Backward-Sort for Time Series in Apache IoTDB

Xiaojian Zhang
Tsinghua University
xj-zhang21@mails.tsinghua.edu.cn

Hongyin Zhang
Tsinghua University
zhanghy22@mails.tsinghua.edu.cn

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

Xiangdong Huang
Timecho Ltd
hxd@timecho.com

Chen Wang
Timecho Ltd
wangchen@timecho.com

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

***Abstract*—While time series data are naturally ordered by timestamps for efficient storage and query processing, the data points in a time series often come out-of-order. We identify two unique features of out-of-order arrivals in Apache IoTDB, i.e., delay-only and not-too-distant. It is not surprising that data points can only be delayed but should never come "earlier" before the generation of its succeeding ones. Moreover, the system employs a separation policy to handle those points delayed for a very long period, and thus only sorts data points delayed to not-too-distant future. Motivated by such unique features, we devise a new algorithm for sorting time series data, Backward-Sort. Intuitively, the delay-only feature leads to the strategy of moving points backward in sorting. Moreover, the not-too-distant feature results in blocks of data points, such that moving points are expected to occur locally inside the blocks. To our best knowledge, this is the first sorting algorithm specially designed for out-of-order arrivals in time series. The algorithm becomes a fundamental component of sorting time series data in Apache IoTDB. The evaluation is conducted over real and synthetic datasets, using IoTDB-benchmark.**

I. Introduction

Apache IoTDB[1] is an open-source time series data management system, developed upon our preliminary study [1]. Data points in a time series are naturally ordered by their timestamps for efficient storage and query processing in the database. Unfortunately, the data points are often delayed on arrival, very prevalent in IoT scenarios, e.g., due to network fluctuations, system failure and so on [2]–[4]. Therefore, ordering data points by timestamps is a fundamental component of Apache IoTDB. While the preliminary study [5] reduces the write amplification on disk in the system owing to out-of-order arrivals, in this paper, we propose to optimize the sorting by timestamps in memory for efficient query processing and flushing.

We first identify the unique features of out-of-order arrivals in the IoT scenarios, i.e., (1) delay only (2) in not-too-distant future. First, note that the time series data are generated in time order, often by IoT devices. That is, a point is impossible to come "earlier" than the generation of its succeeding ones, instead can only be delayed. Moreover, the delay is usually not very distant. The reason is that Apache IoTDB employs a separation policy to handle respectively those points delayed for a very long period [5]. In other words, a point is probably delayed to not-too-distant future.

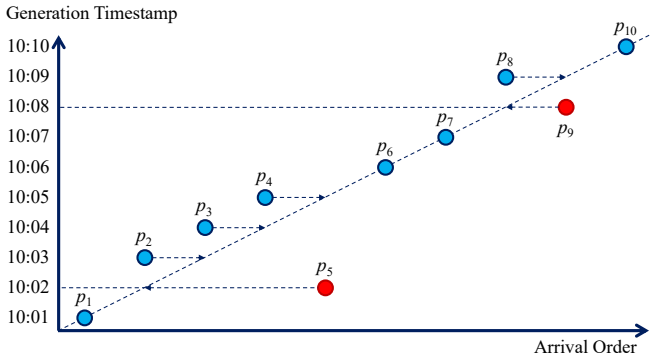

Fig. 1. Example of out-of-order arrivals in Apache IoTDB, where p_5 and p_9 are delayed points

Example 1. *Figure 1 illustrates the unique features of out-of-order arrivals, i.e., delay only, in not-too-distant future. The points $p_1, \ldots, p_{10}$ are labeled referring to their arrival orders. As shown, the points do not arrive in the order of their generation timestamps. For instance, p_5 with generation timestamp* 10:02 *is delayed, as well as p_9 with generation timestamp* 10:08*. To sort the points by timestamps, we need to move backward p_5 and p_9.*

Moreover, p_5 is not delayed for a long period. It only needs to swap with p_2, p_3, p_4. Likewise, p_9 only needs to swap with p_8 with timestamp 10:09 *in a near future.*

Motivated by the aforesaid unique features of out-of-order arrivals, we devise a new algorithm of ordering by timestamps for time series. (1) Referring to the feature of delay-only, we propose a backward strategy, i.e., moving points backward in sorting. (2) For the other feature of not-too-distant, we may divide the data points in blocks, such that moving points is expected to occur locally inside the blocks. For instance, for two blocks of the first 5 points $p_1, \ldots, p_5$ and the last 5 points $p_6, \ldots, p_{10}$, the swapping on p_5 occurs in the first block, while swapping p_9 is in the second. By moving points locally in blocks, it is not surprising that the sorting cost could be reduced.

Shaoxu Song (https://sxsong.github.io/) is the corresponding author.
[1]https://iotdb.apache.org/

TABLE I
THE NOTATIONS AND EXPLANATIONS

Sym.	Explanations
X	The Time Series
N	The maximum points that TVList can hold in memory
L	The block size
τ	The delay of the time series
$\Delta\tau$	The delay difference between two points
Θ	The threshold that IIR cannot surpass
α_L	The interval inversion ratio (IIR) with length L
Q_L	The expected overlap length of two adjacent blocks
P	The expected times of finding block size loops

To the best of our knowledge, this is the first algorithm specially designed for sorting time series data. Our major contributions in this paper are summarized as follows.

(1) We analyze the delay-only, not-too-distant features of out-of-order arrivals, over real-world time series data.

(2) We devise a new algorithm of time series data sorting, Backward-Sort, with backward and blocking strategies for handling the delay-only and not-too-distant features.

(3) We analyze the algorithm performance, where Quicksort is indeed the worst case of our proposal.

(4) We implement the algorithm as the time series sorting component of Apache IoTDB[2].

(5) We evaluate the performance over real and synthetic datasets, using IoTDB-benchmark.

The paper is organized as follows. Section II introduces the preliminary of out-of-order time series. Section III presents the design details of the sorting algorithm. Section IV analyzes the complexity of Backward-Sort. Section V describes the implementation details in Apache IoTDB. Section VI conducts the extensive experiments. And finally, Section VII discusses related studies and Section VIII concludes the paper.

II. FEATURES OF OUT-OF-ORDER ARRIVALS

Out-of-order time series are common in the Internet of Things due to many reasons, such as network fluctuations [2], clock skew [6], system failure [7], etc. The common feature of these disorders is that they all cause data delay. Moreover, since separation policy is applied in Apache IoTDB, any timestamp smaller than the current flushing time will be ingested into the unsequence memtable. Therefore, extreme delays like system recovery from failure are not what we focus on.

A. Preliminary

Definition 1 (Time Series)**.** *A time series X is a collection of tuples, $X = \{p_0, p_1, ..., p_N\}$ where N denotes the current size. $p_i = (t_i, v_i)$ represents the data point containing a timestamp and value, with the array index i denoting its arrival order.*

[2]The code of the sorting algorithm is available in the GitHub repository of Apache IoTDB https://github.com/apache/iotdb/pull/7410

Defining the degree how an array is out-of-order is a more difficult problem. Inversion(*Inv*) is widely-used to measure out-of-order.

Definition 2 (Inversion)**.** *Given any two data points, p_i, p_j, if $i < j$ and $t_i > t_j$, (i, j) is regarded as one inversion.*

The number of X's inversions can measure out-of-order since it is minimized as X is sorted. However, Inversion aggregates all counts of inversions with different lengths.

Definition 3 (Interval Inversion)**.** *Given an interval L any data point p_i if $t_i > t_{i+L}$, (i, L) is regarded as one interval inversion with interval L.*

Generally, we use the ratio of the interval inversion number to the total pair number to measure the out-of-order.

Definition 4 (Interval Inversion Ratio (α))**.** *Given an interval L, the number of interval inversions with interval L equals C. Then the interval inversion ratio $\alpha = C/(N - L)$.*

To simplify the analysis, we treat the equally spaced time series data as the standard interval of **1**, and the corresponding delay distribution is also based on the interval of 1. In actual data analysis, the corresponding interval can be enlarged or reduced proportionally

Definition 5 (Delay)**.** *The order of time series is determined by the generation time t plus the delay time τ where t is evenly distributed at equal intervals(1). The delay $\tau_i, i \in \{0, 1, ..., n\}$ follows an independent and identically distribution $\mathcal{D}$.*

The distribution $\mathcal{D}$ has an absolute dominant effect on the degree of out-of-order. Intuitively, if the delay is large and randomly distributed, there must be many out-of-order points.

Definition 6 (Delay Difference)**.** *since the delay is IID, the inversion relation between two data points depends on the delay time τ_i, τ_j. We define delay difference $\Delta\tau = \tau_i - \tau_j$.*

The distribution of $\Delta\tau$ depends on the delay time distribution. The delay difference is strongly related to the overlap length of adjacent blocks.

B. Time Series Characteristics

Before introducing the Backward-Sort algorithm, it is worth emphasizing the characteristics of the out-of-order situation in industrial time series scenarios, that the algorithm design is inspired by.

