Time Series Representation for Visualization in Apache IoTDB

LEI RUI, Tsinghua University, China
XIANGDONG HUANG, Tsinghua University, China
SHAOXU SONG*, BNRist, Tsinghua University, China
YUYUAN KANG, University of Wisconsin-Madison, USA
CHEN WANG, Tsinghua University, China
JIANMIN WANG, BNRist, Tsinghua University, China

When analyzing time series, often interactively, the analysts frequently demand to visualize instantly large-scale data stored in databases. M4 visualization selects the first, last, bottom and top data points in each pixel column to ensure pixel-perfectness of the two-color line chart visualization. While M4 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 native 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-Tree based storage. That is, a time series is segmented and stored in a number of chunks, with possibly out-of-order arrivals, i.e., disordered on timestamps. To implement M4, a natural idea is to merge online the chunks as a whole series, with costly merge sort on timestamps, and then perform M4 representation as in relational databases. In this study, we propose a novel chunk merge free approach called M4-LSM to accelerate M4 representation and visualization. In particular, we utilize the metadata of chunks to prune and avoid the costly merging of any chunk. Moreover, intra-chunk indexing and pruning are enabled for efficiently accessing the representation points, referring to the special properties of time series. Remarkably, the time series database native operator M4-LSM has been implemented in Apache IoTDB, an open-source time series database, and deployed in companies across various industries. In the experiments over real-world datasets, the proposed M4-LSM operator demonstrates high efficiency without sacrificing preciseness.

CCS Concepts: • **Information systems → Database query processing**.

Additional Key Words and Phrases: time series visualization, database query processing

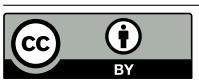
ACM Reference Format:
Lei Rui, Xiangdong Huang, Shaoxu Song, Yuyuan Kang, Chen Wang, and Jianmin Wang. 2024. Time Series Representation for Visualization in Apache IoTDB. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 35 (February 2024), 26 pages. https://doi.org/10.1145/3639290

1 INTRODUCTION

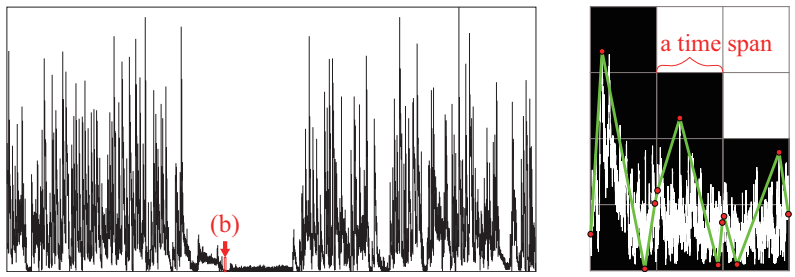
Time series representation is a typical data mining task [18] that reduces the dimensionality while still retaining its essential characteristics such as shape in visualization. M4 representation [25] is known as an error-free method for visualizing time series in two-color (binary) line chart. It

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

Authors' addresses: Lei Rui, Tsinghua University, Beijing, China, rl18@mails.tsinghua.edu.cn; Xiangdong Huang, Tsinghua University, Beijing, China, huangxdong@tsinghua.edu.cn; Shaoxu Song, BNRist, Tsinghua University, Beijing, China, sxsong@tsinghua.edu.cn; Yuyuan Kang, University of Wisconsin-Madison, Madison, USA, yuyuan@cs.wisc.edu; Chen Wang, Tsinghua University, Beijing, China, wang_chen@tsinghua.edu.cn; Jianmin Wang, BNRist, Tsinghua University, Beijing, China, jimwang@tsinghua.edu.cn.



ACM 2836-6573/2024/2-ART35
https://doi.org/10.1145/3639290



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

Fig. 1. M4 representation for time series visualization

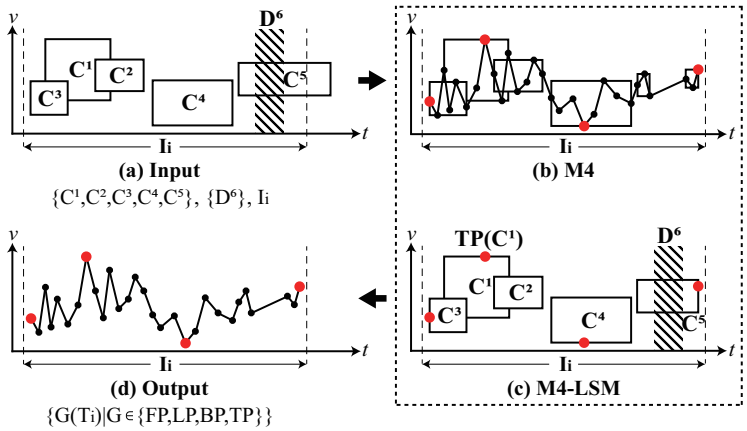
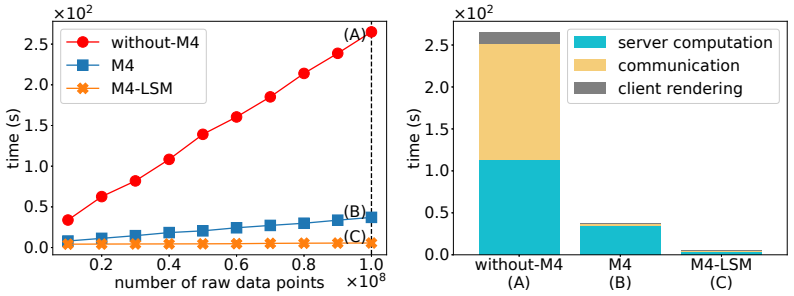


Fig. 2. Two implementations in LSM-Tree store. The original M4 needs to load and merge all chunk data in (b), while our M4-LSM may use directly the chunk metadata in (c).

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 representation points are colored in black for line chart visualization.

1.1 Motivation

Note that M4 is originally designed for visualizing time series data stored in relational databases, reading data points ordered by time. Different from RDBMS, to enable fast writes, the commodity time series database systems, such as Apache IoTDB [6] or InfluxDB [9], employ Log-Structured Merge-Tree (LSM-Tree) [37] 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-Tree based databases is still absent.



(a) Total response time (b) Response time decomposition

Time (s)	**without-M4 (A)**	**M4 (B)**	**M4-LSM (C)**
Server computation	112.85	**34.31**	2.75
Communication	**138.15**	2.42	2.37
Client rendering	14.02	0.38	0.48
Total response time	265.02	37.11	5.6

Fig. 3. Cost of visualizing time series from a database

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 implementation could still be costly, by loading all chunks from disk, ordering data points by time, and scanning the entire time series, as in Figure 2(b). Although M4 representation greatly reduces the time cost, by avoiding transferring and rendering all the raw data points, as illustrated in Figure 3, the cost of computing the representation points in the database server becomes the bottleneck (34.31s among total 37.11s). In order to increase the productivity of a visualization system, it is always desired to further reduce the response time [34].

1.2 M4-LSM Approach

In this study, we propose a novel chunk merge free approach M4-LSM in Section 3 to accelerate the M4 representation. It considers inter-chunk pruning in Section 4 and accelerates intra-chunk accessing in Section 5. (1) Note that the metadata of chunks can be used to avoid loading and merging chunks. 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 results. For example, in Figure 2(c), the candidate point $TP(C^1)$ for the TopPoint representation is obtained from chunk metadata and verified as the latest (i.e., neither updated nor deleted). Therefore, we can directly return $TP(C^1)$ as the result of the TopPoint representation and do not need to load and merge any chunk data. (2) Nevertheless, for those chunks that cannot be pruned and need to access the raw data, we observe the regular intervals of timestamps in time series and introduce a step regression for efficient indexing. Moreover, we use a value regression function to prune the points that cannot be the top or bottom ones.

