Outlier Query Processing in LSM-Tree based Time Series Database

Yunxiang Su, Shaoxu Song, Xiangdong Huang, Chen Wang, Jianmin Wang
Tsinghua University
suyx21@mails.tsinghua.edu.cn, {sxsong, huangxdong, wang_chen, jimwang}@tsinghua.edu.cn

***Abstract*—Querying outliers is highly demanded in time series analysis, e.g., filtering spikes of stock prices, detecting deviations in GPS trajectories, identifying change points of temperature when the cold air rushes in, etc. While outlier detection has been widely studied over streaming data, the query of outliers in time series databases was largely overlooked. The columnar storage of time series with data compression and LSM-tree based storage for intensive writings unfortunately encumber the outlier query performing. In particular, data points could be stored in multiple files, with overlapping time ranges, owing to the delayed data arrivals. Hence, it is not able to simply detect outliers in each file and merge them as the results. In this paper, we propose to utilize the bucket statistics of the values stored in files. The upper and lower bounds for the neighbor counts of data points are derived in buckets and potentially overlapping files for efficient pruning. Extensive experiments demonstrate the efficiency of our proposal in the LSM-tree based time series database, compared to the existing methods for data stream. Remarkably, the proposed outlier query has been included as a function of Apache IoTDB, an open-source time series database.**

I. Introduction

Outliers are frequently queried in time series, e.g., filtering spikes of stock prices [1], detecting deviations in GPS trajectories [2], identifying change points of temperature when cold air rushes in [3]. While some outlier itself is an event of interest, like credit card fraud detection [4], others may be introduced by noises or errors and often eliminated before data science tasks, such as time series classification [5] and forecasting [6].

Hence, outliers are frequently queried with different parameters for various interests. We follow the query convention of distance-based outliers [7]. (1) The neighbor distance threshold r specifies how distant the values are of interest. For instance, over the same GPS data, destination prediction concerns outliers in miles, while trajectory analysis is often in feet. (2) The neighbor count threshold k considers whether points have sufficient neighbors. For example, point-of-interest analysis is more sensitive to outliers with less neighbors than trajectory clustering. (3) The window size w specifies the time range of points under consideration in outlier detection. 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.

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

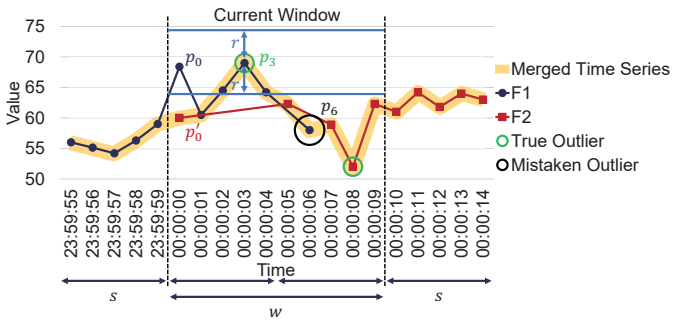

Fig. 1. Querying outliers of a time series stored in two LSM files, F1 & F2, with neighbor distance threshold $r = 5$, neighbor count threshold $k = 4$, window size $w = 10$, slide size $s = 5$

A. Challenge

It is worth noting that data science tasks often need to be iteratively conducted over the time series, stored in databases. Unfortunately, while detecting outliers over streaming data has been widely studied [7], [8], [9], the query of outliers in time series databases was largely overlooked.

Columnar storage is adopted in most time series databases, e.g., Apache IoTDB [10] and InfluxDB [11], for various advantages like efficient data compression [12]. To handle the out-of-order arrivals in time and value columns, the commodity time series database systems employ Log-Structured Merge-Tree (LSM-tree) [13]. 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.

Such separation of delayed data is efficient in handling intensive writes, a must in IoT scenarios, but unfortunately encumbers the query processing [14], including outlier queries. We cannot simply detect outliers in each file and merge the results. However, such a scenario of time-overlapping files is not considered in the existing methods [7], [8], [9] for data streams.

Example 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., there are less than $k = 4$ points (including itself) whose value distance is no greater than $r = 5$ in the current window. They should be detected as outliers referring to distance-based outlier detection [7].

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 in 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. However, the actual point p'_0 at 00:00:00 is received later and stored in F_2. Thereby, the point p'_0 at 00:00:00 in F_2 overwrites p_0 in F_1, which should also be considered when 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.

B. Motivation

To avoid LSM-tree with overlapping time ranges, an alternative implementation is to insert the out-of-order arrivals directly onto the disk. It obviously incurs huge cost of sorting and moving all the subsequent disk-resident data points, and thus it is unacceptable especially in IoT scenarios with extensive writes by millions of sensors [15]. For this reason, most commodity time series databases such as Apache IoTDB [10] and InfluxDB [11] employ LSM-tree to handle the heavy write workloads.

Note that compaction takes care of the overlapping time range problem. Each compaction operation reads the files with overlapping time ranges, merges them as one file, and writes it back to disk. Owing to the overwhelming cost of compaction, it is often triggered periodically, e.g., once each day or week.

The implication of our solution is how to efficiently process outlier queries over the files not compacted yet with overlapping time ranges, as well as the compacted single files. For some applications like anomaly monitoring, it often queries the outliers in the past hours, where the data are not compacted yet and distributed in files with overlapping time ranges. For other analytical applications such as fault diagnosis, the query processes the historical data that have been compacted in single files. Thereby, it is also prevalent and important to process the single file case in the LSM-tree storage.

The simple work-arounds are to costly load all the data points, order them by their timestamps, and apply the existing methods such as CPOD [9] and NETS [16] that process data points in time order. These baselines are obviously inefficient as evaluated in the experiments in Section VII.

C. Contribution

In this paper, we propose LSMOD to efficiently query outliers of time series stored in LSM-tree based time series databases. Different from the existing streaming methods, we focus on efficiently querying from LSM-tree files with overlapping time ranges. Our main contributions are as follows.

(1) We formalize the outlier detection problem in LSM-tree based time series databases in Section III, including the storage structure, and the corresponding SQL statement in databases.

(2) We devise the query processing on single file in Section IV. We utilize pre-computed bucket statistics, which can be further aggregated in LSM-tree based store, to accelerate outlier detection. Propositions 3-6 establish the upper and lower bounds of neighbor count with respect to bucket statistics, to facilitate efficient pruning.

(3) We design the query processing over multiple files in Section V. Given multiple files with overlapping time ranges, though the bucket statistics cannot be directly merged, we propose to bound the bucket statistics in Proposition 7. It leads to the upper and lower bounds on multiple files in Propositions 8-10. These bounds enable efficient pruning for multiple files.

(4) We extend our proposal to density-based and multivariate outlier detection in Section VI. Proposition 11 enables pruning in density-based adaption, and Proposition 12 ensures that our method always returns a subset of the outliers defined on multiple dimensions.

(5) We conduct extensive experimental evaluation in Section VII. The results demonstrate that our proposal with bucket statistics pruning is more efficient than the existing methods for data streams. The evaluation on the extensions also illustrates the effectiveness and efficiency of our proposal.

It is worth mentioning that the outlier query has been deployed as a function in Apache IoTDB [10], an open-source time series database. The document is available in its official website [17]. The code is included in the GitHub repository of IoTDB [18]. The experiment code and data are available in [19] for reproducibility. All proofs can be found in [20].

TABLE I
RELATED WORKS FOR DISTANCE-BASED OUTLIER DETECTION

Method	Time Complexity	Neighbor Search	Input	SOTA
Exact-Storm [21]	$\mathcal{O}(W \log k)$	Index per window	Streaming data	✘
DUE [8]	$\mathcal{O}(W \log W)$	Index per window	Streaming data	✘
Thresh_LEAP [22]	$\mathcal{O}(W^2/S)$	Index per slide	Streaming data	✘
MCOD [8]	$\mathcal{O}((1-c)W \log((1-c)W) + kW \log k)$	Micro-clusters	Streaming data	✘
NETS [16]	$\mathcal{O}(\theta_S + N_{cell}\theta_W)$	Grid index per slide	Streaming data	✔
CPOD [9]	$O(N_{core}S) + N_{candidate}N_{dis\_comp}$	Core points	Streaming data	✔
LSMOD (ours)	$\mathcal{O}(\beta\omega\rho + \rho n + \frac{n^2}{\beta})$	Bucket index per segment	Data stored in time series databases	✔

II. Related Work

A. Outlier Detection

Outliers may indicate abnormal behaviors or events, e.g., credit card fraud or intrusion. There are many outlier detection methods for point outliers [8], [21] and sequence outliers [23], [24], [25]. Among them, distance-based outlier detection is widely used in indicating abnormal behaviors [4], owing to its intuitive definition and high efficiency in implementation. Existing approaches employ various index structures for range query search, such as Exact-Storm [21], DUE [8] and Thresh_LEAP [22]. Besides, micro-cluster is also employed in MCOD [8] to store the neighbor information, instead of range query search, which outperforms the aforesaid methods as shown in [7]. Similar to LSMOD, grid structure is employed in NETS [16], while it cannot handle queries with different parameters efficiently. Moreover, CPOD [9] employs core points to speed up searching neighbors, which is known as the start-of-the-art work of distance-based outlier detection in data streams. These streaming algorithms can be implemented by the series reader in the database, where time series can be loaded from overlapping files and read as a data stream. Pre-computed statistics in databases however are not utilized in these streaming algorithms. Table I compares these streaming methods with our proposal, in terms of time complexity, the structure for neighbor search, input data type and so on.

The key differences between streaming data and time series databases are as follows. (1) The out-of-order arrivals with timestamps earlier than the currently processing window cannot be handled by the streaming methods. With the help of LSM-tree, all the out-of-order data points are stored in time series databases, and thus processed in outlier query. (2) The time series database can utilize some pre-computed statistics for efficient query processing, which are not considered in the streaming methods with online processing.

B. LSM-tree Storage

Time series database systems often need to ingest intensive writes with out-of-order arrivals, especially in the IoT scenarios [14]. Log-Structured-Merged (LSM) Tree [13] naturally handles such workloads, by batching data points in different files referring to their arrival order. These files are then compacted into one file later for more efficient storage and better query processing. To optimize the query processing in LSM-tree based stores, ElasticBF [26] adopts small filters and dynamically loads into memory as needed based on access frequency, instead of adopting a uniform setting for all Bloom filters. Likewise, Rosetta [27] proposes a probabilistic range filter to speed up range queries without losing performance. However, the methods are dedicated to the range query on keys (i.e., time in time series databases), while the outlier queries consider the distances of values.

TABLE II
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 $\lvert B[u]\rvert$

III. Preliminary

In this section, we first introduce the distance-based outliers in time series in Section III-A. The elements of LSM-tree storage for time series are presented in Section III-B. It leads to the query of outliers in the LSM-tree based time series database in Section III-C.

A. 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 [7].

Definition 1 (Neighbor)**.** *Given distance threshold $r(r>0)$, a point p is a r-neighbor of point p' in a window W, if*

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

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

Definition 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 [7] is to detect outliers in each sliding window over the time series T.

Definition 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 $\ldots, W_{t-s}, W_t, W_{t+s}, \ldots$, 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 (Example 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 $O_{00:00:05} = \{p_8\}$.*

B. LSM-tree Storage

A time series T is stored in multiple files, written to the database at different time. We follow the convention of Apache IoTDB [15], a time series database built upon Log-Structured Merge-Tree (LSM-tree) [13].

Definition 4 (Segment)**.** *A time series T is represented by a number of segments, $T = S_1 \cup \cdots \cup S_\tau$, where each segment has*

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

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

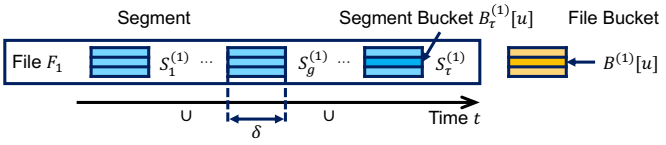

Fig. 2. A file F_1 consisting of a series of segments $S_g^{(1)}$

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 shows a series of segments $S_1^1, \ldots, S_\tau^{(1)}$ stored in a file with ordered timestamps. Each segment has bucket statistics which will be introduced later in Section IV-A.

