Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree

Yuyuan Kang*, Xiangdong Huang†, Shaoxu Song‡✉, Lingzhe Zhang, Jialin Qiao,
Chen Wang, Jianmin Wang, and Julian Feinauer
School of Software, BNRist, Tsinghua University, Beijing, China
pragmatic industries GmbH, Kirchheim, Germany
{kyy19*, zhanglz20, qjl16}@mails.tsinghua.edu.cn, {huangxdong†, sxsong‡, wang_chen, jimwang}@tsinghua.edu.cn,
j.feinauer@pragmaticminds.de

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00315

***Abstract*—LSM-Tree is widely adopted for storing time-series data in Internet of Things. According to *conventional policy* (denoted by π_c), when writing, the data will first be buffered in *MemTable* in memory. When it is full, the data will be written to the disk to form *SSTable*s. Compaction is triggered to sort the data in each layer of the LSM-Tree on the disk. However, the arrival of data can be unordered due to reasons such as transition delay. Apache IoTDB uses *in-order* and *out-of-order MemTable*s to separately buffer the in-order and out-of-order data to accelerate queries, namely the *separation policy* (denoted by π_s). However, given a specific space of memory budget to buffer the data, write amplification (WA) of the leveled LSM-Tree will be influenced by π_s. Whether the influence by separation is positive or negative, and how intense WA is influenced, depend on the properties of workloads and the capacity of the *in-order* and *out-of-order MemTable*s. It is highly demanded to build robust models for estimating the expected amount of data rewritten in each compaction, and predicting the WA under π_c and π_s. Note that as an industrial paper, rather than proposing novel techniques for research problems, we focus on the practice of whether separating or not for lower write amplification. Experiments on synthetic and real-world datasets show that the models for estimating WA are accurate under various delay distributions. In addition, based on the estimation models, we implement an analyzer module in the open-source Apache IoTDB, for choosing the policy with lower WA. We apply the method in the use case of our industrial partner, a service provider of engineering machinery. The use case verifies the effectiveness of deciding whether separation or not by WA estimation.**

***Index Terms*—Leveled LSM-Tree, Write Amplification**

I. Introduction

Apache IoTDB [1] is an open-source, leveled LSM-Tree-based, high-performance data engine tailored for time-series data. A time-series is a collection of data points generated along a timeline [2]. A timestamp recording when the data point is generated, and another indicating when the data point arrives in the database will be assigned to each data point. In this paper, we use the term "tuples", "entries" and "data points" interchangeably. In terms of generation time, time-series data are unordered because there are various delays [3] caused by clock skew [4], batch transmission [5], network delays [6], or system recovery from failure [5] varying from data point to data point. IoTDB distinguishes *in-order* and *out-of-order* data points according to their generation time. If a new-arriving data point is generated later than all of the data points on the disk, then it is an *in-order* data point. Otherwise, it is called an *out-of-order* data point. Accordingly, IoTDB designed in-order and out-of-order *MemTable* (C_{seq} and C_{nonseq} in Figure 1, respectively) to handle the two kinds of data points separately [7], [8], which is called the *separation policy*, compared to the *conventional policy* where there is only one *MemTable* (C_0 in Figure 1). We denote the separation policy and the conventional policy as π_s and π_c.

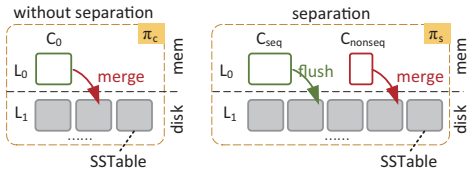


Fig. 1. Writing data with and without the π_s. We consider that the capacity of C_0 equals the sum of the capacities of C_{seq} and C_{nonseq}.

A. Advantages of π_s over π_c

Data are organized in *SSTable*s on the disk. In an *SSTable*, the entries are sorted by the generation time. When writing, π_c first buffers the data in C_0. When C_0 is full, π_c merges the data in C_0 and those in *SSTable*s, which have overlapping key ranges with C_0, to form new *SSTable*s so that the data are sorted on the disk. π_c can lead to data of a period on the disk being rewritten many times in the partially unordered time-series scenario. However, the separation design of IoTDB [7], π_s, only requires compaction when C_{nonseq} is full, and it can just flush C_{seq} to the disk when it is full. π_s exhibits two advantages. First, because the minority of out-of-order data are accumulated together, the remaining in-order data are stored in *SSTable*s without overlap, hence accelerating range queries (with predicates on the generation time) since fewer data need to be read [9]. Second, it attempts to accumulate more out-of-order data before executing a merge operation so that data of a certain period may be rewritten fewer times in the long run.

B. Problems of π_s

However, π_c and π_s should be chosen carefully. In some cases, π_s may suffer comparatively greater write amplification (WA), which is the ratio of the amount of data actually written

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00315

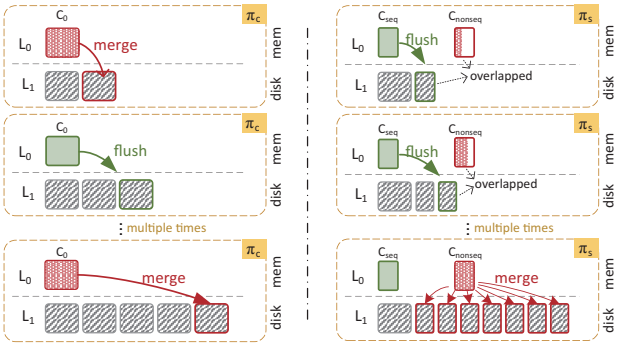


Fig. 2. A case where π_s suffers more intensive WA than π_c, when most of the data are in order.

to the disk divided by the amount required by the user [10]. A high WA means a waste of system resources, and it is detrimental to system performance and the lifespan of the hardware [11]. In Fig. 2, for example, if there are only a few out-of-order data points, then compaction happens at a very low frequency under π_c. WA is close to 1. While under π_s, an out-of-order point is collected in memory earlier. Then arrive a large number of in-order data points. When enough out-of-order data points have arrived, C_{nonseq} is full but contains a long overlapping period among the *SSTable*s on the disk, some of which otherwise would not be rewritten under π_c. In the example of Fig. 2, the separation policy suffers more intensive write amplification than conventional policy, mainly in the last step. That is, C_{nonseq} has to merge with a large number of SSTables in L_1 that have overlapping key ranges with C_{nonseq}. In order to better illustrate this, we use an arrow to point to each SSTable in merging in Fig. 2. Therefore, π_s is not one-size-fits-all. It should depend on the intensity of disorder of the writing workload. The general WA of leveled LSM-Tree has been discussed in [12]. To show the difference between π_c and π_s, an accurate model needs to be built.

Moreover, once the separation policy is chosen, the knob introduced by π_s, the capacity of C_{nonseq} or C_{seq} would also have an impact on the WA, which was not discussed in current literature. Considering limited space in memory for buffering data points, if C_{nonseq} is too small, C_{nonseq} is filled more often, which triggers frequent compaction and leads to intensive WA. While if it is too large, which means that the space left for C_{seq} is circumscribed. Therefore, C_{seq} is flushed more often, which updates the maximum generation time on the disk frequently. As a result, more data points are treated as out of order, which would otherwise be considered in order. It also results in higher WA. Therefore, revealing the impact of the capacity of C_{nonseq} on WA is a non-trivial problem.

C. Decision Problem

However, in the early version of Apache IoTDB, the capacities of *MemTable*s for in-order and out-of-order data are not tunable. Therefore, the following questions demand prompt solutions. Given limited memory for buffering data points (i.e., the number of the tuples that can be buffered in memory is a constant), the delay distribution and generation time interval of the writing workload, then

1) What is the WA under π_c?
2) Under π_s, how does WA change with different capacities of C_{seq}? What is the minimum WA in this case?

Note that as an industrial paper, rather than proposing novel techniques for research problems, we focus on the practice of whether separating or not for lower write amplification.

D. System Practice

We implement a delay analyzer in Apache IoTDB, which will collect time-series data delays and generate the statistical profile of the delays, e.g., the probability distribution function (PDF) and cumulative distribution function (CDF). Then, a statistical model is used to predict WA under π_c and the minimum WA under π_s, as well as the (sub)optimal capacities of C_{seq} and C_{nonseq}. The major components include:

1) A model of the time-series data, delays, and out-of-order data to estimate WA under the conventional policy without separation π_c.
2) A model to estimate WA under the separation policy π_s, where C_{seq} is treated as a parameter. The minimum WA and the (sub)optimal C_{seq} are also given.
3) An analysis module that will analyze the delays of the writing workload, and adopt a (sub)optimal policy such that the WA is minimized.
4) A case study of system practice in the scenario of our industrial partner, verifying the effectiveness of deciding whether separation or not by WA estimation in IoTDB.