As shown in Figure 3, to visualize 100 million point time series, the database server computation time is reduced from 34.31s of M4 to only 2.75s of M4-LSM, comparable to the communication cost. It enables fast visualization of large-scale time series data, without sacrificing preciseness. More extensive experiments over real-world datasets are reported in Section 8 to demonstrate the efficiency of the chunk merge free operator M4-LSM.

1.3 System Deployment

The M4-LSM operator has been implemented in Apache IoTDB [42], an open-source LSM-Tree database. It becomes a built-in function of the system, with the document available on the product website [7]. The source code of M4-LSM has been committed to the GitHub repository of Apache IoTDB by system developer [3]. The code and data of experiments are available in [4] to reproduce.

Remarkably, the system with the visualization function has been deployed and used in many companies across various industries, including rail transit, steel manufacturing, aviation industry, cloud service, and so on. For example, in fault diagnosis during train maintenance, our proposed solution is used to visualize vibration signals at a frequency of about 100Hz in Grafana. In steel manufacturing, domain experts inspect the visualized time series of temperatures to explore the potential gaps among different stages. In the aviation industry, our proposed solution is used to visualize and compare the performance metrics of different parts, with the data collection frequency as high as 20kHz to 400kHz. In cloud service, our solution is employed to visualize multiple metrics on the same dashboard for fault diagnosis of application performance. Please see Section 7 and [5] for more details on use cases.

1.4 Contributions

We highlight the contributions in both research novelty and system deployment.

(1) We formalize the problem of accelerating M4 queries over the LSM-Tree based storage (Section 3). The novel idea is to leverage metadata to prune chunks, with a candidate generation and verification mechanism (Section 4). Moreover, we devise time and value specific regression techniques to accelerate the access to chunks that cannot be pruned (Section 5).

(2) We present the deployment of the proposed M4-LSM approach in Apache IoTDB, without merging any chunk (Section 6). We also introduce a specific application to illustrate the challenges of visualizing large-scale time series, and how M4-LSM tackles the problem (Section 7).

(3) We conduct extensive experiments over real-world datasets (Section 8). The proposed M4-LSM operator demonstrates high efficiency without sacrificing preciseness. It takes about 4 seconds to represent a time series of 127 million points in 1000 pixel columns, enabling instant visualization of the data in four years with a data collection frequency of every second.

2 PRELIMINARIES

We first introduce the M4 representation in Section 2.1, and then present LSM-Tree storage of time series in Section 2.2. Table 1 lists the frequently used notations.

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

(1) FirstPoint representation function, denoted as $FP : T \longrightarrow P$, returns the point with the minimal time, i.e., $P_{first} \in \{P^* \in T \mid P.t \geq P^*.t, \forall P \in T\}$.
(2) LastPoint representation function, denoted as $LP : T \longrightarrow P$, 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 $BP : T \longrightarrow P$, returns any one of the points with the minimal value, i.e., $P_{bottom} \in \{P^* \in T \mid P.v \geq P^*.v, \forall P \in T\}$.

Table 1. Notations and explanations

Notation	Explanation
T	a time series
G	a general notation of functions *FP*, *LP*, *BP*, and *TP*
$[t_{qs}, t_{qe})$	the time range of M4 representation
w	the number of time spans in M4 representation
I_i	the i-th time span of M4 representation, corresponding to the i-th pixel column of line chart
T_i	the subsequence of T that falls in the time span I_i
κ	the version number
C^κ	the chunk with version number κ
D^κ	the delete operation with version number κ
$[t_{ds}, t_{de}]$	the time range of the delete
$\mathbb{C}$	the set of all chunks for the given time series
$\mathbb{D}$	the set of all deletes for the given time series
$M(\mathbb{C}, \mathbb{D})$	the merge function

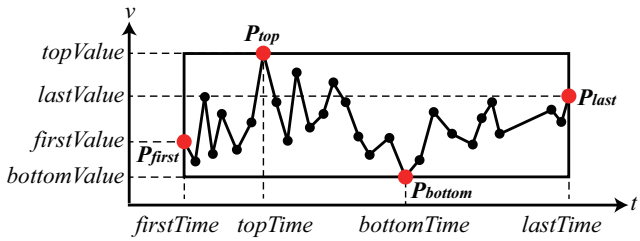


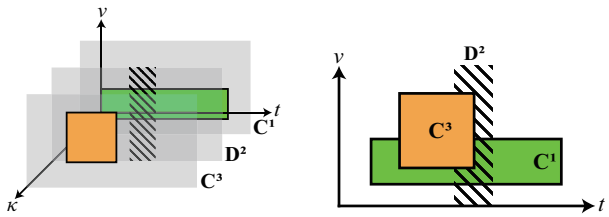
Fig. 4. An example of four representation functions

(4) TopPoint representation function, denoted as $TP : T \rightarrow P$*, returns any one of the points with the maximal value, i.e.,* $P_{top} \in \{P^* \in T \mid P.v \leq P^*.v, \forall P \in T\}$.

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*) from the visualization-driven perspective.

Example 1. *Given a time series* T *in Figure 4, 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.



(a) Three-dimensional (b) Two-dimensional

Fig. 5. Schematic diagrams of chunks and deletes

Definition 2 (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*

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

to group time series T into w time series subsequences

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

and apply the four representation functions on each subsequence

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

2.2 LSM-Tree 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 the LSM-Tree store. That is, time series data are not readily available without applying such updates and deletes.

2.2.1 Elements of LSM-Tree 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 3 (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 4 (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 $t_{ds} \leq t \leq t_{de}$.

Example 2. *Figure 5 gives two ways to better understand the relationship between chunks and deletes. Figure 5(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 5(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) 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 5 (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*

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

where C^{∞} is an empty chunk with the largest version number, and D^{∞} is an empty delete 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.

3 M4-LSM APPROACH

In this section, we give an overview of M4-LSM, an efficient approach to perform M4 representation on LSM-Tree storage.

3.1 Problem Statement

With the M4 representation query on time series and the LSM-Tree storage of time series introduced in Section 2, we now combine them to give the formal definition of the problem of performing M4 representation on LSM-Tree storage.

Definition 6 (M4 representation on LSM-Tree 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 is costly, not only for the heavy cost of I/O but also for the decompression of data [44]. Therefore, we devise novel techniques to avoid merging chunks, reduce chunks to load, and speed up chunk access, for accelerating M4 representation over LSM storage.

To begin with, M4-LSM considers the M4 time span as virtual deletes. As in Definition 2, 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 $\{D^{\infty}_{i(1)}, D^{\infty}_{i(2)}\}$ with the following delete time ranges, respectively,

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

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. Thus 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 deals with the problem of computing

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

where $\mathbb{D}'_i = \{D^{\infty}_{i(1)}, D^{\infty}_{i(2)}\} \cup \mathbb{D}$, having $T_i = M(\mathbb{C}, \mathbb{D}'_i)$. For simplicity, we omit i in $\mathbb{D}'_i$ in the rest of the paper when no ambiguity.