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.

Definition 5 (File)**.** *A file $F_h = S_1^{(h)} \cup \cdots \cup S_\tau^{(h)}$ consists of segments written with version number h.*

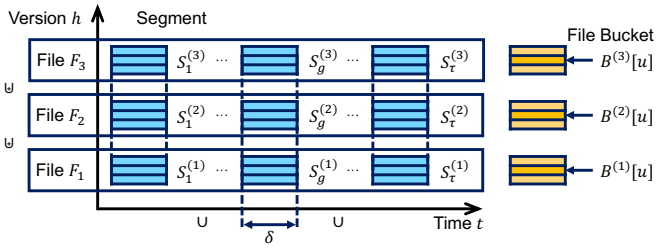

Fig. 3. Multiple file F_h with a series of segments S_g

Figure 3 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 6 (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 occurs by filling a default value of a delayed point for data analysis and updating it on arrival later.

Proposition 1. *A time series T stored in multiple files and segments can be obtained by*

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

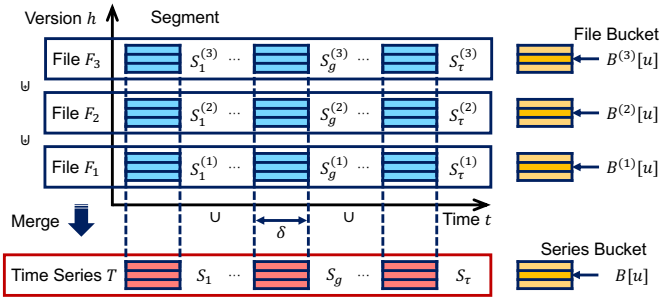

Fig. 4. A time series T stored in segments S_g and files F_h

Example 3. *Figure 4 presents a time series T stored in three files, $\{F_1, F_2, F_3\}$. Each file consists of at most τ segments, with non-overlapping time range δ, e.g., $F_1 = S_1^{(1)} \cup \cdots \cup S_\tau^{(1)}$. 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 6, we obtain the segment S_1 of the time series T in the corresponding time range, i.e., $S_1 = S_1^{(1)} \uplus S_1^{(2)} \uplus S_1^{(3)}$. Finally, the time series T is obtained by concatenating all the segments, $T = S_1 \cup \cdots \cup S_\tau = (S_1^{(1)} \uplus S_1^{(2)} \uplus S_1^{(3)}) \cup \cdots \cup (S_\tau^{(1)} \uplus S_\tau^{(2)} \uplus S_\tau^{(3)})$.*

C. Outlier Query on LSM-tree Storage

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

Definition 7 (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 $\ldots, W_{t-s}, W_t, W_{t+s}, \ldots$, 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 = 1d (1 day) and slide s = 3h (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.

To implement the outlier query in the database, a straightforward idea is to first obtain the queried time series T as in Proposition 1. Existing algorithms [8], [9], [16] are then conducted over the merged time series T. As aforesaid, the query is repeatedly conducted, trying different parameters for various data science tasks. Online merging time series and finding outliers from scratch each time are obviously costly.

IV. Query Processing in Single File

In this section, we first present the outlier query processing in single file, and then extend it to multiple files in Section V. It is worth noting that the single file case is also prevalent and important in LSM-tree storage, as discussed in Section I-B. Thus, we also focus on the performance of our proposal on single file. The method presented below can thus be applied to the merged file.

Intuitively, we may utilize the statistics in files to prune data points with sufficient/insufficient neighbors for sure. Section IV-A introduces the statistics in segments, and Section IV-B aggregates them for the query. Without loading all the data, we derive the bounds of neighbor counts for points in Section IV-C and enable the pruning in Section IV-D.

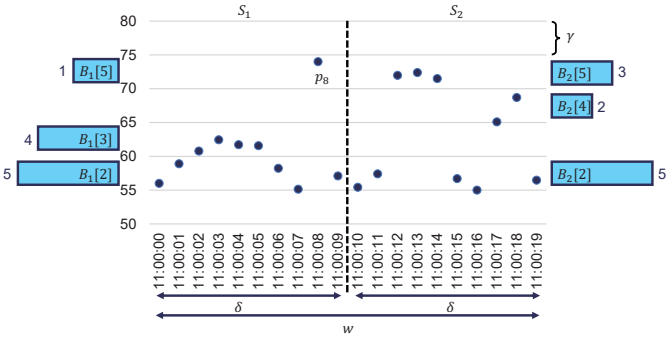

Fig. 5. Buckets with $\gamma = 5$ of two segments with $\delta = 10$

A. 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 illustrated in Figure 2. Since only one file is considered in this section, we omit the default file version number h for simplicity.

Definition 8 (Bucket)**.** *For a time series segment S_g, we consider a number of buckets $\mathscr{B}_g = \{B_g[1], \ldots, B_g[\beta]\}$, where each bucket has a width γ on value range,*

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

$u = 1, \ldots, \beta$ 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 4. *Consider a window containing two segments, S_1 and S_2, with time range $\delta = 10$ in Figure 5. 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., $|B_1[2]| = 5, |B_1[3]| = 4, |B_1[5]| = 1$.*

B. Aggregating Bucket Statistics in a Window

As introduced in Section III-C, 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 2 (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 8. 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 6 presents the buckets by merging/aggregating the two segments in Figure 5.

Example 5 (Example 4 continued)**.** *Consider again the window with two segments in Figure 5. 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 thus be directly aggregated without loading the data, $|B[5]| = |B_1[5]| + |B_2[5]| = 4$. Figure 6 illustrates the aggregated bucket statistics $|B[u]|$ for the window W in Figure 5, where the lengths of bars represent the bucket sizes.*

C. Bounds of Neighbor Count

To identify an outlier, as introduced in Definition 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. The results regarding the bounds of the r-neighbor count are presented as follows.

1) Upper Bound: An important observation is that the query-specified neighbor distance threshold r may exhibit various relationships with the fixed bucket width γ.

Intuitively, considering a threshold r smaller than half of the bucket width γ, the r-neighbors of a point p can only appear in its bucket or one of the adjacent buckets. In contrast, given a threshold r larger than half of the bucket width γ, 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.

a) Condition (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 case of threshold r larger than half of the bucket width γ. 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. *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 $\lambda = \lceil r/\gamma \rceil$.

b) Condition (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 case of threshold r smaller than half of the bucket width γ corresponds to $0 < r \leq \frac{1}{2}\gamma$, for $\lambda = 1$. In this case, the potential r-neighbors of a point $p \in B[u]$ can be further restricted to less

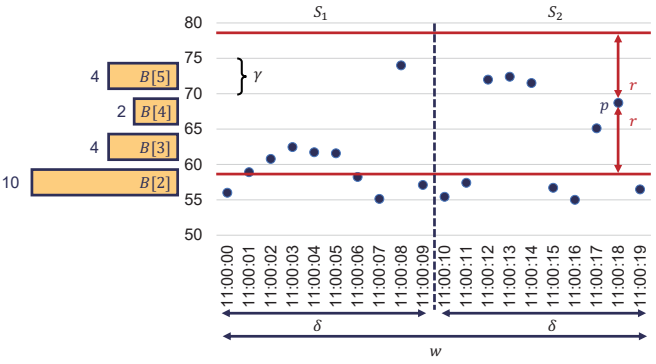

Fig. 6. Aggregating bucket statistics and pruning point p in a window W

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 4. *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 $\lambda = \lceil r/\gamma \rceil$.

2) Lower Bound: Intuitively, for the case of neighbor distance threshold r less than the bucket width γ, $r < \gamma$, it is possible that the r-neighbor count is less than $|B[u]|$, and thus could not be bounded by the bucket statistics. 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 5. *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 $\ell = \lfloor r/\gamma \rfloor$.

Example 6. *Table III presents the bounds of the first 8 example cases by increasing r from 0 to 2γ, corresponding to 8 example cases in Figure 7. For larger r, greater than 2γ, the bounds are derived similarly referring to Propositions 3, 4 and 5.*

(i) *Specifically, for $r \geq \gamma$, i.e., example cases (4)-(8) and so on, the lower bounds are obtained by Proposition 5. For instance, for the example case (4) with $r = \gamma$, we have the lower bound $\sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| = |B[u]|$ where $\ell = \lfloor r/\gamma \rfloor = 1$.*

(ii) *Moreover, for $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$, corresponding to example cases (1)-(2), (5)-(6), ..., the upper bounds are derived by Proposition 4. For instance, for the example case (1) with $r < \frac{1}{2}\gamma$, the upper bound is $\max\left(\sum_{i=u-\lambda}^{u+\lambda-1} |B[i]|, \sum_{i=u-\lambda+1}^{u+\lambda} |B[i]|\right) = \max(|B[u-1]| + |B[u]|, |B[u]| + |B[u+1]|)$ given $\lambda = \lceil r/\gamma \rceil = 1$.*

(iii) *In contrast, for $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$, i.e., example cases (3)-(4), (7)-(8), ..., the upper bounds are given in Proposition 3.*

TABLE III
EXAMPLES OF BOUNDS ON r-NEIGHBOR COUNT $|N(p,r)|$

Example case	Lower bound	Upper bound
(1) $r < \frac{1}{2}\gamma$	–	**Prop. 4:** $\max(\|B[u-1]\|+\|B[u]\|, \|B[u]\|+\|B[u+1]\|)$
(2) $r = \frac{1}{2}\gamma$	–	**Prop. 4:** $\max(\|B[u-1]\|+\|B[u]\|, \|B[u]\|+\|B[u+1]\|)$
(3) $\frac{1}{2}\gamma < r < \gamma$	–	**Prop. 3:** $\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|$
(4) $r = \gamma$	**Prop. 5:** $\|B[u]\|$	**Prop. 3:** $\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|$
(5) $\gamma < r < \frac{3}{2}\gamma$	**Prop. 5:** $\|B[u]\|$	**Prop. 4:** $\max(\|B[u-2]\|+\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|, \|B[u-1]\|+\|B[u]\|+\|B[u+1]\|+\|B[u+2]\|)$
(6) $r = \frac{3}{2}\gamma$	**Prop. 5:** $\|B[u]\|$	**Prop. 4:** $\max(\|B[u-2]\|+\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|, \|B[u-1]\|+\|B[u]\|+\|B[u+1]\|+\|B[u+2]\|)$
(7) $\frac{1}{2}\gamma < r < 2\gamma$	**Prop. 5:** $\|B[u]\|$	**Prop. 3:** $\|B[u-2]\|+\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|+\|B[u+2]\|$
(8) $r = 2\gamma$	**Prop. 5:** $\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|$	**Prop. 3:** $\|B[u-2]\|+\|B[u-1]\|+\|B[u]\|+\|B[u+1]\|+\|B[u+2]\|$
...	...	...

For instance, for the example case (3) with $\frac{1}{2}\gamma < r < \gamma$, the upper bound has $\sum_{i=u-\lambda}^{u+\lambda} |B[i]| = |B[u-1]|+|B[u]|+|B[u+1]|$ where $\lambda = \lceil r/\gamma \rceil = 1$.

D. Pruning by Bucket Statistics

Referring to the upper and lower bounds of neighbor count in Propositions 3, 4 and 5, there are three cases for points p in bucket $B[u]$ to consider in pruning.

1) Case 1: p must be an inlier: If the lower bound in Proposition 5 is no less than the neighbor count threshold k, i.e., $\sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| \geq k$, $\ell = \lfloor r/\gamma \rfloor$, then all the points p in bucket $B[u]$ must have r-neighbor count $|N(p,r)| \geq k$, i.e., inliers. That is, all the data points in bucket $B[u]$ are pruned and have no need to load from disk.

2) Case 2: p must be an outlier: If the upper bound in Proposition 3 or 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. Again, for a query of outlier count, the data points in bucket $B[u]$ have no need to load from disk.

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 check further. Fortunately, 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 $\ell = \lfloor r/\gamma \rfloor$ and $\lambda = \lceil r/\gamma \rceil$. For instance, for the example case (8) in Figure 7, 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 example cases (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 formula of computing the r-neighbor count is given below.

