Distance-based Outlier Query Optimization in Apache IoTDB

Yunxiang Su
Tsinghua University
suyx21@mails.tsinghua.edu.cn

Shaoxu Song*
Tsinghua University
sxsong@tsinghua.edu.cn

Xiangdong Huang
Tsinghua University
huangxdong@tsinghua.edu.cn

Chen Wang
Tsinghua University
wang_chen@tsinghua.edu.cn

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

While outlier detection has been widely studied over streaming data, the query of outliers in time series databases was largely overlooked. Apache IoTDB, an open-source time series database, employs LSM-tree based storage to support intensive writing workloads, yet this storage structure unfortunately encumbers the outlier query performing. In the system, data points of a time series may be stored in multiple files with overlapping time ranges, owing to the far delayed data arrivals, which are simply discarded in streaming outlier detection. Given the overlapping time ranges, it is not able to detect outliers in each file and merge them as the results. In this paper, we focus on optimizing the efficiency of distance-based outlier query in Apache IoTDB, with the consideration of overlapping files for delayed data. We propose to utilize bucket statistics of the values stored in files. Upper and lower bounds on the neighbor counts of data points are derived in buckets and overlapping files for efficient pruning. Extensive experiments demonstrate the efficiency of our proposal in the LSM-tree based time series database, Apache IoTDB, compared to the existing outlier detection methods designed for data streams.

PVLDB Reference Format:
Yunxiang Su, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. Distance-based Outlier Query Optimization in Apache IoTDB. PVLDB, 17(11): 2778 - 2790, 2024.
doi:10.14778/3681954.3681962

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/iotdb-lsmod/iotdb-lsmod.

1 INTRODUCTION

Owing to the simple yet intuitive definition, distance-based outliers [16, 19, 20] are frequently queried in time series, e.g., filtering spikes of stock prices, detecting deviations in GPS trajectories, and identifying sudden change points of temperature when cold air rushes in. It filters out data points that do not have sufficient neighbors on values in a period of time.

By distinct requirements of "sufficient neighbors" in applications, distance-based outliers are often queried with different parameter settings for various interests. (1) By varying the neighbor distance threshold r, it alters how distant a neighbor could be. For instance, over the same GPS data, the destination prediction application concerns outliers in miles, while trajectory analysis is often in feet. (2) Different neighbor count threshold k means distinct requirements on the least number of neighbors. For example, point-of-interest analysis is more sensitive to outliers with less neighbors than trajectory clustering. (3) The window size w specifies various time range of points under consideration in neighbor count. For instance, over the same temperature data, the climate change analysis concerns outliers in years, while the ice forecast of wind turbines in wind farms worries about outliers in hours. (4) The slide size s indicates how often the outliers are queried. It is not necessary to detect outliers and forecast atmospheric temperature in every minute, but meaningful in hours for ice forecast and in days for climate change analysis. Unfortunately, while detecting outliers over streaming data has been widely studied [16, 19, 20], the query of outliers with various parameter settings for different applications in time series databases was largely overlooked.

1.1 Challenge

One of the major differences between streaming outlier detection and in-database outlier detection is on handling the delayed data. Once the timestamps of the delayed points exceed the currently processing window in data streams, the existing outlier detection methods for streaming data, e.g., CPOD [20], NETS [23] and MCOD [19], neglect them. In contrast, all these delayed data are stored in a time series database and processed in the consequent outlier query.

To efficiently handle the out-of-order arrivals, we employ Log-Structured Merge-Tree (LSM-tree) [18] in the design of Apache IoTDB [22], a time series database management system. In short, the data points that arrive within a period are batched and stored in a file. The delayed points are then processed in the subsequent batches and stored in another file with a higher version number, leading to files with possibly overlapping time ranges.

The separation of delayed data is efficient in handling intensive writes, a must in IoT scenarios, but unfortunately encumbers query processing [15], including distance-based outlier queries. Owing to the overlapping time ranges of files, we cannot simply detect outliers in each file and merge the results (see Example 1.1). Existing outlier detection methods for streaming data [16, 19, 20] obviously do not consider such a scenario of time-overlapping files, but simply assume that they can be processed in chronological order.

---

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681962

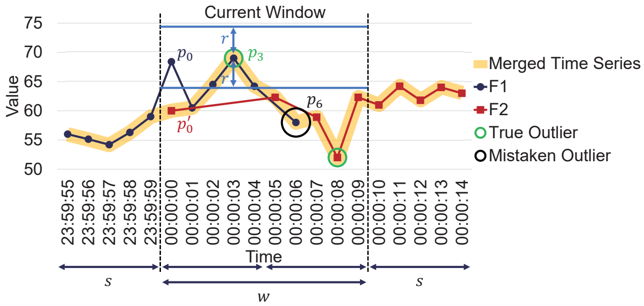

Figure 1: Querying outliers of a time series stored in two files, F1 & F2, with neighbor distance threshold $r = 5$, neighbor count threshold $k = 4$, window size $w = 10$, slide size $s = 5$

Example 1.1. Figure 1 presents a time series (denoted by the yellow thick line), monitoring the water temperature of a vehicle engine. Owing to abnormal operations or sensor failures, two points at 00:00:03 and 00:00:08 have values deviating from others, i.e., less than $k = 4$ points (including itself) whose value distance is no greater than $r = 5$ in the current window. They should be returned as outliers referring to distance-based outlier definition [19].

In practice, data points may arrive in the database out-of-order due to the transmission issues, and thus one time series may be stored in multiple files with overlapping time ranges. For instance, the time series in yellow is stored in two files, F_1 and F_2. When the delayed point at 00:00:05 arrives, the previously received points (in dark blue) including those with larger timestamps such as 00:00:06 have been written to file F_1 on disk. Thereby, the point is batched with other recently received points (in red) and stored in another file F_2 with a higher version number 2. Even worse, some points may be overwritten by later arrivals, stored in different files. For example, the timestamp of point p_0 is occasionally reset to 00:00:00 and stored in F_1. The actual point p_0' at 00:00:00 is received later and stored in F_2. Hence, the point p_0' at 00:00:00 in F_2 overwrites p_0 in F_1, which should also be considered in querying outliers.

Note that detecting outliers separately in each file may lead to mistaken outliers. Consider the point p_6 at 00:00:06. It has only 1 other point at 00:00:01 stored in F_1, whose value distance to p_6 is less than $r = 5$ in the current window, making p_6 an outlier in F_1. However, by considering the points in F_2, there are 4 other points at 00:00:00, 00:00:05, 00:00:07, 00:00:09 stored in F_2, whose value distances are also less than $r = 5$. Thereby, p_6 is mistakenly detected as an outlier if only considering the points in F_1, while p_6 is indeed an inlier given the points stored in both F_1 and F_2.

On the other hand, true outliers could be missed by detecting in an individual file. Though the point p_3 at 00:00:03 has 3 other points within distance $r = 5$ in F_1, i.e., 00:00:00, 00:00:02, 00:00:04, the point at 00:00:00 is overwritten by F_2 due to the aforesaid reason. It is no longer within distance $r = 5$ of p_3 after overwriting. In other words, p_3 is an inlier if only considering the points in F_1, while it is indeed an outlier given the points in both F_1 and F_2.

1.2 Contribution

Our main contributions are as follows.

Table 1: Notations

Symbol	Description
T	a time series with data points p
r	neighbor distance threshold
k	neighbor count threshold
w	window size
s	slide size
$N(p, r)$	the r-neighbor set of point p
W_t	a window starting from time t with ω segments
O_t	the outlier set in the window W_t
F_h	a file of segments with version number h
S_g	the g-th segment with range δ on time
$B[u]$	the u-th bucket with width γ on value
$\lvert\check{B}[u]\rvert, \lvert\hat{B}[u]\rvert$	lower and upper bounds of bucket size $\lvert B[u]\rvert$

(1) We formalize the problem of querying outliers in the LSM-tree based time series database in Section 2, including the storage structure, and the corresponding SQL statement in Apache IoTDB.

(2) We start from query processing on single file in Section 3. The values are organized in buckets with pre-computed statistics, which can be further aggregated for querying outliers. Propositions 3.5-3.7 establish the upper and lower bounds of neighbor count using bucket statistics, to facilitate efficient pruning.

(3) We extend the query processing to multiple files in Section 4. Given the files with overlapping time ranges, though the bucket statistics cannot be directly aggregated, we derive bounds w.r.t. the bucket statistics in Proposition 4.1. It leads to the upper and lower bounds on neighbor count for multiple files in Propositions 4.3-4.6. The bounds enable efficient pruning for multiple files.

(4) We conduct extensive experimental evaluation in Section 5. Under various data loads, the proposed LSMOD with pruning by bucket statistics is always more efficient than the existing streaming outlier detection methods.

The outlier query has been deployed as a function in Apache IoTDB [1]. The document is included in the product website [2]. The code is committed in the GitHub repository of IoTDB [3]. The experiment code and data are available in [4] for reproducibility. All proofs can be found in [5].

2 PRELIMINARY

In this section, we first introduce the distance-based outliers in time series in Section 2.1. The elements of LSM-tree storage for time series are presented in Section 2.2. It leads to the query of outliers in the LSM-tree based time series database in Section 2.3. Table 1 lists the frequently used notations in this paper.

2.1 Distance-Based Outliers

A point $p(t, v)$ is a pair of time t and value v in a univariate time series T of columnar store. Given a distance metric $dist(p, p') = |p.v - p'.v|$ of two points, the neighbors and outliers in a window $W \subseteq T$ are defined as follows [19].