Algorithm 1: M4-LSM algorithm

Input: Time series T, query range $[t_{qs}, t_{qe})$, the number of time spans w
Output: $\{G(T_i) \mid G \in \{FP, LP, BP, TP\}\}, i = 1, \ldots, w$

```
1  determine all time spans I_i by [t_qs, t_qe) and w
2  read the metadata of all chunks ℂ of time series T in the time range [t_qs, t_qe)
3  read all deletes 𝔻 of time series T in the time range [t_qs, t_qe)
4  for each time span I_i do
5      union the virtual deletes of I_i with 𝔻 into 𝔻'
6      for G ∈ {FP, LP, BP, TP} do
7          while G(T_i) is not computed do
8              generate the candidate point P_G in ℂ for G (Section 4.1)
9              verify candidate P_G (Sections 4.2 and 4.3) with (CT) checking if P_G is overwritten
                (Section 5.1)
10             if P_G is not the latest (Propositions 1 and 2) then
11                 if chunk lazy loading applies then
12                     Update chunk metadata without loading chunk data (Sections 4.2 and 4.3)
13                 else
14                     if G ∈ {FP, LP} then
15                         Update chunk metadata for delete using (GT) in Section 5.1
16                     else
17                         Update chunk metadata for delete or update using (MV) in Section 5.2
18             else
19                 set G(T_i) = P_G
20 return {G(T_i) | G ∈ {FP, LP, BP, TP}}, i = 1, ..., w
```

3.2 Solution Overview

Algorithm 1 shows an overview of M4-LSM. The key observation is that we do not need to load and merge chunks if a 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 maximal value and $TP(C^1)$ is the latest, i.e., neither deleted nor updated. Thereby, for each time span I_i, it iteratively generates the candidate point P_G from chunk metadata (line 8) as in Section 4.1, and verifies whether P_G is the latest (line 9) as in Sections 4.2 and 4.3 for each representation function G. If the candidate point is non-latest, i.e., being deleted or updated, we employ the chunk lazy loading strategy in line 11 to update chunk metadata before entering the next round of candidate generation and verification. That is, instead of eagerly loading the chunk to recalculate immediately the chunk metadata under deletes or updates, the idea is to bound chunk metadata by the delete time range as in Section 4.2, or verify first the candidates of other chunks as in Section 4.3.

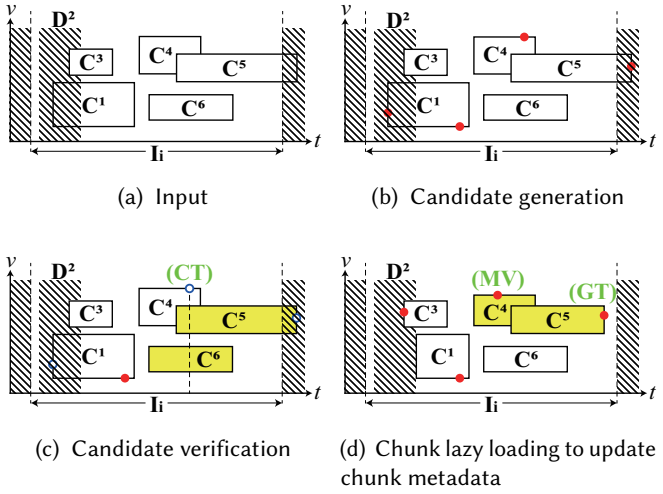


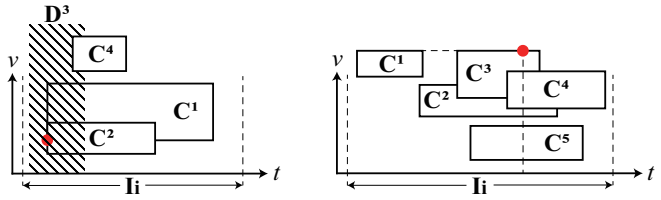
Fig. 6. Example of Algorithm 1 steps. (a) Input of each iteration includes chunk metadata, deletes and the time span. (b) Candidates (red dots) are generated from the chunk metadata in Section 4.1. (c) Candidate verification is performed to verify whether the candidate points are invalid (hollow dots) in Sections 4.2 and 4.3. Chunk access operation (CT) checks if a data point exists at a given timestamp in Section 5.1. (d) If the candidate point is invalid, new candidates are generated. (GT) gets the closest data point after/before a given timestamp in Section 5.1. (MV) gets the undeleted data point with the minimal/maximal value in Section 5.2.

Both candidate verification and generation may need to access specific data points in chunks, if they cannot be pruned by metadata. Simply scanning chunks is obviously costly. Note that when a chunk is loaded in memory, its points are sorted by timestamps. Hence, the data read operation (CT) in line 9 checks if a data point exists at a given timestamp, while (GT) in line 15 gets the closest data point after/before a given timestamp, in an array of sorted timestamps in Section 5.1. Moreover, (MV) in line 17 gets the undeleted data point with the minimal/maximal value in Section 5.2.

Figure 6 is an example of the algorithm steps. It shows an overview of the inter-chunk pruning process in Section 4. Moreover, we also illustrate the steps where the intra-chunk accessing operations (CT), (GT) and (MV) in Section 5 are applied.

4 INTER-CHUNK PRUNING

In this section, we introduce how M4-LSM utilizes chunk metadata to avoid merging chunks and prune chunks to load. To compute $G(M(\mathbb{C}, \mathbb{D}'))$ for representation function G, M4-LSM first generates the candidate point from the precomputed chunk metadata (introduced in Section 2.2.1). Next, it 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 4.1. The candidate verification for *FP*/*LP* and *BP*/*TP* representation functions are presented in Section 4.2 and Section 4.3 respectively, as they require different candidate verification rules and chunk loading strategies.



(a) FP candidate verification (b) TP candidate verification

Fig. 7. (a) $FP(C^2)$ (red dot) is non-latest under delete D^3. (b) $TP(C^3)$ (red dot) is non-latest under updates from C^4 and C^5.

4.1 Candidate Generation

M4-LSM first retrieves 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

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

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,

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

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.

4.2 *FP*/*LP* 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 1 (Latest candidate point for *FP*/*LP*). *For $G \in \{FP, LP\}$, if $P_G.t$ is not covered by 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 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 $FP(C^\kappa).t$ (or at $FP(C^\kappa).t$ with a larger version number than κ), 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. The chunk data are read by operation (GT) and accelerated by the time indexing in Section 5.1.

Example 3. *Take Figure 7(a) as an example, where* $G = FP, \mathbb{C} = \{C^1, C^2, C^4\}$ *and* $\mathbb{D}' = \{D^{\infty}_{i(1)}, D^{\infty}_{i(2)}\} \cup \{D^3\}$. *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* $P_G.t$ *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.

4.3 *BP*/*TP* 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 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 2 (Latest candidate point for *BP*/*TP*). *For* $G \in \{BP, TP\}$, *if* $P_G.t$ *is not covered by any chunk in* $\mathbb{C}$ *with a larger version number than* $P_G.\kappa$ *and* $P_G.t$ *is not covered by any delete in* $\mathbb{D}'$ *with a larger version number than* $P_G.\kappa$, *i.e.,*

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

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 2, 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., $(\bigwedge_{C^\kappa \in \mathbb{C} \wedge \kappa > P_G.\kappa} P_G.t \notin [FP(C^\kappa).t, LP(C^\kappa).t]) \wedge (\bigwedge_{D^\kappa \in \mathbb{D}' \wedge \kappa > P_G.\kappa} P_G.t \notin D^\kappa)$, 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 4.2, 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 $P_G.t \vDash C^\kappa$. The chunk data need to be read for verification by operation (CT) and accelerated by the time indexing technique in Section 5.1.

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 4.1 and 4.3. Again, the chunk data are read by operation (MV) and accelerated by the point pruning technique in Section 5.2.

Example 4. *Take Figure 7(b) as an example, where* $G = TP$, $\mathbb{C} = \{C^1, C^2, C^3, C^4, C^5\}$ *and* $\mathbb{D}' = \{D^{\infty}_{i(1)}, D^{\infty}_{i(2)}\}$. *M4-LSM retrieves the candidate point* $P_G = TP(C^3)$ *(denoted by the red dot) from* $\mathbb{P}'_G = \{TP(C^1), TP(C^3)\}$. *M4-LSM then reads the later appended overlapping chunks (i.e.,* C^4 *and* C^5*)*

to check whether they contain any point that overwrites $\mathbb{P}_G$. Assume that the read 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 $TP(M(\mathbb{C}, \mathbb{D}'))$.

5 INTRA-CHUNK ACCESSING

In this section, we introduce the intra-chunk indexing and pruning techniques to speed up the chunk data accessing operations used in M4-LSM. It accelerate the three types of chunk data read operations (CT), (GT) and (MV) for verifying and generating candidates in Section 4. Machine learning techniques [45] are incorporated into M4-LSM. For example, we learn the step regression for time indexing in Section 5.1. It predicts the position of the target timestamp in the sorted array. Moreover, we use the error-bounded value regression for point pruning in Section 5.2. It bounds prediction error and thus can be used to prune points that cannot be top/bottom ones.