1) Incrementally Nearly Sorted: Different from traditional data, in the industrial time series data scenarios, despite inherent disturbance that may cause out-of-order, all time series data is incrementally nearly sorted. The reason is that data are collected over time. Even if there are out-of-orders, the timestamps must be in an increasing trend as a whole.

As shown in Figure 1, the timestamp in time series data is incremental naturally, not decremental. Take Quicksort as an example, if we choose one pivot, then every element will be compared with the pivot to ensure that the left elements are

smaller than it. However, in time series, the earlier the timestamp, the more likely the pivot is greater than it. Quicksort compares all data equally with it, which is not optimal and may introduce unnecessary comparison operations.

2) Delay-Only: Nevertheless, "incrementally nearly sorted" is not sufficient to fully capture the characteristics of out-of-order time series in real scenarios.

Besides "nearly sorted", "delay-only" needs to start from the cause of out-of-order. As mentioned in Section 2, the main causes of disorder are network delay, time skew, etc. Intuitively, most unordered data points will appear in the form of "delay", that is, unordered data points can be regarded as forced to move behind the data "externally".

Therefore, it is obvious that the data cannot appear "ahead". If it does appear in the data ahead of time, it must be the simultaneous "delay" of many subsequent data points that causes it to appear "ahead", which is a slight probability event.

As shown in Figure 1, p_5, p_9 are delayed with no data points appearing ahead. The real log datasets (AndroidLog and CloudLog) [3] both show the nature of the "delay-only" although they are not available now. "Delay-only" is the primary characteristic that distinguishes time series from other nearly sorted data.

III. Backward Algorithm

A. Overview

1) Sort by Blocks: : The reason for sorting by blocks is to reduce unnecessary comparisons.

Example 2. *In Figure 1, a comparison between p_5 and p_1 or p_{10} is not as necessary as that between p_5 and p_6. The reason is that the arrival order offers some useful information, i.e., p_5 is more likely smaller than p_{10}, and thus $P(p_5 < p_1) < ... < P(p_5 < p_6) < ... < P(p_5 < p_{10})$.*

Therefore, dividing the data into blocks and sorting them all alone can effectively reduce unnecessary comparisons and ensure that the elements that are far apart and more likely to be followed by larger elements are not compared.

2) Backward Merge: : In time series, most disorders come from network delay, system failures, etc. As we mentioned earlier, the "delay-only" characteristic means that few elements will be delayed for a long time. Although they are few, this will proportionally increase the number of move operations in the merging process.

Example 3. *Figure 2 demonstrates the superiority of the backward merge over the traditional straight merge. As shown, data points with timestamps 1 and 3 arrive late. Thereby, they are deferred to the front of the following blocks. Figure 2 I and II show the process of Straight Merge and Backward Merge, respectively, and the corresponding number of movement operations. Let M denote the length of the array block.*

For Straight Merge, it processes the first two blocks and the last two , separately. There are $M+2$ moves for each merge, in which 2 comes from the fact that the 3 is first moved into the additional space and then moved back into the array. The last merge needs $2M$moves, and it is worth noting that the first block is moved again, causing redundant moves.

For Backward Merge, it processes the blocks backward. The number of moves, in processing order, is $M+2$, $M+1$ and $M+4$. The only redundant moves come from 3.

Finally, the total moves in Straight Merge are $4M+4$ while that in Backward is $3M+7$. In other words, the Backward Merge achieves about a 25% reduction of moves in such a case.

Previously, the quantitative measurement of out-of-order data has been well studied [8]–[10], like Inv, Dis, Runs, etc. Straight Insertion Sort is adaptive with respect to Inv, while Patience Sort is designed based on the runs.

Interval Inversion Ratio: As defined in Section II-A, the measures of inversion are extended to interval inversion ratio, so as to measure the probability that different blocks may overlap in the case of block division. The interval inversion ratio α of multi-layer intervals map to the degree of out-of-orders.

Example 4. *In Figure 3, the interval inversion ratios can be calculated as follows.*

$$\alpha_1 = \frac{|\{(4,3),(9,8),(8,5),(11,1),(12,7),(15,2)\}|}{|N-1|} = \frac{6}{14} \tag{1}$$

$$\alpha_3 = \frac{|\{(6,5),(8,1),(12,2),(11,1)\}|}{|N-3|} = \frac{4}{12} \tag{2}$$

$$\alpha_5 = \frac{|\emptyset|}{|N-3|} = \frac{0}{10} \tag{3}$$

3) Set Block Size: The core of the algorithm is to choose the appropriate block size, which directly determines the complexity of the algorithm. In the actual algorithm, collecting that sufficient interval inversion ratio for each block size is time-consuming. Therefore, down-sampling is used to roughly determine the size of the interval inversion rate.

Example 5. *In Figure 3, the empirical interval inversion ratios can be calculated as follows.*

$$\tilde{\alpha_3} = \frac{|\{(12,2)\}|}{|\{(4,9),(9,11),(11,12),(12,2)\}|} = \frac{1}{4} \tag{4}$$

$$\tilde{\alpha_5} = \frac{|\emptyset|}{|\{(4,5),(9,10),(11,15),(12,16)\}|} = \frac{0}{4} \tag{5}$$

B. Pseudo-code and Explanation

Figure 4 shows an overview of Backward-Sort in Algorithm 1. The algorithm is mainly divided into three parts: set block size L, sort in each block, and backward merge of blocks, as also shown in Figure 4. When the Backward Sort algorithm merges adjacent blocks, e.g., in the right part of Figure 4, only the end of the first block (3,4,6,8) and the beginning of the second block (2,5,7) need to be merged. It needs a certain amount of extra space to store overlapping points.

Line 1-8 in Algorithm 1 refers to the part "set block size". Line 9-12 represents "sort by blocks" where Quicksort is used

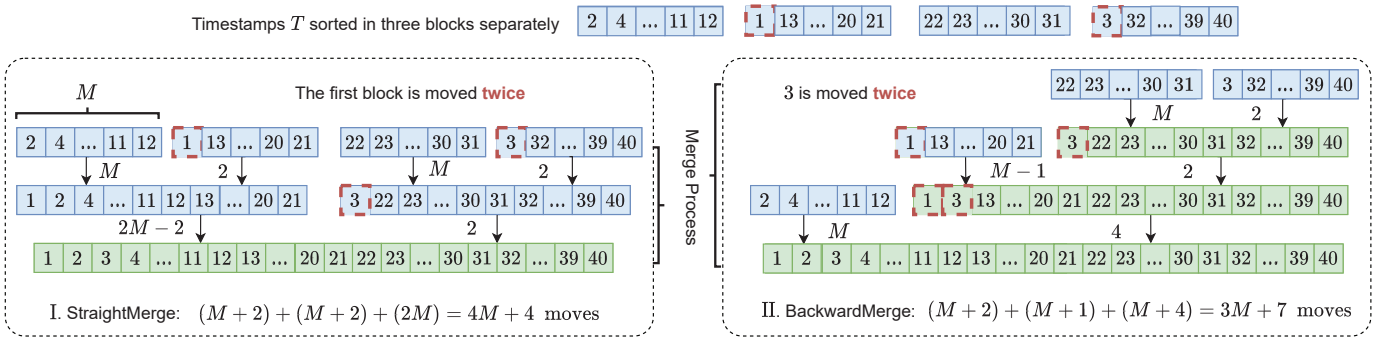

Fig. 2. Comparison between Straight and Backward Merge

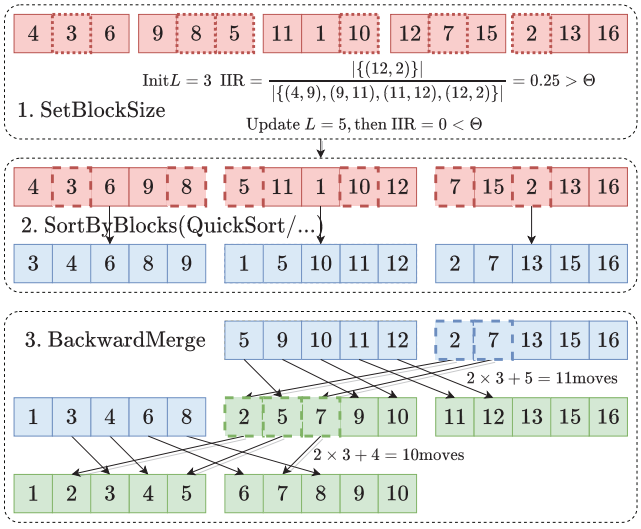

Fig. 3. The example of Backward Sort algorithm

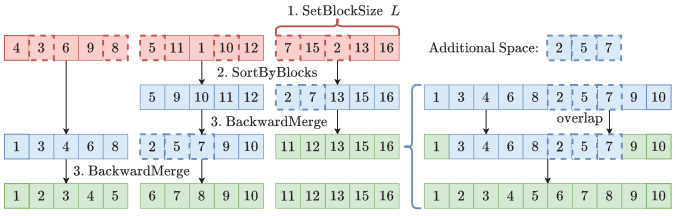

Fig. 4. The overall architecture of Backward Sort algorithm.

in default and can be substituted by other algorithms. Line 13-16 means the "backward merge" needs to find the overlapped blocks firstly and then merge the overlapped blocks. In order to minimize k and reduce the number of repeated blocks, L is better to be larger. However, L is better to be smaller in "sort by blocks". Therefore, there is the trade-off between "sort by blocks" and "backward merge", which is controlled by "set block size".

