Time Series Representation for Visualization in Apache IoTDB

Lei Rui
Tsinghua University
rl18@mails.tsinghua.edu.cn

Xiangdong Huang
Timecho Ltd
hxd@timecho.com

Shaoxu Song
BNRist, Tsinghua University
sxsong@tsinghua.edu.cn

Yuyuan Kang
University of Wisconsin-Madison
yuyuan@cs.wisc.edu

Chen Wang
Timecho Ltd
wangchen@timecho.com

Jianmin Wang
BNRist, Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

When analyzing time series data, often interactively, the analysts frequently demand to visualize instantly a large-scale series. While M4 representation already shows its preciseness of encasing time series in different scales into a fixed size of pixels, how to efficiently support M4 representation in a time series database is still absent. It is worth noting that, to enable fast writes, the commodity time series database systems, such as Apache IoTDB or InfluxDB, employ LSM-based storage. That is, a time series is segmented and stored in a number of chunks, with possibly overlapping time intervals. To implement M4, a natural idea is to merge online the chunks as a whole series, often costly, and then perform M4 representation as in relational databases. In this study, we propose to utilize the metadata of chunks to accelerate M4 representation. In particular, a chunk merge free approach is devised to avoid the costly merging of any chunk. The time series database native operator M4-LSM is deployed in Apache IoTDB, an open-source time series database. Remarkably, in the experiments over real-world datasets, the proposed M4-LSM operator demonstrates high efficiency without sacrificing preciseness. It takes about 700ms to represent a time series of 10 million points in 1000 pixel columns, i.e., enabling instant visualization of the data in four months with a data collection frequency of every second.

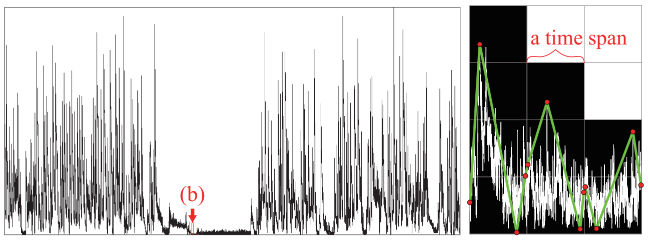
ACM Reference Format:
Lei Rui, Xiangdong Huang, Shaoxu Song, Yuyuan Kang, Chen Wang, and Jianmin Wang. 2023. Time Series Representation for Visualization in Apache IoTDB. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Time series representation is a typical data mining task [16] that reduces the dimensionality while still retaining its essential characteristics such as shape. M4 representation [25] is known as an error-free method for visualizing time series in two-color (binary) line chart. It encases time series in various scales into fixed-size pixels. For instance, Figure 1(a) shows the line chart of 1.2 million data points time series in 1000 × 500 pixels. The time series is divided into 1000 time spans, corresponding to 1000 pixel columns. Figure 1(b) illustrates 3 time spans (pixel columns) out of 1000. For each time span, M4 selects the first, last, bottom and top data points, denoted by red dots in Figure 1(b). Pixels covered by the connecting lines of consecutive selected points are colored in black for line chart visualization.

Note that M4 is originally designed for visualizing time series data stored in RDBMS, reading data points ordered by time. Different from relational databases, to enable fast writes, the commodity time series database systems, such as Apache IoTDB [5] or InfluxDB [8], employ Log-Structured Merge-Tree (LSM-Tree) [35] based storage. That is, time series is segmented and stored in a number of chunks, denoted by rectangles in Figure 2(a). As shown, the data points in the same time period may be stored in different chunks, owing to out-of-order arrivals [26]. Consequently, the data points read from chunks may not be in the order of time either. How to efficiently support M4 representation in such time series native LSM-based databases is still absent.

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



(a) All 1000 pixel columns (b) 3 pixel columns

Figure 1: M4 representation for time series visualization

1.1 M4-LSM Approach

A straightforward idea is to first merge online all the chunks as a whole series, and apply M4 representation over the data points ordered by time as in RDBMS [25]. This baseline method, implemented as UDF in time series database namely M4-UDF, is costly, by loading all chunks from disk, ordering data points by time, and scanning the entire time series, as in Figure 2(b).

In this study, we propose a novel chunk merge free approach M4-LSM to accelerate M4 representation. We show that the metadata of chunks can be utilized to avoid loading and merging chunks,

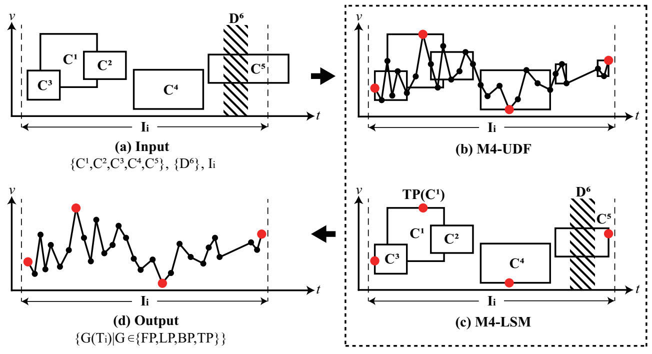


Figure 2: Two M4 implementations (b)/(c) in LSM-based store

thus accelerating the M4 representation. Intuitively, if the candidate points for the first, last, bottom and top representations obtained from metadata are neither updated by other chunks nor deleted by delete operations, we can directly return them as the results, in Figure 2(c), i.e., there is no need to load and merge any chunk. Nevertheless, for those chunks that need to access the raw data points, we observe the regular intervals of timestamps and introduce a step regression for efficient indexing.

1.2 System Deployment and Evaluation

The LSM database native M4 operator has been implemented and deployed in Apache IoTDB [5], an open-source time series database developed based on our preliminary study [39]. The document of the M4 function is available on the product website [6]. The source code has been committed to the GitHub repository of Apache IoTDB [3]. The code and data related to the experiments are available in [4] to reproduce.

The extensive experiments over real-world datasets demonstrate the efficiency of the chunk merge free operator M4-LSM. Remarkably, the time cost is about 700ms to represent a time series of 10 million points in 1000 pixel columns, i.e., enabling instant visualization of the data in four months with a data collection frequency of every second.

2 PRELIMINARIES

We first introduce the M4 representation in Section 2.1, then present LSM-based storage of time series in Section 2.2, and finally define M4 representation on LSM-based storage in Section 2.3.

2.1 M4 Representation

Let $T = \{P_1, \dots P_n\} = \{(t_1, v_1), \dots, (t_n, v_n)\}$ denote a time series with data points (time-value pairs) in the increasing order of time [23], where (t_i, v_i) is the time-value pair of the i-th point P_i. Following the same line of [24, 25], we introduce M4 representation.

Definition 2.1 (M4 representation functions). Given a time series T, the M4 representation functions are as follows.

(1) *FirstPoint* representation function, denoted as $FP : T \to P$, returns the point with the minimal time, i.e., $P_{first} \in \{P^* \in T \mid P.t \geq P^*.t, \forall P \in T\}$.

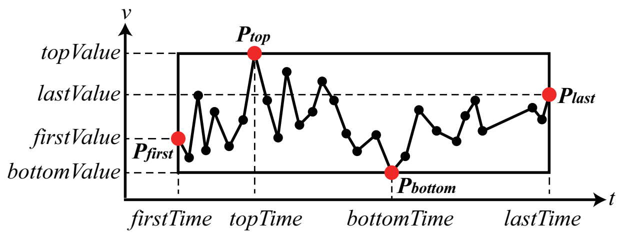


Figure 3: An example of four representation functions

(2) *LastPoint* representation function, denoted as $LP : T \to P$, returns the point with the maximal time, i.e., $P_{last} \in \{P^* \in T \mid P.t \leq P^*.t, \forall P \in T\}$.
(3) *BottomPoint* representation function, denoted as $BP : T \to P$, returns any one of the points with the minimal value, i.e., $P_{bottom} \in \{P^* \in T \mid P.v \geq P^*.v, \forall P \in T\}$.
(4) *TopPoint* representation function, denoted as $TP : T \to P$, returns any one of the points with the maximal value, i.e., $P_{top} \in \{P^* \in T \mid P.v \leq P^*.v, \forall P \in T\}$.

Note that BP (or TP) can return any of the points with the minimal (or maximal) value from the visualization-driven perspective. According to [25], the inter-column pixels are determined by both the times and values of the first and last points, while the inner-column pixels only rely on the values of the bottom and top points. In this sense, any point with the minimal (or maximal) value may sever as BP (or TP).