5.1 Time Index With Step Regression

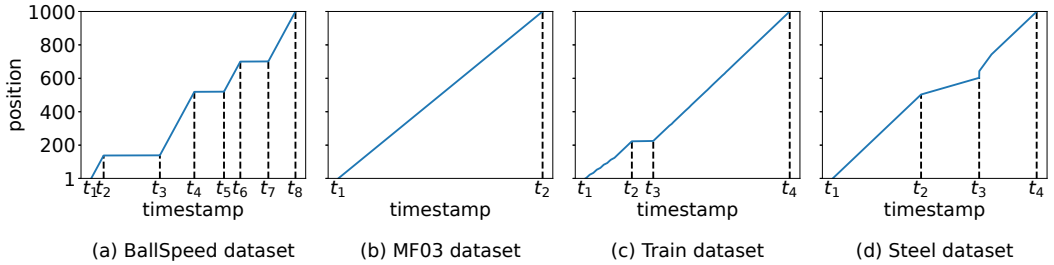
In the following, we observe the regular intervals of timestamps and introduce a step regression for efficient indexing on timestamps and accelerating data read operations (CT) and (GT).

Definition 7 (Time index). *Given a chunk $C^\kappa = \{P_1, \ldots, P_{|C^\kappa|}\}$ in the increasing order of time, and a lookup timestamp t^*,*

(CT) to check if a data point exists at t^*, the time index returns TRUE if $t^* \in \{P_1.t, \ldots, P_{|C^\kappa|}.t\}$, FALSE otherwise;

(GT-1) to get the position of the closest data point after t^*, the time index returns $\arg\min_j\{P_j.t \mid P_j.t > t^*, P_j \in C^\kappa\}$;

(GT-2) to get the position of the closest data point before t^*, the time index returns $\arg\max_j\{P_j.t \mid P_j.t < t^*, P_j \in C^\kappa\}$.



(a) BallSpeed dataset (b) MF03 dataset (c) Train dataset (d) Steel dataset

Fig. 8. Example of timestamp-position steps

Different from the existing learned indexes on an arbitrary cumulative distribution function (CDF) [29, 30, 35], 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(c), 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(c), reflects occasional gaps due to issues such as transmission interruption [19]. 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.

5.1.1 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 $t_m = LP(C^\kappa).t$.

Definition 8 (Step regression). *The step regression function $f : [FP(C^\kappa).t, LP(C^\kappa).t] \rightarrow [1, |C^\kappa|]$ models the map from the timestamp of a data point to its relative position in the chunk,*

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

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

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

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

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

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

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 minimal and maximal output positions, respectively.

Proposition 3 (*FP/LP* position). *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 5. *To model the data in Figure 8(c), the step regression function has slope $K = 1/50$ and split timestamps $\mathbb{S} = \{t_1, t_2, t_3, t_4\} = \{1591728185786, 1591728196886, 1591728205098, 1591728243948\}$,*

$$f(t) = \begin{cases} 1/50 \times t - 31834563714.72, & \text{if } t \in [t_1, t_2), \\ 223, & \text{if } t \in [t_2, t_3), \\ 1/50 \times t - 31834563878.96, & \text{if } t \in [t_3, t_4]. \end{cases}$$

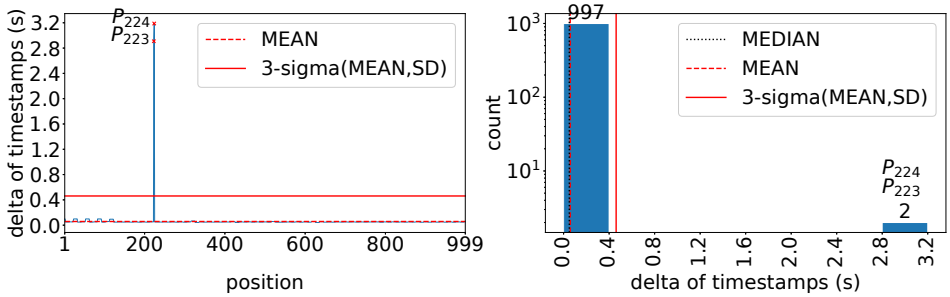
The first point has $f(1591728185786) = 1$ and the last point has $f(1591728243948) = 1000$.

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.

5.1.2 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 6. *Figure 9 plots the deltas of timestamps extracted from the data in Figure 8(c). The timestamp delta for the j-th data point P_j is $P_{j+1}.t - P_j.t$, where $1 \leq j \leq 999$. The median of the*



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

(b) Distribution of the deltas of timestamps

Fig. 9. An example for learning parameters

timestamp delta is 50ms, denoted by the dotted line in Figure 9(b), i.e., mostly collecting data in every 50ms. The slope is $K = 1/50$.

5.1.3 Learning Split Timestamps $\mathbb{S}$. The idea is to first select changing points in 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

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

where

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

Example 7 (Example 6 continued). *Only two data points,* P_{223} *and* P_{224}*, have their deltas of timestamps larger than the threshold* $\mu + 3\sigma$*. As a result, the set of changing points is* $\mathbb{P}_s = \{P_{223}, P_{225}\}$.

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, 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 5.1.1, 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

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

5.2 Point Pruning with Value Regression

In the following, we introduce point pruning for obtaining the bottom and top points by the data read operation (MV). Referring to Proposition 2, for $G \in \{BP, TP\}$, there are two cases for a candidate point P_G to be verified as non-latest. That is, P_G may be deleted by some later appended deletes, or be overwritten by some later appended updates. We unify the two cases into deletes on P_G and formalize the data read operation (MV) as follows.

Definition 9 (*BP/TP* recalculation). *For $G \in \{BP, TP\}$, given the set of chunks $\mathbb{C}$, the set of deletes $\mathbb{D}'$, and the chunk C^κ where the non-latest candidate point P_G belongs as input, the recalculation of the chunk metadata is to compute $G(M(\{C^\kappa\}, \mathbb{D}''))$, where*

$$\mathbb{D}'' = \mathbb{D}' \cup \{D^\infty_{P_G} \mid C^\kappa \in \mathbb{C}, \kappa > P_G.\kappa, P_G.t \vDash C^\kappa\}$$

and $D^\infty_{P_G}$ denotes the delete with delete time range $[P_G.t, P_G.t]$ and version number of infinity.

A straightforward idea is to iterate over all points in the chunk and apply deletes along the way to find the point with the min/max values. We propose to use value regression to prune the impossible positions for minimum/maximum. The regression model should have deterministic error bound guarantees [31, 48] for pruning.

Definition 10 (Value regression). *The value regression function $g : [1, |C^\kappa|] \to \mathbb{R}$ models the map from the relative position of a data point in the chunk to its value, having*

$$g(j) - \epsilon \le P_j.v \le g(j) + \epsilon, \ \ j = 1, ..., |C^\kappa|,$$

where ϵ is the deterministic error bound.

For simplicity, we denote the lower and upper bounds for the value of the j-th data point as $LB(j) = g(j) - \epsilon$ and $UB(j) = g(j) + \epsilon$, respectively. Then, we have the following for pruning.