As a part of the Apache IoTDB project, the code of this paper is available in the official GitHub repository of IoTDB[1].

The remainder of the paper is organized as follows. Section II formally defines time-series data, delays, in-order and out-of-order data points. The arrival rate ratio of in-order and out-of-order data is also given to quantify the disorder intensity, which is strongly related to the WA. In Section III, given the delays' distribution, generation time interval, the WA model under π_c is estimated by counting the subsequent data points. Section IV shows the writing under π_s in detail, and then presents the model of WA under π_s. Subsequently, based on the models, a tuning algorithm is given to minimize the WA. Section V conducts experiments on various datasets to confirm that the models can accurately estimate WA under π_c and π_s. Section VI shows a specific use case of our industrial partner, a service provider of engineering machinery, and confirms the effectiveness of the delay analysis module.

II. Out-of-order Data

In this section, we present some preliminaries on the definitions of time-series data points and delays. To describe the intensity of chaos of the time-series data, we consider *in-order* and *out-of-order data point*s [7]. Then, a model is introduced to quantify the arrival rate ratio of the two types. This model

[1]https://github.com/apache/iotdb/tree/research/separation

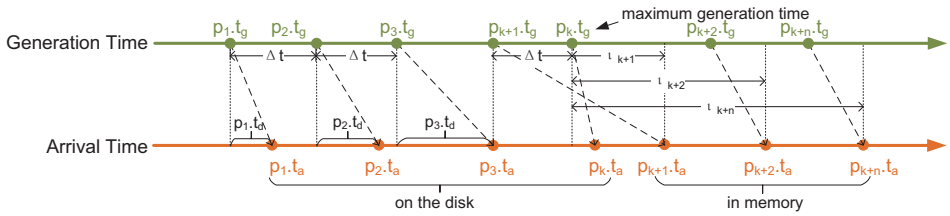


Fig. 3. Time-series data points and their generation time, arrival time and delays.

TABLE I
NOTATIONS AND EXPLANATIONS

Sym.	Explanations
π_c	The conventional LSM-tree leveling merge policy.
π_s	The leveling-based separation policy.
r_c	The WA under π_c.
r_s	The WA under π_s.
n	The maximum entries that can be held in memory.
C_0	The *MemTable* of conventional LSM.
C_{seq}	The *MemTable* for in-order data points.
C_{nonseq}	The *MemTable* for out-of-order data points.
n_{seq}	The capacity of C_{seq}.
n'_{seq}	The expected number of data points to be flushed from C_{seq} for the last time in a phase.
n_{nonseq}	The capacity of C_{nonseq}, $n_{\text{nonseq}} = n - n_{\text{seq}}$.
R	The *run* [13] on level L_1.
r	The write amplification model.
$\zeta(n)$	The expected number of subsequent data points in R when there are n data points buffered in memory.
$f(x)$	The probability distribution function of delays.
$F(x)$	The cumulative distribution function of delays.
ι	The minimum duration of delay making a point p out of order.
LAST(R)	The data point with the latest generation time in R where R is the *run* on level L_1.
p	A data point.
$\mathcal{S}$	A time-series, which is a collection of data points.
π_{adaptive}	The policy chosen by the tuning algorithm under dynamic delay distribution.
$\pi_s(\hat{n}^*_{\text{seq}})$	The separation policy with the recommended capacity of C_{seq}, i.e., $\hat{n}^*_{\text{seq}}$.
$\pi_s(n_{\text{seq}})$	The separation policy with the capacity of C_{seq} being n_{seq}.

is crucial to estimate WA under π_s. Section IV shows how the model is adopted in detail. Several notations used in this paper and their explanations are listed in Table I.

Definition 1. ***Time-series data point***. *A time-series data point p is a triple $p =< t_g, t_a, v >$, where t_g is the timestamp when the data point is generated, which is unique and identifies a specific data point; t_a is the timestamp when the data point arrives in the database; v is the value it carries on.*

Time-series $\mathcal{S}$ is a collection of time-series data points, $\mathcal{S} = \{p_1, p_2, \cdots, p_i, \cdots\}$. As is shown in Fig. 3, the index of each data point indicates the order of their arrivals. Since the data points are generated at a certain frequency, the time interval to generate data points is a constant, denoted by Δt.

Definition 2. ***Delay.*** *The delay of a time-series data point p is the difference between its arrival time and its generation time, denoted by $p.t_d$. Formally, $p.t_d = p.t_a - p.t_g$.*

Fig. 3 shows data points p_1, p_2, $\cdots$, p_{k+n}, and the relationship of their generation time, arrival time, and delay. The green spots refer to their generation time, and the yellow spots refer to their arrival time. A dashed arrow connects the two timestamps of a data point. In this paper, we use the notation $p_i.t_a$ and t_i interchangeably to indicate the arrival time of the i-th data points received.

In IoTDB, the *SSTable*s on level L_1 are organized without overlapping key ranges with each other. As a whole, data points on L_1 are considered as a *run* [13], denoted by R. We use the notation LAST(R) to denote the entry with the latest generation time in R. Similar to [7], the *in-order* and *out-of-order* data points are given as follows.

Definition 3. ***In-order and Out-of-order Data Points***. *For any data point p buffered or to be buffered in memory, if all of the data points in R are generated earlier than p, then p is an in-order data point. Otherwise, p is an out-of-order data point. Formally, p is an in-order data point, iff $p.t_g > LAST(R).t_g$; p is an out-of-order data point, iff $p.t_g < LAST(R).t_g$.*

Note that the *out-of-order* data points are different from the concept of *late event*s in the literature [14], [15]. The former focuses on comparing the generation time of the new data point and the latest generation time on the disk. The latter compares the generation time of two data points that arrive consecutively. In Fig. 3, $p_1, \cdots, p_k$ are on the disk, while $p_{k+1}, \cdots, p_{k+n}$ are not. In this case, $LAST(R) = p_k$. Because $p_{k+1}.t_g < p_k.t_g$, then p_{k+1} is an out-of-order data point. In contrast, $p_{k+2}.t_g > p_k.t_g$, so p_{k+2} is an in-order data point. Consistent with the examples in Fig. 3, Fig. 4 plots the data points in a two-dimensional coordinate according to the two timestamps. In general, these points exhibit a linear upward trend. If all of the data points arrive in order, then the figure should be rigorously monotonically increasing. However, the link between p_k and p_{k+1} violates monotonicity.

It is also worth noting that the decision of separation or not should be efficiently made online. Precise computation on the joint distribution of generation intervals and transmission delays is not affordable to the database system. Therefore, in the following, as a trade-off, three assumptions are used to approximate the computation. The delays of the data points are assumed to obey a specific distribution, whose probability distribution function (PDF) and cumulative distribution function (CDF) are denoted by $f(x)$ and $F(x)$, respectively. Moreover, following the same line of [16], the delay values of different

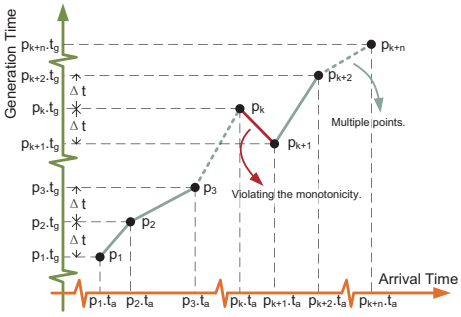


Fig. 4. The arrival time and generation time of the data points in Fig.3.

data points are assumed to be independent and identically distributed. And finally, the time-series data are assumed to be generated at a certain frequency.

To quantify out-of-order intensity, the percentage of *late event*s is widely used in the literature [15], [17]. Nevertheless, it is not suitable in this study, because the definition of *out-of-order* data points is related to the latest generation time on the disk. To take the *MemTable* into account, an analysis is necessary on the quantitative relations between the number of in-order data points and the number of out-of-order data points received in a period. Denote the minimum delay of an in-memory data point p_{k+i} making it out-of-order as $\iota_i = p_{k+i}.t_a - LAST(R).t_g$. Therefore, the probability of a data point p_{k+i} being in order is $F(\iota_i)$. Suppose there are α data points collected in memory. Then, the expected number of out-of-order data points is

$$g(x) = \alpha - x \tag{1}$$

where $x = \sum_{i=1}^{\alpha} F(\iota_i)$ is the expected number of in-order data points.

III. Subsequent Data Points and Write Amplification

In Section II, *in-order* and *out-of-order data point*s in memory are introduced. Motivated by [16], this section considers *subsequent data point*s on the disk to reveal the amount of data to rewrite during compaction given the delay distribution and generation time interval, and then estimates WA under π_c.