Example 2.2. Given a time series T as shown in Figure 3, the four representation functions $FP(T)$, $LP(T)$, $BP(T)$ and $TP(T)$ return four representation points $P_{first} = (firstTime, firstValue)$, $P_{last} = (lastTime, lastValue)$, $P_{bottom} = (bottomTime, bottomValue)$ and $P_{top} = (topTime, topValue)$, respectively, marked with bold red dots. The minimal bounding rectangle of T is also plotted in the figure. Note that the four representation points contain more information than the minimal bounding rectangle (i.e., *firstValue* and *lastValue*).

Below, we use $G \in \{FP, LP, BP, TP\}$ as a general notation of the four representation functions.

Definition 2.3 (M4 representation query). Given a time series T, query time range $[t_{qs}, t_{qe})$, and the number of time spans w, the *M4 representation query* uses the derived time spans

$$I_i = [t_{qs} + \frac{t_{qe} - t_{qs}}{w} * (i-1), t_{qs} + \frac{t_{qe} - t_{qs}}{w} * i), \ i = 1, \dots, w$$

to group time series T into w time series subsequences

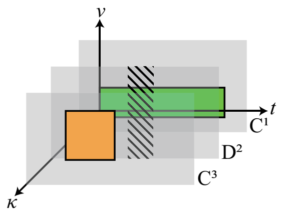
$$T_i = \{P \mid P \in T, P.t \in I_i\}, \ i = 1, \dots, w$$

and apply the four representation functions on each subsequence

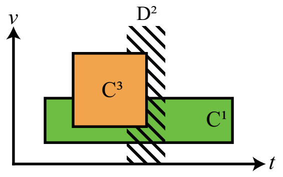
$$\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}, \ i = 1, \dots, w. \quad (1)$$

2.2 LSM-based Storage of Time Series

To enable fast writes, each time series is stored as a set of chunks (containing inserts and append-only updates) together with a set of deletes (containing append-only deletes), in an LSM-based storage system. That is, the time series data are not readily available without applying such updates and deletes.



(a) Three-dimensional



(b) Two-dimensional

Figure 4: Schematic diagrams of chunks and deletes

2.2.1 *Elements of LSM-based Storage.*

A version number κ is a global incremental number assigned to each *chunk* or *delete* to distinguish the append order of updates and deletes. The larger the κ is, the later the operation applies.

Definition 2.4 (Chunk). A *chunk* is a segment of time series that is read-only on disk, denoted by C^κ, where κ is the version number.

When the memory component of the LSM-tree meets the flush trigger condition (such as reaching a threshold size), the time series in memory is flushed to a new location on disk, i.e., a chunk. Each chunk maintains its own metadata, i.e.,

$$\{G(C^\kappa) \mid G \in \{FP, LP, BP, TP\}\}.$$

We say a timestamp t is covered by a chunk C^κ, denoted by $t \vDash C^\kappa$, if there exists a point $\exists P \in C^\kappa$ such that $t = P.t$.

Definition 2.5 (Delete). A *delete* D^κ specifies a time range to delete in the time series, where κ is the version number.

By default, we denote $[t_{ds}, t_{de}]$ as the time range of delete D^κ, where t_{ds} and t_{de} are the left and right endpoints of the range, respectively. We say a timestamp t is covered by a delete D^κ, denoted by $t \vDash D^\kappa$, if $t_{ds} \le t \le t_{de}$.

Example 2.6. Figure 4 gives two schematic diagrams to help better understand the relationship of chunks and deletes. Figure 4(a) shows a three-dimensional space composed of time, value and version number. A version plane is plotted as a gray translucent rectangle, corresponding to a unique version number. Each chunk or delete is drawn on its own version plane, where C^1 and C^3 are represented by their minimum bounding rectangles and D^2 is represented by the slashed region covering its delete time range. Figure 4(b) collapses the version dimension, stacking chunks and deletes in the two-dimensional time-value space. From the version numbers of C^1, D^2, C^3 and the relationship between their time ranges, we know that (1) C^1, D^2 and C^3 are flushed to disk in turn; (2) C^3 might contain some updates to C^1; and (3) the delete D^2 only works on C^1 but not C^3, because C^3 has a larger version number than D^2.

2.2.2 *Merge Function.*

To get the time series with only the latest points, we formally define the merge function as follows.

Definition 2.7 (Merge function). Given a set of chunks $\mathbb{C}$ and a set of deletes $\mathbb{D}$, the merge function $M(\mathbb{C}, \mathbb{D})$ returns the time series

$$M(\mathbb{C},\mathbb{D}) = \{P \in C^\kappa \mid P.t \nvDash C^{\kappa_1}, P.t \nvDash D^{\kappa_2}, \kappa < \kappa_1, \kappa < \kappa_2, \quad C^\kappa \in \mathbb{C}, C^{\kappa_1} \in \mathbb{C} \cup \{C^\infty\}, D^{\kappa_2} \in \mathbb{D} \cup \{D^\infty\}\} \tag{2}$$

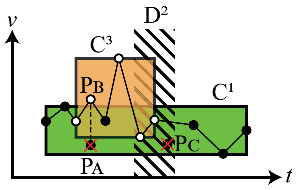


Figure 5: An example of the merge function. Point P_A in C^1 is updated by point P_B in C^3, and point P_C is deleted by D^2.

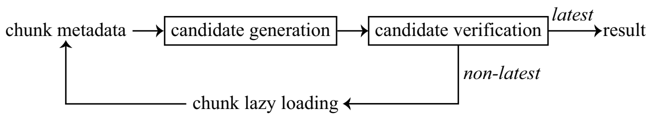


Figure 6: Framework of the M4-LSM approach

where C^∞ is an empty chunk with the largest version number, and D^∞ is an empty delete ($t_{ds} = t_{de}$) with the largest version number.

That is, the point P in the merged time series is from some chunk $C^\kappa \in \mathbb{C}$ such that $P.t$ is not covered by any $C^{\kappa_1} \in \mathbb{C} \cup \{C^\infty\}$ or $D^{\kappa_2} \in \mathbb{D} \cup \{D^\infty\}$ whose version number is higher than κ.

Example 2.8. Figure 4(b) is detailed in Figure 5, with points from C^1 drawn as black dots and points from C^3 drawn as white dots. We can observe that (1) the point P_A from C^1 is non-latest because it is updated by P_B from C^3 (in other words, P_B overwrites P_A); (2) the point P_C from C^1 is non-latest because it is deleted by D^2; and (3) the remaining 11 latest points make up the final time series. Therefore, given $\mathbb{C} = \{C^1, C^3\}$ and $\mathbb{D} = \{D^2\}$, $T = M(\mathbb{C},\mathbb{D})$ returns the time series composed of the 11 latest points (without red cross).

2.3 M4 Representation on LSM-based Storage

With the M4 representation query on time series and the LSM-based storage of time series introduced above, we now combine them to give the formal definition of the problem of performing M4 representation on LSM-based storage.

Definition 2.9 (M4 representation on LSM-based storage). For a time series T with a set of chunks $\mathbb{C}$ and a set of deletes $\mathbb{D}$, given the query parameters of time range t_{qs}, t_{qe} and the number of time spans w, the problem is to compute $\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}$, $i = 1, \ldots, w$, where

$$T_i = \{P \mid P \in T, P.t \in I_i\}, \; T = M(\mathbb{C},\mathbb{D}),$$
$$I_i = [t_{qs} + \frac{t_{qe} - t_{qs}}{w} * (i-1), t_{qs} + \frac{t_{qe} - t_{qs}}{w} * i).$$

It is worth noting that loading chunks from disk and merging them are costly, not only for the heavy cost of I/O but also for the decompression of data [40]. Therefore, in the following, we devise novel techniques to avoid loading and merging chunks for M4 representation over LSM storage.

3 M4-LSM APPROACH

In this section, we present M4-LSM, an efficient approach to perform M4 representation on LSM-based storage. As shown in Figure 2, given the set of chunks $\mathbb{C}$, the set of deletes $\mathbb{D}$ and the M4 time span I_i as input, M4-LSM outputs $\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}$ as defined in Formula (1).

3.1 Overview

The key observation of M4-LSM is that we do not need to load and merge chunks if the representation point can simply be retrieved from the chunk metadata. For example, in Figure 2(c), the representation result of $TP(T_i)$ is $TP(C^1)$, because $TP(C^1)$ has the maximum value and $TP(C^1)$ is the latest, i.e., neither deleted nor updated.

