



PromTCS: An Efficient Two-Dimensional Metadata Management and Tailored Hybrid Storage Architecture for Performance Monitoring Time Series

Tao Kong[1], Hui Li[1*], Yuxuan Zhao[1], Quanqing Xu[2], Liping Li[1], Qian Wang[1]

[1]*Xidian University* [2]*OceanBase, Ant Group*

Abstract

Time series databases (TSDBs) are crucial for performance monitoring in Internet applications and infrastructure. As the scale of time series grows, the storage overhead of metadata (labels) and data points (samples) increases significantly. In spite of prior efforts in hybrid storage to reduce storage costs, they still require large-capacity, expensive, high-bandwidth storage devices to deliver superior performance.

We propose PromTCS, which introduces two novel techniques: (1) Two-dimensional metadata management. We propose a two-dimensional metadata model and a suite of indexes to reduce metadata overhead while ensuring high performance. (2) Tailored hybrid storage architecture. We place hot data on DRAM and NVMe SSDs and cold data on HDDs, and present three optimizations: (i) a compaction-free time series storage engine for NVMe SSDs, TreeSeries, to reduce write amplification (WA) and optimize performance; (ii) hardware-aware write strategies, early migration and lazy flushing, to mitigate the adverse impact of bandwidth cross-tier bandwidth gaps; and (iii) a metadata-driven parallel write-ahead logging (WAL) on SSD, to reduce thread contention during writes and shorten crash recovery time. We implement a PromTCS prototype and evaluate it on synthetic and real-world datasets using both benchmark and production workloads. Compared with advanced TSDBs on pure NVMe SSDs, PromTCS achieves comparable performance at much lower deployment cost. Under the same hybrid storage, it delivers 3.2–5.9× higher insertion throughput and 38.1%–82.1% lower query latency than the advanced TSDBs.

1 Introduction

Performance-monitoring time series databases (TSDBs), such as Prometheus [10], InfluxDB [6], and Cortex [3], continuously collect and query metrics (*e.g.,* CPU utilization and memory usage) from Internet services and infrastructure. They model each time series as a metric from a given source together with its values observed over time. Each time series consists of metadata (labels for metric and data sources) and samples (timestamp–value pairs). However, the rapid growth of monitored entities (hosts, containers, microservices) and exported metrics has led to an explosion in the number of time series and label cardinality, pushing TSDBs to a breaking point in both scalability and cost efficiency [20, 36, 43].

Motivated by our study in Sec. 2, this paper focuses on reducing system cost while delivering high performance through both *metadata management* and *cost-effective storage deployment*. Our study shows that, at scale, metadata and its indexes can dominate the memory footprint of a TSDB [35], while existing metadata models do not sufficiently reuse redundant labels across time series and their index designs struggle to simultaneously achieve high concurrency and storage efficiency [11, 37, 39]. Meanwhile, existing TSDBs typically rely on high-bandwidth storage devices (*i.e.,* NVMe SSDs) and LSM-tree-based storage engines to achieve high performance [1, 6, 34], which is difficult to maintain in cost-constrained deployments.

Thus, we propose **PromTCS**, a TSDB that integrates two key techniques, as follows:

(1) Two-dimensional metadata management. We observe that performance-monitoring metadata naturally contains two orthogonal dimensions: *metric label*s describing the performance metrics and *source label*s describing the monitored entity. PromTCS proposes a new metadata model that explicitly separates these two classes of labels and maps them to a *MetricID* and a *SourceID*, respectively, uniquely identifying each time series by the pair (*MetricID*, *SourceID*). Building on this model, we design a suite of concurrency-friendly and storage-efficient metadata indexes. We further provide a theoretical analysis showing that PromTCS achieves lower metadata overhead than prior metadata models [11, 37, 39].

(2) Tailored hybrid storage architecture. PromTCS adopts a DRAM–NVMe SSD–HDD hierarchy, placing new/hot data and logs on DRAM and NVMe SSDs, and storing old/cold data on HDDs. PromTCS introduces three key optimizations. (i) We design *TreeSeries*, a compaction-free time series storage engine tailored for NVMe SSDs. TreeSeries uses slab-based data management and a concurrent indexing structure,

*Corresponding author (hli@xidian.edu.cn).