Definition 4. ***Subsequent Data Points***. *For an on-disk data point p and the in-memory MemTable C_0, if there is any data point $q \in C_0$, $p.t_g > q.t_g$, then p is a subsequent data point of C_0, subsequent data point in short.*

In Fig. 3, because $p_k.t_g > p_{k+1}.t_g$, then p_k is a subsequent data point. During compaction, any *SSTable* containing subsequent data points should be rewritten. The number of subsequent data points on the disk is thus essential to estimate the WA regarding different amounts of data points temporarily collected in memory.

Denote the CDF and PDF of the specific delay distribution as $f(x)$ and $F(x)$. For the k points on the disk, denote them in the arrival order, $p_1, p_2, \cdots, p_k$, with the respective arrival time $t_1, t_2, \cdots, t_n$. Suppose there are n data points buffered in memory, denoted by $p_{k+1}, p_{k+2}, \cdots p_{k+n}$, with respective arrival time $t_{k+1}, t_{k+2}, \cdots, t_{k+n}$. Then, denote the event that "the generation time of p_{k-i} is earlier than p_{k+j}, where $i \in [0, k-1]$ and $j \in [1, n]$" to be $A_{i,j}^{[k]}$. Denote the event that "p_{k-i} is a subsequent data point" to be B_i^k. Following a similar line of [16], the probability of being a *subsequent data point* is estimated by

$$\begin{aligned} P(B_i^{[k]}) &= 1 - P(\bigwedge_{j=1}^{n} A_{i,j}^{[k]}) \\ &= 1 - \int_0^{+\infty} \left[f(x) \prod_{j=1}^{n} F(t_{k+j} - t_{k-i} + x) \right] \mathrm{d}x \end{aligned}$$

where x indicates the delay of p_i.

Compared with the possible values of the delay, the value of a timestamp is treated as ∞. Therefore, the number of subsequent data points on the disk is $\sum_{i=0}^{k-1} P(B_i^{[k]})$. However, it describes a specific case of the k data points on the disk, where the arrival time is given. Actually, $t_{k+j} - t_{k-i}$ is a sample of a random variable, indicating the difference between the arrival time of the data points for every $i + j$ points. The random variable is denoted by $\widetilde{T}_{i+j}$. Suppose k is large enough, then, the expectation of the number of subsequent data points is

$$\begin{aligned} \zeta(n) &= \mathbb{E}\left[\sum_i P(B_i) \right] \\ &= \sum_i \left\{ 1 - \int_0^{\infty} f(x) \prod_{j=1}^{n} \mathbb{E}[F(\widetilde{t}_{i+j} + x)] \mathrm{d}x \right\} \end{aligned} \tag{2}$$

To confirm the correctness of (2) under different delays' distributions, we test it in synthetic datasets. We adopt $\Delta t = 50$ to set the generation time interval and then add a random variable, which obeys the lognormal distribution, to simulate real-world delays and generate the arrival time. The data points are sorted by the arrival time. We insert the data into a prototype system that records the writing times of each point in an LSM-Tree. Two delay distributions are applied, where $\mu = 4, \sigma = 1.5$ and $\mu = 4, \sigma = 1.75$. Through each compaction, the number of subsequent data points is recorded. In Fig. 5, the scatters indicate the average number of subsequent data points of all compaction. The two lines are the results of Model $\zeta(n)$. Obviously, the model fits the experimental results well.

Generally, if there are n data points collected in memory, the expected number of subsequent data points on the disk is $\zeta(n)$. To compact the in-memory data to the disk, the number of data points that should be rewritten is $\zeta(n)$. As a result, the write amplification can be estimated as $WA = \frac{\zeta(n)}{n} + 1$. This is exactly the case for the WA under π_c. Suppose the capacity of C_0 is n, then the WA under π_c is

$$r_c = \frac{\zeta(n)}{n} + 1 \tag{3}$$

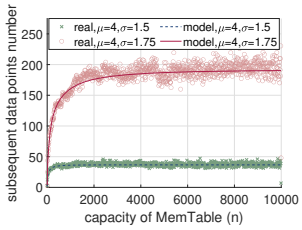


Fig. 5. The numbers of subsequent data points with different capacities of the buffer. Two delay distributions are used. The scatters are the results of experiments. The curves are the estimation results of $\zeta(n)$. The unit of the buffer size is the number of points.

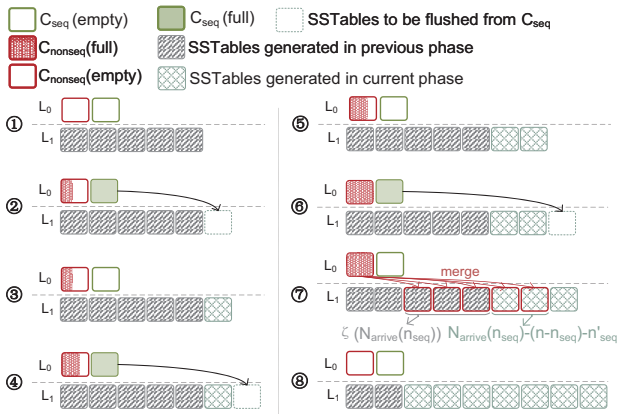


Fig. 6. An example of writing data via π_s in a phase, in which case for n_{seq} in-order data points to arrive, there are $g(n_{\text{seq}})$ out-of-order data points to arrive, and $(n - n_{\text{seq}})/g(n_{\text{seq}}) = 3$.

However, in reality, as long as an *SSTable* contains any subsequent data points, all of the points inside would be rewritten. Nevertheless, the models are based on the subsequent data points. Therefore, the estimated value is a little lower than reality. The upper bound of the difference is 1.

IV. The Separation Policy

In this section, we first introduce the preliminaries of the separation policy π_s that has been studied and implemented in Apache IoTDB [7]. By combining the arrival rate ratio model in Section II, and the subsequent data points number model in Section III, a model could be further introduced to estimate WA under π_s.

The whole writing process is divided into phases, separated by the merges of C_{nonseq}. During each phase, the C_{seq} is filled, flushed, and cleared several times, while C_{nonseq} is filled, merged, and cleared for precisely one time. To analyze WA under π_s, the total number of data points arriving and the number of subsequent data points of C_{nonseq} in a phase are counted.

Fig. 6 demonstrates an example of writing data points via π_s in a phase. It starts at the point when both C_{seq} and C_{nonseq} are empty, just after the compaction of C_{nonseq}. It ends at the point when C_{nonseq} is full again, triggering a subsequent merge and finally being cleared.

Algorithm 1 Separation Policy Tuning Algorithm

Require: n is the maximum number of data points allowed in memory. $F(x)$ is the CDF of delay distribution. $f(x)$ is the PDF of delay distribution. Δt is the generation time interval;

$r_c \leftarrow \text{GET\_WA}_{\pi_c}(n, f, F, \Delta t)$
$g \leftarrow \text{GET\_G}(\Delta t, F)$
$\hat{n}^*_{\text{seq}} \leftarrow -1, r^*_s \leftarrow \infty$
for x from 1 to $n-1$ **do**
  $r'_s \leftarrow \text{GET\_WA}_{\pi_s}(n, x, f, F, \Delta t, g)$
  if $r^*_s > r'_s$ **then**
    $r^*_s \leftarrow r'_s$, $\hat{n}^*_{\text{seq}} \leftarrow x$
  end if
end for
if $r^*_s < r_c$ **then**
  $\pi \leftarrow \pi_s(\hat{n}^*_{\text{seq}})$
else
  $\pi \leftarrow \pi_c$
end if
return π

According to (1), for each time C_{seq} is full, there are $g(n_{\text{seq}})$ out-of-order data points being collected on average. Therefore, C_{seq} will be filled for $(n-n_{\text{seq}})/g(n_{\text{seq}})$ times in a phase. Then, the total number of in-order points to be collected in a phase is $n_{\text{seq}} \cdot (n - n_{\text{seq}})/g(n_{\text{seq}})$. In Fig. 6, $g(n_{\text{seq}}) = (n - n_{\text{seq}})/3$. Besides, the number of out-of-order data points is $n - n_{\text{seq}}$. Therefore, the total number of data points to collect in a phase is

$$N_{\text{arrive}}(n_{\text{seq}}) = \frac{n_{\text{seq}} \cdot (n - n_{\text{seq}})}{g(n_{\text{seq}})} + n - n_{\text{seq}} \tag{4}$$

When C_{nonseq} is full, the *subsequent data point*s on the disk can be divided into two groups. Because they are either written during the current phase or before that. Because the number of *in-order data point*s arriving in the current phase is $N_{\text{arrive}}(n_{\text{seq}}) - (n - n_{\text{seq}})$, while all of the timestamps in the last flushed *SSTable*, which contains $n'_{\text{seq}} = [1 + n_{\text{nonseq}}/g(n_{\text{seq}}) - \lceil n_{\text{nonseq}}/g(n_{\text{seq}}) \rceil] \cdot n_{\text{seq}}$ data points, are earlier than those in C_{nonseq}, then, for the first category, the number of data points to rewrite is

$$N_{\text{cur}}(n_{\text{seq}}) = N_{\text{arrive}}(n_{\text{seq}}) - (n - n_{\text{seq}}) - n'_{\text{seq}}$$