To begin with, M4-LSM merges the M4 time span as virtual deletes. As defined in Definition 2.3, I_i is the i-th time span used to divide time series in the M4 representation query. For a time series subsequence T_i, I_i actually functions as a delete ruling out points that fall outside I_i. Therefore, we transform I_i into two virtual deletes $\{D_1^\infty, D_2^\infty\}$:

$$D_1^\infty = (-\infty, t_{qs} + \frac{t_{qe} - t_{qs}}{w} * (i-1)), D_2^\infty = [t_{qs} + \frac{t_{qe} - t_{qs}}{w} * i, +\infty).$$

These two delete time ranges form the complement of I_i. Also note that the version number is infinity, larger than that of any chunk or delete. M4-LSM thus deals with the problem of computing

$$\{G(M(\mathbb{C}'', \mathbb{D})) \mid G \in \{FP, LP, BP, TP\}\},$$

where

$$T_i = M(\mathbb{C}'', \mathbb{D}), \quad \mathbb{C}'' = \bigcup_{C^\kappa \in \mathbb{C}} M(\{C^\kappa\}, \{D_1^\infty, D_2^\infty\}).$$

The framework of M4-LSM is shown in Figure 6. To compute $G(M(\mathbb{C}'', \mathbb{D}))$ for representation function G, M4-LSM first generates the candidate point from the precomputed chunk metadata which is introduced in Section 2.2.1. Next, M4-LSM performs the candidate verification to check whether the candidate point is the latest or not. If it is the latest, M4-LSM outputs it as the representation result; otherwise, M4-LSM applies a lazy loading strategy to load chunks and update chunk metadata, preparing for the next iteration of the candidate generation and verification. It is worth noting that updates of sensor reading values rarely occur in IoT scenarios. That is, most data points should be the latest, and the iteration is expected to terminate shortly.

The candidate generation is described in Section 3.2. The candidate verification for *FirstPoint*/*LastPoint* and *BottomPoint*/*TopPoint* representation functions are presented in Section 3.3 and Section 3.4 respectively, as they require different candidate verification rules and chunk loading strategies. The chunk index for accelerating data read operations is introduced in Section 3.5.

3.2 Candidate Generation

The first step of M4-LSM is to retrieve the *candidate point* P_G for the representation function G from the metadata of all chunks in $\mathbb{C}''$. For each G, let the points suggested by the metadata of all chunks in $\mathbb{C}''$ be

$$\mathbb{P}_G = \{G(C^\kappa) \mid C^\kappa \in \mathbb{C}''\}, G \in \{FP, LP, BP, TP\}.$$

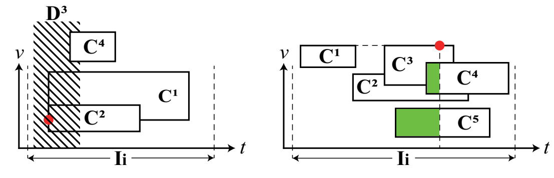


Figure 7: Candidate verification example

Among them, we need to find those points satisfying the representation condition, i.e.,

$$\mathbb{P}'_G = \{P^* \in \mathbb{P}_G \mid P^*.t \le P.t, \forall P \in \mathbb{P}_G\}, G = FP$$
$$\mathbb{P}'_G = \{P^* \in \mathbb{P}_G \mid P^*.t \ge P.t, \forall P \in \mathbb{P}_G\}, G = LP$$
$$\mathbb{P}'_G = \{P^* \in \mathbb{P}_G \mid P^*.v \le P.v, \forall P \in \mathbb{P}_G\}, G = BP$$
$$\mathbb{P}'_G = \{P^* \in \mathbb{P}_G \mid P^*.v \ge P.v, \forall P \in \mathbb{P}_G\}, G = TP$$

Finally, the point with the largest version number in $\mathbb{P}'_G$ is the *candidate point* P_G. Formally,

$$P_G = \arg\max_{P \in \mathbb{P}'_G} P.\kappa,$$

where $P.\kappa$ is the version number of chunk C^κ that P belongs to. To sum up, the candidate point is the one with the largest version number from the metadata satisfying the representation condition.

3.3 *FirstPoint*/*LastPoint* Candidate Verification

We now verify whether the candidate point P_G is valid as the result of $G(M(\mathbb{C}'', \mathbb{D}))$ for $G \in \{FP, LP\}$.

Proposition 3.1 (latest candidate point for *FP*/*LP*). *For* $G \in \{FP, LP\}$*, if* P_G *is not covered by the delete time range of any delete* D^κ *in* $\mathbb{D}$ *with a larger version number* κ *than* $P_G.\kappa$*, i.e.,*

$$\bigwedge_{D^\kappa \in \mathbb{D} \wedge \kappa > P_G.\kappa} P_G.t \not\models D^\kappa,$$

then the candidate point P_G *is the latest, and the result of* $G(M(\mathbb{C}'', \mathbb{D}))$.

Lazy Load. When P_G is verified to be non-latest, i.e., overlapped in time by some later appended deletes, M4-LSM does not eagerly load the chunk C^κ to which P_G belongs and recalculate its metadata under deletes. Instead, it updates $FP(C^\kappa).t = t_{de}$ or $LP(C^\kappa).t = t_{ds}$ by the delete time range $[t_{ds}, t_{de}]$. The updated time interval of C^κ, $[FP(C^\kappa).t, LP(C^\kappa).t]$, might not be tight but can be used to prune C^κ from being loaded. For example, if any other chunk has its first point earlier than $FP(C^\kappa).t$, then C^κ does not need to be loaded thus far. If no such chunks exist, the load of C^κ happens in the next iteration of candidate generation.

Example 3.2. Take Figure 7(a) as an example, where $G = FP$, $\mathbb{C}'' = \{C^1, C^2, C^4\}$ and $\mathbb{D} = \{D^3\}$. Firstly, M4-LSM retrieves the candidate point $P_G = FP(C^2)$ (denoted by the red dot) from $\mathbb{P}'_G = \{FP(C^1), FP(C^2)\}$. Next, M4-LSM verifies P_G as non-latest because P_G is covered by D^3. With the lazy loading strategy, M4-LSM updates the time interval of C^2 to be $[D^3.t_{de}, LP(C^2).t]$ without eagerly loading the chunk data. Likewise, the time interval of C^1 is updated as $[D^3.t_{de}, LP(C^1).t]$. The next iteration of candidate generation and verification starts by finding $FP(C^4)$ as the new candidate point and ends by outputting the verified latest $FP(C^4)$ as the representation result, without loading C^1 and C^2.

3.4 *BottomPoint*/*TopPoint* Candidate Verification

Next, we verify whether the candidate point P_G is valid as the result of $G(M(\mathbb{C}'', \mathbb{D}))$ for $G \in \{BP, TP\}$. Note that *FP*/*LP* can be verified by only checking the deletes in Proposition 3.1. The reason is that for *FP*/*LP*, all candidates in $\mathbb{P}'_G$ are at the same time, and thus the candidate point P_G with the largest version number from $\mathbb{P}'_G$ will never be updated. However, this is not the case for *BP*/*TP* candidate verification. In addition to deletes, we need to further consider whether *BP*/*TP* candidates are updated by other chunks.

Proposition 3.3 (Latest candidate point for *BP*/*TP*). *For $G \in \{BP, TP\}$, if no other chunk in $\mathbb{C}''$ with a version number larger than $P_G.\kappa$ contains a point with the same time as P_G and P_G is not covered by the delete time range of any delete in $\mathbb{D}$ with a larger version number than $P_G.\kappa$, i.e.,*

$$\Big(\bigwedge_{C^\kappa \in \mathbb{C}'' \wedge \kappa > P_G.\kappa} P_G.t \nvDash C^\kappa\Big) \wedge \Big(\bigwedge_{D^\kappa \in \mathbb{D} \wedge \kappa > P_G.\kappa} P_G.t \nvDash D^\kappa\Big),$$

then the candidate point P_G is the latest, and the result of $G(M(\mathbb{C}'', \mathbb{D}))$.

To verify whether the candidate P_G is the latest, M4-LSM first checks whether the time of P_G overlaps with the later appended chunks or deletes (i.e., chunks and deletes with larger version numbers than $P_G.\kappa$). Referring to Proposition 3.3, there are three cases to consider. (1) If P_G is not in the time interval of any later appended chunks or deletes, i.e.,

$$\Big(\bigwedge_{C^\kappa \in \mathbb{C}'' \wedge \kappa > P_G.\kappa} P_G.t \notin [FP(C^\kappa).t, LP(C^\kappa).t]\Big)$$
$$\wedge \Big(\bigwedge_{D^\kappa \in \mathbb{D} \wedge \kappa > P_G.\kappa} P_G.t \nvDash D^\kappa\Big),$$

then P_G is the latest and M4-LSM finishes the computation by returning P_G. (2) If P_G is indeed in the time range of some later appended deletes, similar to the *FP*/*LP* verification in Section 3.3, P_G is non-latest as it is deleted. (3) If P_G is in the time interval of some later appended chunks, then P_G *might* be the latest and needs further verification. The reason is that within the chunk time interval does not necessarily mean the point is overwritten (i.e., updated). That is, $P_G.t \in [FP(C^\kappa).t, LP(C^\kappa).t]$ does not necessarily mean $P_G.t \vDash C^\kappa$.