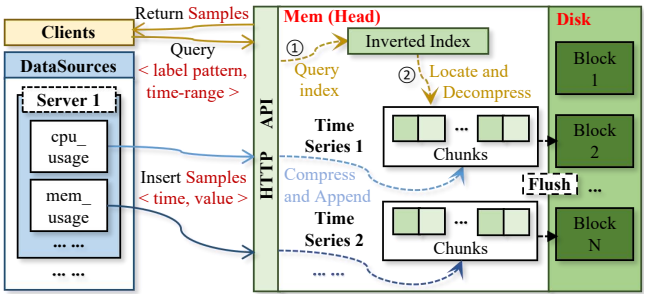


Figure 1: The architecture of TSDB.

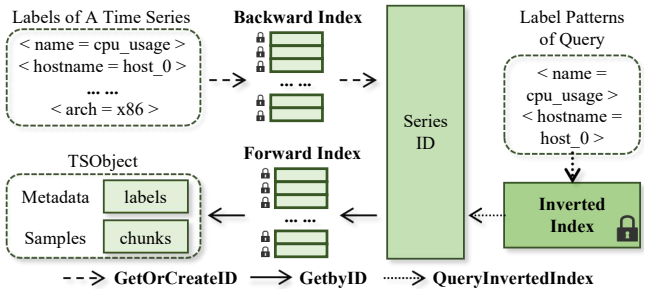


Figure 2: Metadata indexing workflow in existing TSDBs.

and maintains data order via a metadata-driven relaxed ordering scheme, avoiding expensive compactions that can incur severe write amplification (WA) and trigger write stalls [41]. (ii) We optimize cross-tier data migration with *device-aware write strategies*, *early migration*, and *lazy flushing* to mitigate tier bandwidth gaps: early migration avoids hot-tier saturation and ingestion stalls, while lazy flushing retains migrated data on faster media to reduce query latency.
(iii) We propose a *metadata-driven parallel write-ahead log (WAL)* that partitions log records by MetricID into independent log groups, enabling append operations without being blocked by a global WAL lock and supporting parallel log replay for faster crash recovery.

We implemented a PromTCS prototype and evaluated it on synthetic and real-world datasets under benchmark and production workloads. Compared with advanced TSDBs on NVMe SSDs, PromTCS achieves comparable performance on hybrid storage. Under the same hybrid storage, it delivers 3.2–5.9× higher insertion throughput and 38.1%–82.1% lower query latency.

2 Background and Motivation

This section presents the background of performance-monitoring TSDBs and motivates the design of PromTCS. Key terms in the paper are summarized in Tab. 1.

2.1 Performance-Monitoring TSDBs

Time series model. As shown in Fig. 1, performance-monitoring TSDBs model a time series (*e.g., Time Series 1*) as measurements of a metric (*e.g., cpu_usage*) from a particular source (*e.g., Server 1*) over time. Accordingly, each time series consists of two parts: (i) metadata, *i.e.,* a set of labels that describe the metric and the source, and (ii) data points (*i.e.,* samples), where each sample is a ⟨timestamp, value⟩ pair (a 64-bit timestamp and a double-precision value).

Table 1: List of terms utilized in this paper.

Terms	Descriptions
Chunk	A Compressed storage unit for samples
TSObject	A Time series object that contains chunks and metadata
Backward Index	An index that maps labels to unique identifiers (IDs)
Forward Index	An index that maps IDs to TSObjects
Inverted Index	An index that maps labels to matching posting lists (ID lists)

Overall architecture. Fig. 1 illustrates the architecture of a typical performance-monitoring TSDB. Collectors scrape metrics from data sources and write them into the storage engine, while clients query samples using a label pattern (*i.e.,* labels or regular expressions) and a time range. The storage engine (*e.g.,* Prometheus tsdb [11]) consists of an in-memory *Head* and persistent *Blocks*. Newly ingested data is appended to the WAL for crash recovery while the Head, including TSObjects and metadata indexes (*e.g.,* inverted indexes), is updated in parallel. As data accumulates, time-bounded segments in the Head are compacted into immutable on-disk blocks, and small blocks are continuously merged to reduce index overhead and file count.
Metadata index. The TSDB metadata index consists of three components: the *backward index*, *forward index*, and *inverted index*. The backward index maps labels to IDs, the forward index maps IDs to TSObjects, and the inverted index maps label matchers (*e.g., job="api", method="GET|POST"*) to postings lists of time series IDs. In TSDBs (*e.g.,* Prometheus tsdb [11]), the backward and forward indexes are implemented as bucket-locked hash tables, while the inverted index is a two-level hash table protected by a global lock.
Read/Write path. Fig. 1 and Fig. 2 illustrate the read and write paths in a TSDB. ❶ *Write path.* TSDB first appends data to the WAL. It then obtains the SeriesID from the backward index, locates the corresponding TSObject via the forward index, compresses incoming samples (*e.g.,* Gorilla [31]), and appends them to the TSObject. ❷ *Read path.* TSDB first selects the Head and on-disk blocks overlapping the query time range. For each, it intersects posting lists retrieved from the inverted index to obtain candidate SeriesIDs, locates the corresponding TSObjects via the forward index, and fetches matching samples while decompressing them on the fly. Finally, it merges results across blocks.

