TimeUnion: An Efficient Architecture with Unified Data Model for Timeseries Management Systems on Hybrid Cloud Storage

Zhiqi Wang
The Chinese University of Hong Kong
zqwang@cse.cuhk.edu.hk

Zili Shao
The Chinese University of Hong Kong
shao@cse.cuhk.edu.hk

ABSTRACT

Timeseries management systems have attracted considerable attention during the last decade with the rise of IoT and performance monitoring. With the rapidly increasing data scale in the production environment, deploying timeseries management systems on cloud with cloud storage is a natural trend because of its high availability, reliability, and scalability. However, the state-of-the-art designs are not tailored for cloud environments; they suffer from the limited number of timeseries a single compute node can handle because of the imbalanced resource usage.

In this paper, we present TimeUnion, an efficient timeseries management system tailored for hybrid cloud storage services (e.g. fast cloud block stores and slow cloud object stores). First, we propose our unified data model to represent both independent timeseries and groups of timeseries, which is capable of handling diverse use cases of timeseries. Second, since the main bottleneck of the current timeseries systems is the overuse of memory, we introduce our exploration on memory-efficient data structures for timeseries to maximize the number of timeseries a compute instance can maintain. Third, to absorb the memory data chunks quickly, we present our time-partitioned LSM-tree with tailored architecture, compaction mechanism, out-of-order data handling, and dynamic level size adjustment for timeseries data. We prototype TimeUnion with C++ from scratch and evaluate it on AWS EC2 with AWS EBS (cloud block store) and AWS S3 (cloud object store). Compared to the storage engine of Cortex, TimeUnion can handle at least 5× more timeseries, and achieve at least 24.8% higher insertion throughput and 49.8% lower query latency. We have released the open-source code of TimeUnion for public access.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

timeseries systems, cloud storage

ACM Reference Format:
Zhiqi Wang and Zili Shao. 2022. TimeUnion: An Efficient Architecture with Unified Data Model for Timeseries Management Systems on Hybrid Cloud Storage. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526175

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526175

1 INTRODUCTION

Timeseries management systems including performance monitoring systems and timeseries databases play a critical role in IoT and performance monitoring[7, 13, 23, 29, 49, 54, 58, 60]. As data scale increases, deploying the system on cloud becomes a promising option. However, since the state-of-the-art timeseries management systems are mostly targeted at the local machines with locally attached storage, performance degrades when directly deploying them on cloud, especially with hybrid cloud storage with different performance/cost trade-offs.

When deploying systems on cloud, computation and storage are commonly separated. Computation is handled by cloud computing services with attached memory (e.g. virtual machines like AWS Elastic Cloud [3], serverless functions like AWS Lambda [5]). For storage, mainstream cloud storage suppliers mainly provide block storage (e.g. AWS Elastic Block Store (EBS) [4], Google Persistent Disk [15], and Azure Disk Storage [43]) and object storage (e.g. AWS Simple Storage Service (S3) [6], Google Cloud Storage [16], and Azure Blob Storage [42]). Block storage is orders of magnitude faster than object storage, while it is at least 4× more expensive. As a result, users commonly leverage hybrid cloud storage (i.e. combining block and object storage) to strike for a better cost efficiency especially when there is a large volume of hot and cold data in data-intensive applications.

The design of the state-of-the-art open-sourced timeseries management systems such as Prometheus and InfluxDB borrows the philosophy of log-structured merge-tree to pursue a high insertion throughput. Data is partitioned based on time and is first batched in memory before being flushed to disks to form a new partition. Besides, each partition is self-contained with inverted index and data samples. Within the design, there are several assumptions. First, it mainly handles medium data scale (e.g. less than one million timeseries with several days of data). Second, the memory volume is large enough to handle all inverted indexes, metadata, and data samples of the most recent time partition. Third, it assumes that only one type of persistent storage is utilized without considering leveraging tiered storage. Finally, it assumes that timeseries are handled and stored independently. There are several critical issues when deploying the existing systems on cloud to manage a large number of timeseries as follows.

Unbalanced resource utilization. In the design of the existing timeseries management system, an inverted index is built on the fly for each tag pair of each timeseries in the memory partition. In our observation, such an index occupies a large volume of memory, especially with high-cardinality timeseries data (i.e. millions of unique

tag pairs of all timeseries). Besides, for on-disk partitions, metadata (e.g. tag pairs and symbols) is commonly loaded into memory for accelerating querying, which incurs non-negligible memory usage. In addition, to strike for a high compression ratio, data samples of each timeseries are first cached and compressed with relatively large in-memory chunks (e.g. 120 samples in Prometheus). Thus, the memory usage can be explosively increased with the number of timeseries managed by the system, and the system can easily hang with exhausted memory.

Unexplored multi-tiered cloud storage. The deployment of timeseries management systems on multi-tiered storage and hybrid cloud storage is not explored. New challenges emerge with the cost/performance gap when bringing hybrid cloud storage. First, we need to swiftly handle hot/cold data separation and migration in the system write path without blocking data insertion. Second, given the limited fast storage, a new mechanism needs to be designed to dynamically adjust the data size in fast storage. Third, with most cold data maintained in the slow storage, optimizations on the system query path are critical (e.g. caching mechanism and caching granularity). Thus, a new design for timeseries management system to fully exploit different cloud storage tiers is urgently needed.

Limited group support. Timeseries group is naturally formed in numerous real-world use cases (e.g. metrics from the same virtual machine, docker container, IoT sensor, etc.). However, the existing systems such as Prometheus and InfluxDB physically manage and store each timeseries independently. At the data model level, InfluxDB supports simple grouping that timeseries share a set of tag pairs and have their own fields/measurements. However, this scheme can not cover the case that timeseries in a group also have their own unique tags, which is a common case in performance monitoring.

To address the above-mentioned issues, we present TimeUnion, an efficient timeseries management system with a unified data model and hybrid cloud storage support. The contributions are summarized as follows:

- **Cloud storage characteristics.** We conduct a series of experiments to explore the characteristics of cloud storage. Combining with the characteristics of timeseries, we generalize key principles to design a timeseries management system on cloud.
- **Unified data model.** In the high-level data model, we propose our group model abstraction compatible with the existing tag-based identifier, with a natural transition from a single timeseries to a group.
- **Memory efficient data structures.** To address the issue of high memory usage in the existing systems, we redesign memory-efficient data structures for inverted index and data samples, and we extend the memory mapping support for them.
- **Elastic time-partitioned LSM-tree.** To avoid batching an excessive number of data chunks, we design a time-partitioned LSM-tree to absorb those finished data chunks with high insertion throughput. Since timeseries data are naturally ordered by timestamps, we partition the time range of each level and apply different compaction mechanisms for different storage tiers. Specifically, the hot/cold data separation can be naturally done with our three-level design (i.e. recent data in levels 0 and 1 with fast storage; cold data in level 2 with slow storage). Since the majority of data are ordered with high data locality after the compactions in the first two levels, we only maintain one level (i.e. level 2) on slow storage, thus significantly reducing unnecessary compactions and data traffic to slow storage. With this design, out-of-order data and data retention can be efficiently handled based on corresponding time partitions. Besides, considering the high cost of fast storage, the fast storage usage of the tree can automatically adapt to a predefined threshold by dynamically adjusting the partition length in different levels.

2 BACKGROUND AND MOTIVATION

2.1 Cloud Storage Exploration

To facilitate the design of TimeUnion, we conduct a series of experiments on AWS EBS and AWS S3 (widely used cloud block and object storage) to explore the characteristics of cloud storage.

First, we show the pricing of different AWS storage per GB per month in region ap-northeast-1 (Tokyo) in Figure 1a. For the typical cloud storage, the price of AWS EBS (block storage) is around 4× more expensive than that of AWS S3 (object storage). Besides, we have a simple estimation for the memory prices based on the price difference among t3 instances with different memory volumes of ElastiCache and EC2, and we can see they are at least two orders of magnitude more expensive than EBS, which matches the price difference between mainstream RAM and SSD available on the market.

Second, we investigate the read and write performance of block storage and object storage, respectively. We launch an m5.2xlarge EC2 instance on region Tokyo with 200GB EBS volume attached. We set up the S3 bucket in both Tokyo and Virginia regions. As shown in Figure 1b, for small writes, EBS is at least three orders of

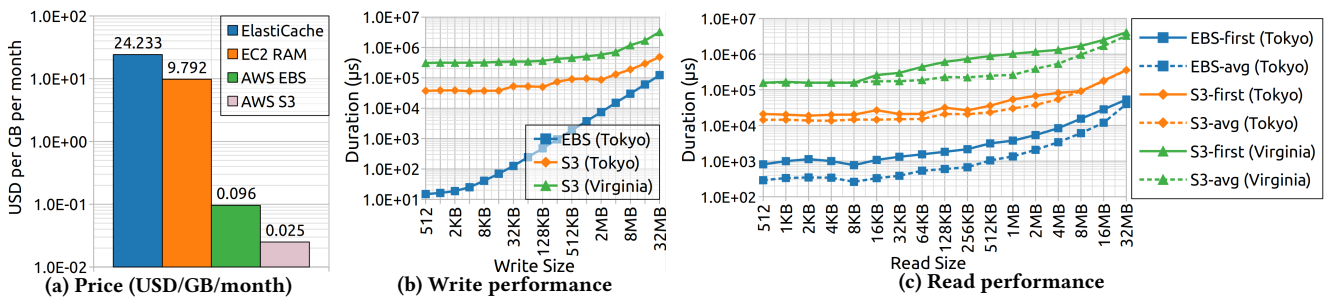


Figure 1: Cloud storage comparison.

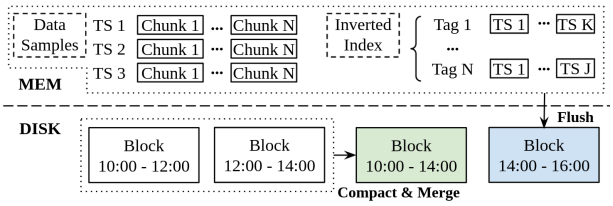


Figure 2: Timeseries management system architecture.

magnitude faster than S3. Although the gap gradually decreases as the write size increases because of the high network bandwidth of S3, EBS is still 3× faster than S3 for 32MB write. As shown in Figure 1c, EBS is 30× faster than S3 on average. Besides, the first read is 1.8× and 71% slower than the following reads for EBS and S3. In addition, for both EBS and S3, the read latency is stable when the read size is smaller than 16KB; this can be leveraged as the unit for efficient reading.