Definition 2.1 (Neighbor). Given distance threshold $r(r > 0)$, a point p is a r-neighbor of point p' in a window W, if

$$dist(p, p') \le r.$$

Note that p is the neighbor of itself. Let $N(p, r) = \{p' \in W \mid dist(p, p') \le r\}$ denote the set of r-neighbors of p in W. An outlier is a point without sufficient neighbors.

Definition 2.2 (Distance-based outlier). Given distance threshold r, and count threshold k, a point p is a distance-based outlier in window W, if

$$|N(p, r)| < k.$$

The problem of Distance-based Outlier Detection [19] is to detect outliers in each sliding window over the time series T.

Definition 2.3 (Distance-based outlier detection). Given distance threshold r, count threshold k, window size w and slide size s of a time series T, the problem is to detect the distance-based outliers in every sliding window $\dots, W_{t-s}, W_t, W_{t+s}, \dots$, where W_t is a window starting from timestamp t and ending at timestamp $t+w$, i.e., $W_t = \{p \in T \mid t \le p.t < t + w\}$.

Example 2.4 (Example 1.1 continued). Consider again the query in Figure 1 with neighbor distance threshold $r = 5$, neighbor count threshold $k = 4$, window size $w = 10$ and slide size $s = 5$. For simplicity, we use p_i to denote the point at time 00:00:i, e.g., p_3 with timestamp 00:00:03. As aforesaid, p_2 is a r-neighbor of p_3 with value distance $dist(p_3, p_2) = 4.5 < r$. It leads to a set of r-neighbors $N(p_3, r) = \{p_2, p_3, p_4\}$. Since the r-neighbor count is less than $k = 4$, point p_3 is detected as an outlier. For the window starting from 00:00:00 and ending at 00:00:09, denoted by $W_{00:00:00}$, its outliers are $O_{00:00:00} = \{p_3, p_8\}$. Likewise, for the next window with slide s, the outliers in $W_{00:00:05}$ are $O_{00:00:05} = \{p_8\}$.

2.2 LSM-tree Storage

We consider the storage structure of Log-Structured Merge-Tree (LSM-tree) [18], for handling intensive writes in Apache IoTDB [22]. The data points in a time series T are stored in multiple files, written to the database at different time.

Definition 2.5 (Segment). A time series T is represented by a number of segments, $T = S_1 \cup \dots \cup S_\tau$, where each segment has

$$S_g = \{p \in T \mid t_{\min} + (g-1)\delta \le p.t < t_{\min} + g\delta\},$$

$g = 1, \dots, \tau$ and $\tau = \lfloor \frac{t_{\max} - t_{\min}}{\delta} \rfloor + 1$.

That is, segments are with non-overlapping time range δ. To support query processing in sliding windows, the window size w and slide s are usually the integral multiples of δ. Figure 2(a) shows a series of segments $S_1^{(3)}, \dots, S_\tau^{(3)}$ stored in a file with ordered timestamps. The values in each segment are furthered divided into buckets for statistics, which will be introduced later in Section 3.1.

If all the points p come in the order of their timestamps t, the segments can be written one by one in the time order. Unfortunately, it is not the case in practice, where points may be delayed. Inserting the delayed points in the segments that have been written to disk is costly. LSM-tree based storage chooses to store the delayed points in other files, leading to multiple versions of a segment.

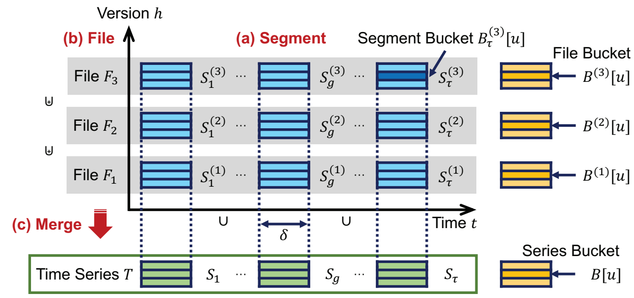

Figure 2: A time series T stored in segments S_g and files F_h

Definition 2.6 (File). A file $F_h = S_1^{(h)} \cup \dots \cup S_\tau^{(h)}$ consists of segments written with version number h.

Figure 2(b) shows multiple files with different version numbers, i.e., F_1, F_2 and F_3, written at different time. One segment may also have multiple versions, for example, 3 versions for segment S_1, i.e., $S_1^{(1)}, S_1^{(2)}, S_1^{(3)}$, written at different time. To obtain the time series T, we need to merge the segments $S_g^{(h)}$ stored in different files F_h.

Definition 2.7 (Merge). Segments of a time series from two files F_i and F_j, $i > j$, can be merged by

$$S_g^{(i)} \uplus S_g^{(j)} = S_g^{(i)} \cup \{p \in S_g^{(j)} \mid p.t \ne p'.t, p' \in S_g^{(i)}\}.$$

That is, a point $p \in S_g^{(j)}$ with lower version number j is overwritten by another point $p' \in S_g^{(i)}$ with higher version number i and the same timestamp $p'.t = p.t$, if exists. Though such updates are not very prevalent in IoT scenarios, it occasionally occurs by filling a default value of a delayed point for data analysis and updating it on arrival later, or simply data correction [24].

Proposition 2.8. *A time series T stored in multiple files and segments can be obtained by*

$$T = \bigcup_g S_g = \bigcup_g \left(\biguplus_h S_g^{(h)} \right).$$

Example 2.9. Figure 2 shows a time series T stored in three files, $\{F_1, F_2, F_3\}$. Each file has at most τ segments, with non-overlapping time range δ, e.g., $F_1 = S_1^{(1)} \cup \dots \cup S_\tau^{(1)}$. By merging the segments in different files with the same time range, e.g., $S_1^{(1)}$ in file F_1, $S_1^{(2)}$ in file F_2, and $S_1^{(3)}$ in file F_3, referring to the merge operator in Definition 2.7, we obtain segment S_1 of time series T in the corresponding time range, i.e., $S_1 = S_1^{(1)} \uplus S_1^{(2)} \uplus S_1^{(3)}$ in Figure 2(c). Finally, the time series T is obtained by concatenating all the segments, $T = S_1 \cup \dots \cup S_\tau = (S_1^{(1)} \uplus S_1^{(2)} \uplus S_1^{(3)}) \cup \dots \cup (S_\tau^{(1)} \uplus S_\tau^{(2)} \uplus S_\tau^{(3)})$.

2.3 Outlier Query on LSM-tree Storage

We are now ready to introduce the outlier query in LSM-tree based time series databases.

Definition 2.10 (LSMOD). Given distance threshold r, count threshold k, window size w and slide size s, the problem is to efficiently detect the distance-based outliers in every sliding window $\dots, W_{t-s}$,

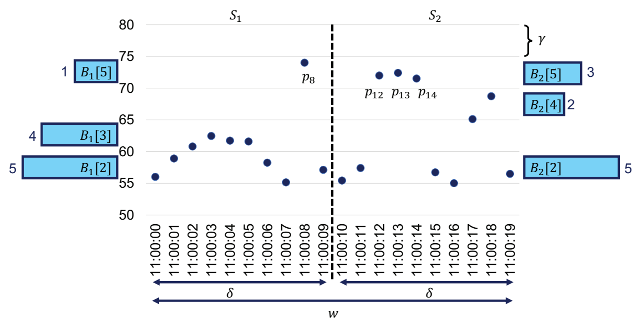

Figure 3: Buckets with $\gamma = 5$ of two segments with $\delta = 10$

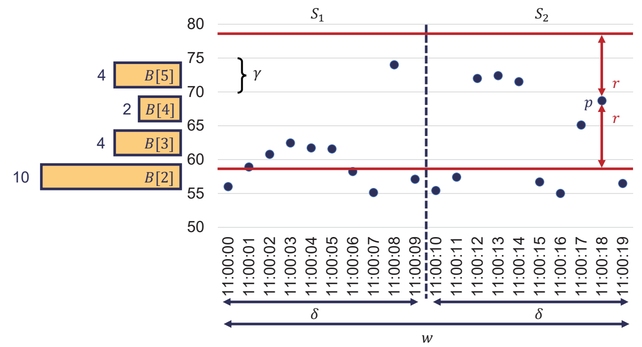

Figure 4: Aggregating bucket statistics and pruning point p in a window W

$W_t, W_{t+s}, \ldots$, of a time series $T = \bigcup_g \left(\uplus_h S_g^{(h)}\right)$ stored in multiple segments and files.

We follow the convention of Apache IoTDB to specify the sliding windows. The SQL statement of LSMOD is given below.

```
select outlier(s0, 'r'='5', 'k'='3',
                   'w'='1d', 's'='3h')
from root.d0
where time >= 2017-11-01T00:00:00
and time <= 2017-11-07T23:00:00
```

It queries outliers in the time range of [2017-11-01T00:00:00, 2017-11-07T23:00:00] of the time series T = root.d0.s0. The window has size $w = 1$d (1 day) and slide $s = 3$h (3 hours). The neighbor distance threshold is $r = 5$, and the neighbor count threshold is $k = 3$. The query output is the set of outlier points O_t for each sliding window.

3 QUERY PROCESSING IN SINGLE FILE

In this section, we first present the outlier query processing in single files, and extend it to multiple files in Section 4. Intuitively, we may utilize the statistics in files to prune data points with sufficient/insufficient neighbors for sure. Section 3.1 introduces the statistics in segments, and Section 3.2 aggregates them for the query. Without loading all the data, we derive the bounds of neighbor counts for points in Section 3.3 and enable pruning in Section 4.3.