2.2 Motivation

Delivering high performance at low cost requires addressing challenges in metadata management and storage architecture.

2.2.1 Metadata Overhead under High Cardinality

As monitored infrastructure scales, the proliferation of data sources drives high cardinality (*i.e.,* a large number of distinct

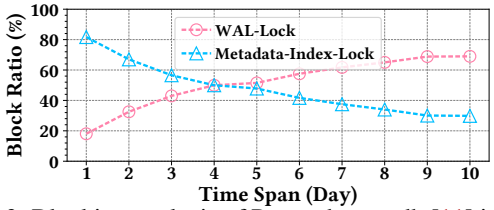


Figure 3: Blocking analysis of Prometheus tsdb [11] inserting 1 million time series at 30s intervals using *pprof* [9].

label combinations). Metadata and its indexes can dominate memory consumption in production. We observe that metadata storage accounts for 35.1%–46.7% of Prometheus tsdb's memory consumption after ingesting 1–10 million time series (12 hours of data, 30s sampling interval) in the DevOps case of TSBS [15]. Meanwhile, sustaining ingestion throughput demands highly concurrent writes, where metadata index contention becomes a critical bottleneck.

Why existing metadata models are inferior. A natural approach is to reduce redundant labels via sharing. Heracles [39] and TimeUnion [37] adopt a group model that clusters time series sharing some common labels and stores the shared labels once per group. While effective in certain scenarios, this model has two limitations. ❶ *Grouping depends on external conventions rather than explicit semantics:* Heracles relies on collector-set group IDs, and TimeUnion relies on user-defined group labels, limiting generality across deployments. ❷ *Limited label sharing:* Grouping mainly reduces redundancy within a group but overlooks redundant labels across groups. For example, when time series are grouped by data source, labels associated with the same metric across different groups cannot be deduplicated. However, this pattern is common in production environments, where the same metric is collected from many sources [13, 15].

Metadata index contention. TSDBs suffer from metadata-index contention under concurrent access (*e.g.,* due to coarse-grained locking in inverted indexes), limiting concurrency. Our experiments on Prometheus tsdb show that metadata-index contention is a major source of blocking (Fig. 3). Although prior work [38, 40] improves index structures, it still relies on a suboptimal metadata model, making it difficult to achieve both high concurrency and low metadata overhead.

Therefore, PromTCS proposes a novel metadata model that accurately separates metadata semantics into two dimensions, namely *metric* and *source*, to maximally eliminate redundant labels. Building on this model, we design a set of storage-efficient and concurrency-friendly metadata indexes.

2.2.2 Performance Limitations of Storage Architectures

Using high-bandwidth devices (*e.g.,* NVMe SSDs) improves ingestion throughput but increases storage cost: enterprise-grade NVMe SSDs are 5–6× more expensive than HDDs [4]. Meanwhile, many TSDBs [1, 6] adopt LSM-tree–based storage engines, whose compactions incur severe WA, increasing SSD writes and shortening device lifetime [42]. Although a DRAM–SSD–HDD hierarchy can reduce storage cost while

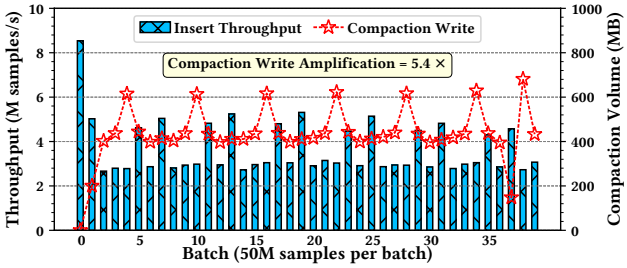


Figure 4: Insert throughput and compactions in Pure-LSM, a PromTCS variant that retains our concurrency-friendly WAL and metadata index designs but uses a time-partitioned LSM-tree [38] on NVMe SSD (TSBS [15] DevOps, 1 million time series, 30s sampling interval). Overall WA = 5.4×.

