational resources are available for a fixed workload. However, as data streaming speed increases, this assumption may no longer hold. To investigate this, we evaluate the recall ratios of different algorithms under various slide durations, with results shown in Figures 6(e1)-(e5). The maximal slide duration is set such that all algorithms can process all updates within a single window slide. We then test slide duration at 100%, 75%, 50%, 25%, and 10% of this maximal duration. Decreasing the slide duration effectively increases the processing pressure on the algorithms, since the same workload must be handled within a shorter time interval. This setting emulates scenarios where the effective workload grows relative to the available computational resources, which can be interpreted as a weak-scaling–like condition in streaming environments.

Knode consistently achieves a near-perfect recall ratio, close to 100%, across different slide duration (except at 10%), outperforming all competitors. This is attributed to Knode's efficient outlier detection framework and its robust performance under reasonable values of ρ. For example,



(a) Memory usage vs. d (b) Data Throughput vs. d

Fig. 7. Performance vs. dimensionality d (Syn.).

Table 7. Impact of Latency Bounds on Gas dataset.

Window slide interval (unit:ms)	10	40	70	100	130	160	200
Average ρ value	4.5	0.6	0.05	0.01	0	0	0
Recall ratio (%)	65	87	98	99	100	100	100
Average range query executed	69	276	483	690	829	829	829

Knode only incurs relatively high running cost for objects located at sparse regions. Therefore, Knode remains efficient and maintains a high recall ratio even when the slide durations are short, demonstrating its strong ability to sustain performance as the effective workload increases.

In contrast, when streaming data moves at very high speeds, competing algorithms experience a significant reduction in recall ratios. Although these algorithms are designed to deliver exact answers for threshold-based outlier detection, they struggle to provide real-time outlier monitoring in high-speed streaming environments. This result indicates that our approach scales more gracefully under increasing workload pressure and is better suited for real-time outlier detection in fast-moving streaming data.

The adjustment of r. When the velocity or distribution of streaming data changes, adjusting the r value helps maintain a stable proportion of outliers. In our experiment, we set δ_0 to 1%. Table 5 reports the adjustment count and running time portion for every 20 slides across different datasets. The adjustment count records the average number of slides (out of 20) that trigger an adjustment of r, while the running time portion records the percentage of the running costs incurred by these adjustments. To visualize the impact of r adjustment, Table 6 reports the outlier ratios produced by the existing algorithms and by Knode on the Gas dataset, where "others" denotes the average outlier ratio of the competing algorithms using a fixed r, and W_i represents the query window after i slides. We draw two main conclusions. First, the proposed r adjustment mechanism is efficient. As shown in Table 5, the runtime overhead for adjusting r does not exceed 10% of the total runtime for most datasets. Second, existing algorithms cannot consistently maintain the desired outlier ratio as data distribution evolves over time, as shown in Table 6. These results indicate that dynamically adjusting r is necessary when the distribution of streaming data changes significantly.

Impact of Window Slide Duration. We also evaluate the impact of window slide duration (i.e., latency bounds) on the average ρ value, the recall, and the total number of range queries performed by Knode. Due to space limitations, we report results only for the Gas dataset; similar trends are observed on other datasets. The stream speed is fixed at 5,000 objects per window slide. We vary the window slide interval from **10 ms** + Δ to **200 ms** + Δ, where Δ ($\approx$ 4ms) accounts for objects insertions and deletions. As shown in Table 7, ρ dynamically adjusts to the window slide interval: it decreases as the interval grows, improving recall, and increases under tighter time constraints. When the interval reaches 130 ms, $\rho = 0$, allowing an exact search on Gas dataset with a 100% recall ratio. In addition, the number of range queries remains significantly smaller than the number of new objects (i.e., 5000). This efficiency stems from our k-node-based framework, which leverages the bound $e(l_k^+)$ to avoid unnecessary searches.

Impact of Dimensionality d. We assess the impact of dimensionality on algorithm performance, measured by data throughput and average memory usage. To this end, we introduce a new Synthetic dataset with up to 20 dimensions, containing 1 million objects, where each dimension follows a normal distribution ($\mu = 0$ and $\sigma = 1$). All parameters are set according to the default values discussed in Section 5.1. Figure 7 reports the performance of different algorithms across varying dimensionalities on Synthetic dataset. As dimensionality grows, throughput decreases and memory consumption rises for all methods. Nevertheless, Knode consistently achieves the best performance. This advantage stems from its ability to identify inliers using node-level distance reasoning.

6 Conclusion

This paper proposes a novel query named RPA-OD, representing the first attempt to integrate the advantages of both threshold-based and k_{max} or k_{avg}-based CDOD queries, while allowing for dynamic tuning of parameters r and ρ. We develop a set of algorithms that adaptively adjust the query parameters in response to real-time changes in streaming data, balancing efficiency and performance across varying data scales, speeds, dimensions, and distributions. Extensive experiments were conducted to demonstrate the superior performance of our proposed algorithms. In the near future, we will explore how the k-node based method can be extended to support RPA-OD in metric spaces.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62472293, U23A20309, U22A2025, 62232007), the National Key Research and Development Program of China (No. 2024YFF0617702), the Dameng Database Research Fund (No. 2025021900057, 2025021900058), and the 111 Project (No.B16009).