2.2 Timeseries Management System

In this section, we discuss the data model of timeseries and the design of the state-of-the-art timeseries management systems.

Data model. A timeseries consists of two components, namely identifier and data samples. An identifier is a set of tag pairs, which are handled differently in the data models of various timeseries management systems. For Prometheus [49], a performance monitoring system, the identifier is composed of a metric name and a set of tags, and there is no support for timeseries grouping. For InfluxDB [29], a timeseries database, they utilize a measurement name to possibly describe a group of timeseries. For a group of timeseries, they share a set of tag pairs, and they are differentiated only by the field name, which is not sufficient for real-world use cases. For instance, to differentiate different timeseries in the CPU resource monitoring group, we need extra tag pairs such as CPU core ID and CPU mode (e.g. idle, user, kernel, etc.) [48].

For each data sample, it contains a timestamp (i.e. 64-bit integer) and metric value (i.e. 64-bit floating-point number). To strike for a high compression ratio, a relatively large number of data samples (e.g. 120 samples in Prometheus) of the same timeseries are compressed into chunks based on the compression algorithms proposed in Facebook Gorilla [46] (i.e. delta-delta for timestamps and XOR'd for metric values). Both Prometheus and InfluxDB store data samples of timeseries independently, even though InfluxDB has a logical group view. Heracles [61] proposes a group storage model for performance monitoring timeseries to strike a balance between compression and query performance. However, for the high-level data model, it is still based on Prometheus without modifications on tags management.

System architecture. Timeseries management systems need to guarantee high insertion throughput for a large number of timeseries with frequent data sample generation. As a result, as shown in Figure 2, current systems batch writes and build indexes on the fly for incoming timeseries in memory. After a specific period (e.g. 2 hours in Prometheus), all the memory data are flushed to disks to form a self-contained partition with a persistent index corresponding to a specific time range. Besides, for the on-disk blocks, they will be merged into larger blocks when the number of them reaches a specific threshold.

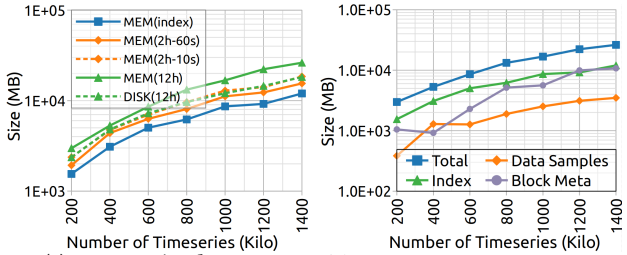
However, such architecture exposes several issues. First, it suffers from high memory pressure because data samples and indexes from all timeseries need to be first batched in memory. The memory usage can explosively increase with the number of timeseries and the data sample density. Second, data flushing can severely affect the system's performance. During data flushing, the system needs to flush and clean all the related in-memory data structures, which incurs severe contention with the incoming insertion and queries at that time. Third, it is awkward to handle out-of-order data samples (e.g. Prometheus does not even support this). The out-of-order data samples can affect the decision of the timing to flush the in-memory data, and we need an extra mechanism to batch and flush those out-of-order data samples. For instance, we need to maintain an extra space in memory for those out-of-order timeseries, and during data flushing, we need to create extra data chunk files and indexes under the existing blocks.

Cloud-based timeseries management system. Even though many companies provide services for managing timeseries data on cloud, their implementations are mostly closed-sourced. For the open-source cloud-based timeseries management systems, Thanos [47] and Cortex [23] are two popular ones in recent years. Thanos launches federated Prometheus with sidecar data uploading components, and data chunks from Prometheus are pushed to the cloud storage (e.g. S3, GCS, Azure). Compared to Thanos, Cortex is more flexible; it allows users to insert their custom timeseries via the Prometheus remote write API [9]. However, they still utilize the storage engine of Prometheus to handle the timeseries data by simply wrapping the file operations with cloud storage APIs, such that the disk files generated by the storage engine are uploaded to cloud storage. Thus, they suffer from the same limitations as mentioned in the previous section.

2.3 Log-Structured Merge-Tree

Log-structured merge-tree [45] is renowned for its decent insertion performance with acceptable query performance degradation. Since the born of LevelDB and RocksDB [51, 55], there are extensive research works on the read/write/space amplification and the deployment on different storage devices of LSM-Tree-based key-value stores [10–12, 14, 17–21, 30, 31, 36–41, 50–52, 55, 56, 63–68, 71, 72].

Data insertions are first handled by the in-memory data structure - *MemTable*, which sorts the key-value pairs in lexicographical order (i.e. skip list in LevelDB and RocksDB). When the MemTable is full (i.e. 64MB), it is treated as an *Immutable MemTable* and a new MemTable is created to handle the new data insertion. Then, the Immutable MemTable is flushed to a disk file (*SSTable*) in level 0, which contains a set of data blocks indexed by the index block, and a filter block with a bloom filter to accelerate queries. When a level accumulates too many SSTables, or the overall size of SSTables exceeds a specific threshold, it will trigger a level-based compaction, which selects an SSTable in that level as the victim, and at the same time searches for all SSTables in the next level that overlaps with the victim SSTable. Then, they will be merged, and new SSTables will be generated at the next level.



(a) Memory/Disk usage (b) Memory usage breakdown

Figure 3: Resource usage of Prometheus tsdb.

2.4 Motivation

To motivate the design of TimeUnion, we conduct a series of experiments on the state-of-the-art timeseries management systems, and a simple combination of timeseries management system and LSM-Tree.

Challenge 1: High memory pressure. In this experiment, we load different numbers of timeseries (each with 20 tags) into the storage engine of Prometheus (i.e. Prometheus tsdb). As shown in Figure 3a, when there is only index and without any data samples, the memory usage linearly increases with the number of timeseries. Then, for each timeseries, we insert 2 hours of random data samples, with data intervals as 10 and 60 seconds, respectively. The overall memory usages of 10 and 60 seconds sample intervals are 51% and 31% higher than that of the case when there is no data sample. The memory usage can be further increased when increasing the time span of data samples to 12 hours because Prometheus needs to load the metadata of the flushed partitions.

Then, we break down the memory usage of the case with 12-hour data samples and 60-second data interval. We observe that the inverted index, block metadata, and data samples account for 51%, 34%, 15% of memory usage, respectively. The main reason for the high memory usage of the inverted index and block metadata is that they are maintained by nested hash tables, which require much extra space to reduce the collision rate.

Therefore, exploiting more memory-efficient data structures to handle the inverted index and block metadata is urgently needed. Besides, we also need to reduce the size of cached data samples by moving the compressed chunks to persistent storage once they are finished.

Challenge 2: Efficient tuple flushing. To avoid batching all the compressed data chunks for a relatively long time, a natural approach is to flush those close chunks to an external high-throughput data structure, such as LSM-Tree. Thus, we conduct experiments to

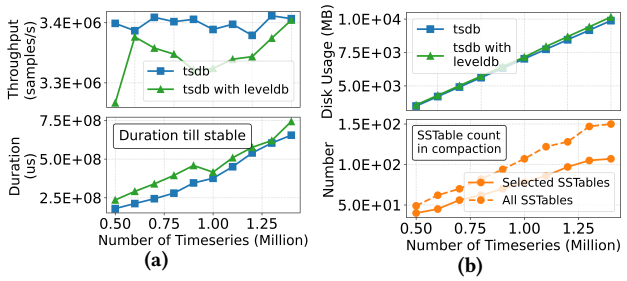


Figure 4: Prometheus tsdb with LevelDB as storage.

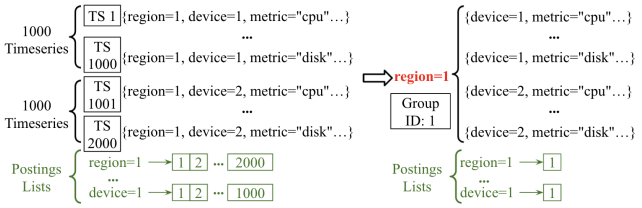


Figure 5: A tags grouping example.

investigate the capability of utilizing the LSM-Tree as the storage data structure for the timeseries management system. We integrate a Golang version LevelDB implementation [59] into Prometheus tsdb. For each compressed chunk, we generate a ULID [44] as the key, and insert the key-value pair into LevelDB. We insert different numbers of timeseries, each with 5 tags and 12 hours data samples with 60 seconds as the data interval, and we compare this integration with the original Prometheus tsdb.

As shown in the top graph of Figure 4a, the insertion throughput of the integration prototype is only 1.6% lower than that of Prometheus tsdb. As shown in the bottom graph of Figure 4a, LevelDB spends 18% more time to finish all the compactions compared to Prometheus tsdb; this can potentially affect the system's performance because compaction can be expensive when bringing slow cloud storage. As shown in the top graph of Figure 4b, LevelDB only writes 2.4% more data to disk compared to Prometheus tsdb. As shown in Figure 4b, on average, we need to read 36% more SSTables in a compaction. Besides, we observe that for each compaction, at least one overlapping SSTable needs to be read from the next level.

In conclusion, LSM-Tree is capable of handling a large volume of timeseries data. However, considering the deployment on hybrid cloud storage, the original compaction mechanism needs to be redesigned to reduce inefficient data reading during compaction. Besides, a new mechanism is needed for LSM-Tree to handle the out-of-order timeseries data.

Challenge 3: Grouping. Figure 5 shows an example of timeseries from two devices under the same region. Without grouping, the lengths of postings lists of "region=1" and "device=1" are both 2000. If we gather all timeseries into a group with shared group tags as "region=1" and group ID as 1, the lengths of posting lists of "region=1" and "device=1" both become 1 because they both indicate group 1. Besides, with grouping, we only need to store the tag "region=1" once for a whole group. However, there are two challenges. First, we need another level of indexing to locate a specific timeseries inside a group. Second, when connecting the group data model with physical data storage, we need to consider (1) the data samples of the new and missing timeseries in the current tuple; (2) the out-of-order data samples.

3 DESIGN

3.1 Unified Data Model

In this section, we first present the logical view of our unified data model represented by tags. Then, we introduce the physical view of the data model, and discuss cases of expanding group and handling missing/out-of-order timeseries data.