Lazy Load. If P_G is found non-latest owing to some later appended deletes or updates, the corresponding chunk does not need to be loaded eagerly, as we can further verify the remaining points in $\mathbb{P}'_G \setminus \{P_G\}$ for *BP*/*TP*. M4-LSM iterates this verification process until a candidate point P_G is verified to be the latest, or all points in $\mathbb{P}'_G$ are non-latest. In the latter case, M4-LSM loads all the corresponding chunks to which the points in $\mathbb{P}'_G$ belong and recalculates their metadata under deletes or updates. Afterwards, M4-LSM starts the next new iteration of generating and verifying candidate points, as described in Sections 3.2 and 3.4.

Example 3.4. Take Figure 7(b) as an example, where $G = TP$, $\mathbb{C}'' = \{C^1, C^2, C^3, C^4, C^5\}$ and $\mathbb{D} = \emptyset$. M4-LSM retrieves the candidate point $\mathbb{P}_G = TP(C^3)$ (denoted by the red dot) from $\mathbb{P}'_G = \{TP(C^1), TP(C^3)\}$. M4-LSM then partially scans the overlapping chunks (i.e., C^4 and C^5) to check whether they contain any point that overwrites $\mathbb{P}_G$. The scanned data are marked in green, indicating that there is no need to scan the time greater than $\mathbb{P}_G.t$. Assume that the partial scan of C^4 and C^5 does find a point that overwrites the current candidate point $P_G = TP(C^3)$. With the lazy loading strategy, M4-LSM considers the remaining points in $\mathbb{P}'_G = \{TP(C^1), TP(C^3)\}$ except the non-latest $TP(C^3)$, and assigns $TP(C^1)$ as the new candidate point for verification. Because $TP(C^1)$ is the latest, M4-LSM outputs $TP(C^1)$ as the result of $TP(M(\mathbb{C}'', \mathbb{D}))$.

3.5 Chunk Index with Step Regression

As summarized in Table 1, M4-LSM employs three types of chunk data read operations. First of all, in the phase of candidate verification for *FP*/*LP*, no data read operation is needed, referring to Proposition 3.1. (a) To verify the candidate point for *BP*/*TP*, M4-LSM needs to read the relevant chunks to check if there exists any data point that overwrites the candidate point. In the phase of candidate generation, (b) to recalculate *FP*/*LP* under deletes for the chunk where the non-latest candidate *FP*/*LP* belongs, M4-LSM needs to get the closet data point after/before the delete boundary. (c) To recalculate *BP*/*TP* under deletes or updates, all data points in the chunk are read. In the following, we observe the regular intervals of timestamps and introduce a step regression for efficient indexing on cases (a) and (b) on timestamps.

Definition 3.5 (Chunk Index). Given a chunk $C^\kappa = \{P_1, \ldots, P_{|C^\kappa|}\}$ in the increasing order of time, and a lookup timestamp t^*,

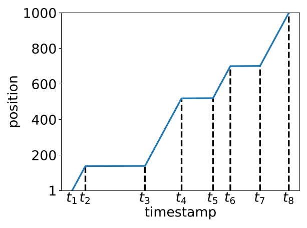
(a) to check if a data point exists at t^*, the chunk index returns TRUE if $t^* \in \{P_1.t, \ldots, P_{|C^\kappa|}.t\}$, FALSE otherwise;
(b-1) to get the position of the closest data point after t^*, the chunk index returns $\arg\min_j\{P_j.t \mid P_j.t > t^*, P_j \in C^\kappa\}$;
(b-2) to get the position of the closest data point before t^*, the chunk index returns $\arg\max_j\{P_j.t \mid P_j.t < t^*, P_j \in C^\kappa\}$.

Different from the existing learned indexes on arbitrary PDF [29, 30, 34], we notice the distinct step features on timestamp-position relationships. Figure 8 illustrates the timestamp-position maps extracted from four real-world datasets. The steep part of the step, e.g., $[t_1, t_2)$ in Figure 8(d), stems from the fact that sensor devices usually collect data with a preset frequency, while the flat part (e.g., $[t_2, t_3)$ in Figure 8(d)) reflects the occasionally delayed timestamps due to issues such as transmission interruption [17]. Therefore, we introduce the step regression function to model such timestamp-position steps. Intuitively, a step regression function has two alternating segments, tilt and level, corresponding to a fixed positive slope and a slope of zero, respectively.

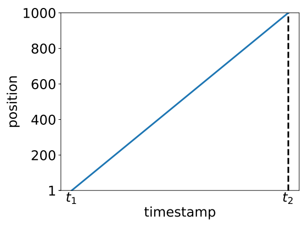
3.5.1 *Step Regression.* Given a set of split timestamps $\mathbb{S} = \{t_1, \ldots, t_m\}$, $m \geq 2$ in the increasing order of time, the range $[t_1, t_m]$ is split into $m-1$ segments, i.e., $[t_i, t_{i+1})$ for $1 \leq i \leq m-2$, and $[t_{m-1}, t_m]$. For a chunk C^κ, we denote $t_1 = FP(C^\kappa).t$ and $t_m = LP(C^\kappa).t$.

Table 1: Different types of chunk data read operations for M4 functions

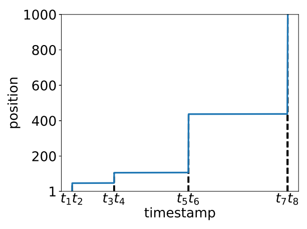
Function	Candidate Verification	Candidate Generation
FP/LP	/	(b) get the closest data point after or before a given timestamp
BP/TP	(a) check if a data point exists at a given timestamp	(c) get all data points



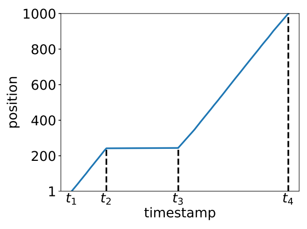
(a) BallSpeed dataset



(b) MF03 dataset

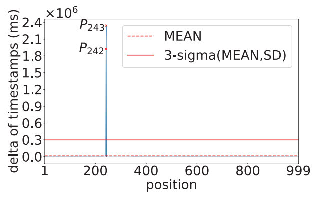


(c) KOB dataset

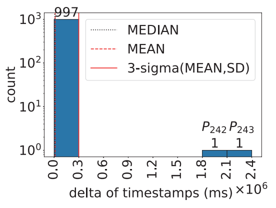


(d) RcvTime dataset

Figure 8: Example of timestamp-position steps



(a) Deltas of timestamps given by each pair of consecutive points



(b) Distribution of the deltas of timestamps

Figure 9: An example for learning parameters

Definition 3.6 (Step Regression). The step regression function $f : [FP(C^\kappa).t, LP(C^\kappa).t] \to [1, |C^\kappa|]$ models the map from the timestamp of a data point to its relative position in the chunk. Formally,

$$f(t) = \mathbf{1}_{I_o}(t) \times K \times t + \sum_{i=1}^{m-1} \mathbf{1}_{I_i}(t) \times b_i, \ t \in [t_1, t_m],$$

where K is the fixed positive slope and the intercepts are determined by $\mathbb{S} = \{t_1, \ldots, t_m\}$.

$$b_1 = 1 - K \times t_1, \ b_{m-1} = \begin{cases} |C^\kappa| - K \times t_m, & \text{if } m-1 \text{ is odd}, \\ |C^\kappa|, & \text{if } m-1 \text{ is even}, \end{cases}$$

$$b_i = \begin{cases} b_{i-2} - K \times (t_i - t_{i-1}), & \text{if } i \text{ is odd}, \ 2 \le i \le m-2, \\ K \times t_i + b_{i-1}, & \text{if } i \text{ is even}, \ 2 \le i \le m-2, \end{cases}$$

$$I_i = \begin{cases} [t_i, t_{i+1}), & \text{if } 1 \le i \le m-2, \\ [t_{m-1}, t_m], & \text{if } i = m-1, \end{cases}$$

$$I_o = \bigcup_{n \in \mathbb{N}^+ \wedge 2n-1 < m} I_{2n-1},$$

and $\mathbf{1}_A(t)$ is an indicator function with intervals A such that:

$$\mathbf{1}_A(t) = \begin{cases} 1, & \text{if } t \in A, \\ 0, & \text{if } t \notin A. \end{cases}$$

The function is a variation of the canonical form $k \times t + b$. Note that the first segment is tilt by default. The first and last points in the chunk always have the minimum and the maximum output positions, respectively.