IV. Performance Analysis

We can derive from Algorithm 1 that it runs $O(\frac{n}{L}P + n\log L + \frac{n}{L}Q)$ time, where P is the number of recursions to find the appropriate block size, and Q denotes the overlapped length between the adjacent sorted arrays in average. In practice, however, it is not easy to directly obtain the specific sizes of P and Q. Thereby, in this section, we first give and prove Proposition 2, 3 and 4 to illustrate the relationship between the delay distribution $\mathcal{D}$ and the interval inversion ratio α. Based on this, we present the time complexity of Backward Sort algorithm w.r.t. the initial block size L_0 in Proposition 6.

Algorithm 1 Backward-Sort

Input: Time series X with size N
Input: Block inversion ratio threshold Θ
1: initial block size $L = L_0$
2: **while** $L <= N$ **do**
3: $\alpha \leftarrow$ getInversionRatioBetweenBlocks(L)
4: **if** $\alpha < \Theta$ **then**
5: break
6: **end if**
7: $L \leftarrow$ updateBlockSizeByRatio(L, α, Θ)
8: **end while**
9: set block number $B = \lfloor N/L \rfloor$
10: **for** $i \leftarrow 1, B$ **do**
11: Quicksort($block_i$)
12: **end for**
13: **for** $i \leftarrow B-1, 1$ **do**
14: $k \leftarrow$ findOverlappedBlock($block_i$)
15: BackwardMerge($block_i, block_{[i+1,\ldots,k]}$)
16: **end for**

A. Analysis for Delay Difference

Proposition 1. *The probability density function of $\Delta\tau$, $f_{\Delta\tau}(t)$, is an even function.*

Proof. As indicated in Section II-A, τ_i, τ_j represent any two i.i.d delay. To obtain the $f_{\Delta\tau}(t)$ with given t, if τ_j is known, then τ_i is naturally restricted. Therefore, the integral only needs to accumulate τ_j, leading to Equation 6. Likewise, Equation 7 can be easily obtained by replacing t with $-t$ in Equation 6.

$$f_{\Delta\tau}(t) = \int f_\tau(\tau_i - \tau_j = t|\tau_j) f_\tau(\tau_j) d\tau_j \tag{6}$$
$$= \int f_\tau(\tau_j + t) f_\tau(\tau_j) d\tau_j$$
$$f_{\Delta\tau}(-t) = \int f_\tau(\tau_i + t) f_\tau(\tau_i) d\tau_i \tag{7}$$

□

The even function property of the probability density function means that we can analyze only the non-negative part of $\Delta\tau$. In practice, when estimating the interval inversion ratio α, the statistics of non-negative $\Delta\tau$ is sufficient. Moreover, $\Delta\tau$'s distribution is closely related to the degree of inversion, because whether two points constitute a reverse order depends on whether their delay difference $\Delta\tau$ is greater than their original interval.

Proposition 2. *The expected value of interval inversion ratio (IIR) α with interval L equals to the probability of $\Delta\tau \geq L$, a.k.a $\bar{F}_{\Delta\tau}(L)$, where $\bar{F}$ denotes the tail distribution.*

Proof. Without loss of generality, choose $i, j = i + L$ to consider their possible inversion relationship.

As Equation 8 shows, the possibility of the interval inversion between two arbitrary points is equal to the tail distribution of $\Delta\tau$. $I\{t_i > t_{i+L}\}$ is the interval inversion indicator, whose expected value is the probability of being an interval inversion. Equation 9 further proves the equivalence between the expected interval inversion ratio $E(\alpha_L)$ and $\Delta\tau$'s tail distribution $\bar{F}_{\Delta\tau}(L)$.

$$P(t_i > t_{i+L}) = P(i + \tau_i > i + L + \tau_{i+L}) = P(\Delta\tau > L) \tag{8}$$

$$E(\alpha_L) = E(\frac{\sum_{i=1}^{n-L} I\{t_i > t_{i+L}\}}{n-L}) = \frac{\sum_{i=1}^{n-L} E(I\{t_i > t_{i+L}\})}{n-L} \tag{9}$$
$$= \frac{\sum_{i=1}^{n-L} P(t_i > t_{i+L})}{n-L} = \frac{\sum_{i=1}^{n-L} P(\Delta\tau > L)}{n-L}$$
$$= P(\Delta\tau > L) = \bar{F}_{\Delta\tau}(L)$$

□

Proposition 2 reveals the correlation between delay distribution and the interval inversion ratio. It provides a theoretical basis for measuring the degree of disorder by empirical interval inversion ratio α. To illustrate Proposition 2, the delay Example 6 of exponential distribution is given. That is, the interval inversion ratio α_L is equivalent to the tail distribution $\bar{F}_{\Delta\tau}(L)$.

Example 6. *Suppose the delay τ follows exponential distribution $\mathcal{E}(\lambda)$, $f_\tau(t) = \lambda e^{-\lambda t}$. We can derive the $\Delta\tau$' distribution as shown below.*

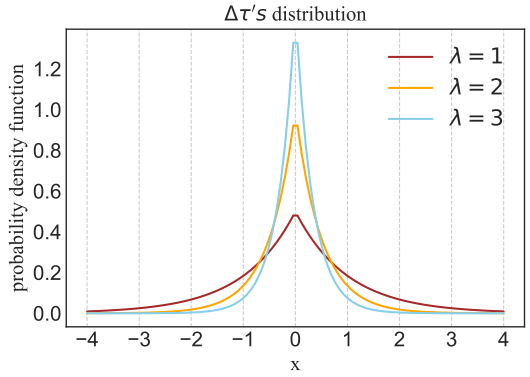

Fig. 5. The PDF of $\Delta\tau$ when $\tau \sim \mathcal{E}(\lambda)$

$$f_{\Delta\tau}(t) = \int_{max(0,-t)}^{\infty} f_\tau(t + \tau_j) f_\tau(\tau_j) d\tau_j \tag{10}$$
$$= \begin{cases} \int_0^\infty \lambda^2 e^{-\lambda(2\tau_j + t)} d\tau_j = \frac{1}{2}\lambda e^{-\lambda t}, & t \geq 0 \\ \int_{-t}^\infty \lambda^2 e^{-\lambda(2\tau_j + t)} d\tau_j = \frac{1}{2}\lambda e^{\lambda t}, & t < 0 \end{cases}$$
$$E(\alpha_L) = \int_L^\infty \frac{1}{2}\lambda e^{-\lambda t} dt = \frac{1}{2e^L} \tag{11}$$

The PDF $f_{\Delta\tau}$ is shown in Figure 5. By artificially adding exponentially distributed delay times for 10^8 points, we count α at intervals of 1 and 3 with $\lambda = 2$. The empirical and theoretical results are as follows.

$$\tilde{\alpha}_1 = 0.067696, \quad \alpha_1 = \frac{1}{2e^2} = 0.067668, \tag{12}$$
$$\tilde{\alpha}_5 = 2.280 \times 10^{-5}, \quad \alpha_5 = \frac{1}{2e^5} = 2.270 \times 10^{-5} \tag{13}$$

B. Separate Complexity

1) Set Block Size: The update process of the block size L can be summarized as an iterative renewal. Recall that Θ is a customized parameter, indicating the threshold of interval inversion ratio. Let t represent the number of iterations, and $L^{(t)}$ be the block size in the t-th iteration. The interval inversion ratio $\alpha^{(t)}$ is estimated by Equation 14, according to its definition.

$$\alpha^{(t)} = P(\Delta\tau > L^{(t)}) \tag{14}$$

When $\alpha^{(t)} > \Theta$, we update $L^{(t)}$ as Equation 15 shows; otherwise, the loop terminates.

$$L^{(t+1)} = 2 \times L^{(t)} \tag{15}$$

How many iterations needed depends on how fast α reaches the desired threshold Θ as Equation 15 shows. Although the loop times differ for different distributions, it is adequate to derive the upper bound of the iterative process's complexity, which is independent of the distributions

Proposition 3. *The complexity of the iterative process is $O(\frac{n}{L_0})$ and the maximal number of loops is $\log(\frac{n}{L_0})$.*

Proof. Suppose the loop stops at the Tth time, since $\forall 0 \leq t < T, \alpha^{(t)} > \Theta$, it is easy to draw $\forall 0 < i < T, \beta \leq 2, L^{(t)} \geq 2L^{(t-1)}$. The total number of points to be scanned is limited as shown in Equation16. When $L^{(0)}$ grows exponentially to n, it takes at most $\log(\frac{n}{L_0})$ iterations.

$$\sum_{t=0}^{T} \frac{n}{L^{(t)}} \leq \sum_{t=0}^{T} \frac{1}{2^t} \frac{n}{L^{(0)}} \leq 2\frac{n}{L^{(0)}}. \tag{16}$$

□

Therefore, the complexity of the process of heuristically finding the favorable block size is $O(\frac{n}{L_0})$.