Logical view. In our unified data model, timeseries can be represented as an individual or one timeseries belonging to a specific

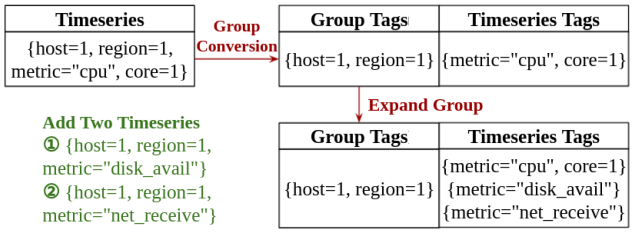


Figure 6: Timeseries and timeseries group.

group. The timeseries identifier is represented by a set of tags, and they can be converted into the group representation as shown in Figure 6. We need to specify the shared group tags for all the timeseries in the group. When adding new timeseries into the group, the group tags are extracted, while the other tags are used to uniquely identify the timeseries inside the group. For each group, we assign a unique ID for it, and the group ID is utilized as the postings ID when building the inverted index for all the tags of the group timeseries.

Physical view. As shown in Figure 7, we manage individual timeseries and groups separately. For a group, we deduplicate the redundant timestamps by sharing the same timestamp column; while the metric values of each timeseries are stored independently. For the data insertion to a group, there are four cases as follows:

(1) **Normal insertion:** We append the timestamp to the timestamp column, and metric values to the columns of all timeseries in the group separately.
(2) **Insertion with new timeseries:** First, we extract the unique tags from the new timeseries and insert them into the timeseries tags array of the group. Second, we create a new metric value column for it, and fill the previous values in this column with *NULL* values, where we extend the XOR'd algorithm in Gorilla [46] with an extra control bit to support NULL value. Finally, a normal insertion is performed.
(3) **Insertion with missing timeseries:** For those timeseries missing in this round, we fill them with NULL values, as shown in *TS3* in Figure 7.
(4) **Out-of-order insertion:** We search for the corresponding slot according to the timestamp, and we determine whether to replace the existing values or to insert new ones. Besides, it may trigger an early flush of the current chunk if the insertion timestamp is older than the current time partition, which will be discussed in §3.3.

When the current chunk is full (i.e. filled with 32 samples), we concatenate and serialize timestamp chunk and metric values

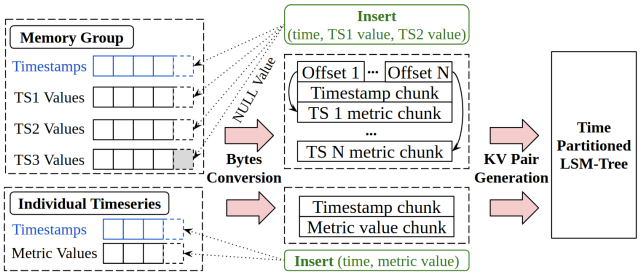


Figure 7: Data flow in the model.

Table 1: Grouping analysis notations.

N	Number of timeseries
T	Average number of tags per timeseries
S_p	Average size (number of bytes) per posting list entry
S_t	Average size (number of bytes) per tag
S_g	Average number of timeseries in a group
T_g	Average number of group tags per group
T_u	Average number of unique tags per group
$Cost_{EBS}$	Latency of reading 1 byte (1/bandwidth) in EBS
$Cost_{S3}$	Latency of 1 Get request (1 data block) to S3
P	Number of time partitions covered in a query
S_{data}	Raw data size per timeseries per time partition
S_{block}	Data block size in SSTables (4KB by default)
L	Number of located timeseries in a query
G	Number of located groups in a query
R_1	Compression ratio of individual timeseries model
R_2	Compression ratio of grouping model

chunks into a byte array, generate key-value pair, and insert it into the time-partitioned LSM-Tree.

Grouping analysis. Next, we analyze the cost of grouping in terms of index space and querying latency, and provide guidelines for users to effectively utilize grouping. The notations are shown in Table 1.

First, we measure the index space cost with the total size of all posting list entries and all tags. The cost without grouping is as follows:

$$Cost_{s1} = N \cdot T \cdot (S_p + S_t) \tag{1}$$

In Equation 1, $N \cdot T \cdot S_p$ is needed as for each tag in each timeseries, a posting list entry is required for mapping the tag to the timeseries ID (as the inverted index), while for each timeseries, we need to store all of its tags as well (so the total tag size from all timeseries is $N \cdot T \cdot S_t$). With grouping, the cost is shown as follows:

$$\begin{aligned} Cost_{s2} = &\frac{N}{S_g} \cdot T_u \cdot S_p + (T - T_g) \cdot N \cdot S_p \\ &+ \frac{N}{S_g} \cdot T_g \cdot S_t + (T - T_g) \cdot N \cdot S_t \end{aligned} \tag{2}$$

In Equation 2, the first line shows the total size of posting list entries of the first and the second level indexes. The posting list entries in the first level index represent group IDs (totally $\frac{N}{S_g}$). Each group contributes T_u (the number of unique tags of a group after deduplication) posting entries to the first level index (a group can be viewed as a big timeseries and each unique tag occupies one entry that points to this group ID). The second level index inside a group is to locate timeseries IDs of group members. Thus, each timeseries contributes $(T - T_g)$ entries. The second line shows the total size of tags where we only need to store the grouping tags once for each group (totally $\frac{N}{S_g} \cdot T_g$); for the other tags of each timeseries, we need to store them individually ($(T - T_g) \cdot N$).

Grouping can save index space when there is a decent number of grouping tags or duplicate tags inside a group. Specifically, grouping can benefit if $S_g > (\frac{T_u}{T_g} S_p + S_t)/(S_p + S_t)$. For instance, this is true for the TSBS DevOps data set (§4.3) as $S_g = 101$, $T_u = 118$, $T_g = 1$, $S_p = 8$ and $S_t = 15$.

Second, we analyze the querying cost with the latency of data reading. For EBS, since it behaves like a local disk, we measure its cost with the reciprocal of the bandwidth. For S3, we measure the cost with the latency of each *Get* request, which fetches one SSTable data block. Then, for the recent data querying, we only consider $Cost_{EBS}$ because recent data are stored on EBS. While for the long-range queries, $Cost_{S3}$ dominates as mentioned in §2.1. For a query covering P time partitions, if without grouping, the cost is shown as follows:

$$Cost_{q1} = \begin{cases} L \cdot P \cdot \dfrac{S_{data}}{R_1} \cdot Cost_{EBS}, & \text{data on EBS} \quad (3) \\ L \cdot P \cdot \left\lceil \dfrac{S_{data}}{S_{block} \cdot R_1} \right\rceil \cdot Cost_{S3}, & \text{data on S3} \quad (4) \end{cases}$$

Similarly, with grouping, the cost is as follows:

$$Cost_{q2} = \begin{cases} G \cdot P \cdot \dfrac{S_{data} \cdot S_g}{R_2} \cdot Cost_{EBS}, & \text{data on EBS} \quad (5) \\ G \cdot P \cdot \left\lceil \dfrac{S_{data} \cdot S_g}{S_{block} \cdot R_2} \right\rceil \cdot Cost_{S3}, & \text{data on EBS} \quad (6) \end{cases}$$

In Equations 5 and 6, with grouping, we have $S_{data} \cdot S_g$ because based on the key format (i.e. group ID + the tuple starting timestamp, which is discussed in details in §3.3), the data of all timeseries of a group in the same tuple are stored together.

Grouping can have a better long-range query performance if the target timeseries are located in fewer groups (e.g. TSBS pattern 5-1-24 in Table 2). Since a time partition only stores a relatively small amount of data for each timeseries (S_{data}) and $R_2 > R_1$ (e.g. 35× compared to 10× in TSBS benchmark), the difference between $\lceil S_{data}/(S_{block} \cdot R_1) \rceil$ and $\lceil S_{data} S_g/(S_{block} \cdot R_2) \rceil$ (e.g. 1 versus 2 in TSBS benchmark) is commonly small with the extra divisor S_{block}. Thus, when G is smaller than L, grouping will have lower cost. For the short-range query, because the cost relates to the data volume, grouping may perform worse when the number of target timeseries in the same group is small. Besides, if $Cost_{s2} < Cost_{s1}$, as the number of timeseries increases, the grouping can perform better because of the higher turning point of triggering swap in the memory-mapped index (e.g. query 1-8-1 in Figure 14).

3.2 Memory Efficiency Exploration

In this section, we redesign the in-memory data structures to reduce the memory footprints of the inverted index, block metadata, and data samples mentioned in §2.4.

Inverted index. Instead of creating a separate index for each time partition and loading it into memory as Prometheus, we manage a single global inverted index in memory. As discussed in §2.4, the nested hash table is not an efficient solution to handle a large

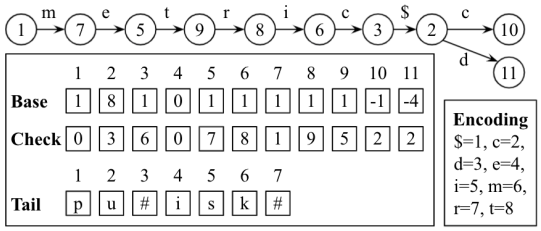


Figure 8: Double-array trie.

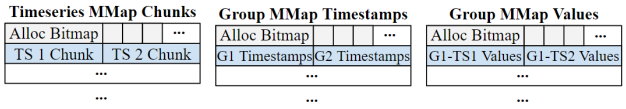


Figure 9: Memory-mapped file arrays for data samples.

number of tag pairs. Since a tag pair is a character strings after concatenating the tag key and value, Trie [8, 34] is a decent choice to index tag pairs with support of prefix search. In our implementation, we choose double-array trie [70] for its compact storage format on a contiguous memory area.

Figure 8 shows an example of storing two tags (metric="cpu", metric="disk") in a double-array trie where we concatenate tag key and tag value with delimiter '$'. Double-array trie essentially is a finite-state machine, which consists of three arrays. First, we need to encode the characters into digital indexes as shown at the top of Figure 8. We start from the root of the trie - the first slot of *Base* array, as shown in the state machine in the middle of Figure 8. With formula $state(xy) = Base(x) + index(y)$, we can calculate the state of character 'm' is 7, which means we need to move to slot 7 of the *Base* array for further traversal. The corresponding slot of the *Check* stores the slot id of the parent state (e.g. 1 in slot 7 means the parent of state 'm' is the root). The traversal stops until we reach the negative-value *Base* slot (e.g. 10 for 'c' and 11 for 'd'). Then, we use the absolute value of the negative number as the slot id, and go to the tail array to read the remaining characters (e.g. "pu" and "isk" starting from slot 1 and 4, respectively).