Proposition 6. *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*

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

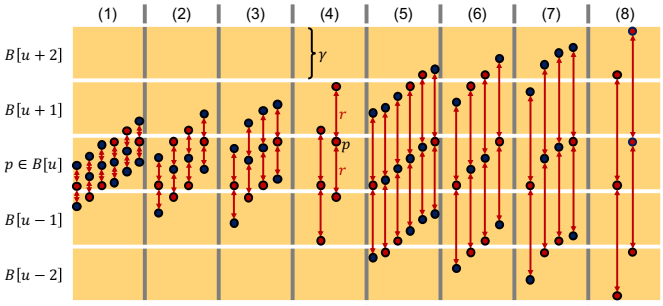

Fig. 7. Possible r-neighbor scenarios for 8 cases with r from 0 to 2γ

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

Specially, when $\ell = \lambda$, e.g., example cases (4) and (8) in Figure 7, the count of its r-neighbors 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 6. Indeed, for $\ell = \lfloor r/\gamma \rfloor < 1$, i.e., example cases (1)-(3) in Figure 7, it also needs only to load 2 additional buckets, $B[u-1]$ and $B[u+1]$.

Example 7 (Example 5 continued)**.** *Consider a query with $w = 20, s = 10, k = 10, r = 10$. Figure 6 illustrates the time series in Figure 5. For a point p in bucket $B[4]$, the lower bound of $|N(p,r)|$ is $|B[3]|+|B[4]|+|B[5]| = 4+2+4 = 10$, referring to Proposition 5. Moreover, the upper bound of $|N(p,r)|$ is $|B[2]|+|B[3]|+|B[4]|+|B[5]|+|B[6]| = 4+2+4+10+0 = 20$, since $r = 2\gamma = 10$ and Proposition 3 applies. Given $|N(p,r)| \geq 10 = k$, point p must be an inlier, i.e., the aforesaid case 1. However, for another query with neighbor count threshold $k = 15$, the points in bucket $B[4]$ cannot be pruned by the aforesaid lower and upper bounds. According to Proposition 6 in case 3, 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 $p \in B[2]$. 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.*

V. 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 IV-A may no longer be valid, since the points may be overwritten by

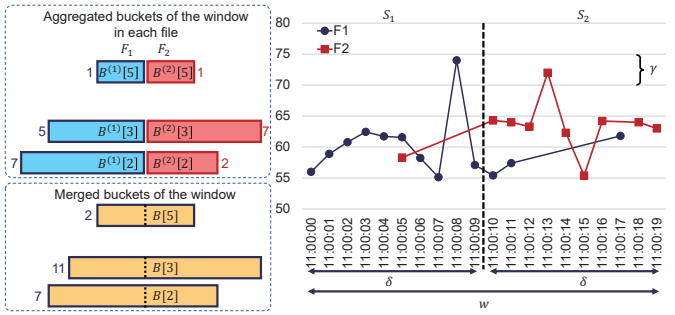

Fig. 8. Buckets of a window in two files F_1 and F_2

others delayed with higher file version numbers. Nevertheless, we study the bounds of bucket statistics considering multiple files in Section V-A. Likewise, the lower and upper bounds of neighbor count can still be derived for a point in pruning in Section V-B. Finally, Section V-C presents the algorithm for outlier query processing in multiple files.

A. Bound of Bucket Statistics in Multiple Files

Though the bucket statistics of a window in single file can be aggregated by Proposition 2, they cannot be further aggregated in multiple files, due to the merge operation in Definition 6 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 7 (File Statistics)**.** *For 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*

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

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 IV.

Example 8. *Figure 8 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 4, 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 $|B[2]| = 7 > |\check{B}[2]| = |B^{(2)}[2]| = 2$.*

B. Pruning by Bucket Statistics on Files

Combining the aforesaid bounds of bucket sizes in Proposition 7 and the bounds of neighbor count w.r.t. bucket sizes in Section IV-C, we can still prune inliers and outliers for multiple files.

1) Pruning by Upper Bound: Analogous to Proposition 3, for $(\lambda - \frac{1}{2})\gamma < r \le \lambda\gamma, \lambda = 1,2,\ldots$, the bound of neighbor count is obtained as follows for pruning, referring to the bucket statistics $|B_g^{(h)}[i]|$ in segment S_g of file F_h.

Proposition 8. *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*

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

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

Likewise, similar to Proposition 4, for $(\lambda - 1)\gamma < r \le (\lambda - \frac{1}{2})\gamma, \lambda = 1,2,\ldots$, we obtain the upper bound for multiple files.

Proposition 9. *For a point p in the bucket $B[u]$, if $\lceil r/\gamma \rceil - r/\gamma \ge \frac{1}{2}$, then the count of its r-neighbors has an upper bound*

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

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

2) Pruning by Lower Bound: Again, based on Proposition 5 for $r \ge \gamma$, we extend the lower bound for pruning inliers.

Proposition 10. *For a point p in the bucket $B[u]$, if $r \ge \gamma$, then the count of its r-neighbors has a lower bound*

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

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

C. 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 7. Based on the bounds of neighbor count in Section V-B, 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 6, instead of $2\lambda+1$ buckets.

Algorithm 1 Outlier Detection Algorithm 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: $\lambda := \lceil r/\gamma \rceil$
2: $\ell := \lfloor r/\gamma \rfloor$
3: initialize outlier set $O_t := \emptyset$
4: initialize buckets $\mathscr{B}$ 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: $|\check{B}[u]| := |B^{(\rho)}[u]|$
12: $|\hat{B}[u]| := \sum_{h=1}^{\rho} |B^{(h)}[u]|$
13: **if** $\sum_{i=u-\ell+1}^{u+\ell-1} |\check{B}[i]| \geq k$ **then**
14: continue
15: **else if** $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$ **and** $\sum_{i=u-\lambda}^{u+\lambda} |\hat{B}[i]| < k$ **then**
16: $O_t := O_t \cup B[u]$
17: **else if** $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$ **and** $\max\left(\sum_{i=u-\lambda}^{u+\lambda-1} |\hat{B}[i]|, \sum_{i=u-\lambda+1}^{u+\lambda} |\hat{B}[i]|\right) < k$ **then**
18: $O_t := O_t \cup B[u]$
19: **else**
20: load buckets from $B[u-\lambda]$ to $B[u+\lambda]$
21: point-wise checking the points in the buckets
22: **return** O_t

Complexity Analysis: Consider ρ files, β buckets and ω segments in a window. The update of bucket statistics takes $\mathscr{O}(\beta\omega\rho)$ time. In the worst case, all the n points in the window may be output as outliers, in $\mathscr{O}(n)$ 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 $\mathscr{O}(\rho n)$ time. Referring to the average number of points in a bucket $\frac{n}{\beta}$, it takes $\mathscr{O}(\frac{n^2}{\beta})$ time.

Finally, Algorithm 1 runs in $\mathscr{O}(\beta\omega\rho + \rho n + \frac{n^2}{\beta})$ time, and needs $\mathscr{O}(\beta\zeta\rho)$ 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.

VI. Extension

In this section, we further extend our proposal to density-based and multi-variate outlier detection methods in Sections VI-A and VI-B, respectively.

A. Adaption to Density-based Outlier Detection

While the distance-based outlier definition [7] is simple, it also has many practical applications, such as anomaly detection in wearable devices [9], abnormal vital sign detection in healthcare [16], computer network attack detection [8], and so on. Nevertheless, our proposal can be adapted to more advanced definitions such as density-based outliers [28]. Compared to distance-based outliers, the density-based definition further distinguishes some points from outliers, which do not have sufficient neighbors but are close enough to some inliers, known as border points.

Definition 9 (Density-based outlier [28])**.** *A point p is a core point if $N(p,r)$ covers more than k points (including itself). If p is not a core point but has a core point r-neighbor p', then p is a border point. Otherwise, p is a density-based outlier.*

Referring to Definitions 2 and 9, the border points in density-based method are considered as outliers in distance-based method. That is, the outlier result set of density-based method is a subset of that of distance-based method.

With our proposal, we can detect core points and non-core points thus far. Intuitively, core points can be determined in the same way as that in Case 1, Section IV-D. In the following, we propose to further distinguish border points from non-core points by modifying the pruning strategy in Section IV-D. For a point p, if $|N(p,2r)| < k$, there should not exist core points in $N(p,r)$, i.e., p must be an outlier. If there is a point p' having sufficient neighbors with $dist(p,p') < r$, then p is a border point. The following proposition formalizes the pruning cases for density-based outlier detection, corresponding to that in Section IV-D.

Proposition 11. *For a point p in $B[u]$, there are three pruning cases for density-based outlier detection as follows:*
(1) point p must be an inlier, if $\sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| \geq k$,
(2) point p must be an outlier, if $\sum_{i=u-2\lambda}^{u+2\lambda} |B[i]| < k$,
(3) point p must be a border point, if there exists another point p' in $B[u']$ satisfying

$$|u-u'| \leq \ell - 1,$$
$$\sum_{i=u'-\ell+1}^{u'+\ell-1} |B[i]| \geq k,$$

where $\ell = \lfloor r/\gamma \rfloor, \lambda = \lceil r/\gamma \rceil$.

By utilizing the above pruning strategy, we can easily design our density-based adaption. We aggregate the bucket statistics in a window referring to Section IV-B, calculate the lower and upper bounds, i.e., the left-side terms in Proposition 11(1)&(2), and prune inliers and outliers. If a point p in $B[u]$ cannot be pruned, we load buckets statistics from $B[u-2\ell+2]$ to $B[u+2\ell-2]$ and verify the condition in Proposition 11(3). Otherwise, we load all the points in $B[u-2\lambda] \cup \cdots \cup B[u+2\lambda]$ and conduct point-wise inspection.

B. Application to Multi-Variate Outlier Detection

Note that this work mainly focuses on univariate time series owing to the columnar storage, which also has many important applications such as filtering spikes of stock prices [1], identifying change points of temperature when cold air rushes in [3], etc. Nevertheless, in order to extend the scope of this work, the proposed techniques can be further applied to

TABLE IV
DATASET AND QUERY SETTINGS

Name	# data points	bucket width γ	neighbor distance threshold r
TAO-OceanGraphic	600k	5	15
UCI-Gas	900k	0.1	3
UCI-PAMAP2	300k	0.1	0.4
WH-Chemistry	100m	2	10
TY-Vehicle	100m	2	10
GW-WindTurbine	100m	5	20

process multi-variate data. Different from the univariate case, multi-variate outliers are defined by Euclidean distance.

Definition 10 (Multi-variate outlier)**.** *A point* $\mathbf{p} = (p_1, \ldots, p_d)$ *is a multi-variate outlier in* $\mathbb{R}^d$*, if* $N(\mathbf{p}, r)$ *covers less than* k *points, i.e.,*

$$\left| \left\{ \mathbf{p}' \middle| \sqrt{\sum_{i=1}^{d} (p_i - p'_i)^2} \leq r \right\} \right| < k.$$

Intuitively, suppose that $\mathbf{p}$ is an outlier on dimension i, that is, no sufficient points p'_i on dimension i with $|p'_i - p_i| \leq r$. Since $dist(\mathbf{p}, \mathbf{p}') = \sqrt{\sum_{i=1}^{d} (p_i - p'_i)^2} \geq |p'_i - p_i|, \forall i = 1, \ldots, d$, there should not exist sufficient points $\mathbf{p}' = (p'_1, \ldots, p'_d) \in \mathbb{R}^d$ such that $dist(\mathbf{p}, \mathbf{p}') \leq r$. The proposition below formalizes the relationship between univariate outliers and multi-variate outliers.

Proposition 12. *For a point* $\mathbf{p} = (p_1, \ldots, p_d) \in \mathbb{R}^d$*, if there exists one dimension* i*, such that* p_i *has less than* k *neighbors in its* r*-neighborhood on the* i*-th dimension,* $\mathbf{p}$ *must be an outlier in* $\mathbb{R}^d$ *w.r.t.* k *and* r*.*

In this sense, to filter outliers in $\mathbb{R}^d$, we may apply our univariate solution to every dimension. Proposition 12 ensures that our method always returns a subset of the outliers defined on multiple dimensions.