Proposition 3 indicates that the complexity of "set block size" part is low and the process takes relatively short time. Since L is exponentially increasing, the while loop ends in $\log(\frac{n}{L_0})$ times. In other words, the amount of scanned data is small.

Nevertheless, it is worth noting that there is always an unavoidable error between the empirical interval inversion ratio α and the real expected one. Therefore, during the estimation, there exists a certain error between the optimal value and the real one.

2) Sort By Blocks: The complexity for sorting blocks is $O(n \log L)$ with no doubt while B denotes the number of blocks partitioned by L in Equation 17. The actual time of sorting an array may get lower for nearly-ordered time series.

$$\sum_{i=1}^{B} O(L \log L) = O(BL \times \log L) = O(n \log L) \tag{17}$$

3) Backward Merge: The key point behind Backward Merge is the overlapped length between adjacent blocks as shown in Figure 4.

Proposition 4. *The expected value of overlapped length Q between adjacent blocks is less than or equal to the expected value of non-negative $\Delta\tau$, i.e., $E(Q) \leq E(\Delta\tau \mid \Delta\tau \geq 0)$.*

Proof. Recall that L denotes the block size and Q is the expected value of overlap. Given a data point m, the inversions, represented by indicator $I\{t_i > t_m\}$, from the first point to the $(m-1)$-th points is calculated cumulatively in Equation 18. The derivation in Equation 18 is similar to Equation 9.

$$\begin{aligned} E(Q) &= E(\lim_{m\to\infty} \sum_{i=0}^{m-1} I\{t_i > t_m\}) && (18)\\ &= \lim_{m\to\infty} \sum_{i=0}^{m-1} P(\Delta\tau > m-i) \\ &= \lim_{m\to\infty} \sum_{i=0}^{m-1} \bar{F}_{\Delta\tau}(m-i) && (19)\\ &= \lim_{m\to\infty} \sum_{k=0}^{m-1} \bar{F}_{\Delta\tau}(k) \end{aligned}$$

If $\Delta\tau$ is discrete, the accumulation of tail distribution function $\bar{F}_{\Delta\tau}(k)$ equals $E(\Delta\tau \mid \Delta\tau \geq 0)$ as shown in Equation 20.

$$E(Q) = \sum_{k=0}^{\infty} \bar{F}_{\Delta\tau}(k) = E(\Delta\tau \mid \Delta\tau \geq 0) \tag{20}$$

Finally, as indicated in Equation 21, the accumulation for any tail distribution is less than or equal to $E(\Delta\tau \mid \Delta\tau \geq 0)$.

$$E(Q) \leq \int_0^{\infty} \bar{F}_{\Delta\tau}(t)dt = E(\Delta\tau \mid \Delta\tau \geq 0) \tag{21}$$

□

Proposition 4 shows that the overlapped length Q could be estimated with the interval inversion ratio α, referring to the relevance between interval inversion ratio α and $\Delta\tau$. If Q is estimated, it can be used to determine the optimal block size L, and thus optimizing the "backward merge" process. Example 7 illustrates and verifies of the inequality in the Proposition 4, where the overlapping length and interval inversion ratio is shown under a specific delay distribution.

Example 7. *Suppose the delay τ obeys discrete distribution $P(\tau = k) = \frac{1}{4}, k \in \{0,1,2,3\}, E(\Delta\tau|\Delta\tau \geq 0) = \frac{10}{16}$, then $E(Q)$ is $\frac{10}{16}$, equal to $E(\Delta\tau|\Delta\tau \geq 0) = \frac{5}{8}$.*

$$\begin{aligned} E(Q) &= \sum_{j=0}^{3} \sum_{i=L-3}^{L-1} P(\tau_i - j \geq L - i) P(\tau_L = j) && (22)\\ &= \sum_{i=L-3}^{L-1} \bar{F}_{\Delta\tau}(L-i) \sum_{j=0}^{3} P(\tau_L = j) \\ &= (\frac{6}{16} + \frac{3}{16} + \frac{1}{16}) \times 1 \\ &= \frac{5}{8} \end{aligned}$$

Proposition 5. *The time complexity of Backward-Sort is $O(n^2)$ when L is set to 1. The time complexity is $O(n \log L)$ when L is set to the optimal.*

Proof. Let η denote the proportion under incomplete equivalence between $O(n \log L)$ and $O(nQ/L)$. The complexity of the algorithm can be summed up as $O(n \log L + \eta nQ/L)$. The algorithm can be regarded as an optimization problem and then the objective function of optimization is set to $g(L)$.

$$\begin{aligned} g(L) =& n(\log L + \eta Q/L), \quad L \in [L_0, n] && (23)\\ g'(L) =& n\frac{L - \eta Q}{L^2} && (24) \end{aligned}$$

Then it's easy to deduce the upper and lower bound of the function.

$$\begin{aligned} \min_L g(L) =& n(\log L + 1) && (25)\\ \max_L g(L) =& \max(n \log n, n(\log L_0 + \eta Q/L_0)) && (26) \end{aligned}$$

□

Proposition 5 shows that Backward-Sort becomes Straight Insertion-Sort with the worst case complexity $O(n^2)$ given $L = 1$. When $L = N$, it becomes Quicksort, as shown in

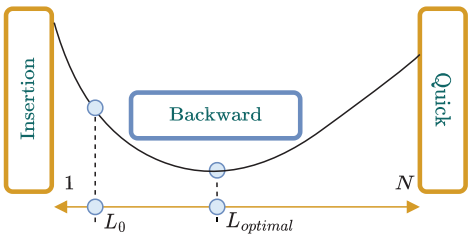

Fig. 6. The relationship between Backward Sort and others

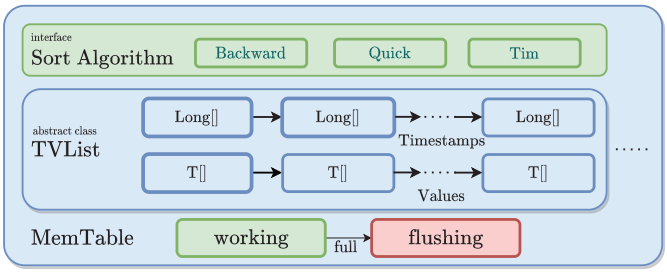

Fig. 7. The structure of IoTDB's Memtable and TVList

Figure 6. In practice, L is expected to be set to ηQ to achieve the best performance $n \log L$.

Proposition 6. *The time complexity of Backward-Sort is* $O(\max\{n \log n, n \log L_0 + \eta n Q / L_0\})$.

Proposition 6 gives the time complexity w.r.t. the initial block size L_0. When the degree of out-of-order is high enough, i.e., Q is large, Backward Sort will set L close to n and degenerate to Quicksort with $O(n \log n)$. On the contrary, when the degree is low, i.e., Q is small, L is set close to L_0. The time complexity turns into $O(n \log L_0 + \eta n Q / L_0)$.

V. Implementation

A. MemTable

In Apache IoTDB, the memtable is divided into two categories, the **active memtable**(working memtable) and **immutable memtable**(flushing memtable). The former is working for continuously writing data, and the latter is prepared for writing data to the disk after the flush condition is triggered.

Each memory table may have multiple chunks, and each chunk contains one TVList that corresponds to one sensor and contains the <T,V>data. T is a unified Long type data and V is an arbitrary type.

Therefore, in the real implementation of IoTDB, in order to reduce the time-consuming of Java template conversion, IoTDB implements a separate class for each custom basic type such as DoubleTVList.

B. TVList

Since time series is generated continuously, every time a new point comes, a simple and straightforward method is to allocate a new buffer for it. The biggest disadvantage of this method is that it costs to allocate memory each time, and the memory access is slower with non-contiguous memory. Another method is to allocate memory large enough at one time. The problem behind this is that it may lead to a waste of memory. Therefore, a common compromise method in databases like IoTDB is to allocate contiguous block memory, similar to the design pattern of Deque, to achieve a trade-off between memory utilization and memory access.

The TVList design in IoTDB is in the form of List<Array>, where timestamps and values both maintain a linked ArrayList to store the TV pair. The size of the array is configurable with its default value 32.

In the industrial scenarios, the data of each sensor corresponds to one TVList in MemTable. Thereby, the timestamps T stored in each TVList are different, that is, each TVList would be sorted separately when flushing or querying.