Considering the size of trie can still grow huge with millions of tag pairs and trigger the *out-of-memory (OOM)* problem, we store these three arrays in three dynamic mmap (memory-mapped) file arrays. Specifically, each mmap file can handle one million slots; when more slots are needed, we create new mmap files and append them to the corresponding arrays.

Timeseries tags. Through the inverted index, we can obtain the postings list, and through the postings IDs in the postings list, we can locate the targeted timeseries. For each timeseries, it is managed as a memory object, which contains all tag pairs and in-memory data samples of that timeseries. Similarly, to avoid the *OOM* problem, we store these tags in mmap files.

Data samples. For each timeseries and group, we batch a small number of data samples (i.e. 32) in memory for on-the-fly compression and efficient flushing. This number can be adjusted by users for the trade-off between compression ratio and memory usage (i.e. larger chunks have a better compression ratio). Figure 9 shows different mmap file arrays for timeseries and groups, which can be dynamically expanded. A mmap file is split into a series of fixed-size chunks to store the compressed bytes of data samples, and the header contains a bitmap to indicate the occupation of each chunk. We store timestamps and metric values together into the same chunk for the individual timeseries. While we separate the share timestamps and the metric values for the group. Finally, as shown in Figure 7, when the current chunk is full, it will be serialized to byte array as the value of a key-value pair and be inserted into the time-partitioned LSM-tree, and the corresponding area of the mmap file will be cleaned.

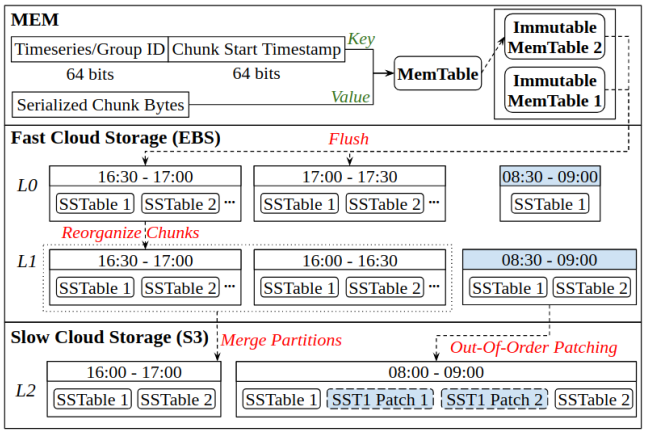


Figure 10: Time-partitioned LSM-tree.

3.3 Elastic Time-Partitioned LSM-Tree

In this section, we present an elastic time-partitioned LSM-tree tailored for hybrid cloud storage, which applies different compaction mechanisms for different storage tiers and can dynamically adapt to a predefined threshold of fast storage usage.

Key-value format. With keys sorted in SSTables, we can group and sort timeseries data efficiently. As shown at the top of Figure 10, we store the timeseries/group ID and the starting timestamp of the timeseries/group chunk in the first and the second 64 bits with big-endian encoding, respectively. Thus, we can group the chunks from the same timeseries/group together, and meanwhile sort them based on starting timestamps, which provides data locality to accelerate scan querying. Besides, such a key format can fully utilize the prefix compression of keys in LevelDB because the 64-bit ID and a large portion of timestamp fields can be saved when storing a series of chunks from the same timeseries/group continuously. For the value of a key-value pair, we directly use the serialized bytes of the data chunk of a specific timeseries/group.

Architecture. As shown in Figure 10, the LSM-tree only maintains three levels that reside in different storage tiers. Level 0 and level 1 manage a relatively small volume of timeseries data (i.e. the recent 2 hours) on the fast cloud storage (i.e. AWS EBS). For the older data, we only manage them in one level (i.e. level 2) stored on the slow cloud storage (i.e. AWS S3) to avoid unnecessary compaction.

Since timeseries data is sorted by timestamp, we partition SSTables based on different time ranges in different levels to control the size of each level. For level 0 and level 1, the time partition length starts with a relatively small value (e.g. 30 minutes), which will be dynamically adjusted based on a predefined size limit of fast storage. The data samples of the data chunks in the SSTables of a specific time partition are strictly bounded by the time range of the partition. When the SSTables are compacted to level 2, several partitions are merged to create larger partitions (i.e. 2 hours). Similarly, this time partition length is also automatically adapted.

Compaction on fast cloud storage. As discussed in §3.1, when the small chunk of data samples batched in memory is full, it will be serialized and inserted into the MemTable. When a MemTable is full, it is switched to an Immutable MemTable and a new MemTable will be created. To mitigate the insertion blocking during the flush of Immutable MemTable, we extend LevelDB with an Immutable MemTable queue to allow multiple flushes at the same time. During the flush of an Immutable MemTable, the key-value pairs are separated into different time partitions (i.e. 30 minutes initially) according to the timestamps contained in the keys.

When the accumulated time partitions in level 0 exceed the threshold (i.e. 2), a compaction from level 0 to level 1 will be triggered to compact the oldest time partition in level 0. There are two considerations of keeping two levels on the fast storage. First, the key-value pairs of the same timeseries/group are merged into larger key-value pairs for a better compression ratio. Second, since the data chunks of the same timeseries/group in a time partition of level 0 may scatter among different SSTables, during the compaction, we gather key-value pairs of the same timeseries/group into the same SSTables for better data locality. If the selected level-0 partition is out-of-order (stale), TimeUnion will merge it with the overlapping partitions in level 1.

Compaction on slow cloud storage. When the overall time span of the time partitions in level 0 exceeds the level-2 partition length (e.g. 2 hours initially), a compaction from level 1 to level 2 will be triggered. The compaction procedure is as follows. First, the oldest partitions whose time ranges are within the time range of the latest time partition in level 2 are selected. Second, key-value pairs are sort-merged to generate new SSTables and the key-value pairs of the same timeseries/group are gathered in the same SSTable. Finally, the new SSTables are uploaded to slow cloud storage and the temporary SSTables in fast cloud storage are deleted. If the selected level-1 partitions are out-of-order (stale) and they are overlapped with existing level-2 partitions, the "patches" will be generated and appended to the corresponding partitions, which will be introduced later in the out-of-order data handling.

With this compaction mechanism, we only need one level for slow cloud storage so as to avoid reading the overlapping SSTables in the slow storage for the majority of fully-ordered data during compaction, which significantly reduces the traffic (i.e. the number of *Get* and *Put* requests) to the slow cloud storage. We show the huge overhead (slow insertion and recent-data queries) of traditional LSM-tree design with baseline *TU-LDB* in the evaluation (§4.3).

Compaction cost analysis. Next, we provide a cost analysis to show the benefit of only keeping one level in slow storage. Since the data traffic to slow storage is the main bottleneck as indicated in §2.1, we measure the cost with the data write size in slow cloud storage, and compare our design with the traditional multi-level LSM-tree. Suppose S_d is the data size, S_b is the size of the topmost level, M is the level size multiplier (the size ratio between adjacent levels), and S_{fast} is the size of fast cloud storage. We can calculate the number of levels L for the traditional LSM-tree to handle S_d as follows:

$$S_d = \sum_{l=0}^{L-1} S_b \cdot M^l \Rightarrow S_d = S_b \frac{1-M^L}{1-M} \Rightarrow L = \frac{log(\frac{S_d \cdot (M-1)}{S_b} + 1)}{logM} \quad (7)$$

Similarly, we can use Equation 7 to calculate the number of levels stored in fast storage L_{fast} with S_{fast}. For the normal compaction in slow storage, since the data are sorted and partitioned by timestamps, we can avoid the traditional level-based compaction which reads and merges all overlapping SSTables on the adjacent levels. Instead, we only need one actual read and write to compact

a SSTable from one level to the next level. The cost (write size) of the traditional multi-level LSM-tree design is as follows:

$$Cost_1 = S_b \cdot \sum_{l=1}^{L-L_{fast}} M^{L_{fast}+l-1} \cdot l \tag{8}$$

In our design, by maintaining only one level in slow storage, we can save unnecessary reads and writes in the deeper levels on slow storage, and the cost is as follows:

$$Cost_2 = S_d - S_{fast} = S_b \cdot \sum_{l=1}^{L-L_{fast}} M^{L_{fast}+l-1} \tag{9}$$

Let $Cost_{saving}$ represent the cost saving. As shown in Equation 10, our one-level design can effectively reduce data traffic by avoiding unnecessary reads/writes from the traditional multi-level. For instance, suppose the topmost level size is 64MB, the size multiplier is 10, the size of fast storage is 1GB, and the total data size is 100GB. Then, L_{fast} is 2.2 and L is 4.2. If we take the floor of L_{fast} and L, we can at least save 64GB of data write to slow storage.

$$Cost_{saving} = S_b \cdot \sum_{l=1}^{L-L_{fast}} M^{L_{fast}+l-1} \cdot (l-1) \tag{10}$$

Out-of-order data handling. In this section, we introduce the mechanism of handling out-of-order data. During the flush of Immutable MemTable, the out-of-order data is detected and inserted into the corresponding time partition (e.g. 8:30-9:00 time partition in L0 in Figure 10).

For a compaction of an out-of-order time partition in level 0, if there are overlapping time partitions in level 1, TimeUnion will sort-merge them together like traditional level-based compaction. If there are multiple data samples of the same timeseries with the same timestamp, TimeUnion will keep the data sample from the newest SSTable. In addition, for the group chunks, we need to handle the inconsistency in two group chunks (i.e. new and missing timeseries) by filling NULL values to those missing timeseries. Because level 0 and level 1 reside in fast cloud storage, the overhead of merging the out-of-order time partitions is relatively small.

For the compaction of the out-of-order time partitions from level 1 to level 2, with the design of time partitioning, we can avoid the huge overhead of merging SSTables in level 2 stored in slow cloud storage. Specifically, we record the timeseries/group ID range of each SSTable in the time partition of level 2. According to the ID ranges of the SSTables, we separate key-value pairs and generate special SSTables - namely patches, and append them to the corresponding time partition in level 2.