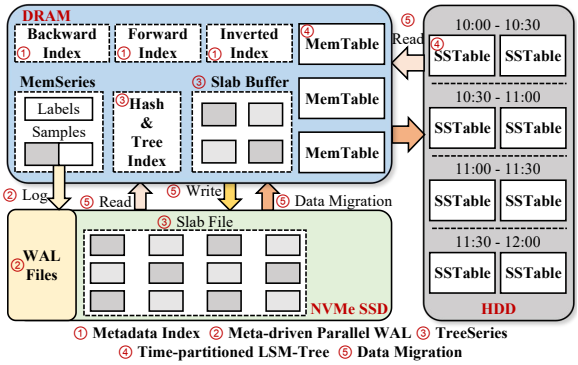


Figure 5: Architecture of PromTCS.

maintaining high performance, existing hybrid-storage TSDBs still rely on large SSD capacity. For example, Goku [34] stores the most recent 80 days of data (tiers 1–4) on SSD.

We find that building a cost-effective yet high-performance hybrid storage for TSDBs requires addressing three key issues. ❶ *Bottlenecks caused by low-bandwidth devices:* Under tight cost constraints, limited SSD capacity forces most data onto HDD, making HDD I/O during flushing and compaction the bottleneck and throttling ingestion throughput. We validate this effect on Prometheus tsdb [11]: limiting the SSD tier to only 24 GB, which is much smaller than the HDD capacity, placing older data on HDD reduces ingestion throughput by 71% compared with an all-NVMe setup. ❷ *Global WAL contention:* As shown in Fig. 3, WAL contention is a major source of *blocking*. Concurrent writers serialize on a global WAL lock, while WAL I/O further extends the critical path, making WAL a key scalability bottleneck. ❸ *Compaction in LSM-tree:* After mitigating WAL and metadata index contention (see Fig. 3), compaction becomes the dominant bottleneck, causing write stalls that throttle ingestion performance, and incurs severe WA that shortens SSD lifespan [24, 41]. Fig. 4 shows that insert throughput repeatedly drops during compactions, coinciding with surges in compaction write volume, with an overall WA of 5.4×.

To address these issues, PromTCS proposes a tailored hybrid storage architecture. It deploys a metadata-driven parallel WAL and the compaction-free time-series engine TreeSeries on NVMe SSD to handle massive writes, and adopts a time-

partitioned LSM-tree on HDD for old data. This distributes the I/O overhead of logging, samples, and compaction across different physical disks and reduces WA on the SSD. It further mitigates cross-tier bandwidth gaps through early migration and lazy flushing while preserving query performance.

3 PromTCS Design

PromTCS introduces a new metadata management and hybrid storage architecture while retaining the Prometheus interface for time series insertion and querying:

- Insert(labels, samples)
- Query(label patterns, start time, end time) → result

In this section, we describe the design of PromTCS.

3.1 Architecture Overview

Fig. 5 illustrates the main components of PromTCS:

- ① **Metadata Management.** We propose a two-dimensional metadata model with concurrency-friendly indexes.
- ② **Meta-driven Parallel WAL.** We partition WAL into groups based on metadata to distribute the logging load, and enable parallel log cleanup and crash recovery.
- ③ **TreeSeries.** We design TreeSeries, a compaction-free time series storage engine for NVMe SSDs, which reduces WA and improves performance for time series workload.
- ④ **Time-partitioned LSM-tree.** We use time-partitioned LSM-trees on HDDs to manage the old data.
- ⑤ **Data Migration Module.** We design tailored write strategies, *early migration* and *lazy flushing* mechanisms.

End-to-end workflow. We walk through PromTCS's read and write paths to illustrate how these components work together. **Write path.** PromTCS resolves labels via the metadata index (①) to locate the *TSObject* (Sec. 3.2.3), logs through the parallel WAL (②), then buffers samples in the *TSObject*'s in-memory chunk. When the chunk fills, it is flushed to TreeSeries (③). Old data is then migrated to the LSM-tree (④) via our migration module (⑤).

Read path. A query resolves labels via the metadata index (①) to locate the TSObject (Sec. 3.2.3), then is routed by time range across tiers: recent data from in-memory chunks, mid-term data from TreeSeries (③) on NVMe SSD, and historical data from the LSM-tree (④) on HDD. HDD chunks are loaded on demand without caching; since queries skew toward recent data, HDD access is infrequent with minimal latency impact. Results from all tiers are merged and returned. The multi-tier layout naturally lends itself to a future query optimizer that skips unnecessary tiers' I/O and parallelizes tier reads for queries according to query time range.