3.1 Statistics on Buckets of Values in a Segment

In order to efficiently identify neighbors, we propose to further divide the points in a segment into multiple buckets, referring to their values. For instance, each segment is divided into three buckets as shown in Figure 2(a). Since only one file is considered in this section, we omit the default file version number h for simplicity.

Definition 3.1 (Bucket). For a time series segment S_g, we consider a number of buckets $\mathcal{B}_g = \{B_g[1], ..., B_g[\beta]\}$, where each bucket has a width γ on value range,

$$B_g[u] = \{p \in S_g \mid v_{\min} + (u-1)\gamma \leq p.v < v_{\min} + u\gamma\},$$

$u = 1, \ldots, \beta$ and $\beta = \lfloor \frac{v_{\max} - v_{\min}}{\gamma} \rfloor + 1$.

That is, all the buckets have non-overlapping value range γ. Any two points in a bucket must have distance less than γ. Such bounds on distances can be utilized below to find neighbors, and the bucket sizes $|B[u]|$ may further contribute to the bounds of neighbor count. Thus, we can identify outliers and inliers without loading data.

Example 3.2. Consider a window containing two segments, S_1 and S_2, with time range $\delta = 10$ in Figure 3. Each segment is divided into a set of buckets with non-overlapping value range $\gamma = 5$, e.g., $S_1 = B_1[1] \cup B_1[2] \cup \cdots \cup B_1[6]$. For instance, we have $B_1[5] = \{p_8\}$. For simplicity, we omit the empty buckets without any point. In addition to the raw data points stored in buckets in a file, we also record the bucket sizes as the statistics for pruning below, i.e., $|B_1[2]| = 5, |B_1[3]| = 4, |B_1[5]| = 1$.

3.2 Aggregating Bucket Statistics in a Window

As introduced in Section 2.3, the query considers outliers in each window. The multiple segments in the window need to be merged for the query. Likewise, the bucket statistics of multiple segments should also be aggregated for the window.

Proposition 3.3 (Window statistics). *For a window with segments, $W = S_1 \cup \cdots \cup S_\omega$, its statistics on buckets can be aggregated as*

$$|B[u]| = \sum_{g=1}^{\omega} |B_g[u]|,$$

where $|B_g[u]|$ is the size of the u-th bucket in segment S_g.

In other words, each window W also has β buckets, with width γ on value range as introduced in Definition 3.1. Each bucket $B[u]$ of the window is the merge of the corresponding buckets $B_g[u]$ in each segment S_g. Since the segments are non-overlapping in time range, i.e., no point overwriting, their bucket sizes can be directly aggregated. Figure 4 presents the buckets by merging/aggregating the two segments in Figure 3.

Example 3.4 (Example 3.2 continued). Consider again the window with two segments in Figure 3. The window buckets can be obtained by merging the corresponding segment buckets, e.g., $B[5] = B_1[5] \cup B_2[5] = \{p_8, p_{12}, p_{13}, p_{14}\}$. Note that p_i denotes the point with timestamp 11:00:i. The corresponding window bucket statistics can thereby be directly aggregated without loading the data, $|B[5]| = |B_1[5]| + |B_2[5]| = 4$. Figure 4 illustrates the aggregated

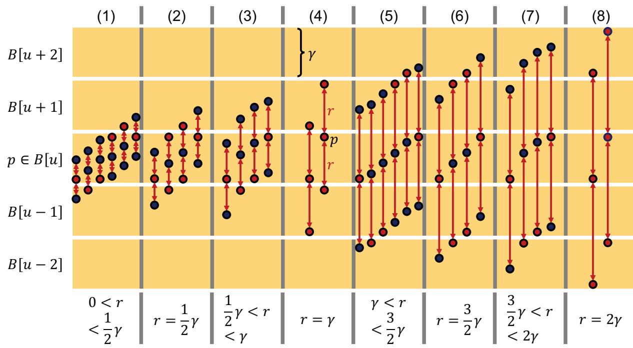

Figure 5: Possible r-neighbor scenarios of point p, for 8 example relationships with r ranging from 0 to 2γ

bucket statistics $|B[u]|$ for the window W in Figure 3, where the lengths of bars represent the bucket sizes.

3.3 Bounds of Neighbor Count

To identify an outlier, as introduced in Definition 2.2, it is essential to determine its r-neighbor count, $|N(p,r)|$. Intuitively, referring to the bucket width γ and size $|B[u]|$, we may derive the bounds of neighbor counts for the points in a bucket. In short, we have two cases for the upper bounds of the r-neighbor count, and another case for the lower bound as follows.

3.3.1 Upper Bound. Note that the query-specified neighbor distance threshold r may exhibit various relationships to the fixed bucket width γ. We present 8 examples on the relationships between r and γ to give an intuitive comprehension, which could further be inducted to cases (i) and (ii) for upper bounds below. Intuitively, consider a threshold r smaller than half of the bucket width γ, as shown in Figure 5(1). The r-neighbors of a point p can only appear in its bucket or one of the adjacent buckets, where the two points above and below p denote its most distant r-neighbors.

In contrast, given a threshold r larger than half of the bucket width γ, as shown in Figure 5(3), the r-neighbors of a point p may appear in its bucket and both of the adjacent buckets. In this sense, the upper bounds of r-neighbors are different regarding the relationships between r and $\frac{1}{2}\gamma$ as follows.

Case (i): $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$. It is the case of $(\lambda - \frac{1}{2})\gamma < r \leq \lambda\gamma$, for $\lambda = 1, 2, \ldots$, in general. For instance, for $\lambda = 1$, $\frac{1}{2}\gamma < r \leq \gamma$ corresponds to the aforesaid intuition of threshold r larger than half of the bucket width γ, i.e., the examples in Figures 5(3) and 5(7). In this case, the points in the buckets $B[u-\lambda], \ldots, B[u+\lambda]$ could be potential r-neighbors of a point $p \in B[u]$.

Proposition 3.5. *For a point p in bucket $B[u]$, if $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$, then the count of its r-neighbors has an upper bound*

$$|N(p,r)| \leq \sum_{i=u-\lambda}^{u+\lambda} |B[i]|,$$

where $\lambda = \lceil r/\gamma \rceil$.

Case (ii): $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$. It is the case of $(\lambda-1)\gamma < r \leq (\lambda - \frac{1}{2})\gamma$, for $\lambda = 1, 2, \ldots$. Likewise, the aforesaid intuition of threshold r smaller than half of the bucket width γ corresponds to $0 < r \leq \frac{1}{2}\gamma$, for $\lambda = 1$, i.e., the examples in Figures 5(1) and 5(5). In this case, the potential r-neighbors of a point $p \in B[u]$ can be further restricted to less buckets, i.e., either the points in the buckets $B[u-\lambda], \ldots, B[u+\lambda-1]$ or in $B[u-\lambda+1], \ldots, B[u+\lambda]$.

Proposition 3.6. *For a point p in bucket $B[u]$, if $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$, then the count of its r-neighbors has an upper bound*

$$|N(p,r)| \leq \max\left(\sum_{i=u-\lambda}^{u+\lambda-1} |B[i]|, \sum_{i=u-\lambda+1}^{u+\lambda} |B[i]|\right),$$

where $\lambda = \lceil r/\gamma \rceil$.

3.3.2 Lower Bound. Intuitively, if the neighbor distance threshold r is less than the bucket width γ, i.e., $r < \gamma$, it is possible that the r-neighbor count is less than $|B[u]|$, and thus could not be bounded by the bucket statistics. For instance, there is no lower bound for the examples in Figure 5(1)(2)(3). On the other hand, for $r \geq \gamma$, the points in the bucket $B[u]$ are at least the r-neighbors of any point $p \in B[u]$. Referring to the intuition, a tighter lower bound could be derived for general cases as follows.

Proposition 3.7. *For a point p in bucket $B[u]$, if $r \geq \gamma$, then the count of its r-neighbors has a lower bound*

$$|N(p,r)| \geq \sum_{i=u-\ell+1}^{u+\ell-1} |B[i]|,$$

where $\ell = \lfloor r/\gamma \rfloor$.

4 QUERY PROCESSING IN MULTIPLE FILES

In this section, we consider multiple files written at various time with different version numbers. Unfortunately, some bucket statistics in a file introduced in Section 3.1 may no longer be valid, since the points may be overwritten by other delayed data with higher file version numbers. Nevertheless, we study the bounds of bucket statistics considering multiple files in Section 4.1. Likewise, the lower and upper bounds of neighbor count can still be derived in Section 4.2, enabling the pruning in multiple files in Section 4.3. Finally, Section 4.4 presents the algorithm for outlier query processing in multiple files.

4.1 Bound of Bucket Statistics in Multiple Files

Though the bucket statistics of a window in single file can be aggregated by Proposition 3.3, they cannot be further aggregated in multiple files, due to the merge operation in Definition 2.7 with the consideration of overwritten points. Nevertheless, the file with the highest version number would not be overwritten by others, and its bucket size can thus serve as a lower bound. Moreover, the sum of the bucket sizes in each file can serve as an upper bound, i.e., for the case without overwriting.