C. System Design

As shown in Figure 7, we abstract the core part of the sorting algorithm as interfaces to reuse the code. The core function of the Backward Sort algorithm, like "sortByBlocks" discussed in Section IV-B1, is implemented using the common interface. Thereby, the facilities of TVList can be used directly.

The sort function is called in two situations, flushing, and querying. For flushing, after the MemTable is full and turning into a flushing state, the time series needs to be sorted and then written to the disk. For querying, the search needs to be based on an ordered time series..

VI. Experiments

In this section, we first introduce the testing settings like IoTDB-Benchmark, which is used for performance tests of IoTDB. Then, the datasets including synthetic and real-world would be summarized and analyzed with the interval inversion ratio. The code and data for experiments are available online[3].

After this, the experiments about the algorithm itself are conducted firstly to evaluate sort time. With IoTDB-Benchmark, we then test the system optimization influenced by Backward-Sort. Finally, the identical experiments will be tested on the two real-world datasets.

A. Experiments Setting

1) Implemented and Tested Algorithms: To compare with the proposed Backward Sort, several previous sorting algorithms are implemented in Java, using the aforesaid interface introduced in Section V-C. Patience Sort [3] is implemented since it is the most recently proposed algorithm for nearly sorted data. Quicksort is also implemented, where the pivot is always chosen as the middle element of arrays due to time series. In addition, CKSort [10], [11] is a hybrid sorting algorithm of Quicksort, Insertion Sort and Merge Sort. YSort [12] is a variation of Quicksort. Java's default sort algorithm Timsort is also considered, which is a variety of Merge Sort and sorts small arrays with Insertion Sort.

The performance factors for algorithm evaluations are as follows. Sort time is the most important metric on evaluating the performance of sorting algorithms. The Inversion ratio with different interval length is considered to give an adequate

[3]https://github.com/thssdb/sort

quantification of the out-of-order degree. As analyzed at the beginning of Section IV, block size L is essential in the time complexity of the algorithm and thus evaluated. Likewise, array size n is also considered referring to the time complexity. The standard deviation of delayed time σ is used to control the degree of out-of-order, which also affects the performance of sorting algorithms.

The performance factors for system evaluation are as follows. The query throughput refers to the number of points queried by IoTDB per second, and the total test latency represents the average execution time of the test. These two factors are client side statistics, meaning user-perceived performance of various sorting algorithms. The flush time refers to the average flush time of TVList in Memtable, as the system design illustrated in Figure 7. It is the performance indicator of different sorting algorithms from the server side.

2) IoTDB-Benchmark: IoTDB-benchmark is a tool for benchmarking IoTDB against other databases and time series solutions. It can generate periodic time series data according to the configuration or load the existed time series file. After data generation, the Benchmark begins to send the data batch by batch to IoTDB-Server. It is worth mentioning that the batch size is configurable and optimal batch size in our experiments is 500.

Later, if the query command is set, Benchmark will query data from Server with support of several kinds of queries. In our test, we choose the basic time range query as our primary test.

To make use of Benchmark, we install it on a Linux server with 16G memory and 8 core CPU.

3) Datasets: To evaluate the efficiency of Backward-Sort algorithm under different out-of-order time series, we generate two kinds of synthetic datasets, AbsNormal [3] and LogNormal [5], [13], while two real-world datasets CitiBike [14] and Samsung [15] are chosen.

For algorithm efficiency tests, the experiment use a data volume from 10 thousand to 10 million while 100 thousand where 100,000 is the appropriate memory points size in the IoTDB. For system performance tests, the experiment uses 10 million data volumes to simulate real-world scenarios and test throughput.

B. Parameter Tuning

As Proposition 6 shows, the block size L determines the performance of the Backward-Sort algorithm. Given a specific delay distribution, there exists an optimal block size to reach minimum complexity. The larger the block size is, the closer the algorithm is to Quicksort. On the contrary, the closer it is to the Insertion-Sort.

If the inversions are rare enough and the array is nearly completely ordered, the Insertion-Sort could perform well. If the inversions are multiple and IIR becomes large coincidently, the straight Insertion-Sort shrinks and Quicksort will be the dominant strategy.

To conduct the parameter tuning test, the array size is set to 1,000,000 and use IntTVList(<long, int >T-V pair). By omitting the first step of the algorithm, we directly set the block size manually to perform the test.

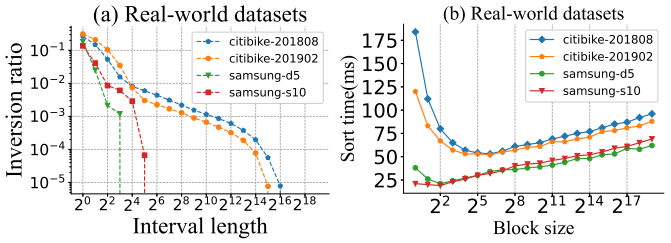

Fig. 8. Varying interval inversion ratio and block size for different datasets

For real-world Datasets as shown in Figure 8, the Backward-Sort shows great improvement compared to Insertion-Sort($L = 1$) and Quicksort($L = N$). As the interval inversion ratio in Figure 8(a) indicates that if $\alpha_L = 0$ $L \geq 2^5$ for Samsung and $L \geq 2^{16}$ for CitiBike, the overlap of the Samsung datasets between different blocks is small and that of Citibike is large. The non-adjacent blocks wouldn't coincide at all if the block size is large. In most cases, too small block size is not optimal since the overlap between non-adjacent blocks will result in more unnecessary move operations. It is crucial to find the appropriate block size. It is observed that the optimal block size roughly corresponds to the interval that the inversion ratio is truncated at some value between 10^{-2} and 10^{-3}.

Fixed Parameter: We choose $\tilde{\Theta} = \mathbf{0.04}$ as our empirical interval inversion ratio threshold which is configurable. In practice, the program estimates the empirical interval inversion ratio as indicated in **SetBlockSize** part. In fact, $\tilde{\Theta}$ could be deliberately chosen to be a little smaller and the block size controlled by it is always a little greater than the found optimal size. The idea behind this is that the empirical value estimates may be smaller, which results in the block size being smaller as well. However, the small block size will make the algorithm degenerate into insertion sort, whose complexity is too high to stand. Therefore, a larger block size is acceptable relatively.

The Backward Sort algorithm can be seen as a "block insertion" sorting process that inserts the sorted black backward. Thereby, its sorting performance is affected by the block size L. As illustrated in Figure 6, the algorithm degenerates to Insertion Sort when L is set to 1, or Quicksort for $L = N$. The optimal block size $L_{optimal}$ is usually in between. Thereby, we search from an initial size L_0 and increase it to find a proper block size. To avoid degenerating into Insertion-Sort, L_0 is expected to be larger than 1. On the other hand, to avoid missing $L_{optimal}$ and sacrificing performance, L_0 should not be too large. As the results shown in Figure 8(b), we can find that $L_{optimal}$ is almost always greater than 4. Therefore, setting L_0 to 4 will not miss the optimal block size in most cases and avoid degradation in a certain degree.

C. Algorithm Comparison

After finding the fixed parameter, then it is indispensable to compare different sort algorithms.

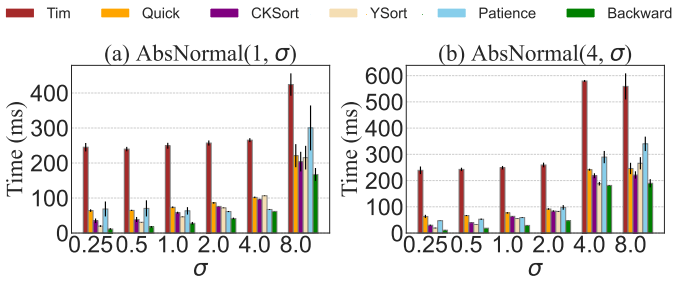

Fig. 9. The sort time of AbsNormal datasets with varying σ

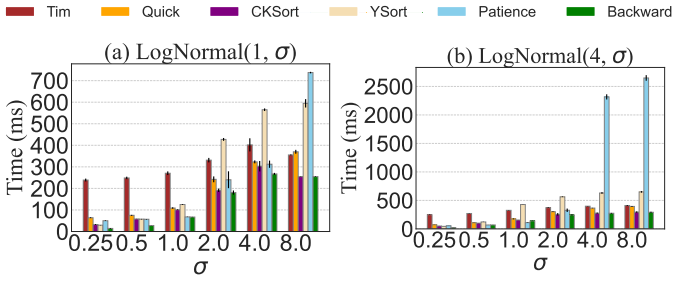

Fig. 10. Sort time of LogNormal datasets with varying σ

1) Varying Inversions: The degree of out-of-order is the main factor influencing the performance of algorithms. Since σ has a greater impact on the inversions, we set $\mu = 1$ or $\mu = 4$ and then vary the standard deviation σ to change the degree of out-of-order.