VII. EXPERIMENTS

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

A. Experimental Settings

To illustrate the efficiency of LSMOD, we compare with three baselines, MCOD [8], NETS [16] and CPOD [9], including the last two state-of-the-art methods, referring to Table I. The implementation employs the series reader in the database to feed point by point to these streaming methods. The series reader loads all files and merges online the data points in different files to handle the out-of-order issues. Since the outlier definition of these methods is exactly the same, i.e., the same outlier query results, the evaluation mainly focuses on the efficiency of approaches.

Table IV lists the real-world datasets employed in the experiments, stored on the disk in the database. While the first three public datasets are small in size, the last three datasets, collected by our industrial partners, contain hundreds of millions data points. The default query is with neighbor count threshold $k = 50$, window size $w = 20$ mins and slide size $s = 10$ mins. Since the datasets are with various value ranges, the default neighbor distance threshold r and bucket width γ, related to values, are also listed in Table IV. The time costs of processing all sliding windows in a query are reported below.

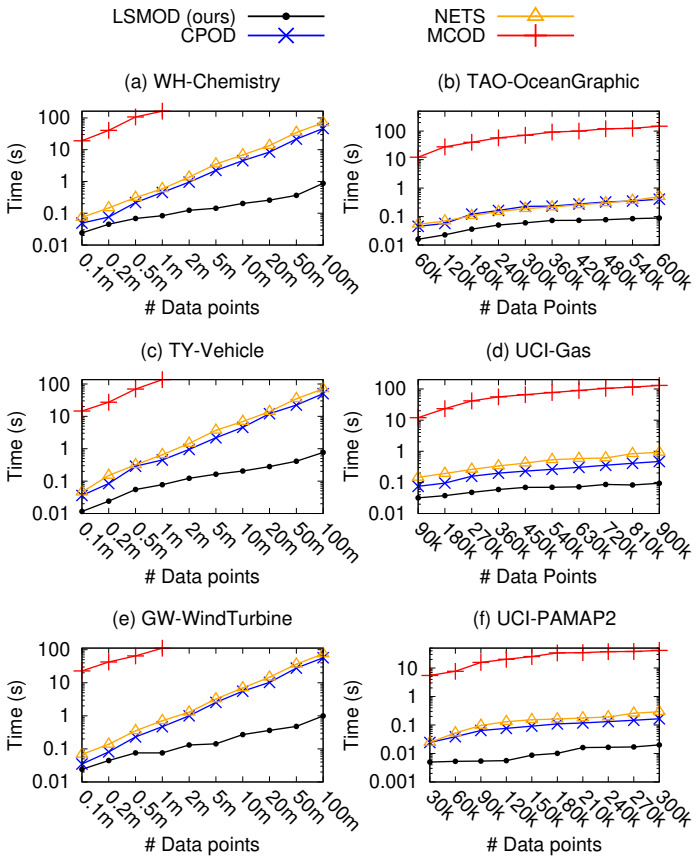

Fig. 9. Query time costs under different data sizes

All the experiments run on a machine with Intel Core i7 (2.80GHz), 16 GB of memory, and Apache IoTDB v0.13.

B. Comparison with Existing Methods

In this experiment, we demonstrate that our LSMOD performs the best in all the datasets, efficiently supporting various queries. It shows orders of magnitude improvement in time costs compared to the existing methods.

1) Scalability in Data Size: Figure 9 reports the query time costs for different numbers of data points. It is not surprising that the time costs of all methods increase with the data sizes. Unfortunately, MCOD cannot handle datasets with more than 1 million points in a limit of 200 seconds. Our proposed LSMOD consistently shows 1–3 order of magnitude improvement compared to the baselines.

Note that NETS and CPOD have similar time costs in Figure 9, which is consistent with the previous evaluation [9].

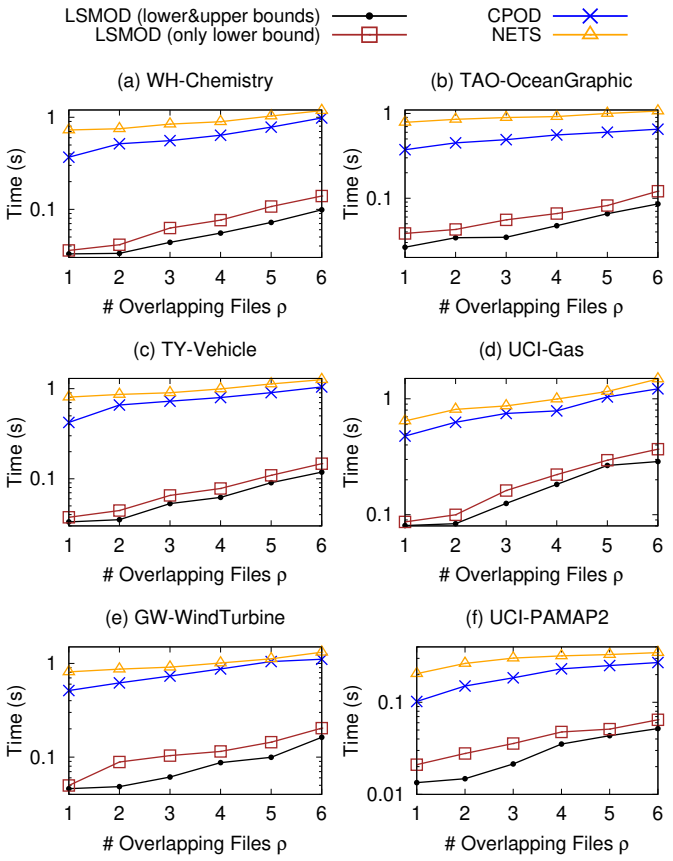

Fig. 10. Varying the number of overlapping files (1 means single file)

This is because all the evaluated time series have only one dimension, where the pruning techniques of CPOD do not obviously reduce the time cost for computing distances.

2) Varying the Number of Overlapping Files ρ: Figure 10 varies the number of overlapping files ρ, under the same data sizes. When a query finds all the data stored in one file, single file solution in Section IV will be used. Otherwise, Algorithm 1 for multiple files will apply.

As shown, the query processing time cost increases with the increase of the number of overlapping files, since more overlapping files lead to more time expending on merging buckets (referring to the complexity analysis in Section V-C) and looser bucket statistics bounds in Proposition 7. For CPOD and NETS, they also takes more time when the overlapping file number increases, because they spend more time loading and merging the overlapping files.

To handle out-of-order data, we may first merge the files with overlapping time ranges and then apply the streaming data methods, which might be unfair in comparison. In this sense, we also compare these methods by querying single files (without merging). Figure 10 shows the results with 1 overlapping file, i.e., the case of single file. As shown, the baselines are still slower, since they need to load all the data points. Instead, our proposal utilizes the pre-computed statistics to further prune the data to load.

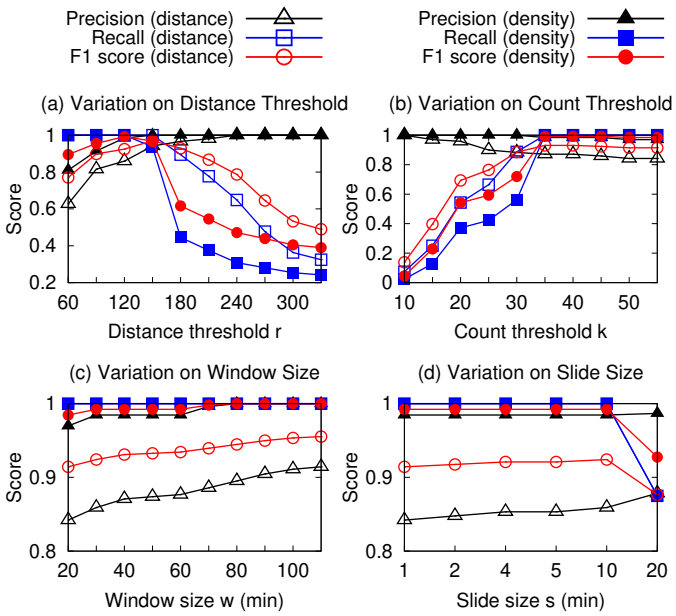

Fig. 11. Precision, recall and F1 score under different query parameters

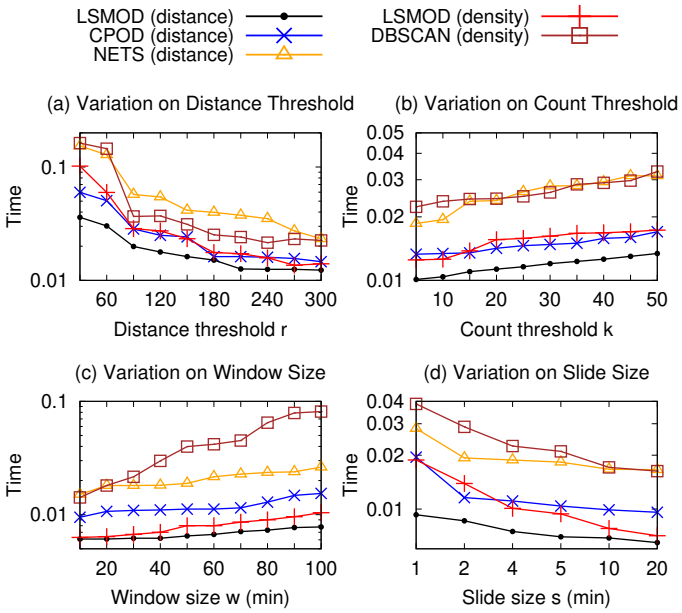

Fig. 12. Time cost under different query parameters

3) Performance under Various Query Parameters: We evaluate precision, recall, F1 score and most importantly time cost to show how parameters r, k, w, s impact the detection of outliers, in Figures 11 and 12. The experiments are conducted on a dataset of stock prices, with true outliers labeled by our industrial partner. Moreover, we also discuss how to choose these parameters referring to the existing studies such as [7].

With the increase of distance threshold r and the decrease of count threshold k, less points are detected as outliers, leading to lower time cost in Figure 12. The corresponding precision increases, while the recall drops as shown in Figure 11. The overall F1 score is the highest given moderately large $r = 150$

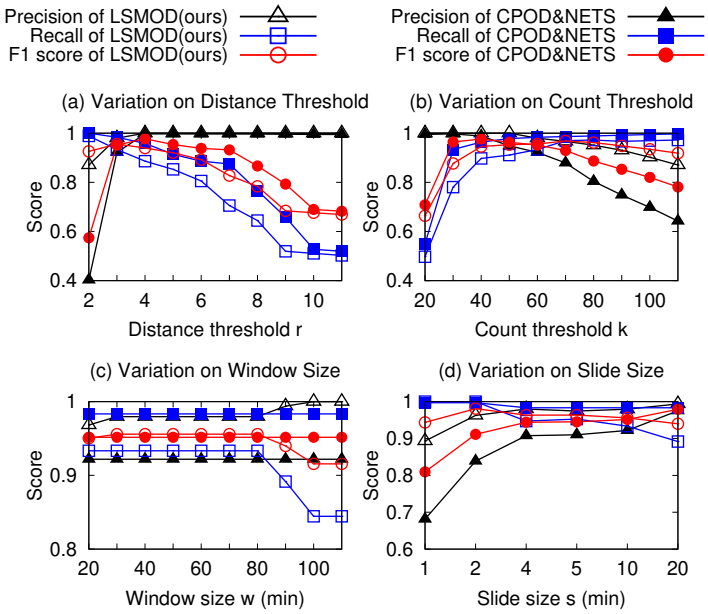

Fig. 13. Precision, recall and F1 score on multi-variant datasets

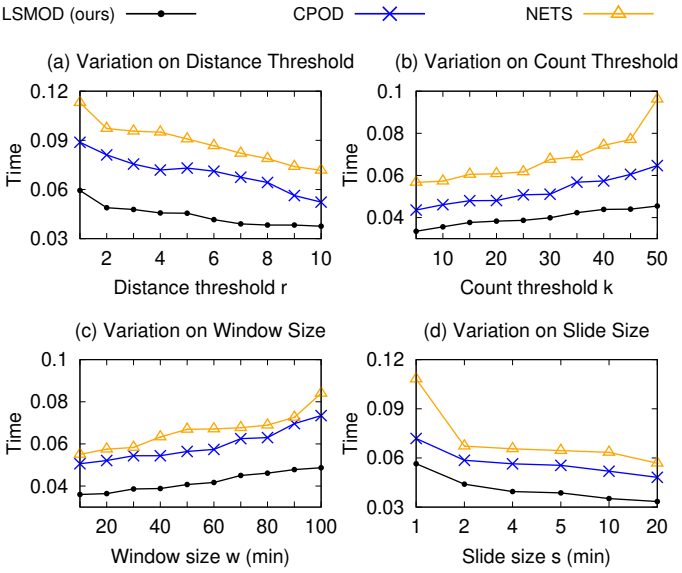

Fig. 14. Time costs under different query parameters on multi-variant datasets

and $k = 40$. A larger window size w means more data points under consideration, with better accuracy in Figure 11 and higher time cost in Figure 12. Finally, a small enough slide size s has no much impact on accuracy but higher time cost.