References

[1] Daichi Amagata, Makoto Onizuka, and Takahiro Hara. Fast and exact outlier detection in metric spaces: A proximity graph-based approach. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, SIGMOD '21, page 36–48, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Susik Yoon, Yooju Shin, Jae-Gil Lee, and Byung Suk Lee. Multiple dynamic outlier-detection from a data stream by exploiting duality of data and queries. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, SIGMOD '21, page 2063–2075, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Chengliang Chai, Lei Cao, Guoliang Li, Jian Li, Yuyu Luo, and Samuel Madden. Human-in-the-loop outlier detection. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 19–33, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Azzedine Boukerche, Lining Zheng, and Omar Alfandi. Outlier detection: Methods, models, and classification. *ACM Comput. Surv.*, 53(3), June 2020.

[5] Yunxiang Su, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. Distance-based outlier query optimization in apache iotdb. *Proceedings of the VLDB Endowment*, 17(11):2778–2790, 2024.

[6] R Lakshmi and TA Sajesh. A robust distance-based approach for detecting multidimensional outliers. *Journal of Applied Statistics*, 52(6):1278–1298, 2025.

[7] Shiblee Sadik and Le Gruenwald. Research issues in outlier detection for data streams. *Acm Sigkdd Explorations Newsletter*, 15(1):33–40, 2014.

[8] Jae-Gil Lee and Minseo Kang. Geospatial big data: challenges and opportunities. *Big Data Research*, 2(2):74–81, 2015.

[9] Shiblee Sadik, Le Gruenwald, and Eleazar Leal. Wadjet: Finding outliers in multiple multi-dimensional heterogeneous data streams. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1232–1235, 2018.

[10] Luan Tran, Liyue Fan, and Cyrus Shahabi. Distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment*, 9(12):1089–1100, 2016.

[11] Ilyeop Yi, Jae-Gil Lee, and Kyu-Young Whang. Apam: Adaptive eager-lazy hybrid evaluation of event patterns for low latency. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2275–2280, 2016.

[12] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. Hades-iot: A practical host-based anomaly detection system for iot devices. In *Proceedings of the 2019 ACM Asia conference on computer and communications security*, pages 479–484, 2019.

[13] Nicholas Duffield, Patrick Haffner, Balachander Krishnamurthy, and Haakon Andreas Ringberg. Systems and methods for rule-based anomaly detection on ip network flow, February 9 2016. US Patent 9,258,217.

[14] Yuhang Lin, Byung Suk Lee, and Daniel Lustgarten. Continuous detection of abnormal heartbeats from ecg using online outlier detection. In *Annual International Symposium on Information Management and Big Data*, pages 349–366. Springer, 2018.

[15] Marina Thottan and Chuanyi Ji. Anomaly detection in ip networks. *IEEE Transactions on signal processing*, 51(8):2191–2204, 2003.

[16] Sean Zhang, Varun Ursekar, and Leman Akoglu. Sparx: Distributed outlier detection at scale. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 4530–4540, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Theodoros Toliopoulos, Christos Bellas, Anastasios Gounaris, and Apostolos Papadopoulos. Proud: Parallel outlier detection for streams. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2717–2720, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A Lozano. A review on outlier/anomaly detection in time series data. *ACM computing surveys (CSUR)*, 54(3):1–33, 2021.

[19] Siddharth Bhatia, Rui Liu, Bryan Hooi, Minji Yoon, Kijung Shin, and Christos Faloutsos. Real-time anomaly detection in edge streams. *ACM Trans. Knowl. Discov. Data*, 16(4), January 2022.

[20] Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and Elke A Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *IEEE 30th International Conference on Data Engineering (ICDE)*, pages 76–87. IEEE, 2014.

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB*, volume 97, page 426. Citeseer.

[22] Allen Klinger and Charles R Dyer. Experiments on picture representation using regular decomposition. *Computer graphics and image processing*, 5(1):68–105, 1976.

[23] Yan Zhao, Xuanhao Chen, Liwei Deng, Tung Kieu, Chenjuan Guo, Bin Yang, Kai Zheng, and Christian S. Jensen. Outlier detection for streaming task assignment in crowdsourcing. WWW '22, page 1933–1943, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Daichi Amagata, Makoto Onizuka, and Takahiro Hara. Fast, exact, and parallel-friendly outlier detection algorithms with proximity graph in metric spaces. *VLDB J.*, 31(4):797–821, 2022.

[25] Tung Kieu, Bin Yang, Chenjuan Guo, Christian S. Jensen, Yan Zhao, Feiteng Huang, and Kai Zheng. Robust and explainable autoencoders for unsupervised time series outlier detection. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3038–3050, 2022.