Proposition 3.7 (*FP/LP*). *The step regression function of a chunk C^κ always has $f(FP(C^\kappa).t) = 1$ and $f(LP(C^\kappa).t) = |C^\kappa|$.*

Example 3.8. To model the data shown in Figure 8(d), the step regression function has slope $K = 1/9000$ and split timestamps $\mathbb{S} = \{t_1, t_2, t_3, t_4\} = \{1639966606000, 1639968775000, 1639972630000, 1639979452000\}$, i.e.,

$$f(t) = \begin{cases} 1/9000 \times t - 182218510.78, & \text{if } t \in [t_1, t_2), \\ 242, & \text{if } t \in [t_2, t_3), \\ 1/9000 \times t - 182218939.11, & \text{if } t \in [t_3, t_4]. \end{cases}$$

The first point has $f(1639966606000) = 1$ and the last point has $f(1639979452000) = 1000$.

Given K and $\mathbb{S}$, the step regression function is fully determined. In the following, we provide a heuristic method to learn the parameters K and $\mathbb{S}$ of the step regression function.

3.5.2 *Learning Slope K.*

Referring to the regular data collection frequency, the slope K is estimated by the median of slopes given by each pair of consecutive points, i.e.,

$$K = 1/\text{median}(\{P_{j+1}.t - P_j.t \mid P_j, P_{j+1} \in C^\kappa\}).$$

Example 3.9. Figure 9 plots the deltas of timestamps extracted from the data shown in Figure 8(d). The delta of timestamps for the i-th data point P_i is $P_{i+1}.t - P_i.t$, where $1 \le i \le 999$. The median for the deltas of timestamps is 9000, as shown with the dotted line in Figure 9(b), i.e., mostly collecting data in every 9s. Therefore, the slope is $K = 1/9000$.

3.5.3 *Learning Split Timestamps $\mathbb{S}$.*

The general idea is to first select changing points from the chunk based on statistics, then calculate the intercept for each segment of the step regression function, and finally derive the split timestamps by intersecting two adjacent segments.

Select Changing Points $\mathbb{P}_s$. Changing points are selected by applying the 3-sigma rule on the deltas of timestamps. As illustrated in Figure 9(a), whenever the delta changes from below the threshold to above the threshold or vice versa, the pivot data point is selected as a changing point. Formally, we have

$$\begin{aligned}\mathbb{P}_s = \{&P_j \mid P_j.t - P_{j-1}.t \le \mu + 3\sigma, P_{j+1}.t - P_j.t > \mu + 3\sigma,\\ &P_{j-1}, P_j, P_{j+1} \in C^\kappa\} \cup\\ \{&P_j \mid P_j.t - P_{j-1}.t > \mu + 3\sigma, P_{j+1}.t - P_j.t \le \mu + 3\sigma,\\ &P_{j-1}, P_j, P_{j+1} \in C^\kappa\},\end{aligned}$$

where

$$\mu = \text{mean}(\{P_{j+1}.t - P_j.t \mid P_j, P_{j+1} \in C^\kappa\}),$$
$$\sigma = \text{std}(\{P_{j+1}.t - P_j.t \mid P_j, P_{j+1} \in C^\kappa\}).$$

Example 3.10 (Example 3.9 continued). Only two data points, P_{242} and P_{243}, have their deltas of timestamps larger than the threshold $\mu + 3\sigma$. As a result, the set of changing points is $\mathbb{P}_s = \{P_{242}, P_{244}\}$.

Calculate Intercepts b_i. Next, we calculate the intercept for each segment. With $|\mathbb{P}_s|$ changing points, we know that the step regression function has $|\mathbb{P}_s| + 1$ segments. In other words, $m = |\mathbb{P}_s| + 2$. According to Proposition 3.7, the first and the last segments of the step regression function should have $f(FP(C^\kappa).t) = 1$ and $f(LP(C^\kappa).t) = |C^\kappa|$, respectively. Therefore, b_1 and b_{m-1} are calculated as defined in Section 3.5.1, i.e.,

$$b_1 = 1 - K \times FP(C^\kappa).t,$$
$$b_{m-1} = \begin{cases} |C^\kappa| - K \times LP(C^\kappa).t, & \text{if } m-1 \text{ is odd,} \\ |C^\kappa|, & \text{if } m-1 \text{ is even.} \end{cases}$$

For the other $m-3$ segments, let the i-th segment have $f(P_j.t) = j$, where P_j is the $(i-1)$-th point in $\mathbb{P}_s$, $2 \le i \le m-2$. Then the intercept b_i for the i-th segment is determined by

$$b_i = \begin{cases} j - K \times P_j.t, & \text{if } i \text{ is odd,} \\ j, & \text{if } i \text{ is even.} \end{cases}$$

Derive Split Timestamps $\mathbb{S}$. Finally, the split timestamps $\mathbb{S} = \{t_1, \ldots, t_m\}$ derived by intersecting two adjacent segments are

$$t_i = \begin{cases} FP(C^\kappa).t, & \text{if } i = 1, \\ (b_{i-1} - b_i)/K, & \text{if } i \text{ is odd, } 2 \le i \le m-1, \\ (b_i - b_{i-1})/K, & \text{if } i \text{ is even, } 2 \le i \le m-1, \\ LP(C^\kappa).t, & \text{if } i = m. \end{cases}$$

4 EXPERIMENTS

In the experiments, we compare M4-LSM with the original M4 algorithm [25] implemented as M4-UDF in Apache IoTDB. The experiments are conducted on a machine running Ubuntu 20.04.3 with 32GB DDR Memory, Intel Core i7 CPU @ 2.50GHz and a 3.65 TB hard disk drive (HDD).

Table 2 gives a summary of the four real-world datasets used in experiments. BallSpeed dataset is a 71-minute soccer monitoring data collected by a speed sensor in a soccer ball at the frequency of 2000Hz [7]. MF03 dataset is a 28-hour manufacturing equipment monitoring data collected by sensor MF03 (i.e., Electrical Power Main Phase 3) at around 100Hz frequency [2]. KOB and RcvTime are two datasets provided by the customers of Apache IoTDB.

Table 2: Dataset summary

Dataset	Entire time range	# Points
BallSpeed	71 minutes	7,193,200
MF03	28 hours	10,000,000
KOB	4 months	1,943,180
RcvTime	1 year	1,330,764

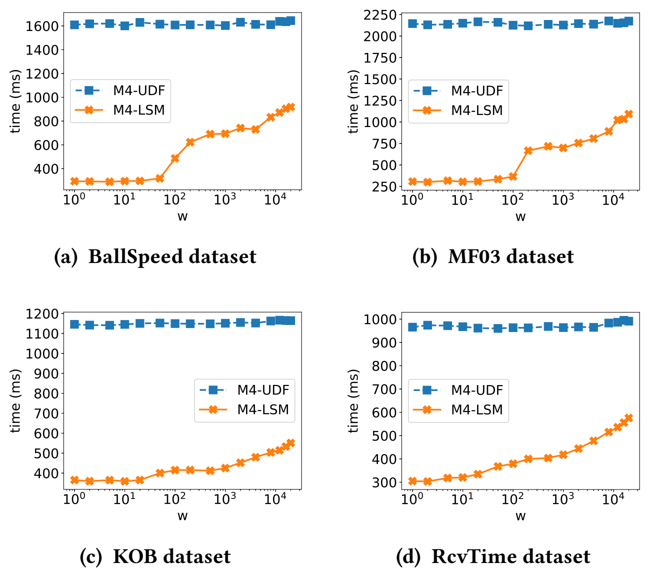


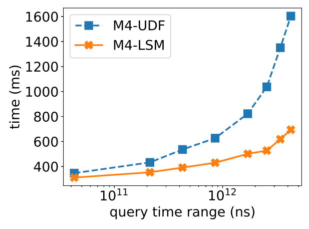
Figure 10: Varying the number of time spans *w*

4.1 Varying the Number of Time Spans *w*

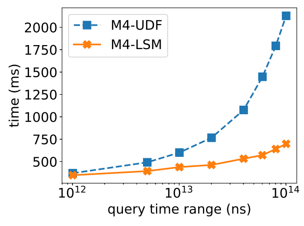
The first experiment evaluates one of the M4 query parameters: the number of spans w, corresponding to the number of pixel columns in a line chart. It usually ranges from 10 to 10,000, while a typical 4K monitor has at most 3840 pixel columns. The M4 representation query latencies of M4-UDF and M4-LSM under different w on four datasets are shown in Figure 10.

The query time costs of M4-UDF under different w are almost constant. It is because M4-UDF loads all the chunks anyway, irrelevant of w. The query latency of M4-LSM increases as w increases. It is because as w increases, the length of a single M4 time span $\frac{t_{qe}-t_{qs}}{w}$ decreases, given a fixed query time range length $t_{qe} - t_{qs}$. Consequently, the number of chunks, which are split by the M4 time spans and thus loaded from disk by M4-LSM, tends to increase. It takes about 700ms for M4-LSM to represent a time series of 10 million points in 1000 pixel columns.