Proposition 4 (Point pruning). *Given a data point $P_c \in C^\kappa$ that satisfies $P_c.t \nvDash D^\kappa, \forall D^\kappa \in \mathbb{D}''$, for any data point whose lower bound is larger than $UB(c)$, its value must be greater than that of $BP(M(\{C^\kappa\}, \mathbb{D}''))$ and thus can be pruned when recalculating BP. Likewise, for any data point whose upper bound is smaller than $LB(c)$, its value must be smaller than that of $TP(M(\{C^\kappa\}, \mathbb{D}''))$ and thus can be pruned when recalculating TP.*

Example 8. *Take Figure 10 as an example. The value regression function $g(j)$ is composed of 18 linear segments and 19 segment points. For $G = TP$, suppose that the candidate point P_G at the k-th position of the chunk is overwritten by some later appended chunks. Therefore, we have $P_k.t \vDash D^\infty_{P_G}$. To recalculate TP, we first find the segment point $(c, g(c))$, which is the point with the maximal value among all non-deleted segment points. Then, according to Proposition 4, we take $LB(c)$, the lower bound of $Pc.v$, as the pruning threshold. Next, the pruning intervals of positions that satisfy $UB(j) < LB(c)$ can be calculated analytically by comparing the linear segments with the threshold. The three regions indicating pruning intervals are colored in green in the figure. Finally, we only need to iterate over the non-pruned positions for recalculating TP as defined in Definition 9.*

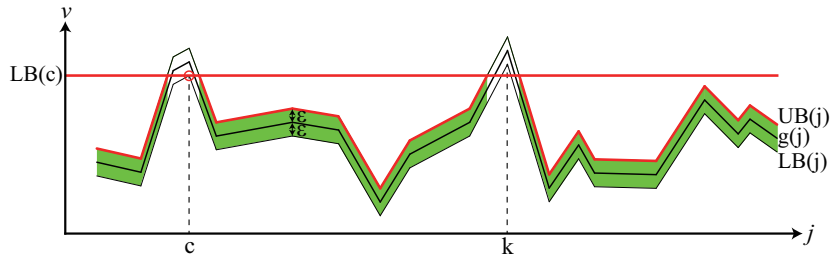


Fig. 10. Example of pruning points in green by the error bounds of value regression

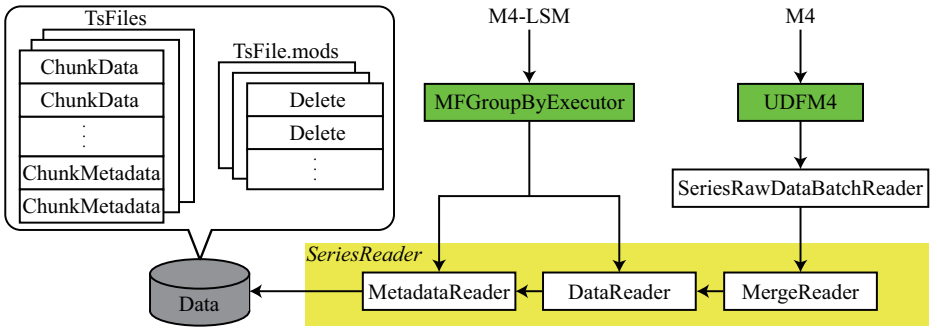


Fig. 11. System deployment in Apache IoTDB

6 SYSTEM DEPLOYMENT

This section describes the system deployment of M4-LSM in Apache IoTDB [6]. The document of the M4 function is available on the product website [7]. The source code has been committed to the GitHub repository of Apache IoTDB by system developer [3]. An overview of the deployment is shown in Figure 11. Let us first introduce some interfaces of the system in Section 6.1, upon which deployment is conducted in Section 6.2.

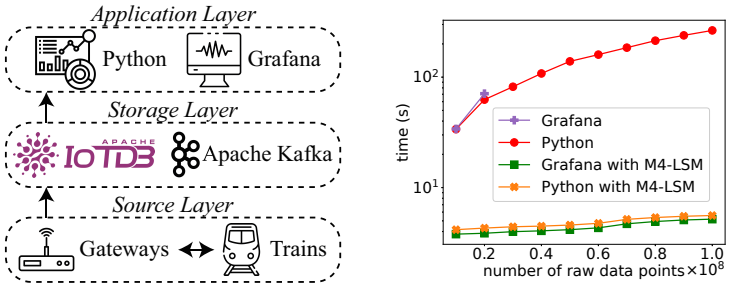
6.1 System Overview

As illustrated in Figure 11, 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-Tree store.

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

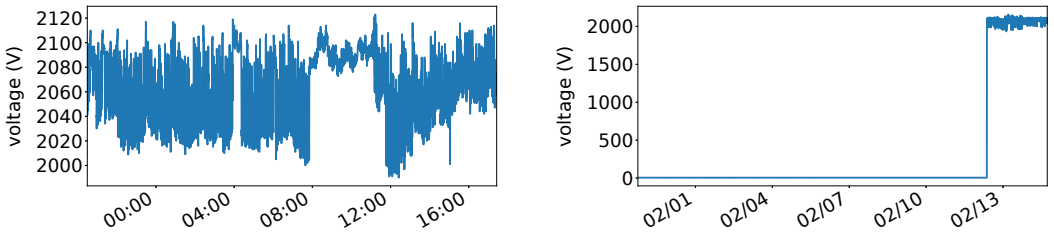
6.2 Deployment Details

We first implement the original M4 method [25] in Apache IoTDB for comparison. 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 in the original M4 design.



(a) System deployment (b) Visualization performance

Fig. 12. Case study of time series visualization



(a) M4 visualizes 8 million points (1 day time series) in 5s

(b) M4-LSM visualizes 130 million points (15 days time series) in 4s

Fig. 13. Train voltage monitoring fails to discover the ad-hoc transmission error by visualizing only one day data with M4 in (a), but successfully identifies it in 02/13 by M4-LSM returning 15 days data with a similar query time in (b)

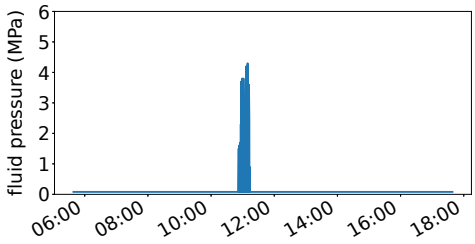
We implement a new `MFGroupByExecutor` for M4-LSM. Rather than reading the system assembled time series, the M4-LSM implementation directly uses `MetadataReader` and `DataReader`. In this way, ChunkMetadata helps in pruning ChunkData, which saves both IO and computation costs. Note that `MFGroupByExecutor`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

7 APPLICATION STUDY

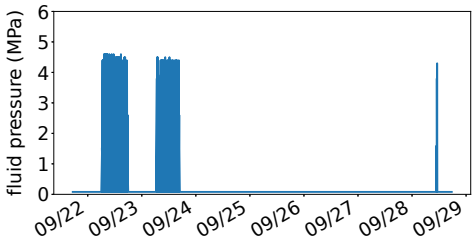
In the intelligent operation and maintenance system of urban rail vehicles, Apache IoTDB manages the sensor data collected in trains. Each train has tens of thousands of sensors installed, measuring current, voltage, pressure, speed, acceleration, temperature, and so on. Twenty trains on an urban rail line generate about 48TB time series data each year. Figure 12(a) presents an overview of system deployment in the company. Note that directly visualizing such a huge volume of data is impractical. As shown in Figure 12(b), it fails to visualize a time series with more than 20 million points in Grafana (with out-of-memory error) for fault diagnosis by domain experts. Python takes more than 265 seconds for 100 million points, which is also unacceptable in exploratory data analysis.

Unfortunately, existing time series visualization techniques such as M4 [25] cannot meet the efficiency requirement either. In the following, we present four tasks in the application to illustrate the challenges and how our proposal M4-LSM tackles the problem.