Since our study focuses on improving the efficiency of distance-based outlier detection [7] under various parameter settings, we choose the parameters as in [9], [16], [29]. That is, the default parameters are set to detect around 1% outliers from the entire dataset. For example, given default $k = 50$, the query with $r = 90$ returns about 1% points in the entire dataset. A sufficiently large window size w and a moderate slide size s generally show stable and better results as shown in Figure 11.

C. Evaluation of Extensions

Finally, we evaluate two extensions of our proposal, density-based and multi-variate outlier detection, in Section VI.

1) Evaluation of Density-based Extension: To validate our density-based extension in Section VI-A, we conduct evaluation on the aforesaid stock dataset with labeled true outliers in Section VII-B3. Figure 11 shows the precision, recall and F1 score under different query parameters. To better demonstrate the efficiency of our adaption, we compare with a classic density-based implementation DBSCAN [28] in Figure 12.

As the experiments shown in Figure 11, the density-based definition has higher precision but lower recall than the distance-based one. The optimal F1 score of the advanced density-based definition is higher. Nevertheless, with the same density definition, our density-based LSMOD implementation has significantly lower time cost than the existing implementation of density-based DBSCAN [28] in Figure 12.

2) Evaluation of Multi-Variant Application: In this experiment, we compare our multi-variate adaption with CPOD and NETS. The evaluation is conducted on a public multi-variate dataset [30] with around 30k points, which records undesirable real events in oil wells.

Figure 13 shows the precision, recall and F1 score under different query parameters, respectively. The precision of our method is higher but with lower recall, since it always returns a subset of outliers defined on multiple dimensions (Proposition 12). The overall F1 score is comparable as shown in Figure 13, whereas our time cost is significantly lower in Figure 14.

VIII. Conclusion

In this paper, we present an efficient method, LSMOD, for querying outliers in 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 in each file and merging them as the results. Based on the bucket statistics in files, we can derive the upper and lower bounds of neighbor count to prune inliers and outliers, respectively. Moreover, we illustrate that the neighbor count may also be determined by loading points in a constant number (at most 4) of additional buckets. Extensive experiments demonstrate the high efficiency of the proposed LSMOD compared to the existing methods. Remarkably, our proposal has been deployed as a function in an LSM-tree based time series database, Apache IoTDB.

Acknowledgment: This work is supported in part by the National Natural Science Foundation of China (92267203, 62021002, 62072265, 62232005), the National Key Research and Development Plan (2021YFB3300500), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] S. Song, A. Zhang, J. Wang, and P. S. Yu, "SCREEN: stream data cleaning under speed constraints," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 827–841. [Online]. Available: https://doi.org/10.1145/2723372.2723730

[2] A. Zhang, S. Song, and J. Wang, "Sequential data cleaning: A statistical approach," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 909–924. [Online]. Available: https://doi.org/10.1145/2882903.2915233

[3] A. Zhang, S. Song, J. Wang, and P. S. Yu, "Time series data cleaning: From anomaly detection to anomaly repairing," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1046–1057, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1046-song.pdf

[4] A. Blázquez-García, A. Conde, U. Mori, and J. A. Lozano, "A review on outlier/anomaly detection in time series data," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 56:1–56:33, 2021. [Online]. Available: https://doi.org/10.1145/3444690

[5] S. Song, F. Gao, A. Zhang, J. Wang, and P. S. Yu, "Stream data cleaning under speed and acceleration constraints," *ACM Trans. Database Syst.*, vol. 46, no. 3, pp. 10:1–10:44, 2021. [Online]. Available: https://doi.org/10.1145/3465740

[6] C. Chen and L.-M. Liu, "Forecasting time series with outliers," *Journal of forecasting*, vol. 12, no. 1, pp. 13–35, 1993.

[7] L. Tran, L. Fan, and C. Shahabi, "Distance-based outlier detection in data streams," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1089–1100, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1089-tran.pdf

[8] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams," in *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, S. Abiteboul, K. Böhm, C. Koch, and K. Tan, Eds. IEEE Computer Society, 2011, pp. 135–146. [Online]. Available: https://doi.org/10.1109/ICDE.2011.5767923

[9] L. Tran, M. Mun, and C. Shahabi, "Real-time distance-based outlier detection in data streams," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 141–153, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p141-tran.pdf

[10] "https://iotdb.apache.org/."

[11] "https://www.influxdata.com."

[12] J. Xiao, Y. Huang, C. Hu, S. Song, X. Huang, and J. Wang, "Time series data encoding for efficient storage: A comparative analysis in apache iotdb," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2148–2160, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2148-song.pdf

[13] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[14] Y. Kang, X. Huang, S. Song, L. Zhang, J. Qiao, C. Wang, J. Wang, and J. Feinauer, "Separation or not: On handing out-of-order time-series data in leveled lsm-tree," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 3340–3352. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00315

[15] C. Wang, J. Qiao, X. Huang, S. Song, H. Hou, T. Jiang, L. Rui, J. Wang, and J. Sun, "Apache iotdb: A time series database for iot applications," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 195:1–195:27, 2023. [Online]. Available: https://doi.org/10.1145/3589775

[16] S. Yoon, J. Lee, and B. S. Lee, "NETS: extremely fast outlier detection from a data stream via set-based processing," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1303–1315, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1303-yoon.pdf

[17] "https://iotdb.apache.org/UserGuide/Master/Operators-Functions/Anomaly-Detection.html#outlier."

[18] "https://github.com/apache/iotdb/tree/research/outlier."

[19] "https://github.com/iotdb-lsmod/iotdb-lsmod."

[20] "https://iotdb-lsmod.github.io/iotdb-lsmod/supplementary.pdf."

[21] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," in *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, M. J. Silva, A. H. F. Laender, R. A. Baeza-Yates, D. L. McGuinness, B. Olstad, Ø. H. Olsen, and A. O. Falcão, Eds. ACM, 2007, pp. 811–820. [Online]. Available: https://doi.org/10.1145/1321440.1321552

[22] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE Computer Society, 2014, pp. 76–87. [Online]. Available: https://doi.org/10.1109/ICDE.2014.6816641

[23] P. Boniol, M. Linardi, F. Roncallo, T. Palpanas, M. Meftah, and E. Remy, "Unsupervised and scalable subsequence anomaly detection in large data series," *VLDB J.*, vol. 30, no. 6, pp. 909–931, 2021. [Online]. Available: https://doi.org/10.1007/s00778-021-00655-8

[24] P. Boniol, J. Paparrizos, T. Palpanas, and M. J. Franklin, "SAND: streaming subsequence anomaly detection," *Proc. VLDB Endow.*, vol. 14, no. 10, pp. 1717–1729, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1717-boniol.pdf

[25] P. Boniol and T. Palpanas, "Series2graph: Graph-based subsequence anomaly detection for time series," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 1821–1834, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1821-boniol.pdf

[26] Y. Zhang, Y. Li, F. Guo, C. Li, and Y. Xu, "Elasticbf: Fine-grained and elastic bloom filter towards efficient read for lsm-tree-based KV stores," in *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*, A. Goel and N. Talagala, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/zhang

[27] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2071–2086. [Online]. Available: https://doi.org/10.1145/3318464.3389731

[28] J. Gan and Y. Tao, "DBSCAN revisited: Mis-claim, un-fixability, and approximation," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2723372.2737792

[29] S. Yoon, Y. Shin, J. Lee, and B. S. Lee, "Multiple dynamic outlier-detection from a data stream by exploiting duality of data and queries," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 2063–2075. [Online]. Available: https://doi.org/10.1145/3448016.3452810

[30] R. E. V. Vargas, C. J. Munaro, P. M. Ciarelli, A. G. Medeiros, B. G. do Amaral, D. C. Barrionuevo, J. C. D. de Araújo, J. L. Ribeiro, and L. P. Magalhães, "A realistic and public dataset with rare undesirable real events in oil wells," *Journal of Petroleum Science and Engineering*, vol. 181, p. 106223, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0920410519306357

To Reviewer 1

R1O1 (MR2O4.a): *Some figures, such as Figure 2, are too complex. To illustrate structures that change over time it would be good to use a series of figures where each figure illustrate the state of the structure at a specific point in time.*

Reply: As suggested, to better illustrate specific parts of the time series storage structure in LSM-tree based store, we introduce the complex Figure 2 gradually by a series of figures, i.e., Figures 2-4 corresponding to Definitions 4-6, respectively.

R1O2 (MR2O1): *The paper is mostly a description of a comprehensive implementation. The level of detail and comprehensiveness are a strength of the paper. What is limited is the research novelty. More space should be allocated to discuss the research contribution of the paper.*

Reply: Following the suggestion, we highlight the research novelty, e.g., establishing bounds of neighbor counts based on statistics in Propositions 3-6 and the bounds derived for multiple files in Propositions 8-10, at the end of Section I-C, Page 2. Moreover, we introduce more contributions on density-based and multi-variate outlier detection, in Section VI, Page 9.

R1O3 (MR2O4.b): *r, k, w, s appear to be crucial and difficult to choose parameters. Please discuss how they impact the detection of outliers and how you choose these parameters.*

Reply: We evaluate precision, recall, F1 score and most importantly time cost to show how parameters r, k, w, s impact the detection of outliers, in Figures 11 and 12. Moreover, we also discuss how to choose these parameters referring to the existing studies such as [7], in Section VII-B3, Page 11.

With the increase of distance threshold r and the decrease of count threshold k, less points are detected as outliers, leading to lower time cost in Figure 12. The corresponding precision increases, while the recall drops as shown in Figure 11. The overall F1 score is the highest given moderately large $r = 150$ and $k = 40$. A larger window size w means more data points under consideration, with better accuracy in Figure 11 and higher time cost in Figure 12. Finally, a small enough slide size s has no much impact on accuracy but higher time cost.

Since our study focuses on improving the efficiency of distance-based outlier detection [7] under various parameter settings, we choose the parameters as in [9], [16], [29]. That is, the default parameters are set to detect around 1% outliers from the entire dataset. For example, given default $k = 50$, the query with $r = 90$ returns about 1% points in the entire dataset. A sufficiently large window size w and a moderate slide size s generally show stable and better results as in Figure 11.

R1O4: *The logical and physical level need to be separated clearly. Columnar storage and LSM trees are one possible implementation. Other choices, such as TimescaleDB, exist. Your work focuses on one concrete implementation of time series databases (LSM trees where the time ranges of files overlap). This is OK but the paper should focus on this issue since your solution does not apply to other implementations.*

Reply: In order to state more clearly the focus of this study on LSM-tree based store, rather than other implementations, we refine the paper title to 'Outlier Query Processing in LSM-Tree based Time Series Database'.

R1O5 (MR2O2): *An alternative to your implementation is to avoid LSM trees with overlapping time ranges. What would be the cost of such a solution? It also seems that compaction takes care of the overlapping time range problem. Is this true? What are the implication for your solution? Overall the sweet spot of your solution is too vague. Is it really necessary to detect outlier in multiple files with overlapping time ranges? What applications require this? Are there simple work-arounds? In section IV you write "It is worth noting that the single file case is also prevalent and important in LSM-tree storage". Please discuss the implications of this.*

Reply: To avoid LSM-trees with overlapping time ranges, an alternative implementation is to insert the out-of-order arrivals directly onto the disk. It obviously incurs huge cost of sorting and moving all the subsequent disk-resident data points, and thus it is unacceptable especially in IoT scenarios with extensive writes by millions of sensors [15]. For this reason, most commodity time series databases such as Apache IoTDB [10] and InfluxDB [11] employ LSM-tree to handle the heavy write workloads.