Proposition 4.1 (File Statistics). *Given a number of files, $\{F_1, \ldots, F_\rho\}$, with time ranges overlapping with window W, the statistics of window W on buckets can be bounded by*

$$|\check{B}[u]| = |B^{(\rho)}[u]| \leq |B[u]| \leq |\hat{B}[u]| = \sum_{h=1}^{\rho} |B^{(h)}[u]|,$$

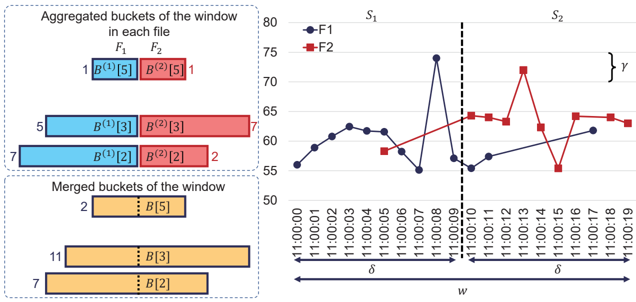

Figure 6: Buckets of a window in two files F_1 and F_2

where $|B^{(h)}[u]|$ is the size of the u-th bucket for window W in file F_h, $|\check{B}[u]|$ and $|\hat{B}[u]|$ denote the lower and upper bounds of bucket size $|B[u]|$ for window W, respectively.

When there is only one file, i.e., $\rho = 1$, we have $|\check{B}[u]| = |B[u]| = |\hat{B}[u]|$, exactly the case in Section 3.

Example 4.2. Figure 6 presents a time series stored in two files F_1 and F_2, denoted by blue and red points, respectively. For each file, as in Example 3.2, we can aggregate its bucket statistics on segments in the window, denoted by blue and red bars, respectively. For instance, we have the aggregated sizes of the 2nd buckets, $|B^{(1)}[2]| = 7$ in F_1 and $|B^{(2)}[2]| = 2$ in F_2. Unfortunately, these two bucket sizes in two files cannot be further aggregated. The reason is that points p_{10}, p_{11} at time 11:00:10, 11:00:11 in $B^{(1)}[2]$ in F_1 are overwritten by the points in F_2. Consequently, the bucket of the time series merging two files (yellow bar) has size $|B[2]| = 7 < |\hat{B}[2]| = |B^{(1)}[2]| + |B^{(2)}[2]| = 9$. In this sense, the aggregated bucket size $|B^{(1)}[2]| + |B^{(2)}[2]|$ serves as an upper bound of the merged bucket. Moreover, the file F_2 with the largest version number is written lastly and should not be overwritten by any other. Therefore, its bucket size is the lower bound of the merged bucket, e.g., having $|B[2]| = 7 > |\check{B}[2]| = |B^{(2)}[2]| = 2$.

4.2 Bounds of Neighbor Count in Multiple Files

Combining the aforesaid bounds of bucket sizes in Proposition 4.1 and the bounds of neighbor count w.r.t. bucket sizes in Section 3.3, we can derive upper bounds and lower bounds for multiple files.

4.2.1 *Pruning by Upper Bound.*

Analogous to Proposition 3.5, for $(\lambda - \frac{1}{2})\gamma < r \leq \lambda\gamma, \lambda = 1, 2, \dots$, the upper bound of neighbor count is obtained as follows, referring to the bucket statistics $|B_g^{(h)}[i]|$ in segment S_g of file F_h.

Proposition 4.3. *For a point p in the bucket $B[u]$, if $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$, then the count of its r-neighbors has an upper bound*

$$|N(p,r)| \leq \sum_{i=u-\lambda}^{u+\lambda} |B[i]| \leq \sum_{i=u-\lambda}^{u+\lambda} \sum_{h=1}^{\rho} |B^{(h)}[i]|$$

where $\lambda = \lceil r/\gamma \rceil$ and $|B^{(h)}[i]| = \sum_g |B_g^{(h)}[i]|$.

Likewise, similar to Proposition 3.6, for $(\lambda - 1)\gamma < r \leq (\lambda - \frac{1}{2})\gamma, \lambda = 1, 2, \dots$, we can further obtain a strict upper bound for multiple files.

Proposition 4.4. *For a point p in the bucket $B[u]$, if $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$, then the count of its r-neighbors has an upper bound*

$$|N(p,r)| \leq \max\left(\sum_{i=u-\lambda}^{u+\lambda-1} \sum_{h=1}^{\rho} |B^{(h)}[i]|, \sum_{i=u-\lambda+1}^{u+\lambda} \sum_{h=1}^{\rho} |B^{(h)}[i]|\right),$$

where $\lambda = \lceil r/\gamma \rceil$ and $|B^{(h)}[i]| = \sum_g |B_g^{(h)}[i]|$.

Example 4.5 (Example 4.2 continued). Consider a query with $w = 20, s = 10, k = 5, r = 5$ on the time series in Figure 6. Since $\lceil r/\gamma \rceil - r/\gamma = 0 < \frac{1}{2}$, Proposition 4.3 can be applied with $\lambda = \lceil r/\gamma \rceil = 1$. For a point p in bucket $B[5]$, its upper bound, w.r.t. files F_1 and F_2, is $|N(p,r)| \leq \sum_{i=5-1}^{5+1} \sum_{h=1}^{2} |B^{(h)}[i]| = |B^{(1)}[4]| + |B^{(2)}[4]| + |B^{(1)}[5]| + |B^{(2)}[5]| + |B^{(1)}[6]| + |B^{(2)}[6]| = 2 < k = 5$.

4.2.2 *Pruning by Lower Bound.*

Again, based on Proposition 3.7 for $r \geq \gamma$, we extend the lower bound for multiple files as follows.

Proposition 4.6. *For a point p in the bucket $B[u]$, if $r \geq \gamma$, then the count of its r-neighbors has a lower bound*

$$|N(p,r)| \geq \sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| \geq \sum_{i=u-\ell+1}^{u+\ell-1} |B^{(\rho)}[i]|,$$

where $\ell = \lfloor r/\gamma \rfloor$ and $|B^{(\rho)}[i]| = \sum_g |B_g^{(\rho)}[i]|$.

Example 4.7 (Example 4.2 continued). Consider another query with $w = 20, s = 10, k = 5, r = 10$ on the time series in Figure 6. Note that $r = 10 \geq \gamma = 5$, and thus Proposition 4.6 can be applied with $\ell = \lfloor r/\gamma \rfloor = 2$. For a point p in bucket $B[3]$, its lower bound, w.r.t. files F_1 and F_2, is $|N(p,r)| \geq \sum_{i=3-1}^{3+1} |B^{(2)}[i]| = |B^{(2)}[2]| + |B^{(2)}[3]| + |B^{(2)}[4]| = 9 > k = 5$.

4.3 Pruning by Bucket Statistics

Referring to the upper and lower bounds of neighbor count in Propositions 4.3, 4.4 and 4.6, there are three cases for points p in bucket $B[u]$ to consider in pruning.

4.3.1 *Case 1: p must be an inlier.*

If the lower bound in Proposition 4.6 is no less than the neighbor count threshold k, having $\sum_{i=u-\ell+1}^{u+\ell-1} |B^{(\rho)}[i]| \geq k$, $\ell = \lfloor r/\gamma \rfloor$, then all the points p in bucket $B[u]$ must have r-neighbor count $|N(p,r)| \geq k$, i.e., inliers. In other words, all the data points in bucket $B[u]$ are pruned and have no need to load from disk.

4.3.2 *Case 2: p must be an outlier.*

If the upper bound in Proposition 4.3 or 4.4 is less than the neighbor count threshold k, it means that the r-neighbor count has $|N(p,r)| < k$ for all the points p in bucket $B[u]$, i.e., outliers. In other words, the entire bucket $B[u]$ can be directly output as outliers without further checking. For a query of outlier count, the data points in bucket $B[u]$ have no need to load from disk.

4.3.3 *Case 3: p is a suspicious point.*

Otherwise, we are not able to directly determine the category of point p by the bucket statistics only, and thus need to further check all related buckets, i.e., $2\lambda + 1$ from $B[u - \lambda]$ to $B[u + \lambda]$, where $\lambda = \lceil r/\gamma \rceil$. Nevertheless, for the case where there is only one file involved in the current processing window, we only need to load the data points in up to 4 additional buckets to determine the r-neighbors of the points p in bucket $B[u]$, i.e., $B[u-\lambda], B[u-\ell], B[u+\ell], B[u+\lambda]$, where $\ell = \lfloor r/\gamma \rfloor$. For instance, for the example in Figure 5(8), all the points in buckets $B[u-1]$ and $B[u+1]$ must be the r-neighbors of any point $p \in B[u]$. That is, we only need to check r-neighbors of p in two additional buckets $B[u-2]$ and $B[u+2]$. For other examples in Figures 5 (5)-(7), the points in $B[u-2]$, $B[u-1]$, $B[u+1]$ and $B[u+2]$ may or may not be the r-neighbors of p, and thus need further checking. The r-neighbor count for this special case is computed as follows.

Proposition 4.8. *If there is only one file involved in the current processing window, for a point p in bucket $B[u]$, the count of its r-neighbors can be computed by loading data points in at most 4 additional buckets, having*

$$|N(p,r)| = \sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| + \left| \left\{ p' \in \bigcup_{j \in \{u \pm \lambda, u \pm \ell\}} B[j] \;\middle|\; dist(p,p') \le r \right\} \right|,$$

where $\ell = \lfloor r/\gamma \rfloor \ge 1$ and $\lambda = \lceil r/\gamma \rceil$.

When $\ell = \lambda$, e.g., the examples in Figures 5(4) and 5(8), the count of the r-neighbors of p can be computed by loading data points in only 2 additional buckets, $B[u-\lambda] = B[u-\ell]$ and $B[u+\ell] = B[u+\lambda]$, referring to Proposition 4.8. Indeed, for $\ell = \lfloor r/\gamma \rfloor < 1$, i.e., the examples in Figure 5(1)-(3), it also needs only to load 2 additional buckets, $B[u-1]$ and $B[u+1]$.