To avoid accumulating an excessive number of patches for each SSTable in level 2, we provide an adjustable threshold number (e.g. 3) for users. When the number of patches of an SSTable exceeds the threshold, a merge will be triggered to merge that SSTable and all its patches. As shown in Figure 11, *SSTable 1* and its patches are merged, and two new SSTables are generated (i.e. *SSTable 1-1* and *SSTable 1-2*) with disjoint ID ranges.

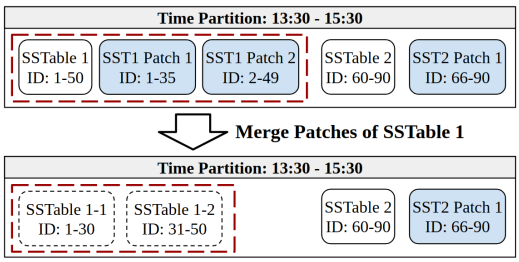


Figure 11: Patches merge in the time partition of level 2.

Algorithm 1: Dynamic Size Control

Input: EBS usage threshold: *ST*; Level-0 partition length: *R1*; Level-2 partition length: *R2*; Partition length lower bound: *LB*; Size of level 0 and 1: *total_size*

```
thres = ST / total_size * R1;
if total_size > ST then
    while R1 / 2 > thres && R1 / 2 > LB do
        R1 /= 2; R2 /= 2;
else if level-1 time span > 0.75 * R2 then
    while R1 * 2 < thres do
        R1 *= 2; R2 *= 2;
```

Dynamic size control. Considering the relatively high cost of fast cloud storage, we need to store as much data as possible on the limited available volume of fast storage. Thus, we propose a mechanism to automatically adapt the fast storage usage to a predefined threshold by adjusting the time partition length, and the basic logic is shown in Algorithm 1. We decrease the partition lengths if the current size of level 0 and level 1 exceeds the threshold. We increase the partition lengths if the overall time span of level 1 is large enough but the total size is smaller than the threshold (e.g. data samples are sparse, or the number of timeseries is small). Besides, for easy time partition alignment during compaction, we multiply/divide the adjustable time partition lengths by 2 during dynamic controlling.

With dynamic size controlling, we may need to handle time partitions with different time partition lengths during a compaction. Thus, careful time partition splitting and aligning are required, and the policy is as follows:

(1) For L0-L1 compaction, since all overlapping SSTables will be read and merged, TimeUnion utilizes the shortest time partition length of all selected partitions as the length of the new partitions, and align them with it.
(2) For L1-L2 compaction, since data in overlapping level-2 partitions will not be merged, TimeUnion needs to keep these partitions and generate patches for them. While for the time ranges not covered by the selected level-2 partitions, they will be split and aligned with the shortest time partition length of the selected level-2 partitions.

An example is provided in Figure 12. As shown in the left part of Figure 12, the L0-L1 compaction generates new partitions with

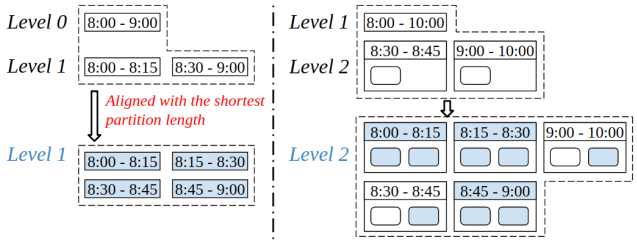


Figure 12: Partition aligning during compaction.

the shortest time partition length (15 minutes of "8:00-8:15" partition). For the L1-L2 compaction in the right, the "8:30-8:45" and "9:00-10:00" level-2 partitions keep unchanged. While for the time ranges that are not covered by these two partitions, they are split and aligned based on the shortest level-2 time partition length (15 minutes of "8:30-8:45" partition).

Logging. To recover the in-memory indexes, data samples, MemTable, and Immutable MemTables from a crash, we disable the original logging in LevelDB and redesign a new logging scheme. For each timeseries and group, we have a sequence ID, which is logged and incremented with each inserted data sample. When a data chunk is flushed to the time-partitioned LSM-tree, the sequence ID is embedded at the beginning of the serialized bytes. Then, when a MemTable is flushed to level 0, for each key-value pair, we will write a special log record with the corresponding sequence ID to indicate that all the log entries of this timeseries/group with sequence IDs before this ID are safe to be removed, and a background worker will purge those stale log records periodically.

Data retention. Since people commonly care more about the recent data, the efficient data retention mechanism for purging too-old data is a critical design decision of timeseries management systems. In TimeUnion, a background worker will periodically check for old time partitions outside the retention time watermark provided by the user. Then, the SSTables contained in those old partitions can be removed efficiently. Besides, we need to purge the memory objects for those old timeseries whose data samples are all removed. Specifically, we record the timestamp of the latest data sample for each timeseries in its memory object, and we will purge those objects that are older than the retention timestamp.

3.4 TimeUnion Operations

In this section, we introduce the operations in TimeUnion to insert and query timeseries data.

Put (Timeseries). Similar to Prometheus tsdb, TimeUnion provides two APIs to insert data samples of individual timeseries. The first API requires the tags of a timeseries, a timestamp, and a metric value of that timeseries, and it returns a 64-bit integer ID for the corresponding timeseries. In the second API, instead of passing timeseries tags, users only pass the timeseries ID returned by the first API for fast insertion, which saves the cost of comparing tags.

Put (Group). Similarly, TimeUnion provides two APIs for group data sample insertion. The first API requires a series of grouping tags, an array of individual timeseries tags, a shared timestamp, and an array of metric values corresponding to each passed group timeseries. For individual timeseries in the group memory object, we manage their tag sets in an appending array with the order when they are inserted. When inserting group timeseries through this API, it will check whether they have been already recorded in the array. If not, it will append a new timeseries to the end of the array, and insert the metric value to the corresponding timeseries object. Finally, it will return a group ID, and a series of timeseries slot indexes which indicate the positions of the inserted timeseries inside the group array. In the second API, users pass a group ID, a series of timeseries slot indexes, a shared timestamp, and an array of metric values for fast insertion.

Get. For a query in TimeUnion, users need to pass a time range and a set of tag selectors, which can be exact or regular-expression match (e.g. *metric="cpu"* or *metric="disk.*"*). TimeUnion first queries the inverted index with the tag selectors to get all the related timeseries/group IDs, then queries the memory objects and the time-partitioned tree using these IDs. Finally, it returns a timeseries set containing all the related timeseries appearing in the given time range. For each timeseries in the timeseries set, users can obtain its iterator to iteratively get its data samples with a merge iterator which connects the individual iterators of all related MemTables and SSTables.

4 EVALUATION

4.1 Experiment Setup

We conduct experiments on an AWS general purpose EC2 instance (type m5.2xlarge in ap-northeast-1 with Ubuntu 20.04 server) with general-purpose EBS SSD volume attached as the fast cloud storage. For slow cloud storage, we create a bucket of AWS S3 under the same data region of EC2. To avoid crashing the EC2 instance during evaluation with high memory pressure, we set up a cgroup [53] with memory restriction and 16GB swap space during the evaluation.

Implementation. We prototype TimeUnion from scratch in C++ with the inverted index derived from Cedar double-array trie [70] and the time-partitioned LSM-tree derived from LevelDB [51]. The source code of TimeUnion is available at [2].

Comparison systems. We first conduct an end-to-end comparison with Cortex [23], a recently popular open-sourced cloud-based timeseries management system, where we utilize HTTP APIs for insertion and querying. Second, to exclude the overhead of network latency via HTTP APIs and the overhead of component communication of Cortex, we directly benchmark the storage engines. Since Cortex's storage engine is based on Prometheus tsdb (the storage engine of Prometheus), we extend Prometheus tsdb with cloud storage support (marked as *"tsdb"* in the following experiments). Besides, we implement two extra baselines as follows: (a) tsdb with LevelDB as data sample storage (*tsdb-LDB*) in which the samples of the flushed time partitions are stored in LevelDB whose SSTables are stored in S3 (b) TimeUnion with LevelDB as data sample storage (*TU-LDB*), which stores the first two levels on EBS and the other levels on S3. These two baselines leverage the same LevelDB key format as TimeUnion (§3.3).

Configurations. For all evaluated systems, we equip a 1GB in-memory LRU cache to cache the data segments fetched from S3 during querying. For the data insertion of *tsdb*-based and *TimeUnion*-based systems in §4.3, we use the fast-path insertion API as introduced in §3.4. For the EBS usage of TimeUnion, since Prometheus tsdb stores the latest 2-hour data in RAM, to make the evaluation fair, when comparing with tsdb-based systems, we turn off dynamic size controlling and fix the level-2 partition length as 2 hours to make TimeUnion only store the latest 2-hour data on EBS, while the older data are stored on S3.

4.2 End-To-End Evaluation

In this experiment, we compare the end-to-end performance of TimeUnion and Cortex [23], an open-sourced timeseries management system with cloud storage support. We utilize HTTP APIs to

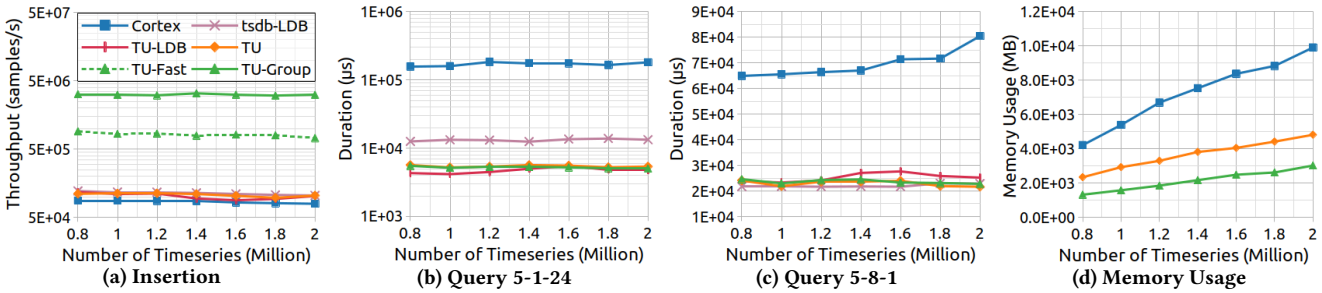


Figure 13: End-to-end evaluation.

insert and query data in both systems, and we utilize TSBS [57], a widely-used benchmark suite for timeseries systems, to generate timeseries tags and query patterns. The insertion API is Prometheus remote write API [9] without streaming support. Thus, we transmit a large batch (i.e. 10,000 samples) in each HTTP request to increase throughput for all systems. For the insertion timeseries, we utilize the DevOps timeseries generated by TSBS, where each host contains 101 timeseries to simulate different system/application metrics (e.g. cpu usage, number of inserted Postgres tuples, number of Redis expired keys, etc.). The query patterns in TSBS can be found in Table 2.