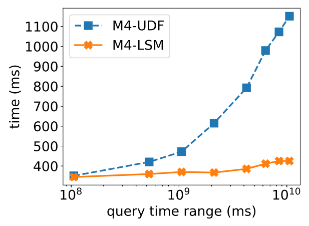
The query latencies of M4-LSM for KOB and RcvTime datasets do not increase as fast as those for BallSpeed and MF03 datasets. This is because KOB and RcvTime datasets have a skewed distribution in time, making chunks vary in time interval length. Consequently, many chunks of small length may not be split by M4 time spans even at a high level of w.



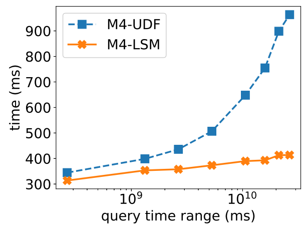
(a) BallSpeed dataset



(b) MF03 dataset

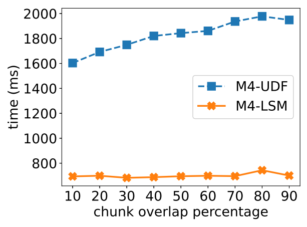


(c) KOB dataset

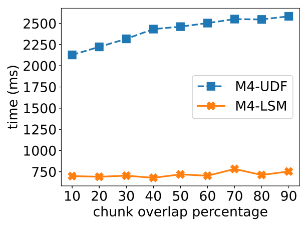


(d) RcvTime dataset

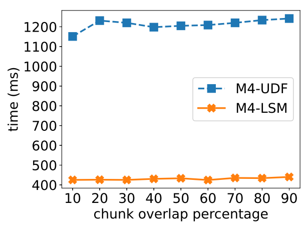
Figure 11: Varying query time range $t_{qe} - t_{qs}$



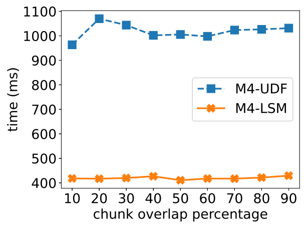
(a) BallSpeed dataset



(b) MF03 dataset



(c) KOB dataset



(d) RcvTime dataset

Figure 12: Varying chunk overlap percentage

4.2 Varying Query Time Range

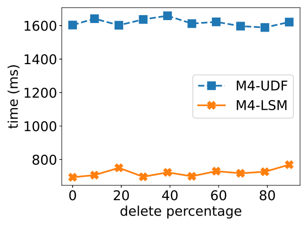
We now consider another M4 query parameter: the query time range length. While different datasets have various data collection frequencies and total time ranges as illustrated in Table 2, a typical time series of 1 million points contains the data collected in two weeks with a data collection frequency of every second, often competent in visual analysis. The M4 representation query latencies of M4-UDF and M4-LSM under different query time range lengths on four datasets are shown in Figure 11.

With the increase of the query range length, the time costs of M4-UDF and M4-LSM increase to varying degrees. The increase of M4-UDF is significant, because as more chunks are involved in the longer query time range, more disk I/O and CPU costs are spent to load and merge these chunks. The query latency of M4-LSM also increases but in a much slower way. The reason is that as the query time range length increases, the proportion of chunks split by M4 time spans decreases. While the chunks split by M4 time spans still need to be loaded, most other chunks can be pruned by the candidate generation and verification framework.

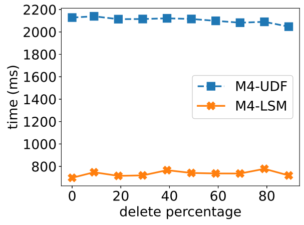
4.3 Varying Chunk Overlap Percentage

In addition to M4 representation query parameters, how the data are written (updated and deleted) will affect the LSM-based storage, and thus the query performance. One of the key issues is chunks overlapping in time intervals, incurring costly chunk loading and merging. In this experiment, we propose to write the points in different orders, leading to various chunk overlap rates. The M4 representation query latencies of M4-UDF and M4-LSM under different percentages of overlapping chunks are illustrated in Figure 12.

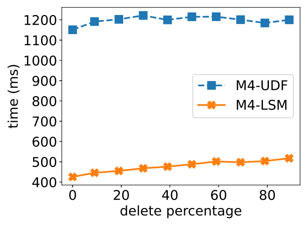
The query latency of M4-UDF increases as the overlap percentage increases. This is because merging more overlapping chunks needs more CPU cost, although the I/O cost does not change. The query latency of M4-LSM is almost constant, owing to the merge free strategy. No chunks need loading as long as the candidate point is not in the time interval of any later appended chunks, and the CPU cost of candidate verification for *BP/TP* is saved with the chunk index.



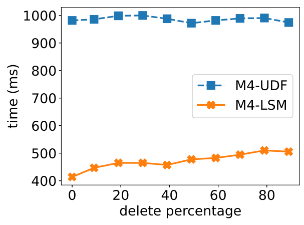
(a) BallSpeed dataset



(b) MF03 dataset



(c) KOB dataset

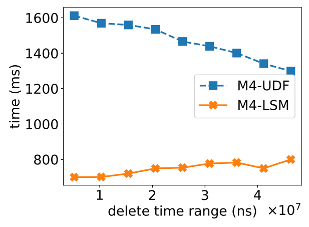


(d) RcvTime dataset

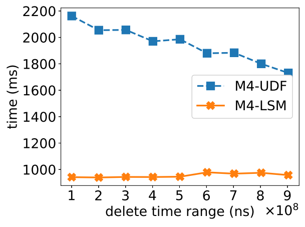
Figure 13: Varying delete percentage

4.4 Varying Delete Percentage

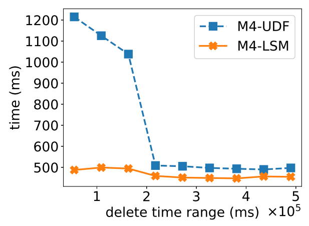
How the data are deleted also affects the LSM-based storage and thereby the M4 representation query. In this experiment, we evaluate the frequency of delete operations. Figure 13 shows the M4 representation query latencies of M4-UDF and M4-LSM under different delete percentages on four datasets.



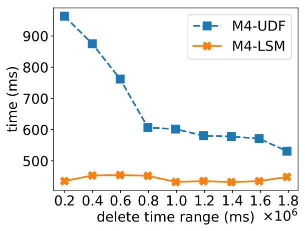
(a) BallSpeed dataset



(b) MF03 dataset



(c) KOB dataset



(d) RcvTime dataset

Figure 14: Varying delete time range $t_{de} - t_{ds}$

The query latency of M4-UDF is almost constant despite the increasing number of deletes, thanks to the CPU-efficient delete sort operation [1] inherent in IoTDB. Although the query latency of M4-LSM has an increasing trend, the overall value is small. This is because, on the one hand, increasing delete percentages will increase the probability of candidate points being deleted, introducing more I/O and CPU costs for recalculating the chunk metadata; on the other hand, the number of deleted candidate points is limited after all, given that the delete time range of each delete is small compared to the chunk time interval length.

4.5 Varying Delete Time Range

We now fix the number of deletes and vary the delete time range. Figure 14 reports the M4 representation query latency of M4-UDF and M4-LSM under various delete time range lengths.

The query latency of M4-UDF decreases as the delete time range length increases. It is more obvious in the KOB and RcvTime datasets, where the skewed time distribution results in many chunks of small length being completely deleted. For M4-LSM, increasing delete time range lengths makes candidate points more likely to be deleted on the one hand, and on the other hand may cause some chunks to be completely deleted. Overall, the query latency of M4-LSM is small, due to the robustness of candidate points under deletes.

5 RELATED WORK

While visualizing time series is highly demanded in practice, e.g., finding interesting patterns [33], the database native representation operator for visualization is surprisingly absent.

5.1 Representing Time Series for Visualization

Time series representation is a cornerstone of time series mining [18]. A number of time series representations have been proposed after decades of research, including sampling [12], Discrete Fourier Transform (DFT) [10], Discrete Wavelets Transform (DWT) [11], Singular Value Decomposition (SVD) [28], Piecewise Aggregate Approximation (PAA) [27], Symbolic Aggregate approXimation (SAX) [32, 37], piecewise polynomials [31], and shapelet-based representations [19, 41]. Among them, sampling is a general way to perform data reduction and is widely used in data mining [9, 38]. In terms of visualization tasks, Park et al. [36] develop a visualization-aware sampling layer between the visualization client and the database backend to speed up queries for the scatterplots and map plots. In contrast, M4 [24, 25] is designed for the line chart, which is more suitable for the visualization of time series, as an in-DB data reduction method. Since M4 shows zero pixel error in two-color (binary) line visualization, which is not possible with other data reduction techniques such as MinMax, we focus on deploying M4 representation in time series databases.