Example 4.9 (Example 4.7 continued). Consider again the query with $w = 20, s = 10, k = 5, r = 10$ on the time series in Figure 6. Referring to the analysis in Example 4.7, for a point p in bucket $B[3]$, the lower bound of $|N(p,r)|$ is 9, according to Proposition 4.6. Moreover, the upper bound of $|N(p,r)|$ is $\sum_{i=3-2}^{3+2} \sum_{h=1}^{2} |B^{(h)}[i]| = 23$, since $r = 2\gamma = 10$ and Proposition 4.3 applies. Given $|N(p,r)| \ge 9 > 5 = k$, point p in $B[3]$ must be an inlier, i.e., the aforesaid Case 1. However, for another query with neighbor count threshold $k = 10$, the points in bucket $B[3]$ cannot be pruned by the aforesaid lower and upper bounds. To check such suspicious points, we need to load additional buckets from $B[3-2]$ to $B[3+2]$, i.e., $B[1]$ to $B[5]$, from files F_1 and F_2 on disk.

For the special case where only one file is involved in the current processing window, e.g., Figure 4, it is easy to check suspicious points by just loading 4 or 2 additional buckets, according to Proposition 4.8. For instance, back to the single file case in Figure 4. Consider a query with $w = 20, s = 10, k = 15, r = 10$ on the time series in Figure 4. Point p in bucket $B[4]$ has bounds $10 \le |N(p,r)| \le 20$, referring to Propositions 3.5-3.7, and cannot be pruned by Case 1 or Case 2. Fortunately, given $\ell = \lambda = 2$, we only need to load two additional buckets, i.e., $B[2]$ and $B[6]$, to determine the r-neighbors of $p \in B[4]$. It follows $|N(p,r)| = 1 + |B[3]| + |B[4]| + |B[5]| + 0 = 11 < k$, where only 1 neighbor is found in $B[2]$. That is, p is an outlier.

4.4 Multi-File Query Algorithm

Algorithm 1 presents the query processing on multiple files with possibly overlapping time ranges. We update the bucket statistics $|B^{(h)}[u]|$ for the expired and new segments in the sliding window. The lower and upper bounds of bucket statistics are obtained in Lines 11 and 12 referring to Proposition 4.1. Based on the bounds of neighbor count in Section 4.2, inliers and outliers are directly pruned and obtained. Otherwise, for those points cannot be pruned, unfortunately, we need to load $2\lambda + 1$ buckets in the window to determine their neighbors, and thus determine whether outlier or not. For the query processing in single file, we only need to load additional 4 buckets in Line 20 referring to Proposition 4.8, instead of $2\lambda + 1$ buckets.

Algorithm 1 Outlier Query on Multiple Files

Input: A window W_t at time t with size w and slide s, neighbor distance threshold r and count threshold k
Output: outlier set O_t of current window W_t

```
1:  λ := ⌈r/γ⌉
2:  ℓ := ⌊r/γ⌋
3:  initialize outlier set O_t := ∅
4:  initialize buckets 𝓑 if algorithm runs for the first window
5:  for each bucket B[u], u := 1 to β do
6:      for each file F_h do
7:          for each expired segment S_g do
8:              |B^(h)[u]| := |B^(h)[u]| − |B_g^(h)[u]|
9:          for each new segment S_g do
10:             |B^(h)[u]| := |B^(h)[u]| + |B_g^(h)[u]|
11:     |B̌[u]| := |B^(ρ)[u]|
12:     |B̂[u]| := Σ_{h=1}^{ρ} |B^(h)[u]|
13:     if Σ_{i=u−ℓ+1}^{u+ℓ−1} |B̌[i]| ≥ k then
14:         continue
15:     else if ⌈r/γ⌉ − r/γ < 1/2 and Σ_{i=u−λ}^{u+λ} |B̂[i]| < k then
16:         O_t := O_t ∪ B[u]
17:     else if ⌈r/γ⌉ − r/γ ≥ 1/2 and
        max(Σ_{i=u−λ}^{u+λ−1} |B̂[i]|, Σ_{i=u−λ+1}^{u+λ} |B̂[i]|) < k then
18:         O_t := O_t ∪ B[u]
19:     else
20:         load buckets from B[u − λ] to B[u + λ]
21:         point-wise checking the points in the buckets
22: return O_t
```

Complexity Analysis. Consider involved ρ files, β buckets and ω segments in a window. The update of bucket statistics takes $O(\beta\omega\rho)$ time. In the worst case, all the n points in the window may be output as outliers, in $O(n)$ time. Otherwise, we need to load a constant number i.e., $2\lambda + 1$, of additional buckets in Line 20 among ρ files, which takes $O(\rho n)$ time. Referring to the average number of points in a bucket $\frac{n}{\beta}$, it takes $O(\frac{n^2}{\beta})$ time.

Algorithm 1 runs in $O(\beta\omega\rho + \rho n + \frac{n^2}{\beta})$ time, and needs $O(\beta\zeta\rho)$ extra space, where ρ is the number of files, β is the number of buckets, ω is the number of segments in a window, n is the number of points in a window, and ζ is the number of segments in a file.

5 EXPERIMENTS

We conduct extensive experiments to demonstrate the higher efficiency of our LSMOD method in processing outlier queries under various settings.

Table 2: Dataset and query settings

Name	# data points	bucket width γ	r	w	# windows
TAO-OceanGraphic	600k	0.5	2.0	10,000	1,180
UCI-Gas	900k	0.01	0.05	100,000	160
UCI-PAMAP2	300k	0.1	0.5	1,200	4,980
WH-Chemistry	100m	2	10	1,200	1,666,647
TY-Vehicle	100m	2	10	1,200	1,666,647
GW-WindTurbine	100m	5	20	1,200	1,666,647

Table 3: The corresponding relationships between out-of-order rates and out-of-window rates in various datasets

out-of-order rate	out-of-window rate					
	TAO	Gas	PAMAP2	WH	TY	GW
0.02	0.009	0.010	0.008	0.009	0.009	0.009
0.04	0.019	0.021	0.018	0.019	0.019	0.019
0.06	0.029	0.031	0.028	0.029	0.029	0.030
0.08	0.039	0.042	0.038	0.039	0.039	0.041
0.10	0.049	0.053	0.048	0.049	0.049	0.051

5.1 Experimental Settings

Table 2 lists the real-world datasets employed in the experiments, including the default window size w and number, i.e., how many windows are tested in the experiments. For example, if the window size is 20 min, the 20-minute window contains 1,200 points (1 point per second). With slide size equal to 1 minute (60 data points), the dataset having 100m points contains (100,000,000 - 1,200) / 60 = 1,666,647 windows. In the experimental evaluation of this paper, we set a large w/s = 20 by default, as in NETS [23] and CPOD [20].

All the baselines are implemented in their own streaming settings. All the experiments run on a machine with Intel Core i7 (2.80GHz), 16 GB of memory.

5.2 Out-of-order Evaluation

In general, we do not assume a known bound on the delay of points. Thereby, like other DBMS, the query answers are based on the data that have already been persisted in the database. In this sense, a new run of the query may lead to a different result, when there are new data received within the query range. Nonetheless, we conduct an experiment to evaluate the correctness of query answers based on the currently persisted data.

5.2.1 Varying Out-of-order Workloads. In the first experiment of Figures 7 and 8, we consider the case that, upon the execution of a query, all relevant points are already persisted. That is, this special experiment assumes the underlying data being periodic and without duplicate points of the same timestamp. As shown in Figure 7, when all the periodic points are persisted without further updates, the database query by our LSMOD can always return the correct answer having F1 score = 1.

It is notable that the existing streaming outlier detection methods, CPOD, NETS and MCOD, cannot always return the correct answers, even knowing that the underlying data being periodic and without duplicate points of the same timestamp. The reason

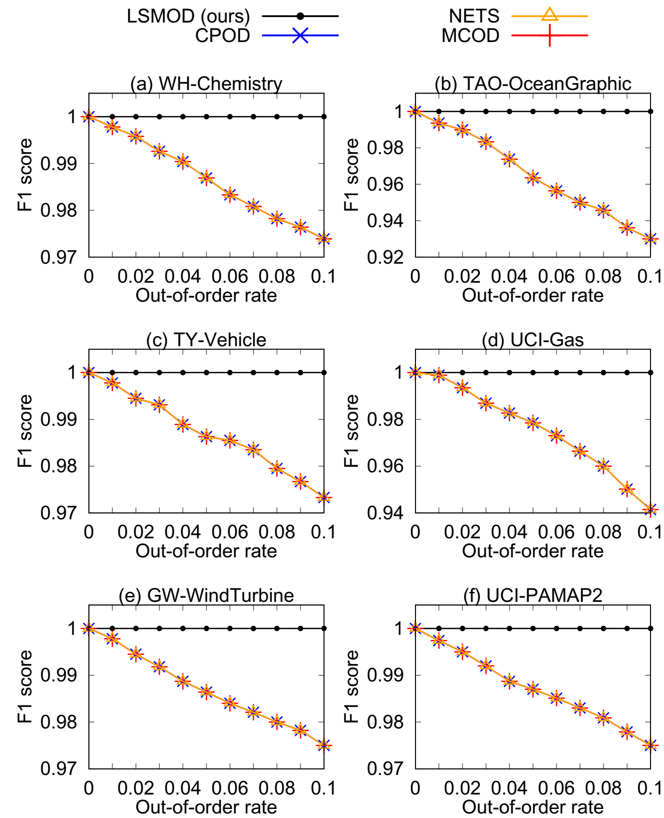

Figure 7: F1 scores under different out-of-order rates

is that the arrivals of data points could be out-of-order. These existing outlier detection approaches cannot handle the delayed data points that are out of the currently processing window, leading to incorrect answers. In contrast, our solution LSMOD processes the query over all the out-of-order points persisted in the database, rather than only those cached in the window of data streams.