As Figure 9 and 10 show, the greater σ is, the longer sorting time is. Moreover, Figure 8 shows that CitiBike is more out-of-order and costs more to sort. Therefore, the higher the degree of out-of-order of time series is, the more time sorting algorithms consume. The performance of Timsort and Quicksort is stable, slowly growing as σ increases. The Patience Sort is not stable, especially in LogNormal Datasets, since the original paper only conducted experiments on AbsNormal Datasets. The reason is that the cost of moves (TV pairs) is higher in IoTDB than that in general arrays. Thereby, the constructions of sorted runs consume more time. Backward-Sort outperforms all other algorithms on the whole and improves performance by about 30% to 100% compared to Quicksort.

Figure 11 shows that YSort performs well when the degree of out-of-order is small like Samsung-D5. However, it is not effective when the out-of-order degree gets large such as in the CitiBike-201808 dataset. CKSort performs stably in most datasets, but still worse than our Backward Sort.

2) Varying Array Size: We choose AbsNormal(0,1), LogNormal(0,1), CitiBike-1808 and Samsung-S10 and vary the array size to test the sort algorithms. Since the sorting time is less than 1ms when the array size is 10000, the error is larger than that with greater array size. Figure 12 shows that Backward-Sort outperforms other algorithms under various datasets in different scales.

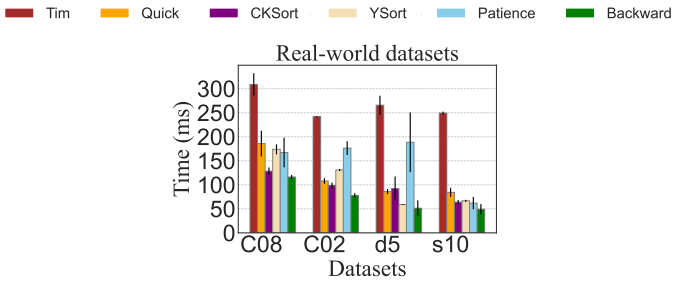

Fig. 11. Sort time of real-world datasets

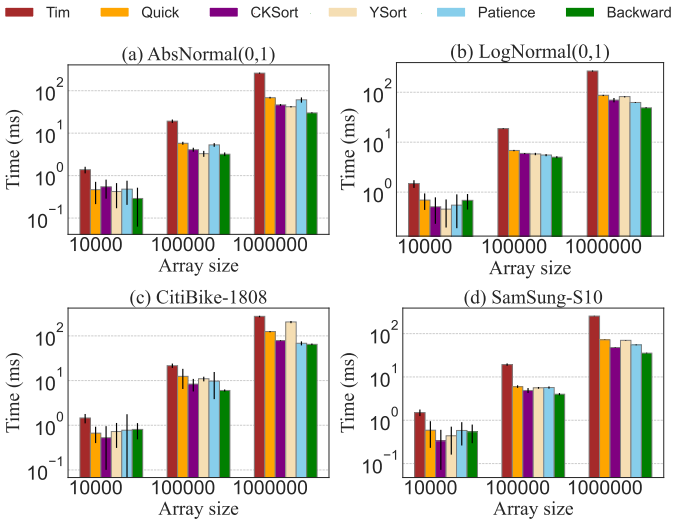

Fig. 12. Varying the array size on different datasets

D. System Comparison

Since write operations are more frequent than query ones in industrial IoT scenarios, the scenarios where the write percentage is higher than 0.5 are mainly considered. Therefore, we choose 25%, 50%, 75%, 90%, 95%, 99% and 100% as our testing write percentage. It is worth noting that when the write percentage is 1, there is no query operation; when the write percentage is 0, that is, no data is written, the querying and sorting processes are not used. Therefore, there is no corresponding query throughput in both cases.

As for query operations in our test, we choose the time range query as our main query statement, which is one of the simplest query and the basis of the aggregation functions. To avoid querying data in the disk which leads to additional I/O cost, we limit the window of the query to the neighborhood of the latest timestamp (current) The query statement is formatted as follows:

```
SELECT *
FROM data
WHERE time > current - window
```

1) Query Throughput: The query process in IoTDB takes the lock and blocks the write process. Thereby, the synchronous query will take the whole time to process query,

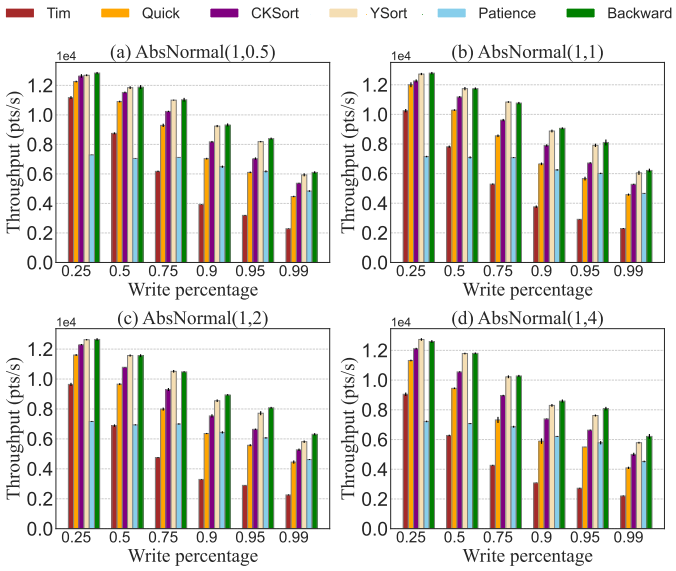

Fig. 13. The query throughput for AbsNormal datasets

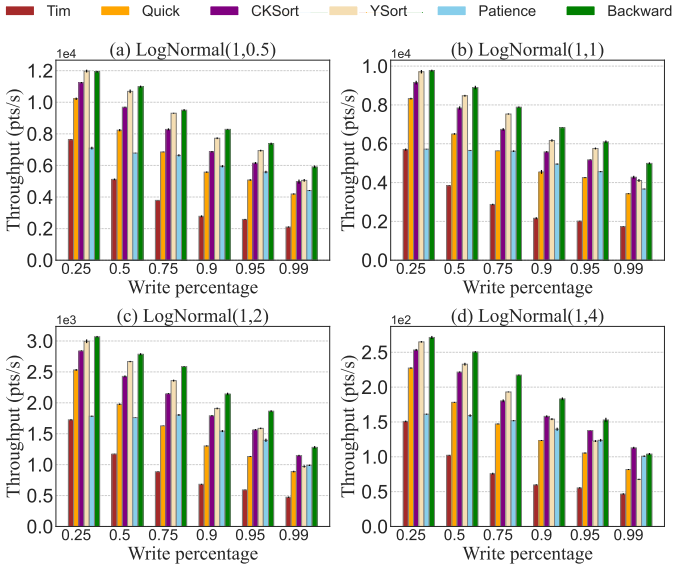

Fig. 14. The query throughput for LogNormal datasets

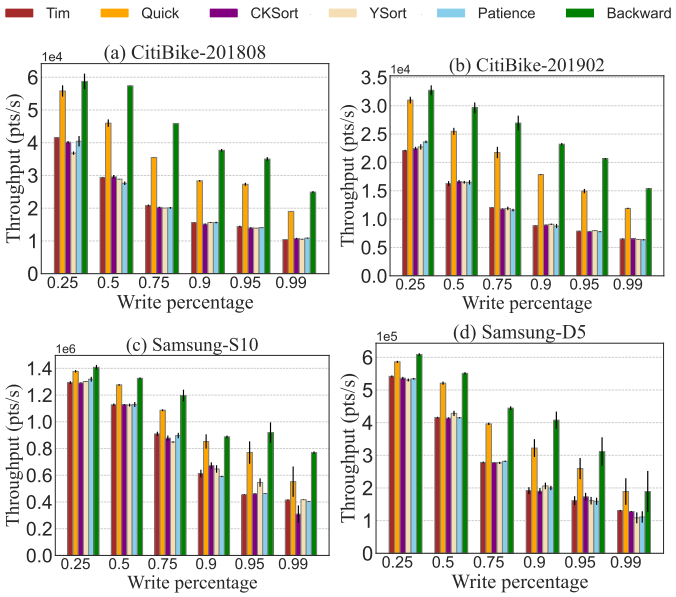

Fig. 15. The query throughput for real-World datasets

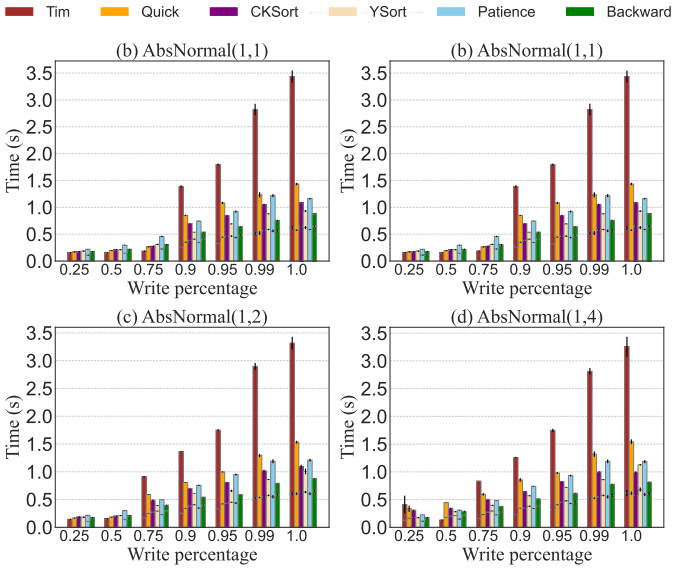

Fig. 16. The flush time for AbsNormal datasets

which firstly sorts the data.