Task 1: Train Voltage Monitoring. Regular visual inspection is conducted on voltage to examine occasional behaviors. M4 can visualize the data in about one day as shown in Figure 13(a), with an

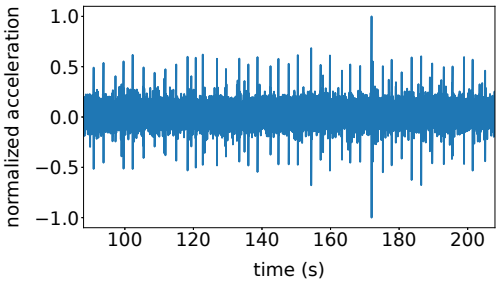


(a) M4 visualizes 8 million points (12 hours time series) in 5s

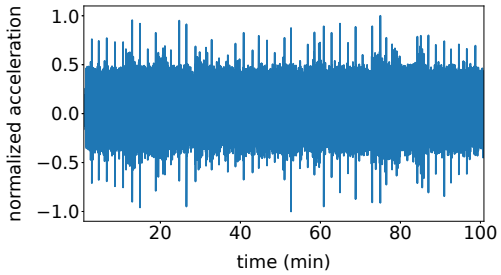


(b) M4-LSM visualizes 120 million points (7 days time series) in 4s

Fig. 14. Cutting fluid pressure diagnosis of a true anomaly at time 11:00 fails in (a) by comparing limited data, but succeeds with M4-LSM in (b) by contrasting with other spikes



(a) M4 visualizes 1.2 million points (2 mins time series) in 5s



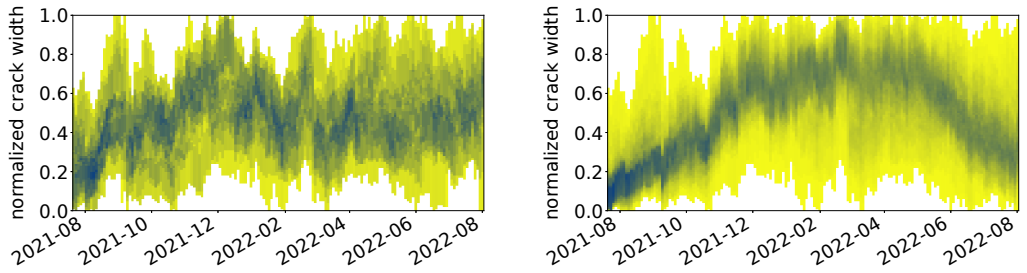
(b) M4-LSM visualizes 60 million points (1.6 hours time series) in 4s

Fig. 15. Track irregularity diagnosis fails to identify the false anomaly at time 172s by comparing limited data in (a), but succeeds in (b) by visualizing similar large pulses in the past with M4-LSM

acceptable processing time of 5s. In contrast, our more efficient M4-LSM visualizes more data in 15 days with the same processing time in Figure 13(b). As shown, there is an irregular shift on voltage found in 02/13 (caused by ad-hoc erroneous transmission from another train). It is missed by M4 which visualizes only the data in the past day, given the requirement of short processing time.

Task 2: Cutting Fluid Pressure Diagnosis. High-pressure jet cutting fluid is used in railway equipment and spare parts processing factories. When some anomaly occurs, the domain experts need to visualize and inspect the historical pressure data, to diagnose whether it is caused by normal tool changing or cutting fluid leaking. Again, the diagnosis query needs to respond in about 5 seconds, to ensure the work efficiency of domain experts. With such a time limit, as shown in Figure 14(b), our proposal M4-LSM can present the historical data in the past 7 days, and successfully illustrate that the spike in 09/28 is very different from those in 09/22 and 09/23. Domain experts highly suspect that it is caused by cutting fluid leaking rather than normal tool changing. Unfortunately, given the 5-second query response time, the original M4 can only visualize the historical data in 12 hours as illustrated in Figure 14(a). With such limited information, the domain experts are not able to diagnose whether it is a true anomaly occurring at noon on 09/28.

Task 3: Track Irregularity Diagnosis. Vehicle vertical acceleration is monitored for track irregularity inspection. Figure 15(a) visualizes the acceleration time series for 2 minutes, returned by M4



(a) M4 visualizes 315 million points (10 time series) in 1min

(b) M4-LSM visualizes 1.6 billion points (50 time series) in 1min

Fig. 16. Track crack width analysis fails to find a clear pattern by visualizing only 10 time series with M4 in a minute, but succeeds by M4-LSM showing 50 time series at a time

Table 2. Dataset summary

Dataset	Source	Entire time range	# Points
BallSpeed	M4 paper [25]	71 minutes	7,193,200
MF03	M4 paper [25]	28 hours	10,000,000
Train	IoTDB customer [10]	5 months	127,802,876
Steel	IoTDB customer [10]	7 months	314,572,100

in 5 seconds. While regular pulses are observed, there is a large pulse at around 172s. With such limited data, domain experts cannot diagnose whether it is normal. Given a similar query time limit, our more efficient M4-LSM visualizes the data for 1.6 hours, and finds that the large pulse occurs regularly in about every 2 minutes. The regular pulses in Figure 15(a) and (b) are owing to different types of rail joints, in about every 100 and 1k meters, respectively, in the railway.

Task 4: Track Crack Width Analysis. In railway track health monitoring, track crack width is analyzed. Domain experts need to visualize together multiple time series of monitored cracks, in order to find patterns. Figure 16(a) plots the DenseLines [36, 49] of 10 time series returned by M4 at a time, within the required response time of 1 minute. Unfortunately, no clear pattern is observed with such a limited number of crack time series. In contrast, Figure 16(b) visualizes 50 time series by our M4-LSM, with a similar response time. As shown, most cracks (in dark green) become larger in width from 2021-12 (winter). However, the crack widths do not drop significantly even in 2022-04, which needs further investigation of other factors like local temperature, track material types, etc.

8 EXPERIMENTS

In the experiments, we compare M4-LSM with the original M4 algorithm [25] implemented 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.

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 [8]. 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]. Train dataset is a 5-month train monitoring data collected by a vibration sensor at around 20Hz frequency. Steel dataset is 7-month steel production monitoring data collected by a vibration sensor at around 20Hz

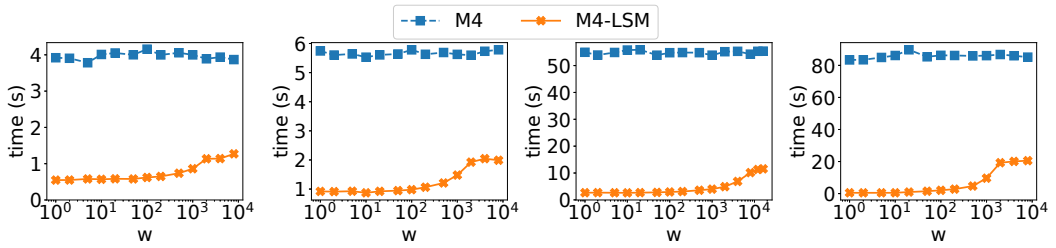


Fig. 17. Varying the number of time spans w

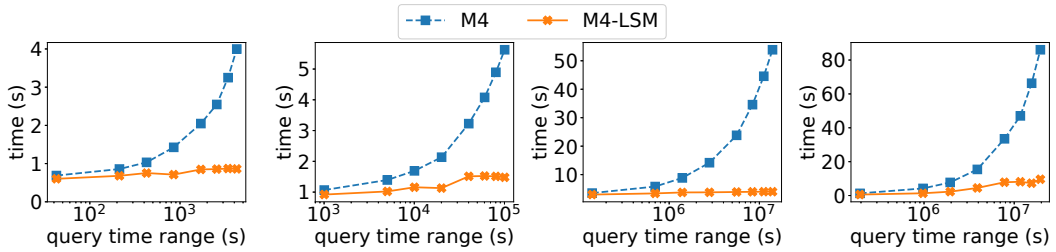


Fig. 18. Varying query time range $t_{qe} - t_{qs}$

frequency. While BallSpeed and MF03 are used in the original M4 paper [25], the large scale Train and Steel datasets are provided by real customers of Apache IoTDB and available in [10].

8.1 Experiments with Varying Parameters

8.1.1 Varying the Number of Time Spans w. This 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, e.g., a typical 4K monitor has at most 3,840 pixel columns. Figure 17 shows the representation query latency of M4 and M4-LSM under different w on four datasets.

The query time costs of M4 under different w are almost constant. It is because M4 loads all the chunks anyway, irrelevant of w. The query latency of M4-LSM increases as w increases. The reason is that 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 4s for M4-LSM to represent a time series of 127 million points in 1000 pixel columns.

8.1.2 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 shown 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 and M4-LSM under different query time range lengths on four datasets are shown in Figure 18.