For example, in Fig. 6, suppose $(n - n_{\text{seq}})/g(n_{\text{seq}}) = 3$. Then, $n'_{\text{seq}} = n_{\text{seq}}$. Therefore, $N_{\text{cur}}(n_{\text{seq}}) = N_{\text{arrive}}(n_{\text{seq}}) - n$. For the second category, the writing could be interpreted in another way, where all of the data collected in the current phase are buffered in memory. Then, the number of subsequent data points is $N_{\text{bef}}(n_{\text{seq}}) = \zeta(N_{\text{arrive}}(n_{\text{seq}}))$. Therefore, the write

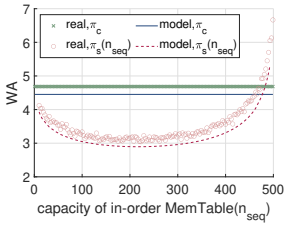


Fig. 7. WA under π_c using synthetic data, where the delays obey lognormal distribution ($\mu = 5, \sigma = 2$), and WA under π_s with different settings of n_{nonseq}. The scatters are the results of experiments, while the two lines/curves are estimation results.

amplification under π_s is

$$\begin{aligned} r_s(n_{\text{seq}}) =& \frac{N_{\text{cur}} + N_{\text{bef}} + N_{\text{arrive}}(n_{\text{seq}})}{N_{\text{arrive}}(n_{\text{seq}})} \\ =& \frac{\zeta(N_{\text{arrive}}(n_{\text{seq}}))}{N_{\text{arrive}}(n_{\text{seq}})} + 1 + \frac{n - n_{\text{seq}} + n'_{\text{seq}}}{N_{\text{arrive}}(n_{\text{seq}})} \end{aligned} \tag{5}$$

Subsequently, it is straightforward to compare $r_c(n)$ with $\min_{n_{\text{seq}}}(r_s(n_{\text{seq}}))$. Algorithm 1 describes the procedure to tune the separation policy given the delays' distribution and generation time interval. The output is a (sub)optimal policy. In line 11, $\pi_s(\hat{n}^*_{\text{seq}})$ means the separation policy, where the capacity of C_{seq} is set to be $\hat{n}^*_{\text{seq}}$.

To have an intuitive view of WA under π_s and π_c, we conduct experiments to show WA with different settings of n_{seq}. To confirm that the model fits more delay distributions, we set $\mu = 5$ and $\sigma = 2$ (different from the case in Fig. 5). The generation time interval is 50, and the size of *SSTable*s is 512 points. In Fig. 7, the scatters are values collected from experiments. The two lines/curves are the estimation results of $r_c(n)$ and $r_s(n_{\text{seq}})$. It shows that the model fits the experimental results well.

V. Experiments

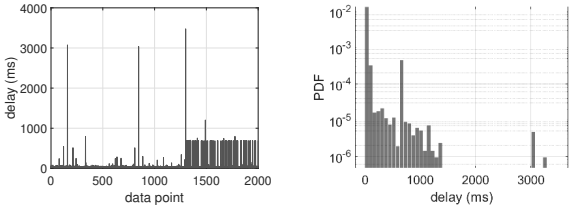
In this section, we first introduce how the datasets were generated. Then, with the synthetic datasets of different delay distributions, we conducted the experiments to show that models, r_c and r_s, could effectively predict the WA. The writing throughput under π_c and π_s was also compared. Finally, various query workloads are considered to show the influence of π_s on query latency.

A. Dataset Description

To evaluate the correctness of the models under different delay distributions and confirm that the recommendations of Algorithm 1 work, we generated several synthetic datasets, which followed the work in [18]. First, we made the generation time by creating an arithmetic progression with the specific time interval Δt. Then, we assigned the delays according to a specific distribution. The sum of the delay and the generation time is the arrival time of the data point. The disorder characteristics are shown in Table II. Totally, there are twelve datasets. For each dataset, there are 10 million tuples. The tuples are written according to the arrival time.

TABLE II
Parameters for the synthetic datasets

	$\Delta t = 50$		$\Delta t = 10$	
	$\mu = 4$	$\mu = 5$	$\mu = 4$	$\mu = 5$
$\sigma = 1.5$	M1	M4	M7	M10
$\sigma = 1.75$	M2	M5	M8	M11
$\sigma = 2$	M3	M6	M9	M12



(a) Delay samples. (b) Delays' distribution.

Fig. 8. Delays of real-world dataset S-9.

We employ a real-world dataset S-9 [19]. The data are sent from mobile devices (Samsung Galaxy Tab 2) to a server (Windows PC), which encounter some delays during the transmission. The dataset contains 27 dimensions and 30 thousand data points. We choose "S.Message.received.time.ms" as arrival time and "C-Send-Time" as the generation time of the data points. The delays of the data points and the histogram are shown in Fig. 8. The dataset exhibits skewness such that some data points suffer much longer delays than others. 7.05% of the data points are considered out-of-order, referring to Definition 3 in Section II. [2]

B. WA Comparisons

The writing times of each data point were recorded to accurately calculate WA, which would increase after the point was rewritten. After writing the datasets, M1– M12, under π_c (n is 512 data points) and π_s (with different settings of n_{seq}), the total number of writing times of all data points was calculated to get WA. Fig. 9 shows the results from experiments and the models, r_s and r_c. The WA of π_c is the result, but it can also reflect the intensity of disorders of the data, which is related to Δt and the distribution parameters, μ and σ.

In Fig. 9, comparing the two subfigures from the same row in the first and the third column, as well as the second and the fourth column, we can see that a greater Δt would reduce the intensity of disorder, hence less WA. While comparing the results on M1 and M4 (and similarly M2 vs M5, M3 vs M6, etc.), we can see that increasing μ would intensify WA. Likewise, the comparisons from M1 to M3 show that a larger σ introduces more severe WA. Similar results are also observed from M4 to M6, from M7 to M9 and from M10 to M12.

The figures (in Figure 9) also show that models r_s and r_c can predict the WA effectively, when the generation time

[2]Because of the limited amount of data in S-9, we set the memory budget to be 8 to trigger merges in experiments.

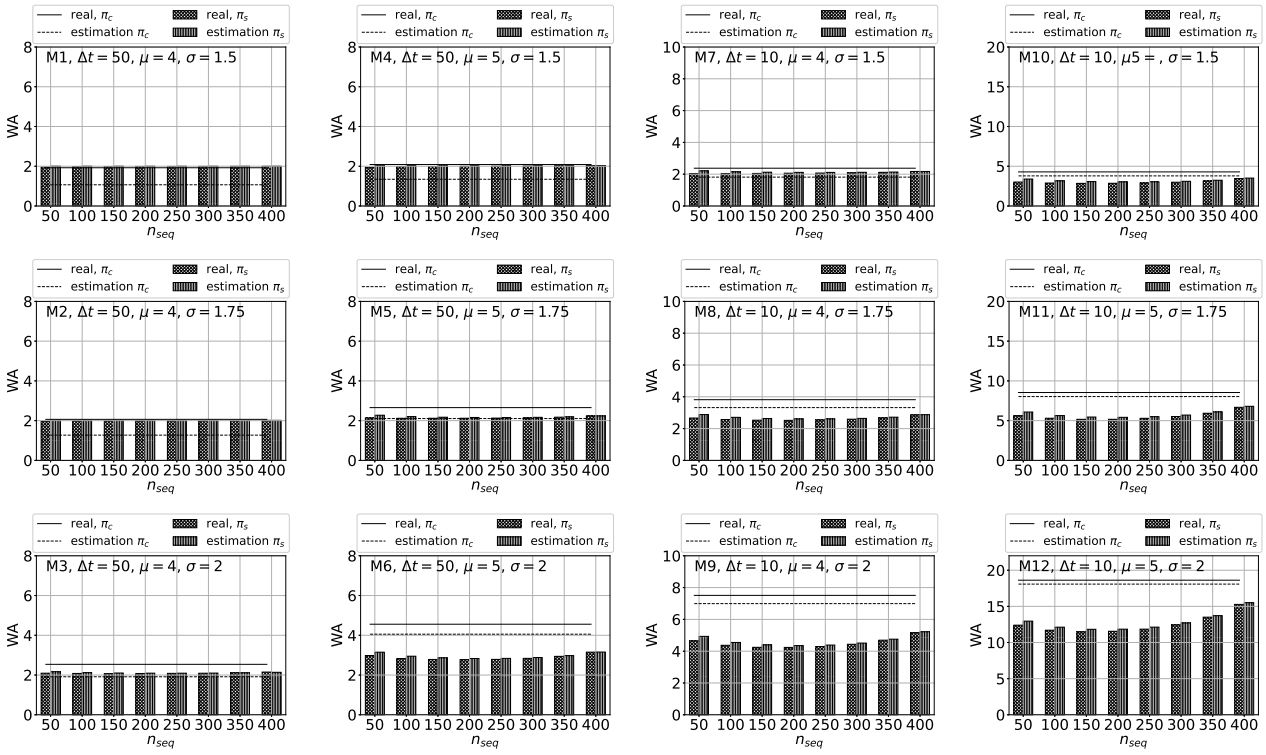


Fig. 9. WA tested from experiments under π_s when using twelve synthetic datasets.