[26] Matteo Corain, Paolo Garza, and Abolfazl Asudeh. Dbscout: A density-based method for scalable outlier detection in very large datasets. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 37–48, 2021.

[27] Yuening Li, Zhengzhang Chen, Daochen Zha, Kaixiong Zhou, Haifeng Jin, Haifeng Chen, and Xia Hu. Autood: Neural architecture search for outlier detection. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2117–2122, 2021.

[28] Jiali Mao, Tao Wang, Cheqing Jin, and Aoying Zhou. Feature grouping-based outlier detection upon streaming trajectories (extended abstract). In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1745–1746, 2018.

[29] Lei Cao, Yizhou Yan, Caitlin Kuhlman, Qingyang Wang, Elke A Rundensteiner, and Mohamed Eltabakh. Multi-tactic distance-based outlier detection. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 959–970. IEEE, 2017.

[30] Mahsa Salehi, Christopher Leckie, James C. Bezdek, Tharshan Vaithianathan, and Xuyun Zhang. Fast memory efficient local outlier detection in data streams. *IEEE Trans. Knowl. Data Eng.*, 28(12):3246–3260, 2016.

[31] Lei Cao, Yizhou Yan, Yu Wang, Samuel Madden, and Elke A. Rundensteiner. Autood: Automatic outlier detection. In *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data*, SIGMOD '23, pages 20:1–20:27, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Zehai Yang and Shimin Chen. Most: Model-based compression with outlier storage for time series data. In *Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*, SIGMOD '24, pages 250:1–250:29, New York, NY, USA, 2024. Association for Computing Machinery.

[33] Ali Degirmenci and Omer Karal. Efficient density and cluster based incremental outlier detection in data streams. *Information Sciences*, 607:901–920, 2022.

[34] Jiaqi Zhu, Shaofeng Cai, Fang Deng, Beng Chin Ooi, and Wenqiao Zhang. Meter: A dynamic concept adaptation framework for online anomaly detection. *Proceedings of the VLDB Endowment*, 17(4):794–807, March 2024.
[35] David Campos, Tung Kieu, Chenjuan Guo, Feiteng Huang, Kai Zheng, Bin Yang, and Christian S. Jensen. Unsupervised time series outlier detection with diversity-driven convolutional ensembles. *Proceedings of the VLDB Endowment*, 15(3):611–623, November 2021.
[36] Di Yang, Elke A Rundensteiner, and Matthew 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*, pages 529–540, 2009.
[37] Maria Kontaki, Anastasios Gounaris, Apostolos N Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *IEEE 27th International Conference on Data Engineering*, pages 135–146. IEEE, 2011.
[38] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. Nets: extremely fast outlier detection from a data stream via set-based processing. *Proceedings of the VLDB Endowment*, 12(11):1303–1315, 2019.
[39] Luan Tran, Min Y Mun, and Cyrus Shahabi. Real-time distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment*, 14(2):141–153, 2020.
[40] Fabrizio Angiulli and Fabio Fassetti. Detecting distance-based outliers in streams of data. In *Proceedings of the 16th ACM International on Conference on Information and Knowledge Management*, pages 811–820, 2007.
[41] Rui Zhu, Tiantian Yu, Zhiyuan Tan, Wei Du, Liang Zhao, Jiajia Li, and Xiufeng Xia. Ptaod: A novel framework for supporting approximate outlier detection over streaming data for edge computing. *IEEE Access*, 8:1475–1485, 2019.
[42] Panthadeep Bhattacharjee, Ankur Garg, and Pinaki Mitra. Kago: an approximate adaptive grid-based outlier detection approach using kernel density estimate. *Pattern Analysis and Applications*, 24(4):1825–1846, 2021.
[43] Sergey Brin. Near neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 574–584. Morgan Kaufmann, 1995.
[44] Yunjun Gao, Lu Chen, Xinhan Li, Bin Yao, and Gang Chen. Efficient k-closest pair queries in general metric spaces. *VLDB J.*, 24(3):415–439, 2015.
[45] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 189–200. ACM, 2000.
[46] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. Efficient metric indexing for similarity search. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 591–602. IEEE Computer Society, 2015.
[47] Rui Zhu, Bin Wang, Xiaochun Yang, and Baihua Zheng. Closest pairs search over data stream. In *Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*, SIGMOD '24, pages 1–26, New York, NY, USA, 2024. Association for Computing Machinery.
[48] *URL https://www.pmel.noaa.gov/tao/drupal/disdel/.*
[49] Dheeru Dua, Casey Graff, et al. Uci machine learning repository, 2017. *URL http://archive. ics. uci. edu/ml*, 7(1):62, 2017.
[50] Elvezio M Ronchetti and Peter J Huber. *Robust statistics*. John Wiley & Sons Hoboken, NJ, USA, 2009.
[51] Luan Tran, Liyue Fan, and Cyrus Shahabi. Fast distance-based outlier detection in data streams based on micro-clusters. In *Proceedings of the 10th International Symposium on Information and Communication Technology*, pages 162–169, 2019.

Received October 2025; revised January 2026; accepted February 2026