Figure 13 shows the experiment results with different numbers of timeseries (each timeseries contains 24 hours of data and the interval between two data samples is 60 seconds). Since HTTP APIs are slow, the experiments are conducted under a medium volume of dataset (maximally 2 million timeseries with a total of 1.44 billion data samples). TU represents TimeUnion with slow-path insertion (each sample insertion with timeseries tags); TU-fast represents TimeUnion with fast-path insertion (the first insertion is with timeseries tags, while the following insertions are with corresponding timeseries IDs); and TU-Group represents TimeUnion with group-style insertion with fast-path (the timeseries from the same host form a group). Besides, Cortex does not support fast-path insertion as discussed in Section 3.4.

As shown in Figure 13a, TU's insertion throughput is 26.6% higher than that of Cortex because of the gRPC [28] communication of internal components of Cortex, which accumulates with HTTP insertion requests. TU-fast's throughput is 6.6× higher than that of TU, because it saves the serialization cost of tags for each data sample. TU-Group is 2.9× faster than TU-fast because it further deduplicates timestamps for a set of timeseries and reduces the number of HTTP requests through grouping.

Table 2: TSBS query patterns.

TSBS Query	Description
1-1-1	Aggregate (MAX) on 1 metric for 1 host, every 5 mins for 1 hour.
1-1-24	Aggregate (MAX) on 1 metric for 1 host, every 5 mins for 24 hours.
1-8-1	Aggregate (MAX) on 1 metric for 8 hosts, every 5 mins for 1 hour.
5-1-1	Aggregate (MAX) on 5 metrics for 1 host, every 5 mins for 1 hour.
5-1-24	Aggregate (MAX) on 5 metrics for 1 host, every 5 mins for 24 hours.
5-8-1	Aggregate (MAX) on 5 metrics for 8 hosts, every 5 mins for 1 hour.
lastpoint	The last reading of 1 CPU metric of one host.

Figure 13b shows the query latencies of query pattern 5-1-24. Cortex and tsdb-LDB are 30.4× and 1.4× slower than TU because they need to read the indexes in the partitions stored in S3, and the index reading of Cortex is inefficient where it needs to load the whole index into memory in advance. For pattern 5-8-1 in Figure 13c, Cortex is 2.0× slower than TU, and the latencies of other systems are close to each other. In Figure 13d, the memory usage of Cortex is 96.8% and 2.4× higher compared with TU and TU-Group.

4.3 Storage Engine Evaluation

To exclude the influence of network latency and internal system components, in this section, we directly benchmark the storage engines with a large data scale. We configure CGroup with a 16GB memory restriction. For all systems, we insert 24 hours of data with 30 seconds (except for big timeseries evaluation) as the sample interval for each timeseries with fast-path insertion. First, we evaluate them with DevOps timeseries generated by TSBS as mentioned in §4.2. For each comparison system, we gradually add more timeseries in each test round until it is killed by OOM or the performance has a significant degradation. Second, we evaluate TSBS with big DevOps timeseries (denser data samples and larger time span). Third, we monitor and compare the memory usage of different systems. Fourth, we evaluate TSBS with all data only stored on EBS. Finally, we evaluate different configurations and properties of TimeUnion, including (a) different EBS usage constraints; (b) different volumes of out-of-order data; (c) the dynamic size controlling algorithm.

DevOps timeseries. Figure 14a shows the insertion throughput, where the performances of tsdb and tsdb-LDB significantly degrade when there are 2.6 and 2 million timeseries, respectively, while TimeUnion can manage up to 12 million timeseries. On average, the throughput of TimeUnion is 24.8% and 13.2% higher than that of tsdb and tsdb-LDB, respectively. This is because the background cleaning of tsdb after flushing incurs contention with foreground insertion. Furthermore, the insertion throughput of TU-Group is 2.4× higher than that of TU because of the coarser-grained index lookup during insertion. TU-LDB dose not perform well because of the inefficient compaction which reads and merges a large number of overlapping SSTables on S3. Besides, tsdb-LDB performs better than TU-LDB because tsdb-LDB flushes in-memory partitions to LevelDB in the background without affecting the foreground

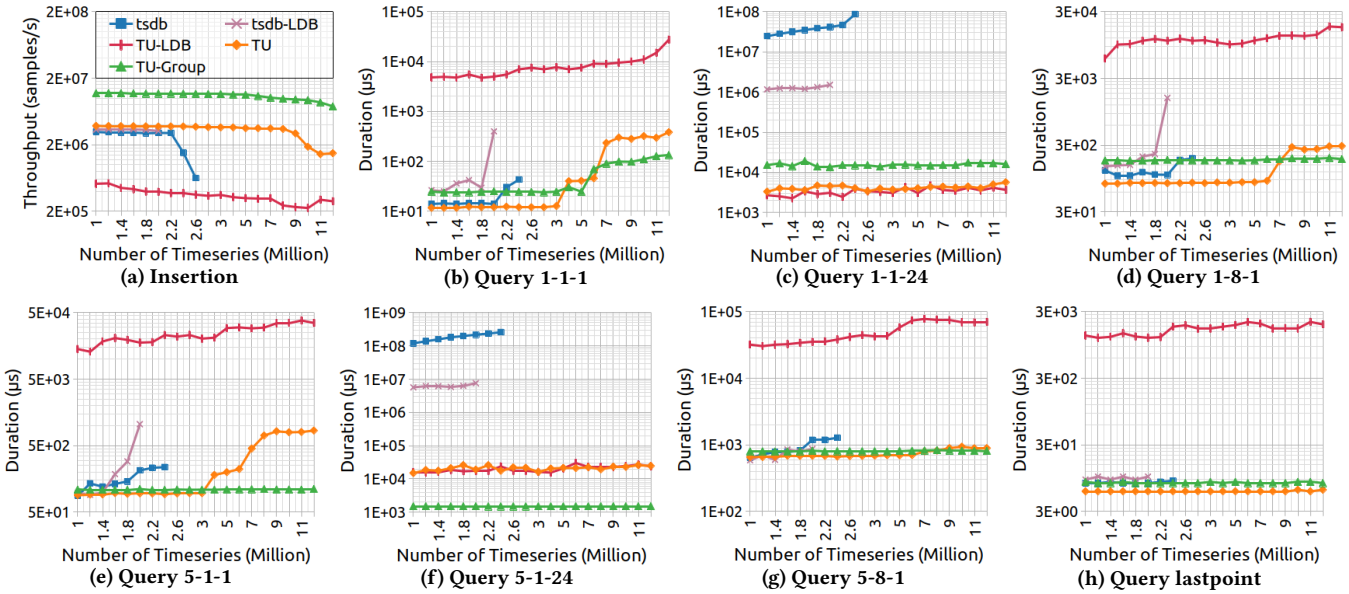


Figure 14: Evaluation with DevOps timeseries (30s sample interval, 24 hours time span).

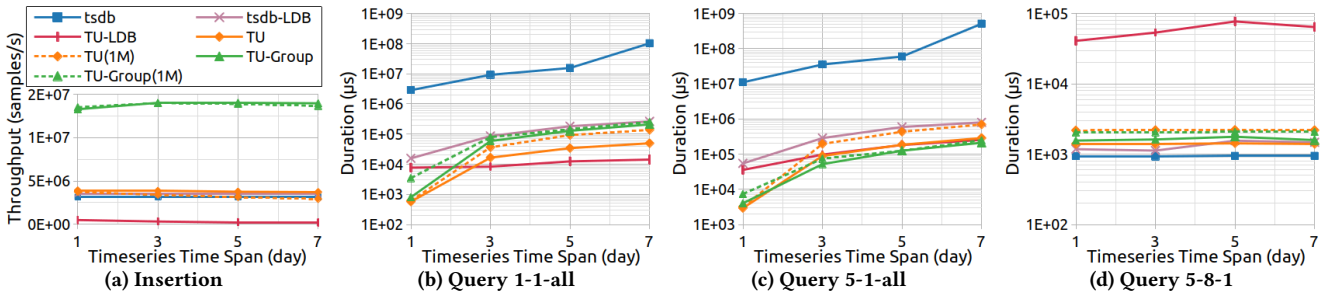


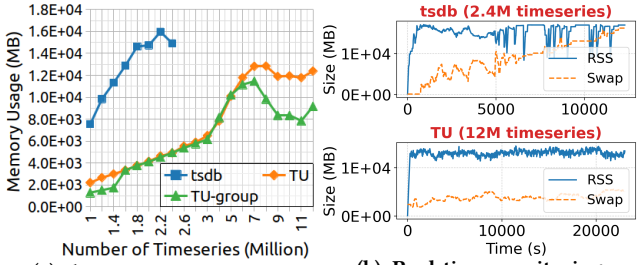
Figure 15: Evaluation with big DevOps timeseries (10s sample interval, 1-7 days time span).

insertion. Thus, tsdb-LDB can accumulate an excessive amount of pending data in memory because its compaction is as slow as TU-LDB.

When querying recent data (1-1-1, 1-8-1, 5-1-1, 5-8-1, and last-point), TU performs generally better than tsdb and tsdb-LDB (e.g. 29.7% and 41.2% lower latency on average). Compared with TU-Group, TU performs better even when the queried timeseries are located in the same group (e.g. 5-1-1 and 5-8-1). This is because the recent data is stored in fast EBS, and latency mainly relates to the queried data size as shown in Equations 3 and 5. Because the queried data size of TU is smaller than that of TU-Group (S_g counteracts the benefit of $G < L$), TU will perform better ($Cost_{q1} < Cost_{q2}$). However, as the number of timeseries increases, the latency of TU-Group is more stable because it is more memory-efficient than TU ($Cost_{s2} < Cost_{s1}$ in Equations 1 and 2). TU-LDB performs the worst (178.8× higher latency than TU) because, without time partitioning, the recent data are scattered in the SSTables on top levels which have not yet been further compacted.