As shown in Figure 15, when the writing percentage is small, the difference between the sorting algorithms is not significant. The reason is that there are fewer write operations, and thus less data needs to be sorted. With the increase of writing percentage, throughput drops in all methods, since less query points are returned due to less query operations. Nevertheless, our proposal Backward sort shows improvement in query throughput in most tests by accelerating sorting for query operations. We add the aforesaid discussion of experiment results on various write percentage at the end of Section VI-D1.

2) Flush Time: The flush time records the range from when the table state transitions(working to flushing) to the completion of writing to the disk. In IoTDB's implementation, it is **asynchronously awaited**, including processes such as sorting, encoding, and I/O, which may be affected by other higher priority processes.

The asynchronous flushing time statistics include two parts, the sorting time and the other, as shown in the Figure 16 17 and 18 by overlay with the corresponding error bars. The higher the write percentage is, the smaller the flush time is. The reason is that write processes are blocked due to more queries and release their resources that flushing takes advantage of.

The results also show that Backward Sort has a huge advantage in flushing time. The improvement comes from two aspects. One is the improvement of the algorithm time;

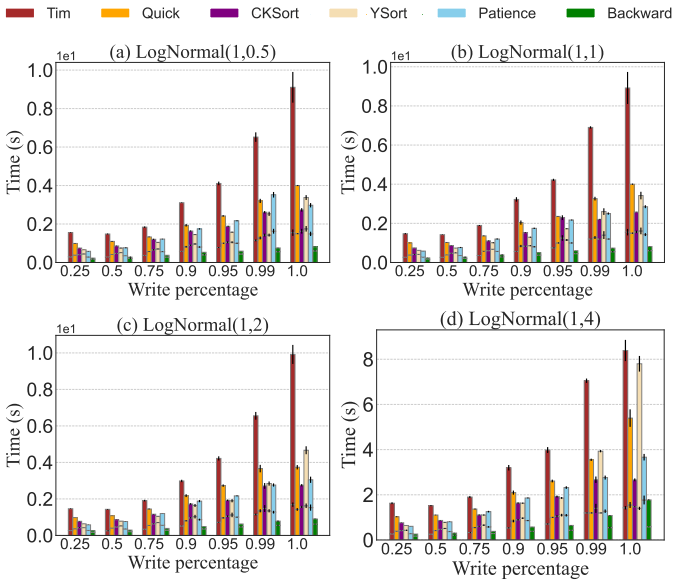

Fig. 17. The flush time for LogNormal Datasets

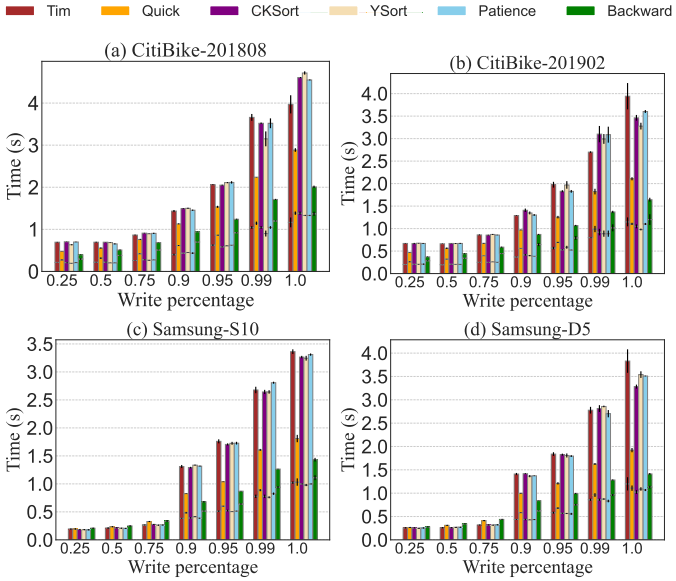

Fig. 18. The flush time for real-world datasets

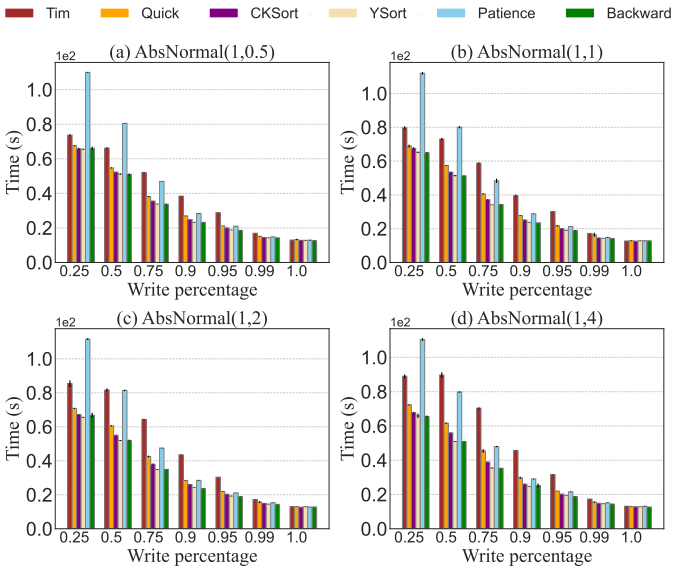

Fig. 19. The total test latency for AbsNormal datasets

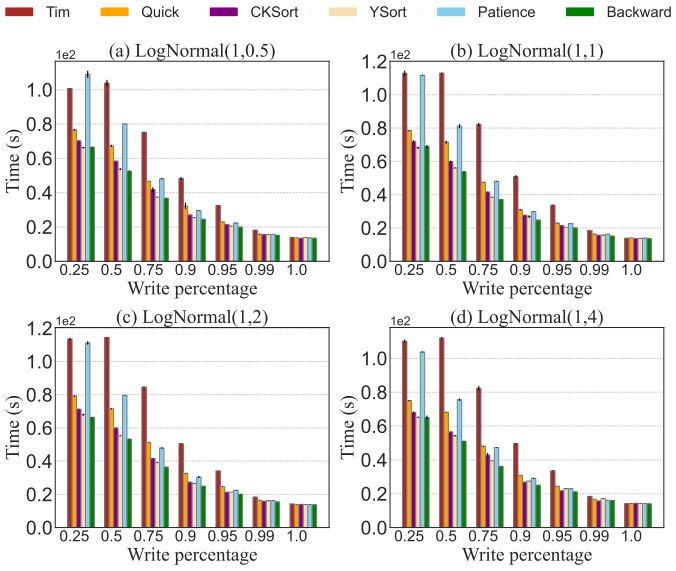

Fig. 20. The total test latency for LogNormal datasets

Backward Sort further optimizes the sorting time compared with YSort and Quicksort. The other is the reduction of system memory usage. For CKSort and Patience, a little extra space is required for Backward.

3) Total Test Latency: The total test latency mainly consists of preprocessing, query and flush, which could indicate the whole performance of the IoTDB system.

Figure 21 shows that when the querying operations gradually dominate, the difference gradually becomes apparent. It is not surprising that owing to the higher sorting costs of CKSort and YSort, their overall test latency is higher than our proposed Backward Sort.

E. Downstream Application

Almost all the downstream applications require time series ordered by time [16], [17]. For example, in time series analytics such as computing the average speed of an engine in every minute, the disordered data points obviously lead to incorrect statistics. Without ordering by time, adjacent points with non-consecutive timestamps may fluctuate on values, as the disordered example illustrated in Figure 22(a). It is not surprising that such fluctuation also affects the deep learning models to some extent.