It is true that compaction takes care of the overlapping time range problem. Each compaction operation reads the files with overlapping time ranges, merges them as one file, and writes it back to disk. Owing to the overwhelming cost of compaction, it is often triggered periodically, e.g., once each day or week.

The implication of our solution is how to efficiently process outlier queries over the files not compacted yet with overlapping time ranges, as well as the compacted single files. For some applications like anomaly monitoring, it often queries the outliers in the past hours, where the data are not compacted yet and distributed in files with overlapping time ranges. For other analytical applications such as fault diagnosis, the query processes the historical data that have been compacted in single files. Thereby, it is also prevalent and important to process the single file case in the LSM-tree storage.

The simple work-arounds are to costly load all the data points, order them by their timestamps, and apply the existing methods such as CPOD [9] and NETS [16] that process data points in time order. These baselines are obviously inefficient as evaluated in the experiments in Section VII.

As suggested, we add the aforesaid discussion in Section I-B on Page 2.

R1O6 (MR2O4.c): *The bounds in Table III are difficult to understand and should be explained better. How many cases must be considered?*

Reply: To better explain the example bounds in Table III, we indicate the corresponding proposition where each bound is derived, and add more discussions on the example cases at the end of Section IV-C on Page 6. In short, Table III presents the bounds of the first 8 example cases by increasing r from 0 to 2γ. For larger r, greater than 2γ, the bounds are derived similarly referring to Propositions 3, 4 and 5.

R1O7: *The writing is quite rough and the paper should be proofread carefully.*

Reply: We improve writing and proofread the paper carefully, e.g., in Section I-A Page 1, Section II-B Page 3, etc.

To Reviewer 7

R7O1 (MR2O4.d): *Table I summarizes methods for streaming data, except the proposed method in this study. What is the key differences between streaming data and time series databases?*

Reply: The key differences between streaming data and time series databases are as follows. (1) The out-of-order arrivals with timestamps earlier than the currently processing window cannot be handled by the streaming methods. With the help of LSM-tree, all the out-of-order data points are stored in time series databases, and thus processed in outlier query. (2) The time series database can utilize some pre-computed statistics for efficient query processing, which are not considered in the streaming methods with online processing.

As suggested, we discuss these key differences in the end of Section II-A on Page 3.

R7O2 (MR2O4.e): *The definition of outliers is very simple [7], which may hinder the practical applications. (**MR2O4.e:** Please provide a better definition.)*

Reply: While the distance-based outlier definition [7] is simple, it also has many practical applications, such as anomaly detection in wearable devices [9], abnormal vital sign detection in healthcare [16], computer network attack detection [8], and so on. Nevertheless, our proposal can be adapted to more advanced definitions such as density-based outliers [28]. Compared to distance-based outliers, the density-based definition further distinguishes some points from outliers, which do not have sufficient neighbors but are close enough to some inliers, known as border points. Please see the adaption details in Section VI-A on Page 9.

As the experiments shown in Figure 11, the density-based definition has higher precision but lower recall than the distance-based one. The optimal F1 score of the advanced density-based definition is higher. Nevertheless, with the same density definition, our density-based LSMOD implementation has significantly lower time cost than the existing implementation of density-based DBSCAN [28] in Figure 12. The experiments are presented in Section VII-C1 on Page 12.

R7O3 (MR2O4.f): *Experimental study: the compared methods are for streaming data - Hence, it is kind of unfair comparison. BTW, would it be better querying single files for this comparison methods?*

Reply: To handle out-of-order data, we may first merge the files with overlapping time ranges and then apply the streaming data methods, which might be unfair in comparison. Following the suggestion, it is better to also compare these methods by querying single files (without merging). Figure 10 shows the results with 1 overlapping file, i.e., the case of single file. As shown, the baselines are still slower, since they need to load all the data points. Instead, our proposal utilizes the pre-computed statistics to further prune the data to load.

As suggested, we add more discussions on fair comparison at the end of Seciton VII-B2 on Page 11.

R7O4: *The proposed technique is coupled with LS-tree. This needs a further clarification.*

Reply: Thanks for the suggestion. For a better clarification on techniques coupled with LSM-Tree, we refine the paper title to 'Outlier Query Processing in LSM-Tree based Time Series Database'.

To Reviewer 8

R8O1 (MR2O3): *If the reviewer understands correctly, it seems this work only target univariate timeseries, that is, each object in the timeseries is one dimensional. This makes the scope of this work narrow. (**MR2O3:** Please clarify.)*

Reply: It is true that this work mainly focuses on univariate time series owing to the columnar storage, which also has many important applications such as filtering spikes of stock prices [1], identifying change points of temperature when cold air rushes in [3], etc. Nevertheless, in order to extend the scope of this work, the proposed techniques can be further applied to process multi-variate data, as illustrated in Section VI-B on Page 9. Proposition 12 ensures that our method always returns a subset of the outliers defined on multiple dimensions.

We evaluate the application on a multi-dimensional dataset in Section VII-C2, Page 12. In short, the precision of our method is higher but with lower recall, since it always returns a subset of outliers defined on multiple dimensions (Proposition 12). The overall F1 score is comparable as shown in Figure 13, whereas our time cost is significantly lower in Figure 14.

R8O2 (MR2O3): *Moreover, the authors target distance-based outlier, which is very basic and not necessarily popular in timeseries outlier detection. (**MR2O3:** Please provide better comparison with other approaches that are relevant to the paper.)*

Reply: While the distance-based outlier definition [7] is basic, it also has many practical applications, such as anomaly detection in wearable devices [9], abnormal vital sign detection in healthcare [16], computer network attack detection [8], and so on. Nevertheless, our proposal can be adapted to more advanced definitions such as density-based outliers [28]. Compared to distance-based outliers, the density-based definition further distinguishes some points from outliers, which do not have sufficient neighbors but are close enough to some inliers, known as border points. Please see the adaption details in Section VI-A on Page 9.

As the experiments shown in Figure 11, the density-based definition has higher precision but lower recall than the distance-based one. The optimal F1 score of the advanced density-based definition is higher. Nevertheless, with the same density definition, our density-based LSMOD implementation has significantly lower time cost than the existing implementation of density-based DBSCAN [28] in Figure 12. The experiments are presented in Section VII-C1 on Page 12.

R8O3: *The paper is unnecessarily over-formulated. There are a lot of theories and proofs which are in fact very straightforward. This makes the paper a little bit hard to read*

Reply: As suggested, we move some straightforward facts to Supplementary [20]. Moreover, we add more explanations, e.g., on the bounds of example cases in Table III derived by Propositions 3, 4 and 5 at the end of Section IV-C, Page 6.

APPENDIX

A. Proof of Proposition 3

Proof. For a point p in bucket $B[u]$, its r-neighbors must locate in $[p.v-r, p.v+r]$. Referring to Definition 8, the points in bucket $B[u]$ must fall into $[v_{\min}+(u-1)\gamma, v_{\min}+u\gamma)$, i.e., $v_{\min}+(u-1)\gamma \le p.v < v_{\min}+u\gamma$.

Since $(\lambda-\frac{1}{2})\gamma < r \le \lambda\gamma$, the following inequality can be derived,

$$v_{\min}+(u-\lambda-1)\gamma \le p.v-r < v_{\min}+(u-\lambda+\frac{1}{2})\gamma,$$
$$v_{\min}+(u+\lambda-\frac{3}{2})\gamma < p.v+r < v_{\min}+(u+\lambda)\gamma.$$

That is,

$$[p.v-r, p.v+r] \subseteq [v_{\min}+(u-\lambda-1)\gamma, v_{\min}+(u+\lambda)\gamma),$$

which corresponds to the union of the ranges of buckets $B[u-\lambda], B[u-\lambda+1], \ldots, B[u+\lambda]$, referring to Definition 8. Thus we derive the upper bound $|N(p,r)| \le \sum_{i=u-\lambda}^{u+\lambda} |B[i]|$. □

B. Proof of Proposition 4

Proof. Similar to the proof of Proposition 3, we also have $v_{\min}+(u-1)\gamma \le p.v < v_{\min}+u\gamma$. Since $(\lambda-1)\gamma < r \le (\lambda-\frac{1}{2})\gamma$, we have,

$$v_{\min}+(u-\lambda-\frac{1}{2})\gamma \le p.v-r < v_{\min}+(u-\lambda+1)\gamma.$$

Intuitively, with the constraint $(2\lambda-2)\gamma < 2r \le (2\lambda-1)\gamma$, $N(p,r)$ at most overlaps with 2λ consecutive buckets.

Thus, for the case with $v_{\min}+(u-\lambda-\frac{1}{2})\gamma \le p.v-r < v_{\min}+(u-\lambda)\gamma$, the r-neighbor of p overlaps with 2λ buckets $B[u-\lambda], B[u-\lambda+1]\ldots, B[u+\lambda-1]$, leading to

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

Otherwise, for the case with $v_{\min}+(u-\lambda)\gamma \le p.v-r < v_{\min}+(u-\lambda+1)\gamma$, the r-neighbor of p overlaps with 2λ buckets $B[u-\lambda+1], B[u-\lambda+2]\ldots, B[u+\lambda]$, leading to

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

Combining the above cases, the upper bound can be obtained as the maximum, i.e.,

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

□

C. Proof of Proposition 5

Proof. For a point p in bucket $B[u]$, its r-neighbors must locate in $[p.v-r, p.v+r]$. Referring to Definition 8, the points in bucket $B[u]$ must fall into $[v_{\min}+(u-1)\gamma, v_{\min}+u\gamma)$, and we thus have $v_{\min}+(u-1)\gamma \le p.v < v_{\min}+u\gamma$. Since $\ell\gamma \le r < (\ell+1)\gamma$, we can derive

$$v_{\min}+(u-\ell-2)\gamma < p.v-r < v_{\min}+(u-\ell)\gamma,$$
$$v_{\min}+(u+\ell-1)\gamma \le p.v+r < v_{\min}+(u+\ell+1)\gamma.$$

That is,

$$[p.v-r, p.v+r] \supseteq [v_{\min}+(u-\ell)\gamma, v_{\min}+(u+\ell-1)\gamma),$$

which leads to the lower bound $|N(p,r)| \ge \sum_{i=u-\ell+1}^{u+\ell-1} |B[i]|$. □

D. Proof of Proposition 6

Proof. Referring to the proofs of Propositions 3 and 5, for a point p in bucket $B[u]$, we have

$$[v_{\min}+(u-\ell)\gamma, v_{\min}+(u+\ell-1)\gamma) \subseteq [p.v-r, p.v+r] \subseteq [v_{\min}+(u-\lambda-1)\gamma, v_{\min}+(u+\lambda)\gamma),$$

Combining with Definitions 1 and 8, we can further derive

$$\bigcup_{i=u-\ell+1}^{u+\ell-1} B[i] \subseteq N(p,r) \subseteq \bigcup_{i=u-\lambda}^{u+\lambda} B[i].$$

With the lower bound obtained, we only need to further check the points contained in $\bigcup_{i=u-\lambda}^{u+\lambda} B[i]$ but not contained in $\bigcup_{i=u-\ell+1}^{u+\ell-1} B[i]$, that is $\bigcup_{i=u-\lambda}^{u-\ell} B[i] \cup \bigcup_{i=u+\ell}^{u+\lambda} B[i]$. In particular, since

$$\ell = \lfloor r/\gamma \rfloor \le \lceil r/\gamma \rceil = \lambda \le \lfloor r/\gamma \rfloor + 1 = \ell + 1,$$

we have $\lambda = \ell$ or $\ell+1$. For $\lambda = \ell+1$, only the points in 4 additional buckets $B[u-\lambda], B[u-\ell], B[u+\ell]$ and $B[u+\lambda]$ need to be checked. For $\lambda = \ell$, only the points in 2 additional buckets $B[u-\lambda], B[u+\lambda]$ need to be checked. □

E. Proof of Proposition 7

Proof. If all the points in $F_1, \ldots, F_\rho$ do not overwrite each other, then the bucket sizes can be simply aggregated when merging, i.e., the maximum size $\sum_{h=1}^{\rho} |B^{(h)}[u]|$. On the other hand, if all the points in $F_1, \ldots, F_{\rho-1}$ are overwritten by points in F_ρ, then the bucket size $|B[u]|$ should be the same as $|B^{(\rho)}[u]|$, the minimum size. □

F. Proof of Propositions 8, 9, 10

Proof. Propositions 8, 9, 10 can be easily derived by combining Propositions 3 and 7, Propositions 4 and 7, Propositions 5 and 7, respectively. □

G. Single File Query Algorithm

Now, we present the pseudo-code of querying outliers in a file via sliding window in Algorithm 2. Following the convention of query processing in data stream [7], we also consider the current sliding window W, with a number of points expired and some others new compared to the previous window. Thereby, Lines 7 and 9 update the window statistics for each bucket $B[u]$, referring to the aggregation property in Proposition 2 in Section IV-B.