3.2 Two-Dimensional Metadata Management

In this section, we present a two-dimensional metadata model and indexes to reduce overhead and improve scalability.

3.2.1 Two-Dimensional Metadata Model

We propose a new metadata model to reduce metadata redundancy across metrics and data sources, as follows:

Definition 1 (Two-Dimensional Metadata Model)**.** *Let l denote the metadata of a time series, represented as a set of labels. We partition l into two disjoint subsets, the* metric label set l_m *and the* source label set l_s*, such that*

$$l = l_m \cup l_s, \qquad l_m \cap l_s = \emptyset. \tag{1}$$

We assign a unique MetricID *to each distinct l_m and a unique* SourceID *to each distinct l_s, i.e.,*

$$f_m(l_m) = MetricID, \qquad f_s(l_s) = SourceID, \tag{2}$$

where f_m and f_s are injective. Accordingly, each time series is uniquely identified by the pair $(MetricID, SourceID)$.

Example 1. *In Fig. 2, the metadata of a time series is partitioned into a metric label set l_m* (e.g., $\{\langle name = cpu_usage\rangle\}$) *and a source label set l_s* (e.g., $\{\langle hostname = host_0\rangle, \ldots\}$)*, mapped to a MetricID (mid) and a SourceID (sid) that uniquely identify the time series as (mid, sid).*

This two-dimensional decomposition is not arbitrary. Performance-monitoring metadata naturally consists of metric and source dimensions: each time series corresponds to a metric (e.g., *cpu_usage*) on a source (e.g., a server). Since metrics are shared across sources and each source produces multiple metrics, indexing these dimensions separately exploits their inherent reuse, reducing redundancy and indexing overhead while preserving full metadata reconstruction.

3.2.2 Metadata Index

We design efficient and scalable metadata indexes based on the following three data structures:

- **Thmap [14].** It is a concurrent hash-trie that maps keys to 32-bit *MurmurHash3* [19] hashes and organizes them as a trie for fast lookups. It provides lock-free reads via atomic CAS and uses fine-grained node locks for updates.
- **MergeSet.** It is a simplified single-level LSM-tree in VictoriaMetrics [18] that encodes each key as a concatenation of a tag and a posting ID, so posting lists emerge naturally from the maintained key order.
- **Flattened Array.** It stores entries sequentially in a compact layout, achieving fast address-based access.

Our evaluation shows that ❶ Thmap outperforms MergeSet on reads and writes, whereas MergeSet has lower memory overhead. ❷ A flattened array provides better read performance and storage efficiency than Thmap due to direct addressing and no load factor, but it has slower insertions.

Backward index. Every read and write must first resolve labels to IDs, so the backward index demands fast, concurrent lookups. We implement the backward index with two Thmaps, the *metric backward index* and the *source backward index*, whose lock-free reads and fine-grained node-level locking suit this high-frequency, multi-threaded access pattern. The former maps metric labels to *MetricID*s, while the latter maps source labels to *SourceID*s (see Fig. 6).

Forward index. The forward index maps (*MetricID*, *SourceID*) pairs to *TSObject*s. Queries dominate this index (updates occur only when new time series are registered), so

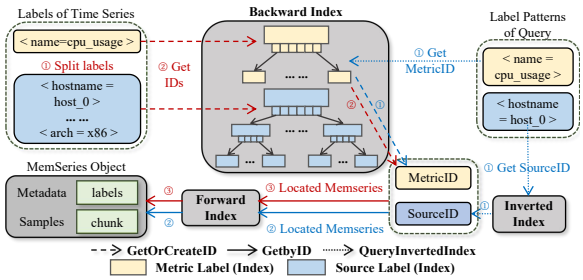


Figure 6: Operations on metadata indexes.