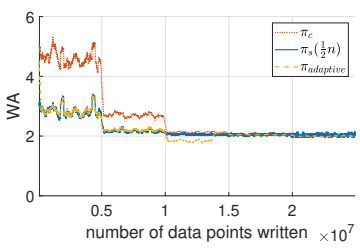


Fig. 10. WA of different strategies under dynamic delay distribution. $\pi_s(\frac{1}{2}n)$ is the original settings in Apache IoTDB, where the $n_{\text{seq}} : n_{\text{nonseq}} = 1 : 1$.

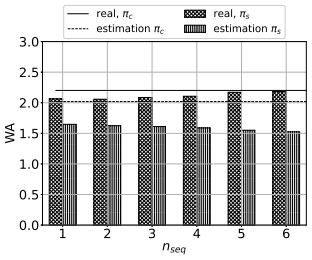


Fig. 11. WA under π_c and π_s on dataset S−9.

interval is shorter, e.g., in M7-M12 with $\Delta t = 10$. The differences between the predicated WA and the corresponding truth could also be relatively large, e.g., in M1 - M4 in Fig. 9. The reason is that as long as an *SSTable* contains subsequent data points, all of the data inside would be read and rewritten. Since the model is based on *subsequent data points*, it will introduce estimation error when rounding up. Nevertheless, the difference between the estimated WA and the real WA should be less than 1. When the generation time interval is short, e.g., $\Delta t = 10$, there would be more subsequent data points in each merge, hence resulting in higher WA. The relative difference between prediction and truth is thus smaller.

When the disorder is severe, such as M12, WA of n_{seq} from 50 to 400 exhibits U shape more obviously. That is because when n_{seq} is too large, the space left for C_{nonseq} is too limited, so that compaction happens more frequently. While on the other hand, if n_{seq} is too small, then C_{seq} will be flushed more often, which will also update the maximum generation time on the disk. So, more data would be considered out-of-order. When the "increase of out-of-order data" outweighs the "increase of n_{nonseq}", WA is intensified.

We also implemented an auto-tuning program to confirm the correctness of Algorithm 1. We used π_c to initialize the system, which then continuously collected delays when writing. If it finds that the distribution of delays changes, it would trigger the Separation Policy Tuning Algorithm (Algorithm 1) to update the policy. We use the notation π_{adaptive} to denote the recommended policy. Fig. 10 shows the results of WA

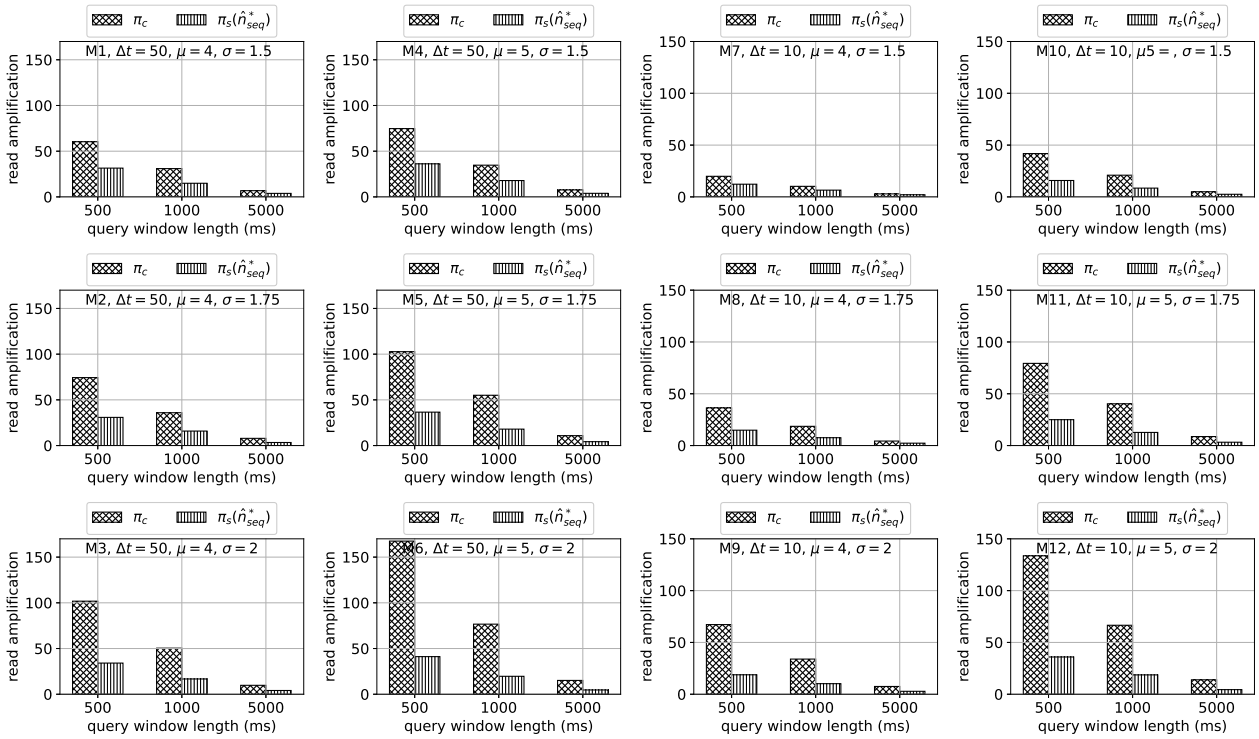


Fig. 12. Read amplification when executing the recent data query workload.

under a dynamic delay environment. The synthetic dataset was generated from delays of 5 different distributions. With fixed $\mu = 5$ and $\Delta t = 50$, the parameter σ was changed from 2, 1.75, 1.5, 1.25 to 1, respectively, for every 5,000,000 data points. The total writing times of all data points were recorded for each 512 data points to write from the user's view. We used a sliding window to smooth the WA for the records and plotted the data in Fig. 10. It shows that the auto-tuning method could detect the difference in the delays' distribution and switch different policies to reduce WA. It confirms the correctness of Algorithm 1.

Note that the methods in comparison in Fig. 10 are π_c the conventional policy without separation, and $\pi_s(1/2n)$ the separation policy with a straightforward space allocation used in the current version of Apache IoTDB. For $\pi_s(1/2n)$, the capacities of the in-order and the out-of-order *MemTable*s are both halves of the memory budget. Obviously, such a straightforward split of space may not be the best choice. As illustrated in Figure 7, WA changes with various n_{seq}, where $\pi_s(1/2n)$ may not lie in the middle of the U-shaped curve. By dynamically tuning the capacity of the in-order *MemTable*, the $\pi_s(\text{adaptive})$ may show even lower WA than the minimum one of π_c and $\pi_s(1/2n)$.

The estimations and real results of the WA of π_c and π_s over the real-world dataset `S-9` are shown in Fig. 11. The estimations show that the WA under π_s is lower than π_c, which is consistent with the real WA results. Because of the skewness, some data points suffering very long delays may have a lot of subsequent data points. Under π_c, those points may share the same subsequent data points. Since they are distributed in multiple merges, some subsequent data points would be rewritten multiple times. However, under π_s, those out-of-order data points are buffered and merged with the shared subsequent data points together, hence reducing the WA. The high proportion of the out-of-order data points also makes the advantage of π_s more prominent.