The pruning of points in bucket $B[u]$ is then applied, i.e., cases 1 and 2 in Section IV-D. Specifically, we prune the entire bucket $B[u]$ as inliers in Line 10, according to the lower bound in Proposition 5 in Section IV-C2. Likewise, we directly output the points in bucket $B[u]$ as outliers in Lines 12 and 14, referring to Propositions 3 and 4, in Section IV-C1.

Otherwise, we need to load at most 4 additional buckets to determine the r-neighbors of points p in bucket $B[u]$, i.e., case 3 in Section IV-D. Lines 19 and 22 aggregate the r-neighbor count according to Proposition 6, to determine the outlier.

Algorithm 2 Outlier Detection Algorithm on Single File

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: $\lambda := \lceil r/\gamma \rceil$
2: $\ell := \lfloor r/\gamma \rfloor$
3: initialize outlier set $O_t := \emptyset$
4: initialize buckets $\mathscr{B}$ if algorithm runs for the first window
5: **for** each bucket $B[u]$, $u := 1$ to β **do**
6: **for** each expired segment S_g **do**
7: $|B[u]| := |B[u]| - |B_g[u]|$
8: **for** each new segment S_g **do**
9: $|B[u]| := |B[u]| + |B_g[u]|$
10: **if** $\sum_{i=u-\ell+1}^{u+\ell-1} |B[i]| \geq k$ **then**
11: continue
12: **else if** $\lceil r/\gamma \rceil - r/\gamma < \frac{1}{2}$ **and** $\sum_{i=u-\lambda}^{u+\lambda} |B[i]| < k$ **then**
13: $O_t := O_t \cup B[u]$
14: **else if** $\lceil r/\gamma \rceil - r/\gamma \geq \frac{1}{2}$ **and** $\max\left(\sum_{i=u-\lambda}^{u+\lambda-1} |B[i]|, \sum_{i=u-\lambda+1}^{u+\lambda} |B[i]|\right) < k$ **then**
15: $O_t := O_t \cup B[u]$
16: **else**
17: load additional buckets $B[u \pm \lambda], B[u \pm \ell]$
18: **for** each point p in $B[u]$ **do**
19: $cnt := \sum_{i=u-\ell+1}^{u+\ell-1} |B[i]|$
20: **for** each point p' in $B[u \pm \lambda] \cup B[u \pm \ell]$ **do**
21: **if** $dist(p, p') \leq r$ **then**
22: $cnt := cnt + 1$
23: **if** $cnt < k$ **then**
24: $O_t := O_t \cup \{p\}$
25: **return** O_t

Example 9 (Example 7 continued)**.** *Figure 15 shows the next sliding window of Figure 6, where a segment S_1 is expired and a new segment S_3 comes. The corresponding bucket statistics are updated for the new window $W_{11:00:10}$ starting from time 11:00:10. For instance, we have $|B[2]| = |B[2]| - |B_1[2]| + |B_3[2]| = 10 - 5 + 8 = 13$, where $|B_1[2]|$ and $|B_3[2]|$ are the 2nd bucket sizes in segments S_1 and S_3, respectively. Once all the bucket statistics are updated, similar to the previous window in Example 7, it checks whether the pruning is applicable. If not, i.e., case 3 in Section IV-D, the algorithm loads the additional buckets for evaluating r-neighbors.*

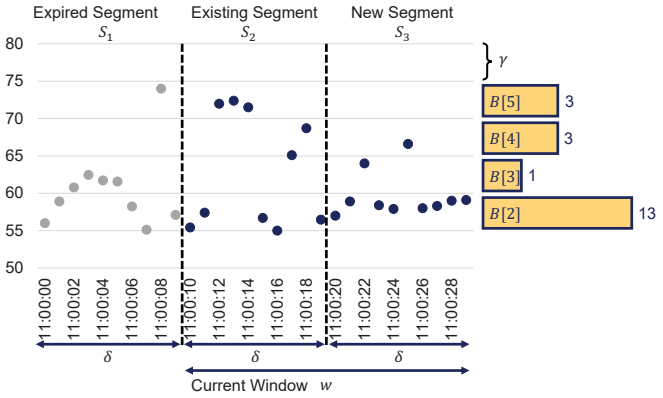

Fig. 15. Query in sliding window with bucket statistics

Complexity Analysis: We have β buckets as introduced in Definition 8. Consider all the ω segments in a window as in Definition 2. There are at most ω segments expired and ω segments new in the sliding window. The update of bucket statistics in Lines 7 and 9 thus takes $\mathscr{O}(\beta\omega)$ time. In the worst case, all the n points in the window may be output as outliers, referring to the upper bounds in Lines 12 and 14, in $\mathscr{O}(n)$ time. Otherwise, we need to load point p and check its r-neighbors. Luckily, we only need to load a constant number (up to 4) of additional buckets in Line 17. Referring to the average number of points in a bucket $\frac{n}{\beta}$, it takes $\mathscr{O}(\frac{n^2}{\beta})$ time. To sum up, Algorithm 2 runs in $\mathscr{O}(\beta\omega + \frac{n^2}{\beta})$ time, and $\mathscr{O}(\beta\zeta)$ extra space, where 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.

H. Supporting Various Queries

As introduced in Section I, outliers are often queried with different parameters for various interests. Same as the existing methods, our LSMOD also supports queries with different (1) neighbor distance threshold r, (2) neighbor count threshold k, (3) window size w, and (4) slide size s. We evaluate our proposal and other baselines under different query parameters, and present the results on the dataset WH-Chemistry in Figure 16. Owing to the limited space, similar results on other datasets can be found in the supplementary [20].

When neighbor distance threshold r increases in Figure 16(a), each data point has more neighbors in a window, and more inliers can be directly pruned by the index structures of NETS, CPOD and MCOD. Therefore, the time costs of all the baselines decrease. Similarly, for LSMOD, it may aggregate

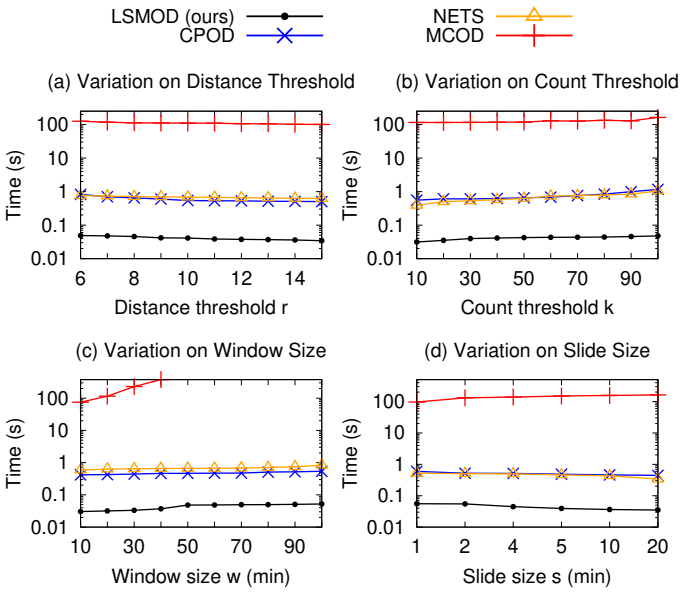

Fig. 16. Varying query parameters on WH-Chemistry

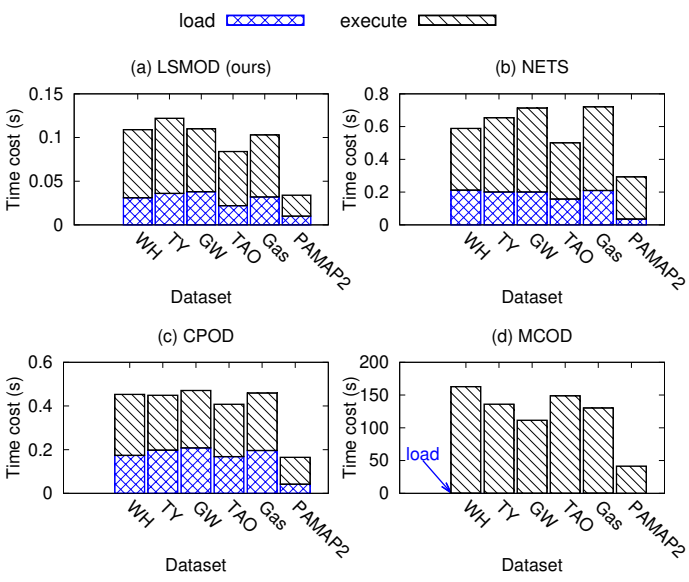

Fig. 17. Running time breakdown evaluation

more buckets with the increase of r, and the lower bounds for the neighbor count of each point become larger. Thus, more inliers can be pruned by comparing the lower bounds with the fixed k, also leading to the decrease of time cost.

When varying k in Figure 16(b), the baselines need to check more points to determine the point status no matter by which index structure, resulting in a slight increasing tendency. For our LSMOD, when k is small, inliers are easy to be determined by Proposition 5. However, fewer inliers can be pruned with the increase of k, and the time cost thus increases as well.

In Figure 16(c), with window size w increasing, the time cost of MCOD increases rapidly, since the points and micro-clusters in a window become more time-costly to maintain. For others, their time costs slightly increase due to the more costly initialization phases, while our LSMOD remains 1 order of magnitude faster than other baselines.

When varying slide size s in Figure 16(d), NETS, CPOD and our LSMOD have lower time costs under large slide sizes. This is because a larger slide size s means fewer windows to check, for a time series with fixed length. However, MCOD takes more time to maintain its micro-clusters, with more points expired and arrived for each slide.

I. Running Time Breakdown

We conduct breakdown evaluation for running time in Figure 17 to show whether LSM-tree designs impede the streaming methods. For LSMOD, we measure the time cost for loading bucket statistics, outliers and suspicious points from disk, taking about 20% of the total time. For the baselines, they use the same series readers as aforesaid, and thus they all have almost the same time costs of loading data from the LSM-tree store. It takes about 20%, 40% and 0.1% of their total time costs, for NETS, CPOD and MCOD, respectively. That is, the execution time costs of the baselines make up the major part of their overall time costs, much higher than the running time of LSMOD. In other words, the baselines are still more time-consuming than our proposal even if not considering the loading costs in LSM-tree storage, i.e., LSM-tree designs are not the bottleneck that impedes the streaming methods.