Table 3 indicates the corresponding out-of-window rates, i.e., how much data is out of the windows that should contain the data. As illustrated in Table 3, with the increase of out-of-order data, there are more out-of-window data as well. As aforesaid, since out-of-window data are ignored by the existing streaming outlier detection methods, CPOD, NETS and MCOD, they cannot always return the correct answers with worse F1 score in Figure 7.

The SOTA approaches for in-order streaming data such as NETS and CPOD can indeed process out-of-order data, as long as late data is still within the current windows. Rather than sorting all timestamps in windows, the compared approaches directly process the out-of-order data stream in the experimental evaluation. As illustrated in Figure 8, even without the overhead of sorting the streaming data, the online streaming computation by NETS and CPOD is still much slower than our LSMOD, thanks to the pre-computation and pruning in the database.

5.2.2 Varying Updating Workloads. Similar to other DBMS, if points might also be updated after being received, it is challenging to know whether the result of a query is "final" even if all data has

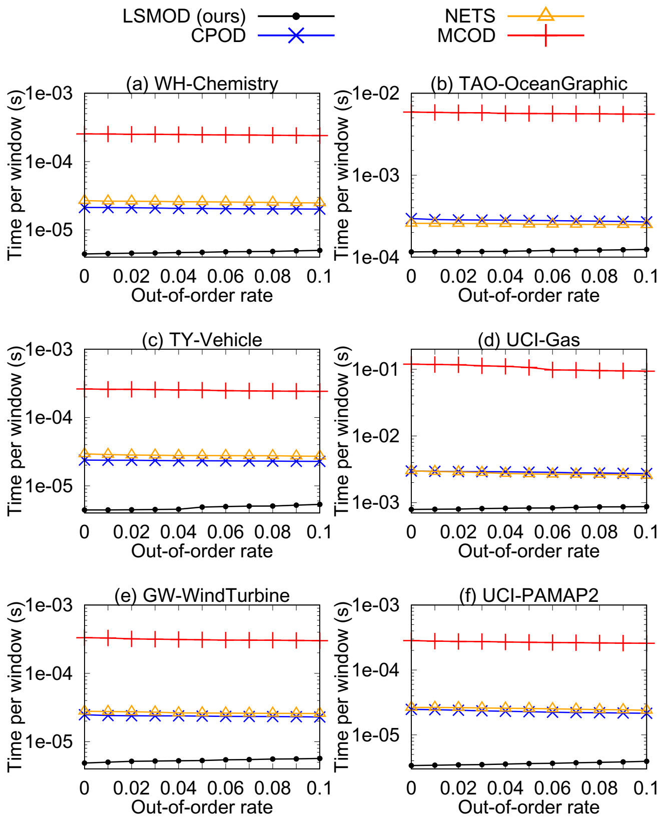

Figure 8: Time costs under different out-of-order rates

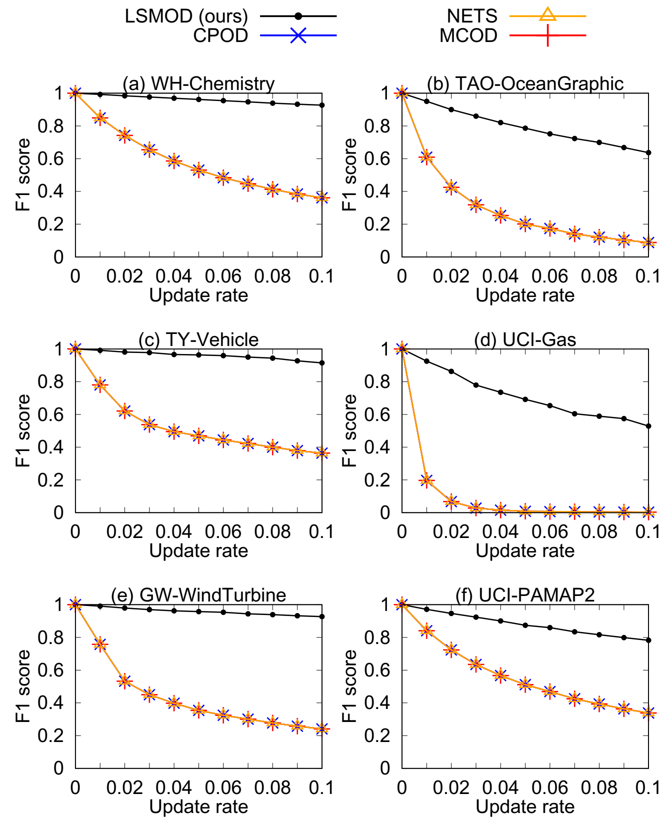

Figure 9: F1 scores under different update rates

been delivered. Thereby, rather than considering out-of-order arrivals without updates, Figures 9 and 10 evaluate the query answers over the data that may be updated after the query.

As illustrated in Figure 9, the more the data will be updated after the query, the lower the F1 score of the query answer would be. Again, the correctness of the existing streaming outlier detection methods, CPOD, NETS and MCOD, is worse than our database solution LSMOD. The reason is still that the updates may be delivered out of the currently processing window, and thus lead to wrong answers. In contrast, the database query LSMOD considers all the delivered data points and their updates thus far.

It is true that there is a significant overhead during the merge process, where late data points could overwrite previous ones. As illustrated in Figure 10, the larger the update rate is, the more our LSMOD costs to merge them. The existing streaming outlier detection methods, CPOD, NETS and MCOD, are less affected, since they simply neglect the updates that are out of the current windows (leading to lower F1 score in Figure 9).

5.2.3 *Discussion on Correctness.*

When there is no bound for the delay assumed, a delayed data point may come after the issue of a query. In this case, the correctness of the query is not guaranteed, i.e., the query result could be different when the delayed data point finally arrives. On the other hand, supposing all delayed data points arrive within a limited bound, if we execute the query after the maximum delay of the last point, the result must be correct. It is because all the relevant data points are persisted.

5.3 Parameter Evaluation

5.3.1 *Varying Slide Size s Relative to Window Size w.*

As in the CPOD paper [20], we conduct a direct repetition of one of the previous experiments for the comparison, in Figure 11, reporting the average processing time for each sliding window, with various slide sizes (relative to window sizes).

Indeed, although the hardware is different, the performance of CPOD is not profoundly different. Similar to [20], with the ratio of slide sizes to window sizes increasing, the method takes more time, because there are more points to process in each new slide. It also has a better understanding of how the two systems would compare. That is, no matter where the outlier detection query is processed, in the streaming system CPOD or the database system with our LSMOD, a larger slide size relative to the window size leads to higher processing time. Nevertheless, the improvement by our proposal is consistently observed under various slide sizes.

This of course could extend to other baselines/datasets that could be matched to experiments in previous work. Note that the datasets (b) TAO-OceanGraphic and (d) UCI-Gas are also used in CPOD [20], NETS [23] and MCOD [19]. As shown in Figure 11, similar results of these existing methods and improvements by our proposal are observed in both datasets.