C. Writing Throughput Comparisons

To study the impact of π_s on writing throughput, we conducted experiments on IoTDB using the synthetic dataset. The implementation of π_s was a little different from the design that was introduced in Section III. That is, when a *MemTable* is full, the data will be flushed to a file on the disk on level 1. A compaction thread consumed the *SSTable*s on level 1, and organized them to new *SSTable*s on level 2 in the background. Therefore, on level 1, the SStables may have overlapping data with each other. But on level 2, there's no overlap at all. So, the writing will not be blocked to wait for compaction. Instead, new data points will be buffered in *MemTable*s before they are flushed again. The results of writing throughput are shown in Table III. There is no significant impact on the writing throughput because the compaction happens in the background. From the user's view, the throughput is calculated

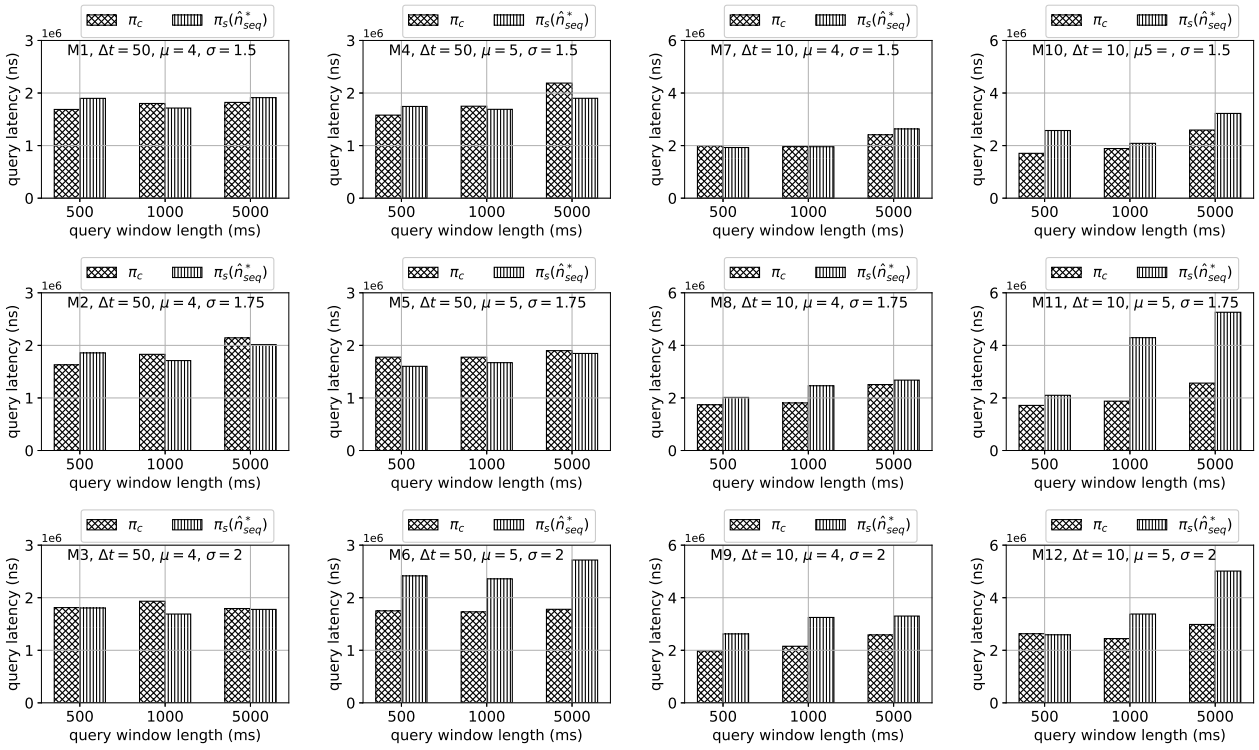


Fig. 13. Query latency (ns) when executing the recent data query workload on synthetic dataset.

TABLE III
WRITING THROUGHPUT (POINTS/MS) WHEN ADOPTING π_c AND π_s(SETTING OF IOTDB, I.E., $n_{\text{SEQ}} = \frac{1}{2}n$)

Dataset	π_c	π_s	Dataset	π_c	π_s
M1	89.98	89.94	M7	88.47	91.51
M2	90.60	88.07	M8	91.67	91.27
M3	92.52	91.17	M9	89.54	90.49
M4	83.56	89.73	M10	84.70	90.12
M5	90.71	90.51	M11	88.71	92.65
M6	88.53	89.55	M12	90.34	90.04

once the data are written to the database, while the compaction may not have happened yet.

D. Query Latency Comparisons

We design two categories of query workloads to study the impact of π_s on query performance. The experiments were conducted on IoTDB.

1) Recent data query workload: This workload simulated the queries in the real-time monitoring system, where the latest period of time-series is queried and visualized in real-time. The queries were generated when the data were being written to the database. The client recorded the maximum generation time currently written to the database. When writing the data, for every 100 ms, a query was generated and executed, with a predicate on generation time. We use different "window" lengths for the query, which was a period of time (500ms, 1000ms and 5000ms). For example, those statements were like

```
SELECT   *
  FROM   TS
 WHERE   time > (max_time - window)
```

When writing the data, we set the capacity of *MemTable* to be 512 data points under π_c. While under π_s, we used the values recommended by the system to set the capacity of C_{seq} and C_{nonseq}. The average read amplification is shown in Fig. 12. There are two remarkable phenomena: (1) Given a fixed window size, π_s would have less read amplification than π_c. That is because the *SSTable*s under π_s contain fewer data points. When querying data from the files, fewer useless data points were read. (2) The more extended the query window, the less the read amplification. That is because a longer query window means more data points to be queried from the database, whose number is increased by multiple times accordingly and is the divisor when calculating read amplification. Even though the number of *SSTable*s to be read will increase, the factor is less than that of the size of the result sets. Therefore, the overall read amplification is reduced.

The average query latency is shown in Fig. 13. There are two evident phenomena: (1) If the query windows are enlarged, then the query latency will also increase. That is because the number of data points to be queried is increased. (2) Under π_s, the query performance is negatively impacted. Even though the

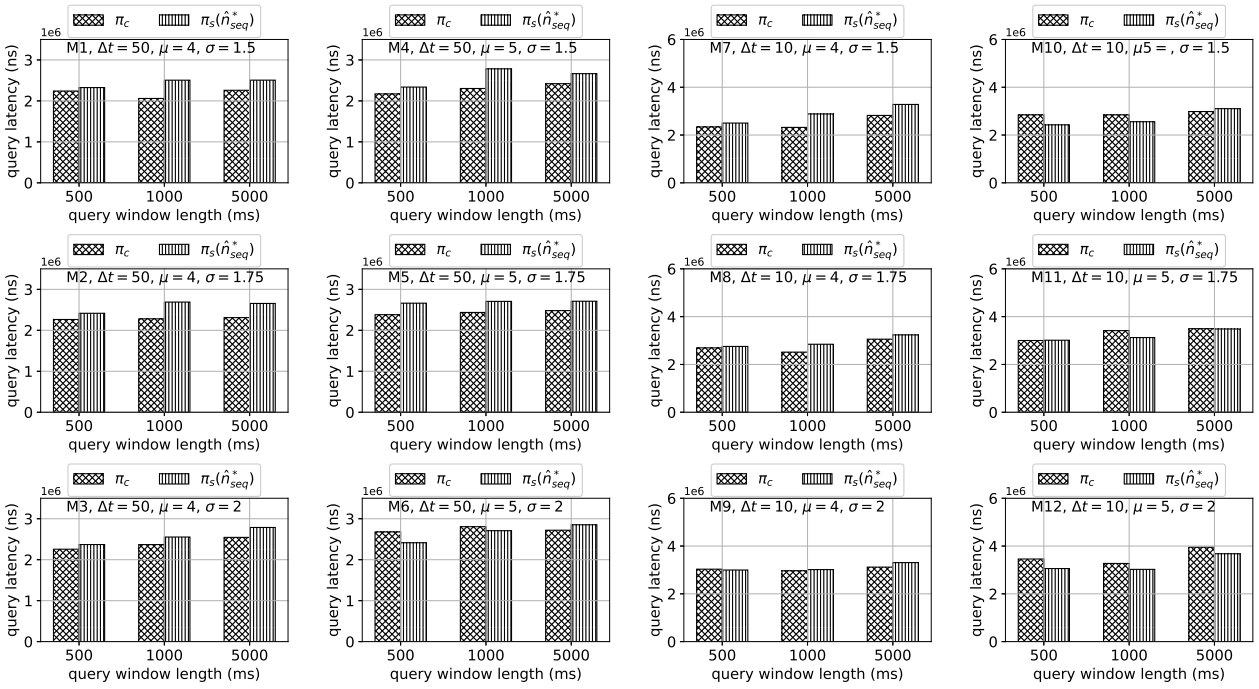
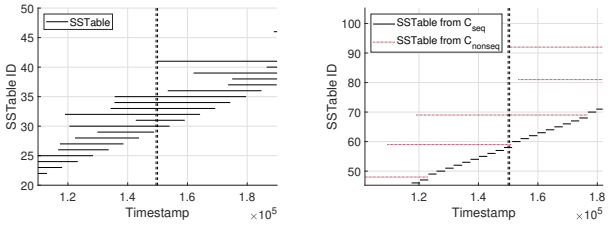


Fig. 14. Query latency (ns) when executing historical data query workload on synthetic dataset.