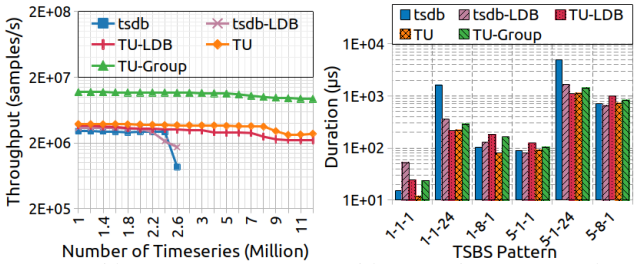
When querying long-range data (1-1-24 and 5-1-24), TU performs orders of magnitude better than tsdb and tsdb-LDB, because tsdb needs to fetch those large indexes in old time-partitions from S3 and incurs frequent cache eviction and memory swapping. Compared with TU, in query 1-1-24, TU-Group has 2.8× higher latency. The reason is $L = 1$ in Equation 4 and $G = 1$ in Equation 6, and the ceiling function in Equation 6 generates a slightly larger value than that of Equation 4, which results in $Cost_{q1} < Cost_{q2}$; while for query 5-1-24, TU has 13.1× higher latency because $L = 5$ in Equation 4 incurs a larger $Cost_{q1}$. The latency of TU-LDB is similar to that of TU because compaction in deeper levels of TU-LDB gathers the data from the same timeseries with our key format, which increases data locality.

Big DevOps timeseries. In Figure 15, we evaluate TSBS with big timeseries (10s sample interval and 1-7 days of time span). By default, the number of timeseries is 100K for all systems. This is because, for tsdb and tsdb-LDB, the throughput of data flushing is much lower than that of insertion, thus a huge amount of data samples can be temporarily accumulated in memory, which triggers OOM (Out of Memory) killing, and 100K is the limit for both tsdb and tsdb-LDB. For TimeUnion, because of its memory efficiency and high flushing throughput, we provide another set of experiments with more timeseries (1M). For insertion, the throughput of TU is 21.0%, 8.8%, and 12.2× higher compared with tsdb, tsdb-LDB, and TU-LDB, respectively, and TU-Group performs 2.6× better than TU. For query, we add two new patterns (1-1-all, 5-1-all), which



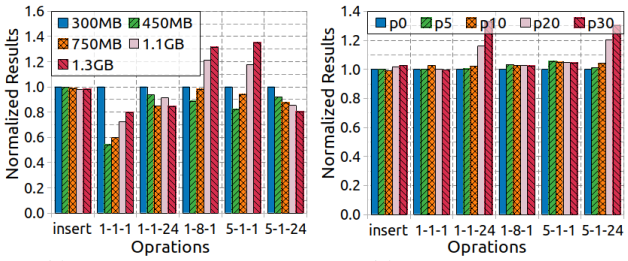
(a) Average memory usage (b) Real-time monitoring

Figure 16: Memory usage monitoring.



(a) Insertion (b) Query (1M timeseries)

Figure 17: Evaluation with only EBS.



(a) Different EBS usage (b) Out-of-order data

Figure 18: Evaluation with different constraints.

query the whole time span in the corresponding test run. For 1-1-all, the latencies of tsdb, tsdb-LDB, and TU-Group are three orders of magnitude, 9.8×, and 2.2× higher compared with TU. 5-1-all has a similar trend except that TU-Group performs 15.7% better than TU because queried timeseries are from the same group. For 5-8-1, the latencies of tsdb and tsdb-LDB are 33.1% and 5.1% lower compared with TU. This is because tsdb only needs to read the in-memory block and tsdb's hash-based indexing is more efficient than TimeUnion's mmaped-trie-based indexing with ample memory (we carry out the experiment after all pending samples are flushed). For TU-LDB, it performs better (49%) than TU for the long-range query because, without time partitioning, the data of the same timeseries are gathered in fewer SSTables in the deeper levels after many compactions, while it performs the worst for recent data queries because of the poor data locality of SSTables in top levels.

Memory usage monitoring. Figure 16a shows the average memory usage of the experiments in Figure 14. We omit tsdb-LDB and TU-LDB because they are similar to tsdb and TU, respectively. The memory usage of tsdb reaches the CGroup 16GB memory restriction when the number of timeseries is 2.2 million. For TimeUnion, when the number of timeseries becomes larger than 7 million, the OS positively swaps out those not frequently used memory-mapped pages to mitigate memory pressure. On average, the memory usage of tsdb is 2.6× and 3.6× higher than that of TU and TU-Group, respectively.

Figure 16b shows the real-time memory usage (*resident set size*) of tsdb and TU in the test rounds of 2.4 million and 12 million timeseries in the experiments in Figure 14. For tsdb, the data insertion finishes at 4550 seconds (for 2.4M timeseries), and the memory usage is skyrocketing until it reaches the 16GB memory limit and is forced to swap. Then, the system waits for all compactions to be finished, and finally, the system executes the TSBS queries, which incur intensive memory fluctuation. For TU, data insertion finishes at 12530 seconds (for 12M timeseries), and the memory usage remains stable at around 12GB. The not frequently used memory-mapped pages are swapped out in advance to mitigate the memory pressure.

EBS-only evaluation. In Figure 17, we repeat the experiments in Figure 14 with all data stored on EBS. For insertion, the results are similar to Figure 14a, the throughput of TU is 28.8% and 34.0% higher compared with tsdb and tsdb-LDB, respectively, and TU-Group performs 2.1× better than TU. Besides, TU-LDB is only 19.4% worse than TU because the compaction on EBS is faster than on S3. For query, because of the space limitation, we show the results of the test round of 1M timeseries and we observe similar trends as Figure 14. For query patterns that cover recent data, tsdb and tsdb-LDB exhibit similar performance to TU because they only need to access the in-memory block. For larger queries (1-1-24, 5-1-24), tsdb and tsdb-LDB are 4.9× and 55.6% slower than TU because of inefficient index reading across blocks. Besides, TU performs better than TU-Group because, with EBS only, we have $Cost_{q1} < Cost_{q2}$ based on Equations 3 and 5.

Different EBS limits. Figure 18a shows the normalized results (throughput for insert, latency for query) of TimeUnion under different EBS limits (1M timeseries, 10s sample interval). Based on our dynamic size control algorithm, with a larger EBS limit, the time partition length can be larger so that more SSTables can be placed at the first two levels. For insertion, the performance is stable because of the efficient compaction. For short-range queries (e.g. 1-1-1), the latency is high when EBS cannot cover all the data in the last hour. As the EBS limit increases, the latency first drops, and then gradually increases because more SSTables need to be checked as the partition length increases. For long-range queries (e.g. 5-1-24), the latency gradually decreases as the EBS limit increases because more data can be stored on EBS.

Different amounts of out-of-order data. Since Prometheus tsdb does not support out-of-order data, we only evaluate TimeUnion with different amounts of out-of-order data in Figure 18b (1M timeseries with 10s sample interval). After the normal data insertion, we randomly insert different portions of out-of-order data of randomly picked timeseries (p5 represents that the volume of out-of-order data is 5% of the normal data). For insertion, the influence of out-of-order data is subtle because of efficient compaction. For short-range queries, the latency increase is small (3%) because data is on fast EBS. On the other hand, the latency of long-range queries gradually increases with more out-of-order data because more SSTables on S3 need to be read.

Dynamic size control. In Figure 19, we evaluate our dynamic size control algorithm with 1M timeseries and 512MB EBS limit. We

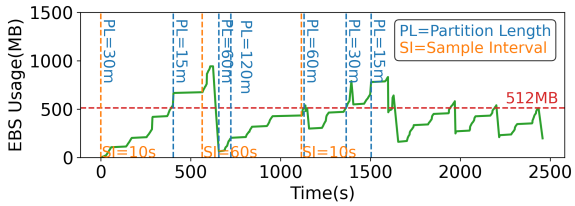


Figure 19: Dynamic size control with 512MB EBS limit.

start with a 30-minute time partition length and insert data with a dense sample interval of 10 seconds. As the EBS usage exceeds 512MB, the partition length decreases to 15 minutes to reduce the size of the first two levels. Next, we increase the sample interval to 60 seconds. Since the data size in a time partition decreases, the partition length gradually increases to 120 minutes. Then, we pressure the system again by inserting data with a 10-second sample interval. It can be observed that the partition length gradually decreases to 15 minutes, and the EBS usage remains stably under 512MB.

Index/Data size comparison. Table 3 shows the index and data sizes in the test round of 2 million timeseries in Figure 14. The index size of tsdb is 21.4% and 48.6% higher than that of TU and TU-Group, respectively, because tsdb generates a separate index for each time partition, which incurs duplicate data. We calculate timeseries data size by summing all the data chunks files for tsdb and all the SSTables for TimeUnion, respectively. The data size of tsdb is 1.35× higher than that of TU because data blocks in SSTables are further compressed by Snappy [1]. Furthermore, the data size of TU-Group is 71.9% lower than that of TU because group data chunk deduplicates timestamps.

5 RELATED WORK

The design of TimeUnion refers to extensive past works of timeseries management systems, LSM-tree-based key-value stores, and systems deployed on hybrid storage.

Timeseries management systems. In addition to Prometheus, InfluxDB, and Cortex as discussed in §2, there are previous works on timeseries management systems. BtrDB [7] targets at the telemetry timeseries with nanosecond level precision, which proposes a novel time partition tree for each timeseries as the index. However, this design can only handle a limited number of timeseries because the indexing structure is memory-consuming. Gorilla [46] presents an in-memory timeseries database for performance monitoring in Facebook; the delta-of-delta timestamps and XOR'd floating-point values proposed in this paper are widely used in the existing timeseries management systems. Timon [13] aims at handling the out-of-order timeseries data with the TS-LSM-Tree and SEDA programming model [62]. It is different from our time-partitioned LSM-tree because we have different compaction and out-of-order data handling mechanisms for different cloud storage tiers. Besides, we have dynamic level size adjustment for cost-efficiency. IBM Db2 Event Store [27] leverages hybrid MPP (massively parallel processing) share nothing/share disk cluster architecture, which manages data in different zones with Parquet file format [22] for fast ingestion. ByteSeries [54] aims at high-dimensional and dynamic timeseries by reducing the metadata memory footprints. Peregreen [60] focuses on handling a large volume of historical timeseries data totally on S3; however, timeseries are simply indexed by an integer ID without utilizing tags. Thus, it is inconvenient to differentiate timeseries and support complex queries.