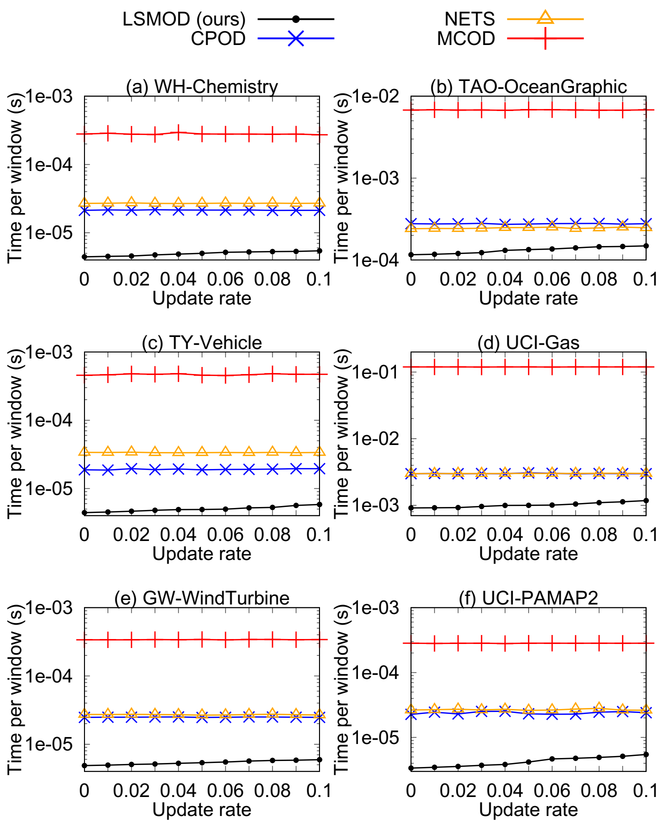

Figure 10: Time costs under different update rates

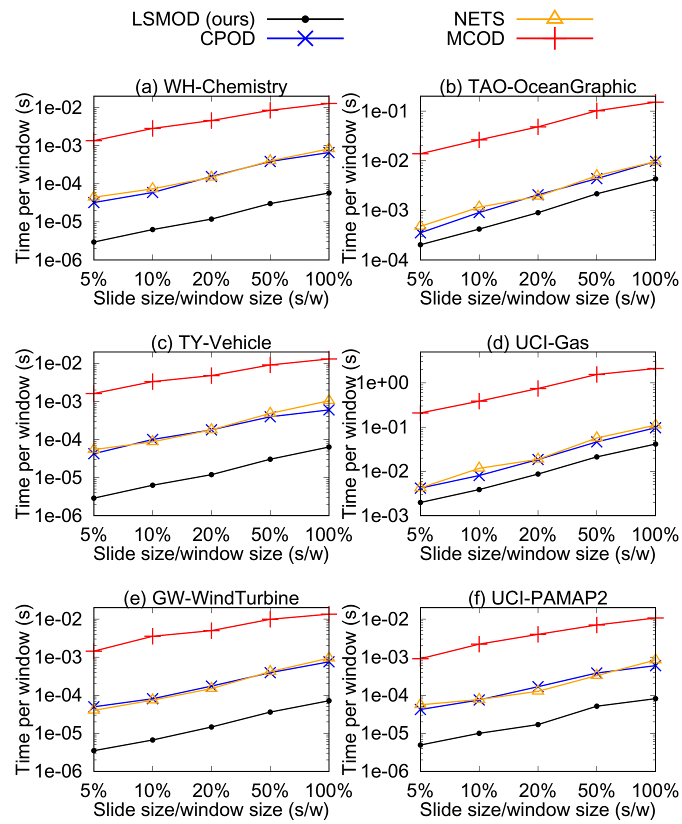

Figure 11: Varying slide size relative to window size

It is true that for a small s/w (i.e., large w/s), SOTA approaches having specific designs for incremental computations perform well. Nevertheless, our LSMOD benefits from such small increments as well, and consistently show the improvement.

5.3.2 Varying File Numbers. Figure 12 evaluates the time costs under different file numbers. The number of files is chosen by the corresponding page size setting of the database, i.e., the maximum number of points in a file. By default, the system sets the page size as 50,000 points. Thus, for a dataset with 1,000,000 data points, the data are read from 1,000,000/50,000 = 20 files. By setting a smaller page size, the corresponding file number will be larger. It is not surprising that LSMOD costs a bit more time to handle more files.

It is true that parameters that can "tilt" the results in one direction or another. As shown in Figure 12, a single file with almost all data in order would perform better, since LSMOD has tighter bounds for pruning in this case as presented in Section 3.

5.3.3 Varying File Disorder Degree. Figure 13 evaluates the time costs under different disorder degree. A disorder degree, e.g., 0.02, is chosen by randomly making 2% points in a file delayed and batched with other files having higher versions. With the increase of disorder degree, LSMOD costs a bit more time to handle the out-of-file disorder issues, whereas the streaming outlier detection methods neglect them if not in the current window (with correctness issue).

Moreover, Figure 13 shows that the lower the disorder degree is, the better the proposed LSMOD performs. It is again owing to the stronger pruning power. When there is no out-of-file disorder issue, the corresponding file statistics could be directly utilized.

5.3.4 Varying Consecutive Missing Point Numbers. Note that the default value used for missing points may result in the detection of an outlier or not. Thereby, Figure 14 evaluates the detection correctness by varying missing points. As shown, when the consecutive missing points are less than $k = 50$, the outlier detection F1 score is low. The reason is that the default values of missing points are insufficient to form inliers, and thus mistakenly detected as outliers. On the other hand, when the consecutive missing points exceed $k = 50$, the default values become inliers. They would not affect largely the detection of other outliers, having higher F1 score.

5.3.5 Varying Width γ on Value Range of Buckets. To illustrate how to choose bucket size, we conduct an experiment to evaluate the difference of using various bucket sizes. It considers a persistent query with the neighbor distance threshold r. We will not choose $\gamma > r$, since lower bounds do not apply in this case according to the analysis in Section 3.3.2. Thereby, Figure 15 varies the bucket size γ from $0.01r$ to r. As shown, with the increase of bucket size γ, LSMOD takes significantly less time to execute, since there are less buckets to traverse. In this sense, it is the best to choose a bucket size γ equal to the neighbor distance threshold r, if possible.

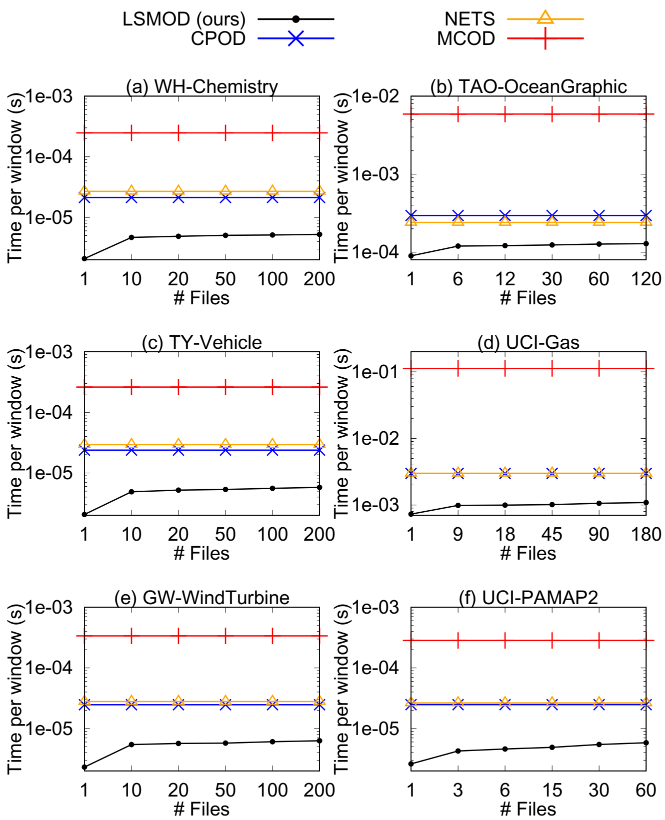

Figure 12: Varying file number (baselines not affected)

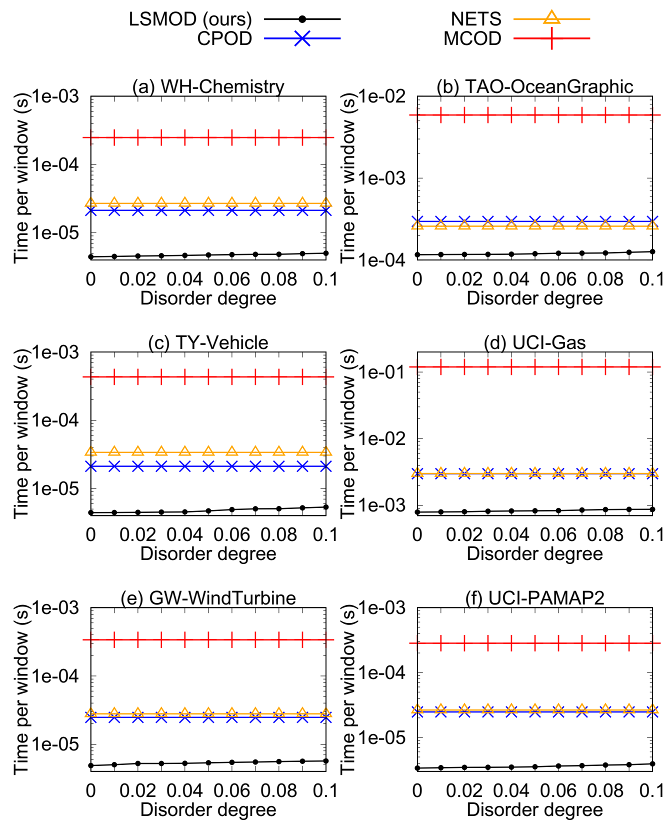

Figure 13: Varying file disorder degree

6 RELATED WORK

Modern SPEs like Flink also provide dedicated mechanisms for late arrivals [6]. For stateful analysis such as aggregates, the tuples are immediately added to the according windows in Flink. To produce the results of each window, Flink utilizes watermarks to trigger the window operators, when all tuples in the windows have been received. Moreover, Flink allows to specify a maximum delay time, called an allowed lateness. It explicitly specifies how long each window should wait for late arrivals before being finally evicted. Analogously, in our scenario, if the query is issued after such a known limit of delay, the correctness of query answer is also guaranteed.

TinTiN also discusses dedicated approaches to handle late arrivals [21]. A determinism scheme is proposed to ensure timely output based on the timely available input. As a middleware for streaming system, it can top-up the guarantees of aggregation applications with out-of-order arrivals, if the application conforms to the strict determinism.

The Borealis paper [12] in CIDR 2005 was the first to point out the possibility of using revision tuples. The authors envision dynamic revision of query results, with the consideration of revision records. It processes and generates "delta" showing only the effects of revisions, rather than regenerating the entire result.

The CIDR 2007 paper [13] was the first to work out how to do this systematically for relational algebra. CEDR formally defines a spectrum of consistency levels to deal with latency or out-of-order delivery. Its implementation is based on a set of relational algebra operators, most of which are based on view update semantics.

Microsoft StreamInsight's query processor [7] also worked this way. It utilizes the "current time increment" events to signal the engine that no more events earlier than what have already been received will arrive. Current time increment events effectively cue the engine to process the events that have arrived and subsequently adjust any with timestamps earlier than the current time.