(a) The case of π_c. (b) The case of π_s.

Fig. 15. *SSTable* generation time range and the queried range.

read amplification under π_s is less than π_c, since the number of data points in each *SSTable* is reduced, the number of files to read is increased. Considering the overhead of disk seeks on HDD, the overall query performance is reduced.

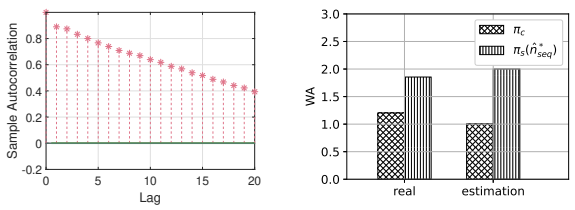
2) Historical query workload: This query workload was irrelevant to the latest generation time that had been written. Following the concept of "window" in Section V-D1, the lower bound of the constraints on time was generated randomly. The upper bound was the sum of the lower bound and the "window" length. It was guaranteed that the upper bound would not be greater than the maximum generation in the database. For example, those statements were like

```
SELECT  *
  FROM  TS
 WHERE  time>rand_value  AND time<rand_value+window
```

The results of the query latency are shown in Fig. 14. Compared with Fig. 13, it is obvious that under π_s, the system has better performance on historical query workload than the recent data query workload, sometimes even better than π_c (`M6`, `M11` and `M12` in Fig. 14). That is because under π_c, more *SSTable*s share the same queried period, and they are still in level 1, not compacted yet. Fig. 15 demonstrates an example to explain the situation. Each horizontal segment indicates an *SSTable* on the disk. Its left and right endpoints mean the earliest and latest generation time of that *SSTable*. The two vertical dashed lines indicate the lower bound and upper bound of the query. In Fig. 15(a), there are 7 *SSTable*s overlapping with the required range. While in Fig. 15(b), the number is 4. With fewer *SSTable*s to read, there will be fewer seeks on the disk. Therefore, the query is accelerated. The analysis is aligned with the case in [9]. But the smaller size of *SSTable* under π_s would increase query latency to some extent. We also find that based on `M1`, `M2`, `M4` and `M5`, the difference of π_s and π_c remains almost the same when compared to those in Fig. 13. That is because the σ of the delay distribution is small. Subsequently, the majority of data points are in order. Therefore, the overlap among *SSTable*s which are generated under π_c is not severe at all. The trouble caused by small *SSTable*s outweighs the benefits of π_s.

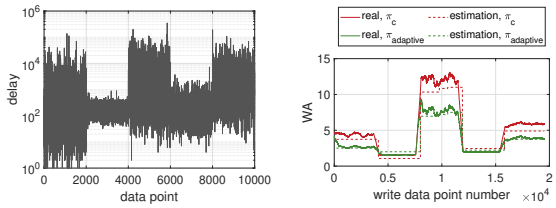
E. Robustness Evaluation

For the case that the delays of data points are not independent of each other, we use the `autocorr` function in MATLAB to test the real-world dataset `H` (the details of the dataset `H` are given in Section VI), in Fig. 16(a). The



(a) Autocorrelation function. (b) WA under π_c and $\pi_s(\hat{n}^*_{\text{seq}})$ on H.

Fig. 16. Performance on real-world dataset H with non-independent delays.



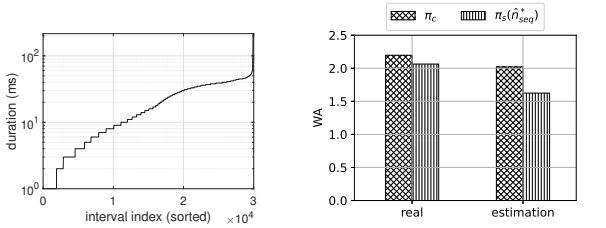
(a) Delays of the data points. (b) WA under π_c and π_{adaptive} on the synthetic dataset.

Fig. 17. The performance of dynamically determining separation policy over the synthetic dataset without a certain delay distribution.

correlated value on the Y-axis of the red star indicates the autocorrelation coefficient given the lag. A high absolute value of the autocorrelation coefficient, close to 1, indicates that there is a strong relationship. For example, when the lag is 1, the autocorrelation coefficient means that 2 consecutive delays in this time series have a strongly positive relation. Moreover, there are indeed two very close green lines near 0, which indicates the upper bound and lower bound for considering the delays as independent given the lag. Fig. 16(a) shows that in dataset H, the delays are not independent of each other. Fig. 16 shows the estimated WA and the corresponding truth. Remarkably, even though the real-world delays do not follow the independence assumption, the approximate computation can still successfully detect that the π_c outperforms $\pi_s(\hat{n}^*_{\text{seq}})$ in this case.

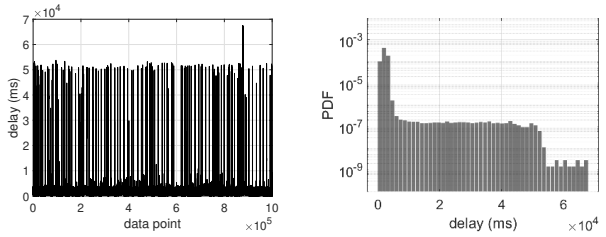
For the case that the delays do not obey a certain distribution, we synthetically generate a dataset, which is composed of 5 different delay distributions, changing over time as illustrated in Fig. 17(a). Fig. 17(b) demonstrates the results of WA when ingesting the dataset. Again, the estimation could successfully detect the change of the delay and dynamically adopt the best policy to minimize the WA.

For the case of data not generated at a certain frequency, we consider another real-world dataset S−9. The generation time intervals for two consecutive data points are collected, sorted in order of duration, and then visualized in Fig. 18(a). As shown, the generation time interval varies significantly from pair to pair. The results of WA estimation under π_c and $\pi_s(\hat{n}^*_{\text{seq}})$ as well as the real WA are shown in Fig. 18(b). Again, the estimation can successfully predict that the WA under $\pi_s(\hat{n}^*_{\text{seq}})$ is lower than π_c in this case.



(a) Generation time interval. (b) WA under π_c and $\pi_s(\hat{n}^*_{\text{seq}})$ on S−9.

Fig. 18. The performance on real-world dataset S−9 with data not generated at a certain frequency.



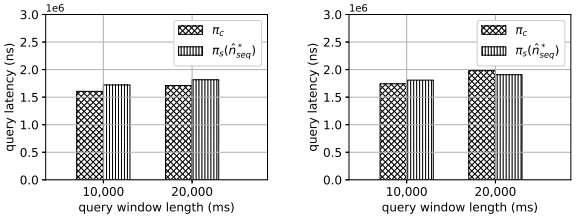
(a) Delays in H. (b) Delays' PDF in H.

Fig. 19. The delay set and its distribution in H.

VI. A Use Case in Vehicle Industry

To evaluate the effectiveness of π_{adaptive} in IIoT, we implemented a delay analyzer module in IoTDB based on Algorithm 1, deployed it, and conducted experiments on a real-world scenario. Our industrial partner is an enterprise in the area of IIoT. It builds the monitoring system for industrial vehicle vendors, who have first-hand data in the areas of IIoT. The company deploys one IoTDB instance for each vendor to store the data. The data are generated from the devices on vehicles, and then it is sent to the data center, where Apache IoTDB is deployed to store the time-series data. For each vehicle, more than two thousand time-series are recorded, such as the location and velocity of the vehicle, the temperature of the engine, and the status of the pilot lamp, etc. Unordered time-series data are common in this scenario, because the transition duration varies from point to point. More than one-third of the time-series contain out-of-order data points. Denote the dataset as H, which contains 1 million data points. The delays of each point are shown in Fig. 19(a), and the distribution is shown in Fig. 19(b). The generation time interval is one second.

According to the statistics in this experiment, the average delay of the out-of-order data points, referring to Definition 3 in Section II, is about 2.49s. The percentage of out-of-order data points is about 0.0375%. As illustrated in Figure 19(b), there exist some systematic patterns of the delays in the real-world dataset H. That is, most of the delays are indeed less than about 5×10^4ms. The reason is that normally the device would send the data points immediately when they are collected. However, when the network is unstable or the



(a) Query latency (ns) when executing recent data workload.

(b) Query latency (ns) when executing historical data query workload.

Fig. 20. Querying data in dataset H.

transmission fails, the device is able to buffer the data points locally. Moreover, a system triggers re-sending for about every 5×10^4ms, transmitting the buffered data points in a batch to the data center. Therefore, it appears some clear systematic delays about 5×10^4ms.