With the increase of the query range length, the time costs of M4 and M4-LSM increase to varying degrees. The increase of M4 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.

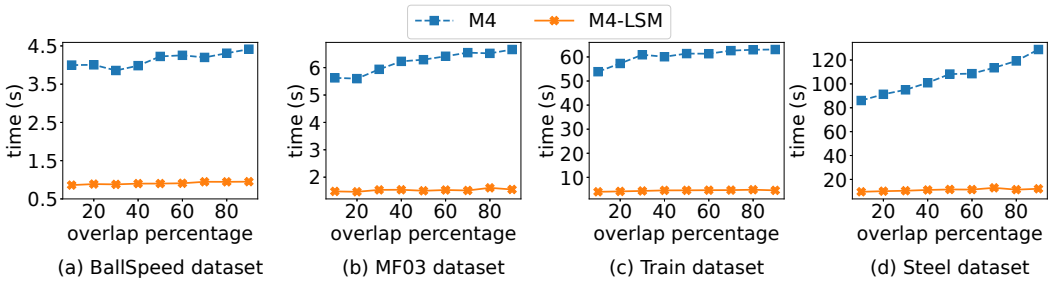


Fig. 19. Varying chunk overlap percentage

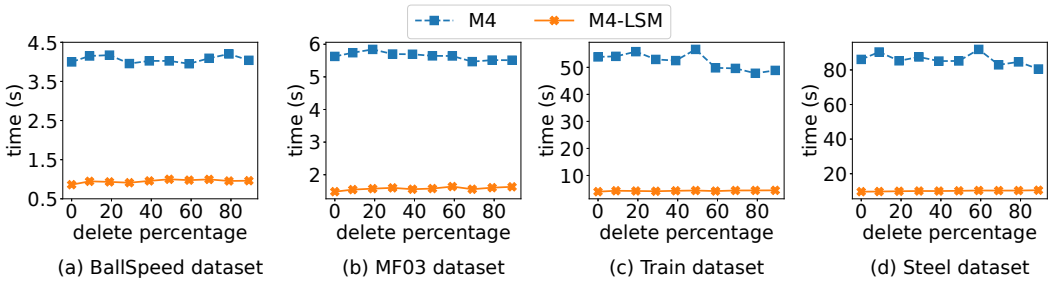


Fig. 20. Varying percentage of chunks with deletes

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.

8.1.3 Varying Chunk Overlap Percentage. In addition to M4 representation query parameters, how the data are written (updated and deleted) will affect the LSM-Tree 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 and M4-LSM under different percentages of overlapping chunks are illustrated in Figure 19.

The latency of M4 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 or deletes. The CPU cost of candidate verification for *BP/TP* is saved with the time index.

8.1.4 Varying Delete Percentage. How the data are deleted also affects the LSM-Tree storage and thus the M4 representation query. In this experiment, we evaluate the frequency of delete operations. Figure 20 shows the M4 representation query latencies of M4 and M4-LSM under different delete percentages on four datasets.

The query latency of M4 is almost constant despite the increasing number of deletes, thanks to the CPU-efficient delete sort operation [1] inherent in IoTDB. The overall time cost of M4-LSM is small. This is because the number of deleted candidate points is limited, given that the delete time range of each delete is small compared to the chunk time interval length.

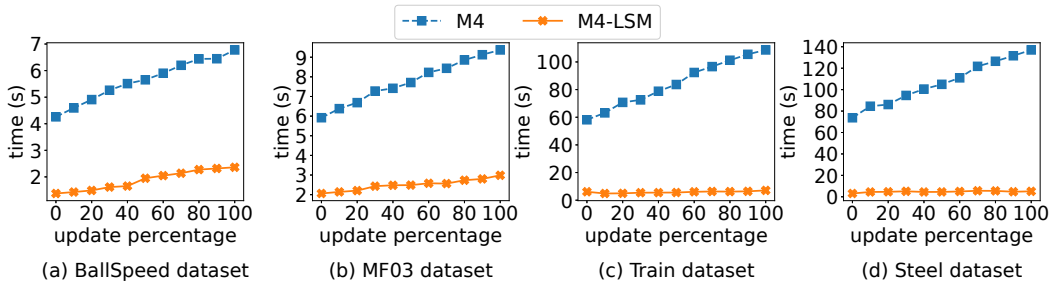


Fig. 21. Varying update percentage

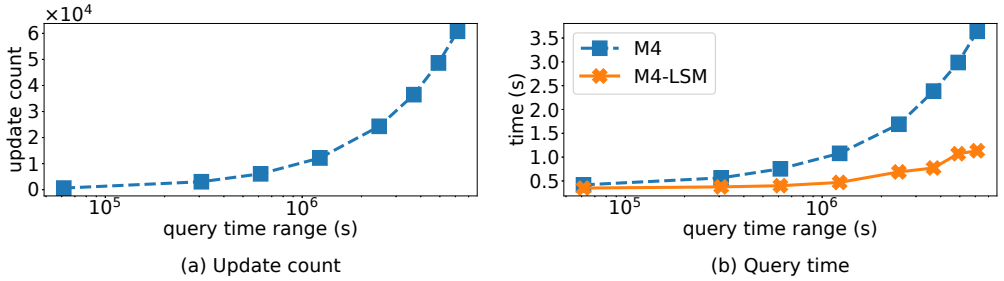


Fig. 22. Varying query time range $t_{qe} - t_{qs}$ on CQD1 (a dataset with real updates)

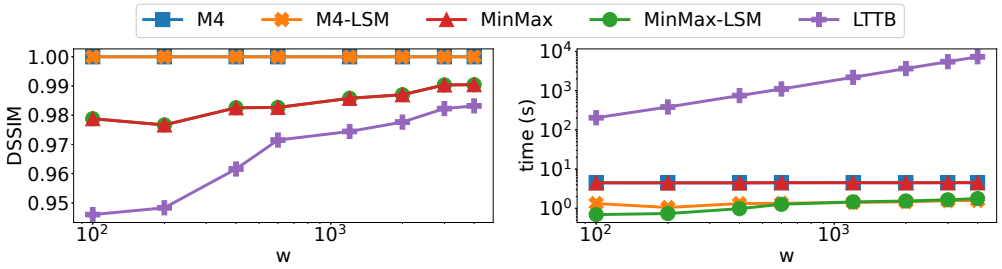
8.1.5 Varying Update Percentage. We vary the frequency of update operations to evaluate the impact on query performance. The update operation is implemented by adding a normally distributed random value with a mean of 0 to the original value. As shown in Figure 21, M4-LSM still has much better time performance than M4, under a large number of overwrites. It means more overlapped chucks as illustrated in Figure 7 in Example 4. The improvement is thus not surprising referring to Figure 19 on chunk overlap.

8.1.6 Varying Query Time Range on the Dataset with Real Updates. We consider a dataset CQD1 with updates from real usage. The time series records the average length of all the observations in every 500 milliseconds. Note that 16% observations are delayed [43]. Thereby, the computed average length of the corresponding time slot needs to be updated when the delayed observations finally arrive. Figure 22(a) shows the number of updated points in various query time ranges. Our M4-LSM still has significantly better time performance than the original M4, as in Figure 22(b), when there are more updated points. The reason is that M4-LSM is chunk merge free, without merging chunks with updates.