Table 3: Index and data size of 2M timseries (GB).

	tsdb	TU	TU-Group
Index	3.27	2.70	2.20
Data	20.28	8.61	2.42

LSM-tree-based key-value stores. PebblesDB [50] proposes the fragmented LSM-tree, which partitions the key range of each level into guards (partitions), and the boundary of each guard is determined by the inserted keys with a certain probability. During a compaction, PebblesDB does not need to read the overlapping SSTables in the next level; instead, it simply splits the selected SSTable according to the guard boundaries and appends the split parts to the corresponding guards. Dostoevsky [19] targets at the trade-off between tiered and leveled compaction [33], which proposes lazy leveling that remove the sorted runs merging from all levels but the largest. Mutant [69] proposes a storage layer for LSM-tree data stores to strike a cost-performance balance with different types of cloud storage tiers; with Mutant's cost model, SSTables are migrated among different storage tiers correspondingly.

Systems with hybrid cloud storage. Alluxio [35] builds an indirection layer above the existing cloud storage systems to support the storage of different data systems such as Spark, Flink, and Presto [24–26]. Pangea [73] proposes a monolithic general-purpose storage system that can handle all different types of data. Pocket [32] proposes a distributed data store for ephemeral objects of the serverless analysis with sub-second response time; it strikes cost efficiency among different volume types (HDD, SSD, NVMe) of EBS.

6 CONCLUSION

In this paper, we present TimeUnion, an efficient timeseries management system with a unified data model tailored for hybrid cloud storage services. We explore the characteristics of cloud storage, propose a unified tiemseries data model, mitigate the imbalanced resource usage of the timeseries management system, and present our time-partitioned LSM-tree with tailored compaction mechanism, out-of-order data handling, and dynamic level size adjustment. Compared to the storage engine of Cortex, TimeUnion can handle at least 5× more timeseries, and achieve at least 24.8% higher insertion throughput and 49.8% lower query latency.

ACKNOWLEDGMENTS

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15224918), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151).

REFERENCES

[1] 2021. *Snappy | A fast compressor/decompressor.* https://github.com/google/snappy.
[2] 2021. *TimeUnion Source Code.* https://github.com/naivewong/timeunion.
[3] Inc. or its affiliates 2020, Amazon Web Services. 2021. *Amazon EC2.* https://aws.amazon.com/ec2/.
[4] Inc. or its affiliates 2020, Amazon Web Services. 2021. *Amazon Elastic Block Store.* https://aws.amazon.com/ebs/.
[5] Inc. or its affiliates 2020, Amazon Web Services. 2021. *Amazon Lambda.* https://aws.amazon.com/lambda/.
[6] Inc. or its affiliates 2020, Amazon Web Services. 2021. *Amazon S3.* https://aws.amazon.com/s3/.
[7] Michael P Andersen and David E. Culler. 2016. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 39–52. https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen
[8] J. Aoe, K. Morimoto, and Takashi Sato. 1992. An efficient implementation of trie structures. *Software: Practice and Experience* 22 (1992).
[9] Prometheus Authors. 2020. *Prometheus remote write.* https://prometheus.io/docs/prometheus/latest/configuration/configuration/#remote_write.
[10] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 363–375. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau
[11] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau
[12] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3064176.3064193
[13] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. 2020. 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 739–753. https://doi.org/10.1145/3318464.3386136
[14] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1007–1019. https://www.usenix.org/conference/atc18/presentation/chan
[15] Google Cloud. 2021. *Cloud Persistent Disk.* https://cloud.google.com/persistent-disk.
[16] Google Cloud. 2021. *Google Cloud Storage.* https://cloud.google.com/storage/.
[17] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway
[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054
[19] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927
[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 449–466. https://doi.org/10.1145/3299869.3319903
[21] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 1414–1425. https://doi.org/10.14778/1920841.1921015
[22] Apache Software Foundation. 2020. *Apache Parquet.* https://parquet.apache.org/.
[23] Cloud Native Computing Foundation. 2020. *Horizontally scalable, highly available, multi-tenant, long term Prometheus.* https://cortexmetrics.io/.
[24] The Apache Software Foundation. 2021. *Apache Flink - Stateful Computations over Data Streams.* https://flink.apache.org/.
[25] The Apache Software Foundation. 2021. *Apache Spark - Lightning-fast unified analytics engine.* https://spark.apache.org/.
[26] The Presto Foundation. 2021. *Presto - Distributed SQL Query Engine for Big Data.* https://prestodb.io/.
[27] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, Alexander Cheung, Gidon Gershinsky, Hamid Pirahesh, David Kalmuk, Yuanyuan Tian, Matthew Spilchen, Lan Pham, Darren Pepper, and Gal Lushi. 2020. Db2 Event Store: A Purpose-Built IoT Database Engine. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3299–3312. https://doi.org/10.14778/3415478.3415552
[28] 2021 gRPC Authors. 2020. *gPRC, A high performance, open source universal RPC framework.* https://grpc.io/.
[29] influxdata. 2020. *InfluxDB 1.7 Documentation.* https://docs.influxdata.com/influxdb/.
[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet
[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan
[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic
[33] Bradley C Kuszmaul. 2014. A comparison of fractal trees to log-structured merge (LSM) trees. *Tokutek White Paper* (2014).
[34] V. Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. *Proceedings - International Conference on Data Engineering*, 38–49. https://doi.org/10.1109/ICDE.2013.6544812
[35] Haoyuan Li. 2018. Alluxio: A Virtual Distributed File System.
[36] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 739–752. https://www.usenix.org/conference/atc19/presentation/li-yongkun
[37] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558
[38] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 429–444.
[39] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (March 2017), 28 pages. https://doi.org/10.1145/3033273
[40] F. Mei, Q. Cao, H. Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. *Proceedings of the ACM Symposium on Cloud Computing* (2018).
[41] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2014. Optimizing Key-Value Stores for Hybrid Storage Architectures. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '14)*. IBM Corp., USA, 355–358.
[42] Microsoft. 2021. *Azure Blob Storage | Microsoft.* https://azure.microsoft.com/en-us/services/storage/blobs/.
[43] Microsoft. 2021. *Azure Disk Storage | Microsoft.* https://azure.microsoft.com/en-us/services/storage/disks/.
[44] OKLog. 2020. *Universally Unique Lexicographically Sortable Identifier.* https://github.com/oklog/ulid.
[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (01 June 1996), 351–385. https://doi.org/10.1007/s002360050048
[46] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078
[47] Bartlomiej Plotka. 2020. *Thanos - Highly available Prometheus setup with long term storage capabilities. A CNCF Incubating project.* https://github.com/thanos-io/thanos.
[48] Prometheus. 2020. *Node exporter - Exporter for machine metrics.* https://github.com/prometheus/node_exporter.
[49] Prometheus. 2020. *Prometheus - From metrics to insight, power your metrics and alerting with a leading open-source monitoring solution.* https://prometheus.io/.

[50] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[51] Jeff Dean Sanjay Ghemawat. 2021. *LevelDB*. https://github.com/google/leveldb.

[52] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[53] Michael Kerrisk Serge Hallyn. 2021. *cgroups — Linux manual page*. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[54] Xuanhua Shi, Zezhao Feng, Kaixi Li, Yongluan Zhou, Hai Jin, Yan Jiang, Bingsheng He, Zhijun Ling, and Xin Li. 2020. ByteSeries: An in-Memory Time Series Database for Large-Scale Monitoring Systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 60–73. https://doi.org/10.1145/3419111.3421289

[55] Facebook Open Source. 2021. *RocksDB - A persistent key-value store for fast storage environments*. https://rocksdb.org/.

[56] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang. 2017. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 68–79. https://doi.org/10.1109/ICDCS.2017.70

[57] Timescale. 2020. *Time Series Benchmark Suite, a tool for comparing and evaluating databases for time series data*. https://github.com/timescale/tsbs.

[58] Inc. Timescale. 2020. *Time-series data simplified | Timescale*. https://www.timescale.com/.

[59] Suryandaru Triandana. 2020. *LevelDB key/value database in Go*. https://github.com/syndtr/goleveldb.

[60] Alexander Visheratin, Alexey Struckov, Semen Yufa, Alexey Muratov, Denis Nasonov, Nikolay Butakov, Yury Kuznetsov, and Michael May. 2020. Peregreen – modular database for efficient storage of historical time series in cloud environments. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 589–601. https://www.usenix.org/conference/atc20/presentation/visheratin

[61] Zhiqi Wang, Jin Xue, and Zili Shao. 2021. Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1080–1092. https://doi.org/10.14778/3447689.3447710

[62] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 230–243. https://doi.org/10.1145/502059.502057

[63] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[64] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[65] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[66] T. Yao, Jiguang Wan, P. Huang, Xubin He, Q. Gui, F. Wu, and C. Xie. 2017. A Light-weight Compaction Tree to Reduce I / O Amplification toward Efficient Key-Value Stores.

[67] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, USA) *(FAST'19)*. USENIX Association, Boston, MA, 159–171. https://www.usenix.org/conference/fast19/presentation/yao

[68] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. 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)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[69] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 162–173. https://doi.org/10.1145/3267809.3267846

[70] Naoki Yoshinaga. 2021. *cedar - C++ implementation of efficiently-updatable double-array trie*. http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/.

[71] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2017. Building an Efficient Put-Intensive Key-Value Store with Skip-Tree. *IEEE Trans. Parallel Distrib. Syst.* 28, 4 (April 2017), 961–973. https://doi.org/10.1109/TPDS.2016.2609912

[72] Q. Zhang, Y. Li, P. P. C. Lee, Y. Xu, Q. Cui, and L. Tang. 2020. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 313–324. https://doi.org/10.1109/ICDE48307.2020.00034

[73] Jia Zou, Arun Iyengar, and Chris Jermaine. 2019. Pangea: monolithic distributed storage for data analytics. *Proceedings of the VLDB Endowment* 12 (02 2019), 681–694. https://doi.org/10.14778/3311880.3311885