The results of WA are shown in Fig. 16(b). It shows that the analyzer module can effectively estimate the WA, and π_c requires less WA than $\pi_s(\hat{n}^*_{\text{seq}})$. In this case, the analyzer will choose π_c for IoTDB to reduce the WA. Fig. 20(a) shows the query latency when executing the recent data query workload on dataset H. The results are similar to the cases in the synthetic dataset. Fig. 20(b) shows the average query latency when executing the historical data query workload on dataset H. Compared to Fig. 20(a), the difference of query latency under π_c and π_s is reduced when the query window length is 10 s. When it is 20 s, π_s shows better query performance.

VII. Related Work

A. LSM-Tree-based Storage Engine

Although it has been decades since Log-Structured-Merge Tree (LSM-Tree) was proposed [13] in 1996, it is widely used in modern database systems for handling write-intensive workload, e.g., LevelDB [20], RocksDB [21], Cassandra [22], [23], to name a few. Some made exploration to reduce WA in LSM-tree [24]–[27]. WiscKey [27] separates the keys and values to only store keys in the LSM-tree, which lessens the size and makes the compaction less expensive. Luo *et al*. [12] reviewed modern LSM-based storage techniques and concluded the techniques for reducing WA, including tiering, merge skipping, and data skew exploitation. However, those were improvements on LSM-Tree. The characteristics and applications of LSM-tree in time-series were seldom considered. Moreover, [12] also studied the WA of leveled LSM-Tree, but in a general form, which was $O(T \cdot \frac{L}{B})$, where T is the capacity ratio from a higher level to a lower level, B is block size, L is the number of levels. However, it is not acute enough to detect the difference between π_c and π_s.

B. Time-series Database

With the rapid development of IIoT, the demand for industrial data storage has risen sharply, a lot of time-series databases came into being, e.g., InfluxDB [28], Prometheus [29], IoTDB [1], etc. However, Prometheus rejects to store out-of-order data. IoTDB [1] is an integrated data management engine designed for time-series data, aiming to provide excellent data ingestion and query performance. Experiments [30], [31] show that IoTDB is a promising database, which outperforms InfluxDB [28] and Open-TSDB [32] in many fields. Although it designed in-order and out-of-order *MemTable*s, the capacities are not tunable. Timon [9] is a new timestamped event database, that is also aware of out-of-order event data. It designs two kinds of *MemTable* in memory, one for the late event data, and another for the rest. The data persistence is based on LSM-Tree, too. They focus on data processing and analysis. The purpose of the separation is to reduce the time span of normal *SSTable*s.

C. Time-series Data Processing

Time-series data are widespread in life. It is observed that time-series data are not always strictly in order of a timeline [3]–[6], [9], [33]. Our preliminary study [34] shows that even the timestamps could be dirty. The disorder in time-series has attracted wide attention. Grulich *et al*. [4] even proposed a stream generator that could produce out-of-order data on purpose to simulate time-series data. FiBA [6] provided a sliding window aggregation algorithm for dealing with out-of-order data. Weiss [3] studied the dynamic time-out buffering algorithm for reordering the out-of-order data. Ji *et al*. [17] proposed *AQ-K-slack*, a buffer-based method to handle out-of-order data stream, which keeps low query latency but high accuracy for the user at the same time. The dataset they used is the time-series dataset proposed by Mutschler *et al*. [35]. The data were collected from sensors embedded in the football players' shin guards. More than 50% of the data were *late event*s [17]. These works dealt with out-of-order data in a time window. How to permanently store the data was not discussed.

VIII. Conclusion

To handle the unordered time-series data, Apache IoTDB adopts a separation policy to store the out-of-order data points and reduce query latency. However, it is concerned that how the separation policy would affect WA. Over the time-series data as well as the out-of-order data, a robust model can be built to estimate WA with and without the separation policy. To enable efficient determination, the model could be approximated under certain assumptions. Note that as an industrial paper, rather than proposing novel techniques for research problems, we focus on the practice of whether separating or not for lower write amplification. Therefore, we implement a delay analyzer in Apache IoTDB to choose the better policy so that the WA is reduced. Experiments confirm that the models can accurately predict the WA.

Acknowledgement: This work is supported in part by the National Key Research and Development Plan (2021YFB3300500, 2019YFB1705301, 2019YFB1707001), the National Natural Science Foundation of China (62021002, 62072265), BNR2022RC01011, the MIIT High Quality Development Program 2020 and the MIIT Industry Internet Innovative Development Program 2021.

REFERENCES

[1] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. A. McGrail, P. Wang *et al.*, "Apache iotdb: time-series database for internet of things," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2901–2904, 2020.

[2] H. Yuan and G. Li, "A survey of traffic prediction: from spatio-temporal data to intelligent transportation," *Data Sci. Eng.*, vol. 6, no. 1, pp. 63–85, 2021. [Online]. Available: https://doi.org/10.1007/s41019-020-00151-z

[3] W. Weiss, V. J. E. Jimenez, and H. Zeiner, "Dynamic buffer sizing for out-of-order event compensation for time-sensitive applications," *ACM Transactions on Sensor Networks (TOSN)*, vol. 17, no. 1, pp. 1–23, 2020.

[4] 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.

[5] 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.

[6] K. Tangwongsan, M. Hirzel, and S. Schneider, "Optimal and general out-of-order sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1167–1180, 2019.

[7] K. Liu, "Improvement and implementation of lsm on time series data storage," Master's thesis, Tsinghua University, 2019.

[8] T. I. Authors. (2021, May) Apache iotdb's comparison with tsdbs. The Apache Software Foundation. [Online]. Available: https://iotdb.apache.org/

[9] W. Cao, Y. Gao, F. Li, S. Wang, B. Lin, K. Xu, X. Feng, Y. Wang, Z. Liu, and G. Zhang, "Timon: A timestamped event database for efficient telemetry data processing and analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 739–753.

[10] Q. Mao, S. Jacobs, W. Amjad, V. Hristidis, V. J. Tsotras, and N. E. Young, "Experimental evaluation of bounded-depth lsm merge policies," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 523–532.

[11] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in lsm-tree based {KV} stores with matrix container in {NVM}," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 17–31.

[12] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[14] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *Proceedings of the Second International Conference on Distributed Event-Based Systems*, ser. DEBS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 265–275. [Online]. Available: https://doi.org/10.1145/1385989.1386023

[15] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, "Quality-driven processing of sliding window aggregates over out-of-order data streams," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 68–79. [Online]. Available: https://doi.org/10.1145/2675743.2771828

[16] L. Chen, "Research on out-of-order message of real-time streaming system in condition monitoring," Master's thesis, Tsinghua University, 2017.

[17] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, "Quality-driven continuous query execution over out-of-order data streams," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 889–894.

[18] D. Basin, F. Klaedtke, and E. Zălinescu, "Runtime verification of temporal properties over out-of-order data streams," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 356–376.

[19] W. Weiss, V. J. E. Jiménez, and H. Zeiner, "A dataset and a comparison of out-of-order event compensation algorithms," in *2nd International Conference on Internet of Things, Big Data and Security*, 2017.

[20] Google. (2021, May) google/leveldb: Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values. Google. [Online]. Available: https://github.com/google/leveldb

[21] Facebook. (2021, May) Rocksdb — a persistent key-value store. Facebook Open Source. [Online]. Available: https://rocksdb.org/

[22] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[23] Cassandra. (2021, May) Apache cassandra. The Apache Software Foundation. [Online]. Available: https://cassandra.apache.org/

[24] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb." in *CIDR*, vol. 3, 2017, p. 3.

[25] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, 2013, pp. 257–270.

[26] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 71–82.

[27] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[28] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and influxdb," *Studienarbeit, Université Libre de Bruxelles*, p. 12, 2017.

[29] B. Rabenstein and J. Volz, "Prometheus: A next-generation monitoring system (talk)." Dublin: USENIX Association, May 2015.

[30] Y. Cheng, M. Cheng, H. Ge, Y. Guo, Y. Hao, X. Sun, X. Qin, W. Lu, Y. Chen, and X. Du, "Midbench: Multimodel industrial big data benchmark," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2018, pp. 172–185.

[31] IoTDB. (2021, May) Apache iotdb's comparison with tsdbs. The Apache Software Foundation. [Online]. Available: https://iotdb.apache.org/UserGuide/Master/Comparison/TSDB-Comparison.html

[32] OpenTSDB. (2021, May) Opentsdb - a distributed, scalable monitoring system. The OpenTSDB Authors. [Online]. Available: http://opentsdb.net/

[33] A. Dey, K. Stuart, and M. E. Tolentino, "Characterizing the impact of topology on iot stream processing," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018, pp. 505–510.

[34] S. Song, Y. Cao, and J. Wang, "Cleaning timestamps with temporal constraints," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 708–719, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p708-song.pdf

[35] C. Mutschler, H. Ziekow, and Z. Jerzak, "The debs 2013 grand challenge," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 289–294.