Apache Beam [8] by Google built systems later which could correct prematurely issued incorrect answers. It tries to determine when all data have arrived (based on data source types) and then advances the watermark past the end of the window. One can use triggers to decide when each individual window aggregates and reports its result, including how the window emits late elements [9].

Similarly, Spark Streaming [10] also adopts the idea of watermarking to deal with out-of-order arrivals. It allows users to specify the threshold of late data, and lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly [11].

Trill [14] introduces a method to handle and preaggregate late arriving data. It maintains several ordered runs for a stream, implemented as blocks of MemTable in Apache IoTDB [25], which is similar to a run/file in LSM-Trees. While the idea is alike, to preaggregate chunks of data and combine with late arriving data in the impatience framework part, our studied problem is more difficult

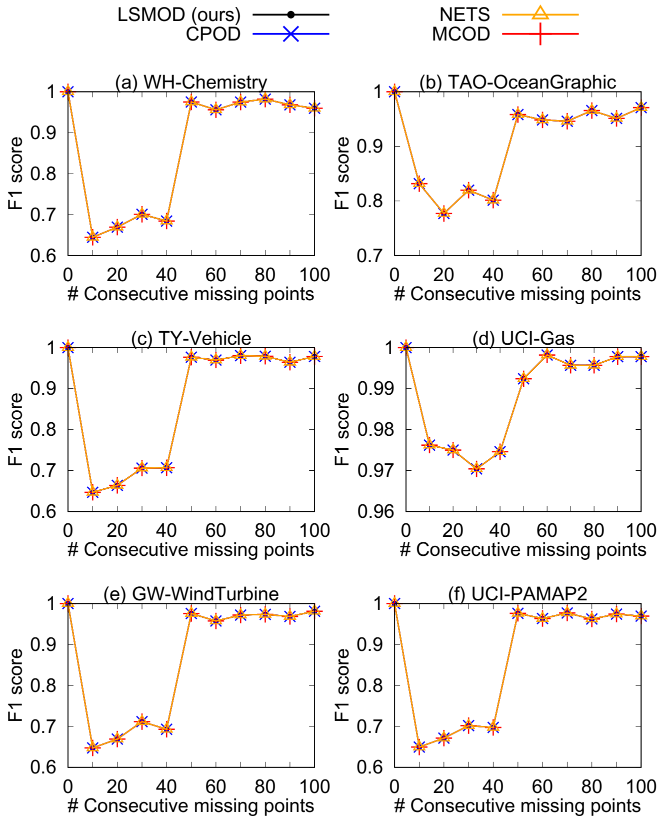

Figure 14: F1 score under various consecutive missing points

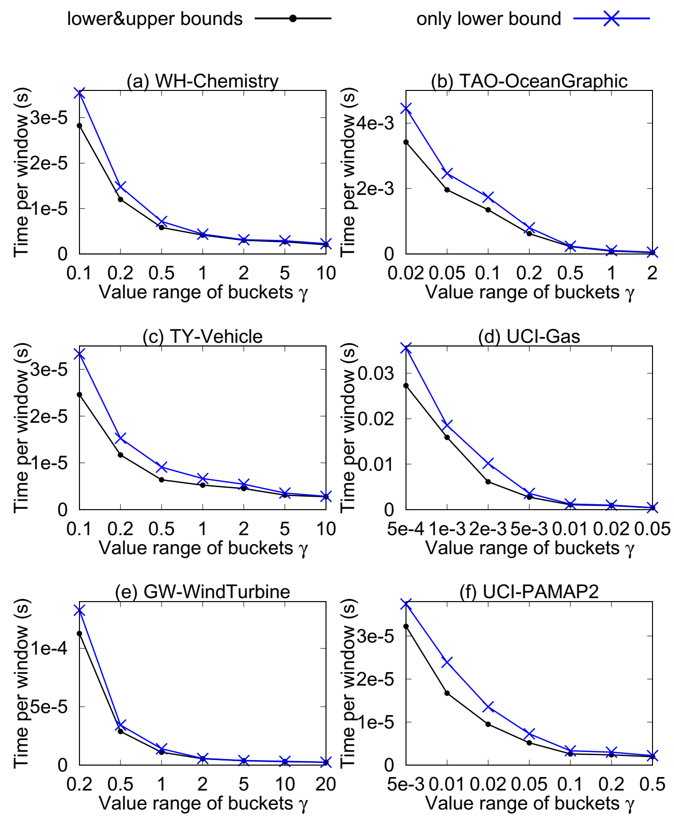

Figure 15: Varying the width γ on value range of buckets

in two folds. (1) The outlier detection problem is more complicated to aggregate, as illustrated in Section 3. (2) Loading all runs/files from disk into memory incurs significant extra overheads.

The VLDB 2008 work [17] pushes incomplete data through query plans to partially compute results, and revises as late arriving data invalidates earlier results, similar to Microsoft StreamInsight. Both of these systems hold back the final answer until punctuation to avoid exposing listeners to revisions. Again, while they consider aggregation and join operations for out-of-order processing systems, we study the outlier detection problem with more complicated aggregation in an LSM-Tree based database system.

Note that none of the work described above specifically works out for outlier detection. (1) For the partially processed results for outlier detection, since the outliers in different segments cannot be directly aggregated, we design the bucket statistics in each segment for outlier pruning, in Section 3.1. (2) Moreover, to combine things to produce the final answers, we show that unlike outliers, bucket statistics of different segments among different files on disk can be combined, and thus enable the outlier pruning, in Section 4.1. Indeed, in addition to database systems, it is interesting to extend our solution to the streaming systems in the future work.

7 CONCLUSION

In this paper, we present an efficient method, LSMOD, for querying outliers in Apache IoTDB, an LSM-tree based time series database. Owing to the out-of-order arrivals, a time series may be stored in multiple files written at different time. Even worse, some points formerly recorded may be overwritten by later arrivals, e.g., for data correction. Unfortunately, such storage strategies prevent detecting outliers separately in each file and merging them as the results. In this sense, we optimize the efficiency of distance-based outlier query in Apache IoTDB, by addressing the issue of overlapping files for delayed data. Based on bucket statistics in files, we derive the upper and lower bounds of neighbor count to prune outliers and inliers, respectively. Moreover, we illustrate that the neighbor count may also be determined by loading points in a constant number of additional buckets, e.g., at most 4 buckets for single file in Proposition 4.8. Extensive experiments demonstrate the high efficiency of the proposed LSMOD compared to the existing streaming outlier detection methods. It has now been deployed as a function in Apache IoTDB.

ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation of China (92267203, 62072265, 62021002, 62232005), National Key Research and Development Plan (2021YFB3300500), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] 2024. Apache IoTDB. https://iotdb.apache.org/.
[2] 2024. Documentation. https://iotdb.apache.org/UserGuide/V1.2.x/Reference/UDF-Libraries.html#outlier.
[3] 2024. Code. https://github.com/apache/iotdb/tree/research/outlier.
[4] 2024. Experimental code. https://github.com/iotdb-lsmod/iotdb-lsmod.
[5] 2024. Supplementary. https://iotdb-lsmod.github.io/iotdb-lsmod/supplementary.pdf.
[6] 2024. Apache Flink. https://flink.apache.org/.
[7] 2024. Microsoft StreamInsight. https://download.microsoft.com/documents/uk/bieb/Microsoft_CEP_Overview.pdf.
[8] 2024. Apache Beam. https://beam.apache.org/.
[9] 2024. Apache Beam Docmentation. https://beam.apache.org/documentation/basics/.
[10] 2024. Apache Spark. https://spark.apache.org/streaming/.
[11] 2024. Spark streaming Documentation. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.
[12] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*. www.cidrdb.org, 277–289.
[13] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*. www.cidrdb.org, 363–374.
[14] Badrish Chandramouli, Jonathan Goldstein, and Yinan Li. 2018. Impatience Is a Virtue: Revisiting Disorder in High-Performance Log Analytics. In *ICDE*. IEEE Computer Society, 677–688.
[15] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree. In *ICDE*. IEEE, 3340–3352.
[16] Maria Kontaki, Anastasios Gounaris, Apostolos N. Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. 2011. Continuous monitoring of distance-based outliers over data streams. In *ICDE*. IEEE Computer Society, 135–146.
[17] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288.
[18] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.
[19] Luan V. Tran, Liyue Fan, and Cyrus Shahabi. 2016. Distance-based Outlier Detection in Data Streams. *Proc. VLDB Endow.* 9, 12 (2016), 1089–1100.
[20] Luan V. Tran, Minyoung Mun, and Cyrus Shahabi. 2020. Real-Time Distance-Based Outlier Detection in Data Streams. *Proc. VLDB Endow.* 14, 2 (2020), 141–153.
[21] Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. 2020. TinTiN: Travelling in time (if necessary) to deal with out-of-order data in streaming aggregation. In *DEBS*. ACM, 141–152.
[22] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27.
[23] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. 2019. NETS: Extremely Fast Outlier Detection from a Data Stream via Set-Based Processing. *Proc. VLDB Endow.* 12, 11 (2019), 1303–1315.
[24] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S. Yu. 2017. Time Series Data Cleaning: From Anomaly Detection to Anomaly Repairing. *Proc. VLDB Endow.* 10, 10 (2017), 1046–1057.
[25] Xiaojian Zhang, Hongyin Zhang, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2023. Backward-Sort for Time Series in Apache IoTDB. In *ICDE*. IEEE, 3196–3208.