8.2 Applications to Other Visualizations

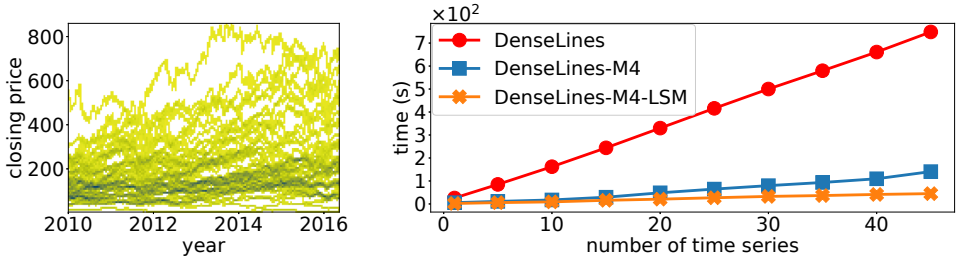
8.2.1 Apply to MinMax Representation. According to [17, 41], we further implement LTTB [40] and MinMax [25] in Apache IoTDB, and compare M4-LSM with them. Moreover, since M4 returns the first/last/bottom/top points, our approach can be naturally applied to MinMax visualization by returning only bottom/top points. Thus, we also conduct experiments to demonstrate how the proposed approach improves MinMax visualization, namely MinMax-LSM.

The visualization quality (i.e., DSSIM) comparison results in Figure 23(a) show that M4-LSM is as precise as M4, with DSSIM always equal to 1, meaning perfect (error-free) visualization. This is not



(a) DSSIM (b) Query time

Fig. 23. Comparing M4-LSM with baselines in terms of DSSIM and query time. A fair comparison is achieved by ensuring that all methods return the same number of points.



(a) DenseLines of 45 time series (b) Cost of visualizing time series (DenseLines) from a database

Fig. 24. Apply M4-LSM to DenseLines visualization

surprising because M4-LSM returns exactly the same query results as M4. Similarly, MinMax-LSM is as precise as MinMax, but with DSSIM smaller than 1. The experimental results of query time are reported in Figure 23(b). As shown, our proposed M4-LSM has high efficiency without sacrificing perfect preciseness. Moreover, MinMax-LSM, with our proposed techniques applied as aforesaid, shows clearly lower time cost than the original MinMax.

8.2.2 Apply to DenseLines Visualization. In addition to line charts, M4 can also be used to maintain the visual integrity of density-based visualizations such as DenseLines [36, 49], which use the same rendering mechanism. Thereby, M4-LSM can naturally accelerate DenseLines visualization by enabling faster M4 query processing. In this experiment, we evaluate the efficiency improvement of DenseLines visualization on large-scale time series stored in Apache IoTDB. Figure 24(a) presents an example of DenseLines, visualizing 45 stock time series together. The results in Figure 24(b) are generally similar to those in Figure 3(a). Without applying M4, the original DenseLines is extremely costly. By integrating our proposal, DenseLines-M4-LSM shows significantly lower time costs.

9 RELATED WORK

While visualizing time series is highly demanded, e.g., finding interesting patterns [33, 47], the time series database native representation operator for visualization is surprisingly absent.

9.1 Representing Time Series for Visualization

A number of time series representations have been proposed after decades of research, including sampling [13], Discrete Fourier Transform (DFT) [11], Discrete Wavelets Transform (DWT) [12],

Singular Value Decomposition (SVD) [28], Piecewise Aggregate Approximation (PAA) [27], Symbolic Aggregate approXimation (SAX) [32, 39], piecewise polynomials [31], and shapelet-based representations [20, 46]. In terms of visualization tasks, Park et al. [38] 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], as an in-DB data reduction method, is designed for the line chart, which is more suitable for the visualization of time series. Since M4 shows zero pixel error in two-color (binary) line visualization, which is impossible with other data reduction techniques such as MinMax, we focus on M4 representation in time series databases.

9.2 LSM-Tree based Storage

Log-Structured Merge Tree (LSM-Tree) [37] 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-Tree 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 [14–16]. 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-Tree systems.

10 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 disordered points in the LSM-Tree 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-Tree store. Metadata are utilized to prune chunks, which do not contain representation points for sure. To access data points in chunks that cannot be pruned, we observe the regular intervals of timestamps and introduce a step regression for efficient indexing. Moreover, we use a value regression function to prune the points that cannot be the top or bottom ones. The method has been deployed in Apache IoTDB, an open-source LSM-Tree time series database [42], and used in many companies across various industries, including rail transit, steel manufacturing, aviation industry, cloud service, etc. Extensive experiments over real-world datasets demonstrate that M4-LSM takes about 4 seconds to represent a time series of 127 million points in 1000 pixel columns, enabling instant visualization of data in four years with a data collection frequency of every second.

However, our approach cannot directly accelerate the time series visualizations like Largest-Triangle-Three-Buckets (LTTB) [40, 41]. This is because LTTB selects in each group the point with the largest effective triangle area. Therefore, such triangle representations require maintaining different statistics to avoid merges. We leave this extension as future work. As discussed in [24], visualization-driven data aggregation queries for scatter plots select the last record per pixel. Since M4 aggregation for line charts is at the pixel column level, M4-LSM cannot be directly used for scatter plots. We leave the extension of our techniques to support scatter plots also as future work.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Plan (2021YFB3300500), the National Natural Science Foundation of China (62072265, 62021002, 62232005, 92267203), and Beijing National Research Center for Information Science and Technology (BNR2022RC01011). 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/thssdb/M4-LSM.
[5] https://github.com/thssdb/M4-LSM/blob/supplement/supplement.pdf.
[6] https://iotdb.apache.org.
[7] https://iotdb.apache.org/UserGuide/V1.1.x/Operators-Functions/Sample.html#m4-function.
[8] https://www.iis.fraunhofer.de/en/ff/lv/dataanalytics/ek/download.html.
[9] https://www.influxdata.com/.
[10] https://www.kaggle.com/datasets/xxx123456789/exp-datasets.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] J. V. D. Donckt, M. J. V. D. Donckt, M. Rademaker, and S. V. Hoecke. Data point selection for line chart visualization: Methodological assessment and evidence-based guidelines. *CoRR*, abs/2304.00900, 2023.
[18] P. Esling and C. Agón. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, 2012.
[19] C. Fang, S. Song, and Y. Mei. On repairing timestamps for regular interval time series. *Proc. VLDB Endow.*, 15(9):1848–1860, 2022.
[20] 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.
[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, Kuala Lumpur, Malaysia, May 9-12, 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, mar 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] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[35] 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.

[36] D. Moritz and D. Fisher. Visualizing a million time series with the density line chart. *CoRR*, abs/1808.06019, 2018.

[37] 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.

[38] 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.

[39] 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.

[40] S. Steinarsson. Downsampling time series for visual representation. Master's thesis, University of Iceland, 2013.

[41] J. Van Der Donckt, J. Van der Donckt, E. Deprost, and S. Van Hoecke. Plotly-resampler: Effective visual analytics for large time series. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pages 21–25. IEEE, 2022.

[42] 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*, 1(2):195:1–195:27, 2023.

[43] W. Weiss, V. J. E. Jiménez, and H. Zeiner. A dataset and a comparison of out-of-order event compensation algorithms. In M. Ramachandran, V. M. Muñoz, V. Kantere, G. B. Wills, R. J. Walters, and V. Chang, editors, *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security, IoTBDS 2017, Porto, Portugal, April 24-26, 2017*, pages 36–46. SciTePress, 2017.

[44] 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(10):2148–2160, 2022.

[45] H. Yang, J. Fang, M. Cai, and Z. Cai. A prefetch-adaptive intelligent cache replacement policy based on machine learning. *J. Comput. Sci. Technol.*, 38(2):391–404, 2023.

[46] 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.

[47] H. Yu, X. Guo, X. Luo, W. Bian, and T. Zhang. Construct trip graphs by using taxi trajectory data. *Data Sci. Eng.*, 8(1):1–22, 2023.

[48] D. Zhang, M. Ding, D. Yang, Y. Liu, J. Fan, and H. T. Shen. Trajectory simplification: An experimental study and quality analysis. *Proc. VLDB Endow.*, 11(9):934–946, 2018.

[49] Y. Zhao, Y. Wang, J. Zhang, C. Fu, M. Xu, and D. Moritz. Kd-box: Line-segment-based kd-tree for interactive exploration of large-scale time-series data. *IEEE Trans. Vis. Comput. Graph.*, 28(1):890–900, 2022.

Received July 2023; revised October 2023; accepted November 2023