5.2 LSM-tree based Storage

Log-Structured Merge Tree (LSM-tree) [35] is widely adopted as a storage backend by state-of-the-art key-value stores [21] including time series databases. This is because LSM-tree meets the performance requirement of both high-throughput writes and fast point reads of key-value stores. Research on LSM-based storage has flourished in recent years. For example, Idreos et al. [22] propose a unified design space spanning LSM-trees, B-trees, Logs, etc., and optimize the design of these data structures to improve the performance of NoSQL storage systems [13–15]. These lines of work are orthogonal to our work, as our focus is on the optimization of the (M4) query execution algorithm in the LSM-based systems.

6 CONCLUSIONS

M4 representation [25] has been found error-free in two-color line visualization of time series data. The method however is originally designed for relational databases, without considering the LSM-based storage, which is widely adopted in the commodity time series database systems. In this paper, we present M4-LSM without merging any chunk in the LSM-based store. To access data points in chunks, we observe the regular intervals of timestamps and introduce a step regression for efficient indexing. The method has been deployed in Apache IoTDB, an open-source LSM-based time series database developed upon our preliminary studies [39]. Extensive experiments over real-world datasets demonstrate that M4-LSM takes about 700ms to represent a time series of 10 million points in 1000 pixel columns, enabling instant visualization of data in four months with a data collection frequency of every second.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (under grants 62072265, 62232005, 62021002), the National Key Research and Development Plan (under grants 2021YFB3300500, 2019YFB1705301, 2019YFB1707001), 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] https://cwiki.apache.org/confluence/display/IOTDB/Query+Fundamentals.

[2] https://debs.org/grand-challenges/2012/.

[3] https://github.com/apache/iotdb/tree/research/M4-visualization.

[4] https://github.com/LeiRui/M4-visualization-exp.

[5] https://iotdb.apache.org.

[6] https://iotdb.apache.org/UserGuide/Master/Operators-Functions/Sample.html#m4-function.

[7] https://www.iis.fraunhofer.de/en/ff/lv/dataanalytics/ek/download.html.

[8] https://www.influxdata.com/.

[9] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 29–42. ACM, 2013.

[10] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms, 4th International Conference, FODO'93, Chicago, Illinois, USA, October 13-15, 1993, Proceedings*, volume 730 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 1993.

[11] K. Chan and A. W. Fu. Efficient time series matching by wavelets. In M. Kitsuregawa, M. P. Papazoglou, and C. Pu, editors, *Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia, March 23-26, 1999*, pages 126–133. IEEE Computer Society, 1999.

[12] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[13] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–94. ACM, 2017.

[14] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 505–520. ACM, 2018.

[15] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 449–466. ACM, 2019.

[16] P. Esling and C. Agón. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, 2012.

[17] C. Fang, S. Song, and Y. Mei. On repairing timestamps for regular interval time series. *Proc. VLDB Endow.*, 15(9):1848–1860, 2022.

[18] T. Fu. A review on time series data mining. *Eng. Appl. Artif. Intell.*, 24(1):164–181, 2011.

[19] J. Grabocka, N. Schilling, M. Wistuba, and L. Schmidt-Thieme. Learning time-series shapelets. In S. A. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 392–401. ACM, 2014.

[20] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[21] S. Idreos and M. Callaghan. Key-value storage engines. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2667–2672. ACM, 2020.

[22] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[23] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time series management systems: A survey. *IEEE Trans. Knowl. Data Eng.*, 29(11):2581–2600, 2017.

[24] U. Jugel. *Visualization-driven data aggregation: rethinking data acquisition for data visualizations*. PhD thesis, Technical University of Berlin, Germany, 2017.

[25] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *Proc. VLDB Endow.*, 7(10):797–808, 2014.

[26] 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*, pages 3340–3352. IEEE, 2022.

[27] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.

[28] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 289–300. ACM Press, 1997.

[29] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[30] Y. Li, Z. Wang, B. Ding, and C. Zhang. Automl: A perspective where industry meets academy. In F. Zhu, B. C. Ooi, and C. Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 4048–4049. ACM, 2021.

[31] C. Lin, E. Boursier, and Y. Papakonstantinou. Plato: Approximate analytics over compressed time series with tight deterministic error guarantees. *Proc. VLDB Endow.*, 13(7):1105–1118, 2020.

[32] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.

[33] C. Liu, K. Zhang, H. Xiong, G. Jiang, and Q. Yang. Temporal skeletonization on sequential data: patterns, categorization, and visualization. In S. A. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1336–1345. ACM, 2014.

[34] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[35] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[36] Y. Park, M. J. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 755–766. IEEE Computer Society, 2016.

[37] J. Shieh and E. J. Keogh. *i*sax: indexing and mining terabyte sized time series. In Y. Li, B. Liu, and S. Sarawagi, editors, *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 623–631. ACM, 2008.

[38] C. Wang and B. Ding. Fast approximation of empirical entropy via subsampling. In A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 658–667. ACM, 2019.

[39] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun. Apache iotdb: Time-series database for internet of things. *Proc. VLDB Endow.*, 13(12):2901–2904, 2020.

[40] 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.*, 15, 2022.

[41] L. Ye and E. J. Keogh. Time series shapelets: a new primitive for data mining. In J. F. E. IV, F. Fogelman-Soulié, P. A. Flach, and M. J. Zaki, editors, *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, pages 947–956. ACM, 2009.

A SUPPLEMENTARY MATERIAL

A.1 SQL

A SQL-like expression of the M4 representation query is as follows,

```
SELECT  FirstTime(T),    /* time of FP(T) */
        FirstValue(T),  /* value of FP(T) */
        LastTime(T),     /* time of LP(T) */
        LastValue(T),   /* value of LP(T) */
        BottomTime(T),   /* time of BP(T) */
        BottomValue(T), /* value of BP(T) */
        TopTime(T),      /* time of TP(T) */
        TopValue(T)     /* value of TP(T) */
FROM    T
GROUPBY floor(@w*(t-@tqs)/(@tqe-@tqs))
```

where `@tqs`, `@tqe` and `@w` are the query parameters. $floor(x)$ is a mathematical function that returns the largest integer no greater than the input number x. For $i = 1, \dots, w$, $floor(\frac{w*(t-t_{qs})}{t_{qe}-t_{qs}}) = i-1$ means grouping together the points that fall within the time span $I_i = [t_{qs} + \frac{t_{qe}-t_{qs}}{w} * (i-1), t_{qs} + \frac{t_{qe}-t_{qs}}{w} * i)$.

A.2 M4-LSM Algorithm

Algorithm 1 illustrates M4-LSM. For each time span I_i, it iteratively generates the candidate point P_G from chunk metadata (line 9) as in Section 3.2, and verifies whether P_G is the latest (line 10) as in Sections 3.3 and 3.4 for each representation function G. If non-latest, it lazily loads chunks and updates chunk metadata (line 12).

```
Algorithm 1: M4-LSM algorithm
   Input: Time series $T$, query range $[t_{qs}, t_{qe})$, the number of
          time spans $w$
   Output: $\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}, i = 1, \dots, w$
 1 determine all time spans $I_i$ by $[t_{qs}, t_{qe})$ and $w$
 2 read the metadata of all chunks $\mathbb{C}$ of time series $T$ in the
     time range $[t_{qs}, t_{qe})$ by MetadataReader
 3 read all deletes $\mathbb{D}$ of time series $T$ in the time range
     $[t_{qs}, t_{qe})$ by DataReader
 4 for each time span $I_i$ do
 5     find the chunks $\mathbb{C}'' \subseteq \mathbb{C}$ having time intervals
         overlapping with $I_i$
 6     apply the deletes of $I_i$ to $\mathbb{C}''$
 7     for $G \in \{FP, LP, BP, TP\}$ do
 8         while $G(T_i)$ is not computed do
 9             generate the candidate point $P_G$ in $\mathbb{C}''$ for $G$
10             verify the candidate point $P_G$
11             if $P_G$ is not latest then
12                 chunk lazy loading
13             else
14                 set $G(T_i) = P_G$
15 return $\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}, i = 1, \dots, w$
```

A.3 Proof of Proposition 3.1

Proof. For $G = FP$ (or $G = LP$), as defined in Section 3.2, P_G is the point with both the minimum time (or the maximum time) and the largest version number. Therefore, no other chunk in $\mathbb{C}''$ with a version number larger than $P_G.\kappa$ contains a point with the same time as P_G. In other words, P_G is never updated by later appended chunks, having $P_G.t \nvDash C^{\kappa_1}$ for any $C^{\kappa_1} \in \mathbb{C}''$ with $P_G.\kappa < \kappa_1$. Moreover, P_G is not covered by the delete time range of any delete in $\mathbb{D}$ with a larger version number than $P_G.\kappa$, i.e., $P_G.t \nvDash D^{\kappa}, D^{\kappa} \in \mathbb{D}, \kappa > P_G.\kappa$. Referring to Formula 2 in Definition 2.7, P_G is a point in the merged time series $M(\mathbb{C}'', \mathbb{D})$, i.e., the latest.