We apply the deep network LSTM [18] to forecast the time series. Again, multiple out-of-order datasets are prepared by adding the delay time of LogNormal$(1, \sigma)$. The larger the variance σ is, the higher the degree of out-of-order is. The

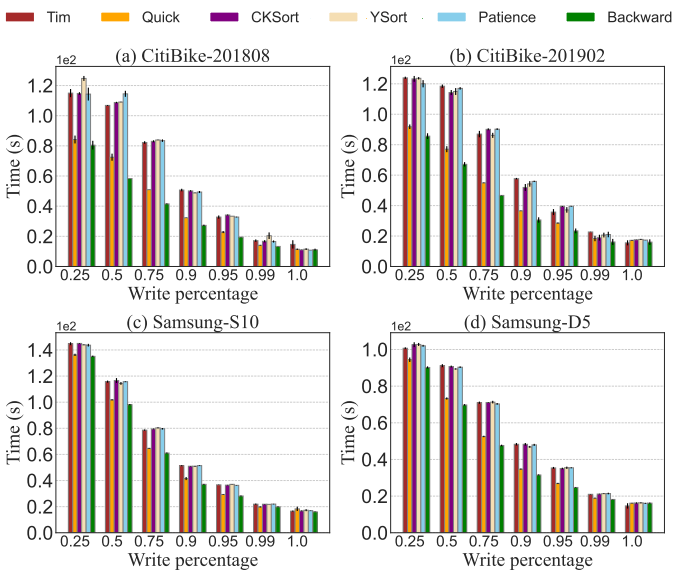

Fig. 21. The total test latency for real-world datasets

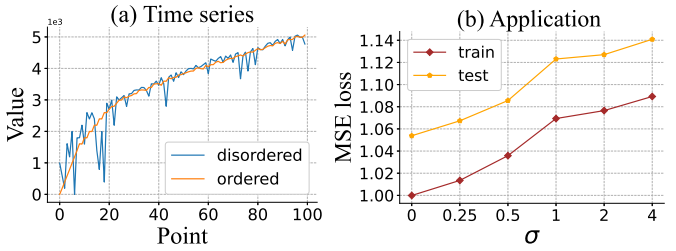

Fig. 22. The downstream application of ordered and disordered time series

first 70% data are used for training, with the last 30% for testing. The input size and hidden size are set to 10 and 2, while other parameters are default.

The train and test MSE loss is reported in Figure 22(b). LogNormal(1,0) with variance $\sigma = 0$ means no delayed points, i.e., exactly ordered by time. As shown, with the increase of the disordered degree σ, it is generally harder to train and the application performance degrades. The results indicate that the downstream applications indeed benefit from the ordered time series.

VII. Related Work

A. In-memory Structure

Traditional relational databases use classical data structures in memory to store sorted data. For example, RocksDB [19] maintains a probabilistic data structure SkipList, which allows search and insertion with $O(\log n)$ complexity. RocksDB implements Skiplist to maintain sorted data. There is no best choice of the design, but for the specific situation. With the tremendous development of IoT, the demand for industrial data process and storage arises. Some time series databases like InfluxDB [20], TimescaleDB [21] and so on use B-tree to maintain the in-memory data in order. Apache IoTDB makes use of a list-array structure, similar to the deque, to buffer the streaming time series.

B. Out-of-Order

Out-of-order has been well studied especially in the sliding window. Since the aggregation should be executed during the latest window, there is always a trade-off between accuracy and latency.

Timsort, a variety of Merge-Sort, is used everywhere due to its stability. The Apache IoTDB's current method is Timsort. Quicksort does not use extra space for sorting, which is very friendly to system performance [12], [22]. Many sort algorithms are designed for almost-sorted data and gain great performance [8], [23].

CKSort [10], [11] is based on three sorting algorithms: Quicksort, Insertion Sort and Merge Sort. It extracts the unordered pairs into another array, then sorts and merges the two arrays. The downside of CKSort is that it requires $O(n)$ extra space and may bring multiple redundant moves. YSort [12], a variation of Quicksort, ensures that the minimum and maximum elements of each sublist are located on the left and right. Therefore, it requires fewer partitioning steps.

Smoothsort [24] is inspired by heapsort, and maintains a priority queue to extract the maximum. Though its upper bound is $O(n \log n)$, it is unstable. Patience Sort [3], Impatience-Sort [25], are state-of-the-art algorithms specifically designed for nearly sorted data, where Impatience also takes advantage of some modern processors. Patience Sort makes full use of the characteristics of little runs in the nearly sorted data, and achieves further optimization through some memory tricks, such as ping-pong.

VIII. Conclusion

In this paper, we first analyze the unique features of out-of-order arrivals in Apache IoTDB, i.e., delay-only and not-too-distant. Referring to such features, a new algorithm Backward-Sort is then devised for sorting time series data by their timestamps. While the idea of moving points backward is motivated by the feature of delay-only, we further propose to divide data points in blocks, referring to the non-too-distant feature, such that moving points is expected to occur locally inside blocks. To the best of our knowledge, this is the first sorting algorithm specially designed for time series. We show that Quicksort is indeed the worst case of the proposed algorithm. Remarkably, the algorithm has been a fundamental component of sorting time series data in Apache IoTDB. The experimental evaluation is conducted, using IoTDB-benchmark, over real and synthetic datasets.

Acknowledgement: This work is supported in part by the National Key Research and Development Plan (2021YFB3300500), the National Natural Science Foundation of China (62021002, 62072265, 62232005), 31511130201, 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] 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.*, vol. 13, no. 12, pp. 2901–2904, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2901-wang.pdf

[2] K. Tangwongsan, M. Hirzel, and S. Schneider, "Optimal and general out-of-order sliding-window aggregation," *Proc. VLDB Endow.*, vol. 12, no. 10, pp. 1167–1180, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1167-tangwongsan.pdf

[3] B. Chandramouli and J. Goldstein, "Patience is a virtue: revisiting merge and sort on modern processors," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 731–742. [Online]. Available: https://doi.org/10.1145/2588555.2593662

[4] X. Cui, K. Wu, T. Wei, and E. H. Sha, "Worst-case finish time analysis for dag-based applications in the presence of transient faults," *J. Comput. Sci. Technol.*, vol. 31, no. 2, pp. 267–283, 2016. [Online]. Available: https://doi.org/10.1007/s11390-016-1626-6

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

[6] P. M. Grulich, J. Traub, S. Breß, A. Katsifodimos, V. Markl, and T. Rabl, "Generating reproducible out-of-order data streams," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 256–257.

[7] A. Awad, M. Weidlich, and S. Sakr, "Process mining over unordered event streams," in *2020 2nd International Conference on Process Mining (ICPM)*. IEEE, 2020, pp. 81–88.

[8] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 441–476, 1992. [Online]. Available: https://doi.org/10.1145/146370.146381

[9] V. Estivill-Castro, "Generating nearly sorted sequences - the use of measures of disorder," *Electron. Notes Theor. Comput. Sci.*, vol. 91, pp. 56–95, 2004. [Online]. Available: https://doi.org/10.1016/j.entcs.2003.12.006

[10] C. R. Cook and D. J. Kim, "Best sorting algorithm for nearly sorted lists," *Commun. ACM*, vol. 23, no. 11, pp. 620–624, 1980. [Online]. Available: https://doi.org/10.1145/359024.359026

[11] I. Finocchi and G. F. Italiano, "Sorting and searching in the presence of memory faults (without redundancy)," in *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, L. Babai, Ed. ACM, 2004, pp. 101–110. [Online]. Available: https://doi.org/10.1145/1007352.1007375

[12] R. L. Wainwright, "A class of sorting algorithms based on quicksort," *Commun. ACM*, vol. 28, no. 4, pp. 396–403, 1985. [Online]. Available: https://doi.org/10.1145/3341.3348

[13] D. A. Basin, F. Klaedtke, and E. Zalinescu, "Runtime verification of temporal properties over out-of-order data streams," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10426. Springer, 2017, pp. 356–376. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_18

[14] citibike. Citi Bike Trip Histories. (2022, Oct). [Online]. Available: https://ride.citibikenyc.com/system-data

[15] W. Weiss, V. J. E. Jiménez, and H. Zeiner, "A dataset and a comparison of out-of-order event compensation algorithms," in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security, IoTBDS 2017, Porto, Portugal, April 24-26, 2017*, M. Ramachandran, V. M. Muñoz, V. Kantere, G. B. Wills, R. J. Walters, and V. Chang, Eds. SciTePress, 2017, pp. 36–46. [Online]. Available: https://doi.org/10.5220/0006235400360046

[16] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are transformers effective for time series forecasting?" *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

[17] A. Abdullatif, F. Masulli, and S. Rovetta, "Tracking time evolving data streams for short-term traffic forecasting," *Data Sci. Eng.*, vol. 2, no. 3, pp. 210–223, 2017. [Online]. Available: https://doi.org/10.1007/s41019-017-0048-y

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[19] FaceBook. RocksDB. (2022, Oct). [Online]. Available: http://rocksdb.org/

[20] influxData. InfluxDB. (2022, Oct). [Online]. Available: https://www.influxdata.com/

[21] timescale. TimescaleDB. (2022, Oct). [Online]. Available: https://www.timescale.com/

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[23] J. Wright, "Algorithms for sorting nearly sorted lists," 1986.

[24] E. W. Dijkstra, "Smoothsort, an alternative for sorting in situ," *Sci. Comput. Program.*, vol. 1, no. 3, pp. 223–233, 1982. [Online]. Available: https://doi.org/10.1016/0167-6423(82)90016-8

[25] B. Chandramouli, J. Goldstein, and Y. Li, "Impatience is a virtue: Revisiting disorder in high-performance log analytics," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 677–688. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00067