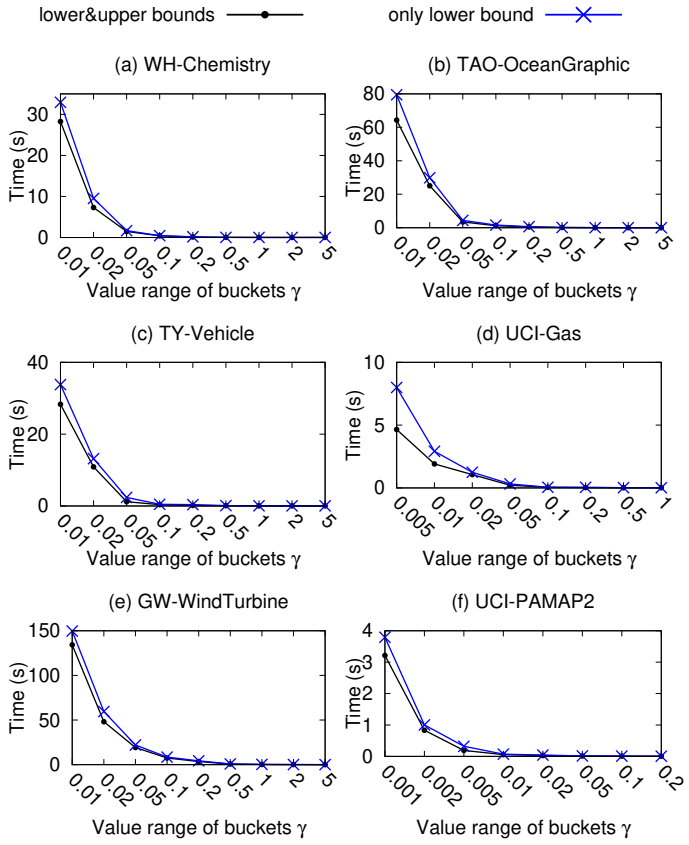

Fig. 18. Varying the width of buckets on value γ

J. Evaluation of Proposed Techniques

Recall that time series T is stored in files F_h and segments S_g with buckets $B[u]$, as illustrated in Figure 4. Therefore, in addition to the scalability in data size in Figure 9, we also evaluate how the numbers of files, segments and buckets affect the performance of LSMOD, by varying the number ρ of overlapping files, the time range δ of splitting segments and the value width γ of buckets, in Figures 10, 19 and 18, respectively. Ablation study is also presented to compare the proposed LSMOD with both upper and lower bounds, and with only lower bound.

1) Varying the Time Range of Segments δ: Figure 19 varies the time range δ for splitting time series into segments in Definition 4. Under the same data sizes, a larger time range δ leads to fewer segments, and thus the processing time cost decreases. Recall that to support query processing in sliding window, the window size w and slide s are usually the integral multiples of δ, as discussed in Section III-B. A larger δ thus means less supported queries.

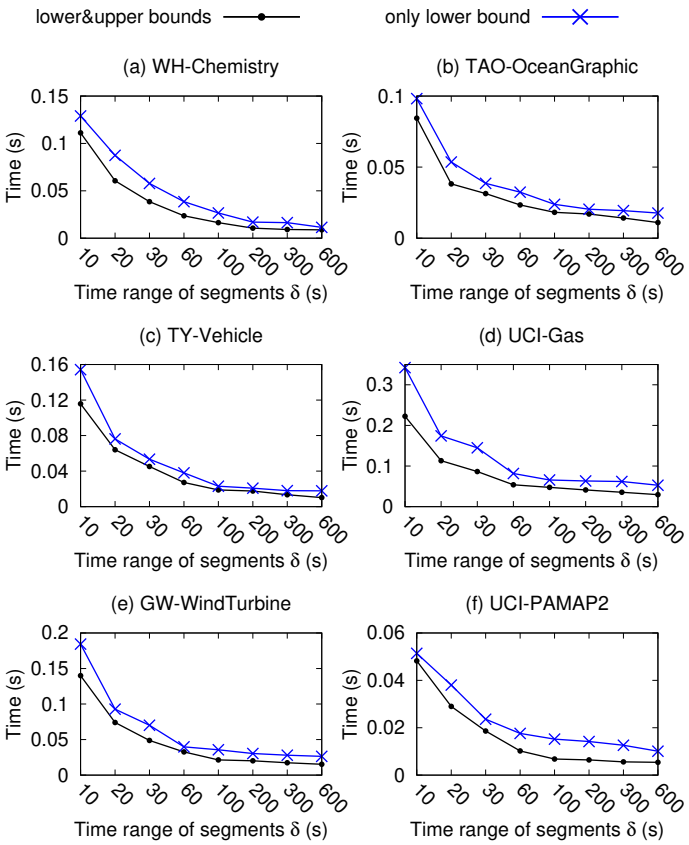

Fig. 19. Varying the range of segments on time δ

2) Varying the Value Width of Buckets γ: Figure 18 varies the bucket width γ of value range. Again, under the same data sizes, a larger γ leads to less buckets, and thus lower time cost. As also discussed at the end of Section IV-C, to enable the lower bound for pruning, we prefer to set a bucket width $\gamma \leq r$ in practice, and a larger γ means less supported queries. However, if γ is set very small, there will be a large number of buckets, even more than the number of points in a window, leading to fairly high time cost. In practice, a smaller γ supports more queries, while a larger γ decreases the query time cost, again a trade-off.

K. Evaluation on Supporting Various Queries

As introduced in Section I, outliers are often queried with different parameters for various interests. Same as the existing methods, our LSMOD also supports queries with different (1) neighbor distance threshold r, (2) neighbor count threshold k, (3) window size w, and (4) slide size s. We evaluate our proposal and other baselines under different query parameters. Figures 20, 21, 22, 23 present the evaluation on 6 datasets under different (1) neighbor distance threshold r, (2) neighbor count threshold k, (3) window size w, and (4) slide size s, respectively.

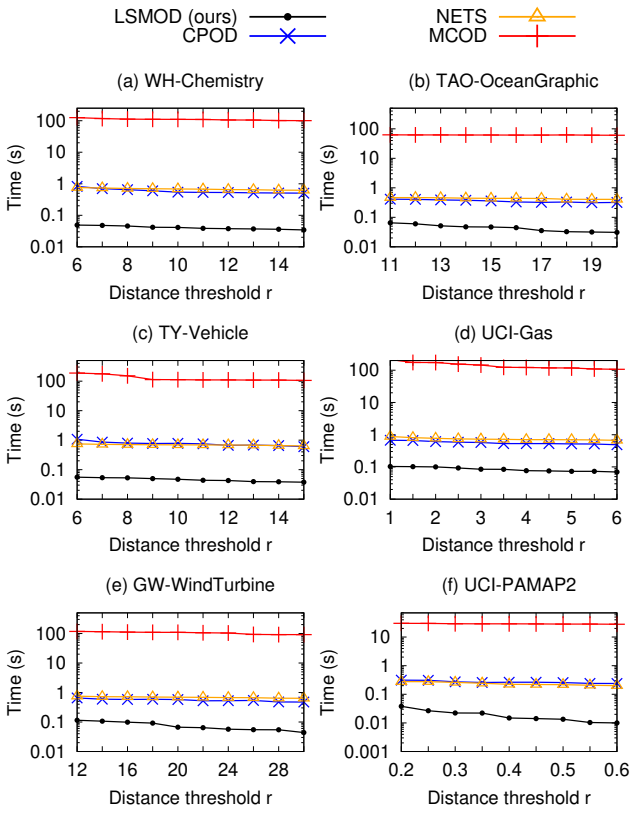

Fig. 20. Varying neighbor distance threshold r of query

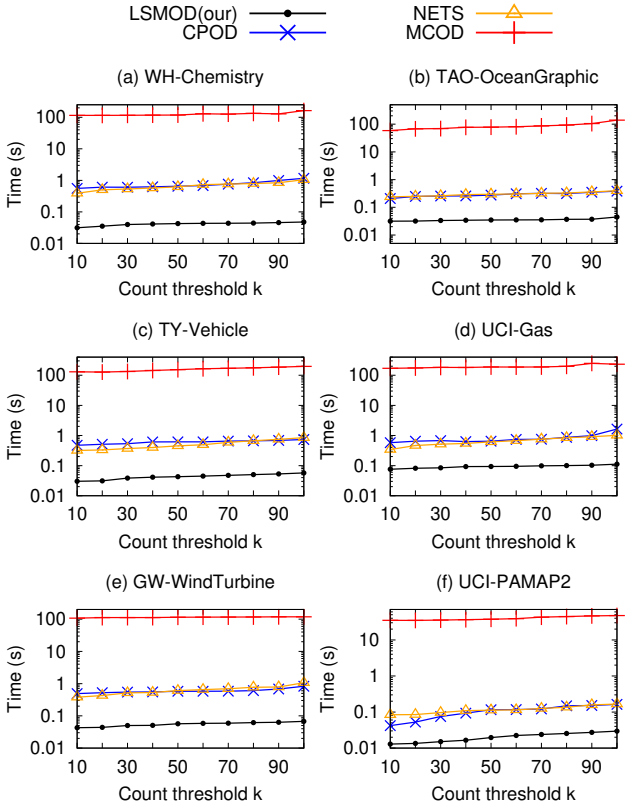

Fig. 21. Varying neighbor count threshold k of query

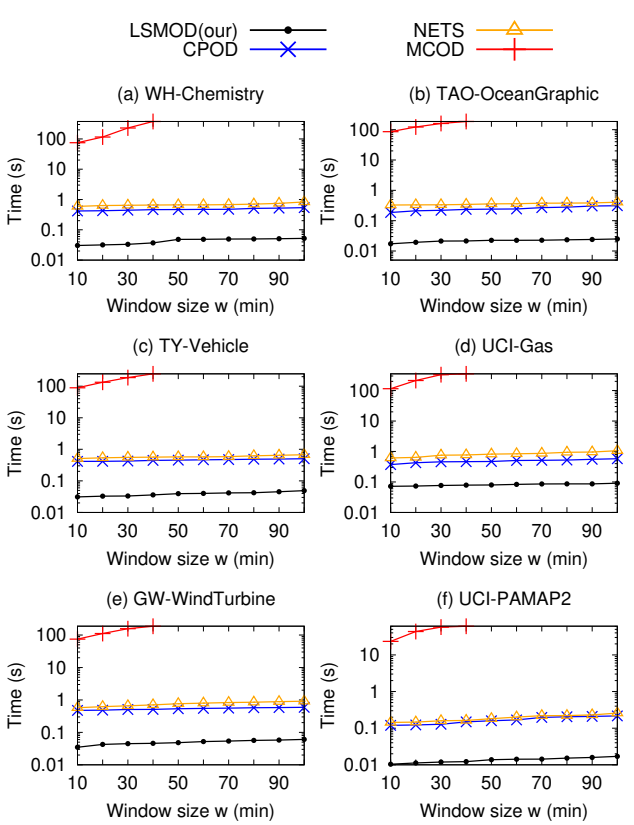

Fig. 22. Varying window size w of query

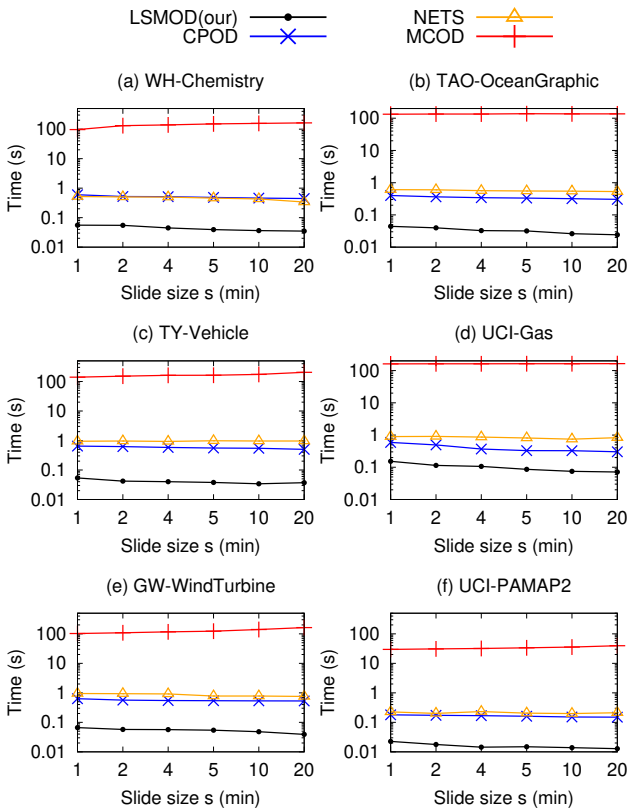

Fig. 23. Varying window slide s of query