Next we prove that the latest candidate point is the representation result for $G \in \{FP, LP\}$. From Definition 2.7, we know that if P_G is the latest, P_G is a point in $T_i = M(\mathbb{C}'', \mathbb{D})$. Therefore, T_i can be divided into three disjoint subsequences based on P_G,

$$T_l = \{P \mid P \in T_i, P.t < P_G.t\},$$
$$T_m = \{P_G\},$$
$$T_r = \{P \mid P \in T_i, P.t > P_G.t\}.$$

From the distributive property [20] of G, we have

$$G(T_i) = G(T_l \cup T_m \cup T_r) = G(\{G(T_l), G(T_m), G(T_r)\})$$
$$= G(\{G(T_l), P_G, G(T_r)\}).$$

It is easy to know that the chunk metadata bounds the time and value range of all points in the chunk. Since P_G is extracted from the boundary points among all chunk metadata, we have

$$P_G.t \le G(T_i).t \le \min(G(T_l).t, G(T_r).t), G = FP$$
$$P_G.t \ge G(T_i).t \ge \max(G(T_l).t, G(T_r).t), G = LP$$

i.e., $G(T_i) = G(\{G(T_l), P_G, G(T_r)\}) = P_G$, for $G \in \{FP, LP\}$. □

A.4 Proof of Proposition 3.3

Proof. The first condition states that P_G is not updated by any later appended chunks, having $P_G.t \nvDash C^{\kappa}$ for any $C^{\kappa} \in \mathbb{C}''$ with $\kappa > P_G.\kappa$. The second condition states that P_G is not covered by any delete in $\mathbb{D}$ with a larger version number than $P_G.\kappa$, i.e., $P_G.t \nvDash D^{\kappa}$ for any $D^{\kappa} \in \mathbb{D}$ with $\kappa > P_G.\kappa$. Referring to Formula 2 in Definition 2.7, P_G is a point in the merged time series $M(\mathbb{C}'', \mathbb{D})$, i.e., the latest. With the latest candidate point, the proof of the representation result follows the same line of Proposition 3.1. □

A.5 System Deployment

This section describes the system deployment of M4-LSM in Apache IoTDB [5]. The document of the M4 function is available on the product website [6]. The source code has been committed to the GitHub repository of Apache IoTDB [3]. An overview of the deployment is shown in Figure 15. Let us first introduce some interfaces of the system in Section A.5.1, upon which deployment is conducted in Section A.5.2.

A.5.1 System Overview. As illustrated in Figure 15, the storage of data in Apache IoTDB consists of TsFiles, carrying ChunkData and ChunkMetadata, as well as TsFile.mods recording the delete operations. Note that these delete operations will not be applied to modify the read-only TsFiles until the files are compacted for a new one, which is a typical strategy in LSM-based store.

The system built-in `SeriesReader` contains `MetadataReader`, `DataReader` and `MergeReader`. `MetadataReader` and `DataReader` are responsible for loading chunk metadata, chunk data, and delete

Table 3: Experiment settings

Section	# Time Spans w	Query Time Range $t_{qe} - t_{qs}$	Chunk Overlap Percentage	Delete Percentage	Delete Time Range $t_{de} - t_{ds}$
4.1	**vary**	entire time range	10%	0%	–
4.2	1000	**vary**	10%	0%	–
4.3	1000	entire time range	**vary**	0%	–
4.4	1000	entire time range	10%	**vary**	10% of chunk time interval
4.5	1000	entire time range	10%	49%	**vary**

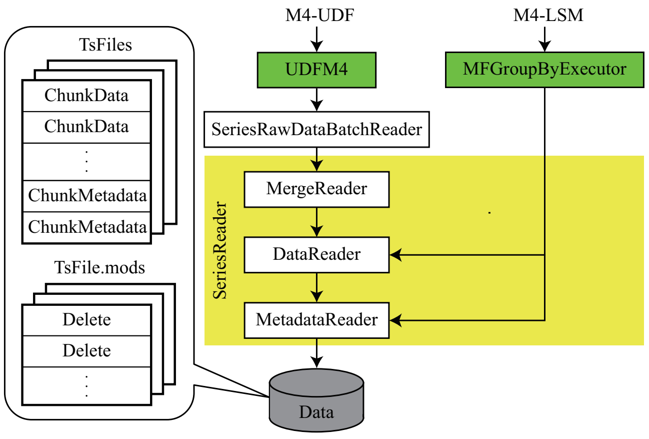


Figure 15: System deployment in Apache IoTDB

data from disks, while `MergeReader` merges the chunks with possible overlapping time intervals and data overwrites referring to the version numbers. Of course, the delete operations are also applied if any, to form a whole time series.

A.5.2 Deployment Details. We implement the original M4 method as M4-UDF, a user-defined function (UDF) `UDFM4` in Apache IoTDB. It reads the assembled time series directly from the system built-in `SeriesRawDataBatchReader`, and performs the representation computation on the time series. Note that ChunkMetadata is not accessed by `UDFM4`. The M4-UDF deployment thus serves as the baseline of the original M4 implementation [25].

We implement `MFGroupByExecutor` for M4-LSM. Rather than reading the system assembled time series as `UDFM4`, the M4-LSM implementation directly uses `MetadataReader` and `DataReader`. ChunkMetadata helps in pruning ChunkData, which saves both IO and computation cost. Note that `MFGroupByExecutor` does not use `MergeReader`, i.e., merge free.

A.6 Experimental Settings

A.6.1 Experiment Setup. As summarized in Table 3, we consider (1) various numbers of time spans w and time ranges $t_{qe} - t_{qs}$ for the query, as well as (2) different chunk overlap percentages, delete percentages and delete time ranges $t_{de} - t_{ds}$ for the storage. For each experiment, we vary one of the five parameters while fixing the other four.

A.6.2 Datasets. Figure 16 illustrates the visualization of four real-life datasets. As summarized in Table 2, the four datasets collect

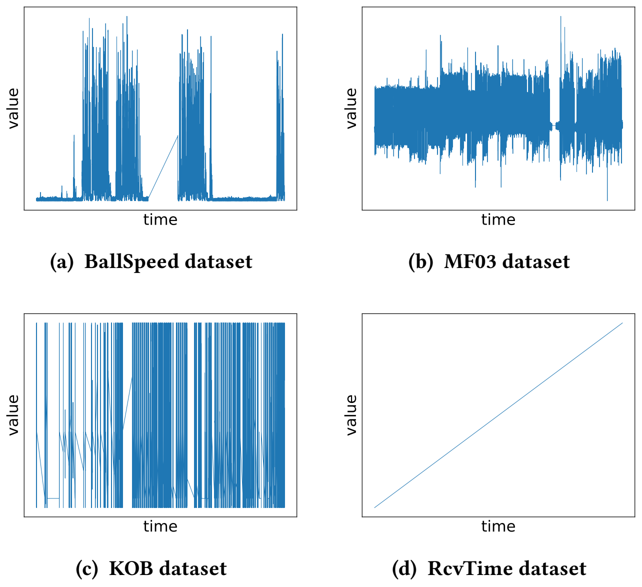


Figure 16: Visualization of four real-life datasets

Table 4: System settings

Parameter	Value
unseq_tsfile_size	1073741824
seq_tsfile_size	1073741824
avg_series_point_number_threshold	1000
page_size_in_byte	1073741824
compaction_strategy	NO_COMPACTION
enable_unseq_compaction	false

data at different frequencies. BallSpeed and MF03 datasets collect data over a short period of time at a high frequency. In contrast, KOB and RcvTime datasets collect data over a long period of time at a low frequency.

A.6.3 System Setup. We use IoTDB v0.12.4 as the test database. Important parameter settings for IoTDB are summarized in Table 4. Among them, "unseq_tsfile_size" and "seq_tsfile_size" are used to control the size of TsFiles. "avg_series_point_number_threshold" and "page_size_in_byte" are used to control the number of points in a chunk. "compaction_strategy" and "enable_unseq_compaction" are used to turn off the LSM compaction strategy.

B ADDITIONAL MATERIAL

C RESPONSE TO REVIEWERS