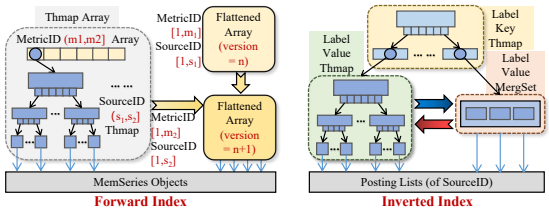


Figure 7: Forward index and inverted index.

read performance and storage density are the primary concerns. We therefore split it into a read-optimized *flattened array* for the bulk of entries, which supports direct address-based access without hash-table overhead, and a *Thmap array* that buffers newly added entries with efficient insertion. Periodically, the Thmap entries are merged into the flattened array to reclaim space. As shown in Fig. 7, Thmap array stores entries of newly added time series, *i.e.,* $MetricID \in (m1, m2], SourceID \in (s1, s2]$, while the flattened array with version number n stores entries of historical time series, *i.e.,* $MetricID \in [1, m1], SourceID \in [1, s1]$. These are merged into a flattened array with version number $n+1$, *i.e.,* $MetricID \in [1, m2], SourceID \in [1, s2]$.

Inverted index. It is a two-level inverted index that maps each source label to a posting list, i.e., a sorted list of SourceIDs whose sources contain that label (see Fig. 7). For example, the posting list for $\langle hostname = server_1 \rangle$ is $[s_1, s_3, s_7]$, meaning these three sources share this hostname. Since $N_s \ll N$, storing SourceIDs greatly shrinks posting lists. In the first level, we use a Thmap for label keys, ensuring performance with minimal memory usage given the small cardinality of label keys. Although the number of label values scales with data sources and can be large in practice, only a subset of labels is queried with significant frequency (*e.g., hostname* in TSBS [15]). Thus, in the second level, we use Thmap for hot label values and MergeSet for cold ones. Newly written label values are always inserted into Thmap first for insertion performance. We design a cold-hot awareness mechanism that monitors the access frequency of each label-key leaf node and dynamically migrates the label-value indexes between Thmap and MergeSet according to their access frequency.

3.2.3 Metadata indexing interface and workflow

Locating the *TSObject* via the metadata index is essential for both reads and writes. We next describe the interfaces and walk through a concrete example using the data in Fig. 6.

- WriteIndex(labels)→TSObject. Following the red arrows, labels are split into metric labels (<*name=cpu_usage*>) and source labels (<*hostname=host_0*>, ..., <*arch=x64*>), mapped to a MetricID and SourceID via the backward indexes, and combined to locate the *TSObject* in the forward index. For new time series, PromTCS assigns new IDs and updates the indexes.
- ReadIndex(label patterns) → vector<TSObject>. Following the blue arrows, label patterns are split into metric and source parts (<*name=cpu_usage*> and <*hostname=host_0*>). The backward index resolves metric labels to MetricIDs. The inverted index retrieves posting lists of SourceIDs for source labels and intersects them. The resulting MetricIDs and SourceIDs are used to retrieve all matching *TSObject*s from the forward index.

3.2.4 Metadata Storage Cost Study

We analyze PromTCS's metadata cost and compare it with Prometheus tsdb [11] and TimeUnion [37] (group model). Detailed derivations are provided in the technical report [12].

Metadata cost model. Let N_s, N_m, N be the number of data sources, metrics, and time series, respectively, and S_p, S_l, S_h be the average size per posting list entry, per label, and per single hash entry, respectively, and T be the average number of labels per time series. Then the metadata storage cost for PromTCS, denoted as $Cost_{TCS}$:

$$Cost_{TCS} = N_m \cdot S_l + (N + N_m + N_s) \cdot S_h + N_s \cdot (T-1) \cdot (S_l + S_p)$$

Comparison. We summarize the corresponding cost for both Prometheus tsdb [11] and TimeUnion [37] in Tab. 2, where T_u denotes the average number of unique labels per group. From the cost formulas above, PromTCS compares favorably to the baselines. Let $Cost_{TU}$ and $Cost_{tsdb}$ be the costs for TimeUnion [37] and Prometheus tsdb [11], respectively:

$$\begin{cases} Cost_{TCS} \le Cost_{tsdb} & \text{if} \quad N_m \ge 2 \quad \text{and} \quad N_s \ge 2 \\ Cost_{TCS} \le Cost_{TU} \end{cases}$$

PromTCS often yields lower overhead (*e.g.,* DevOps case in TSBS [15]: $N_m = 100$, $N_s \gg 2$).

3.3 TreeSeries

In this section, we introduce TreeSeries, a compaction-free time series storage engine designed for multi-core CPUs and NVMe SSDs (see Fig. 8). It serves as a write buffer on NVMe SSDs to mitigate WA and deliver superior performance.

3.3.1 Slab Data Management

The basic unit for managing data in TreeSeries is the slab, which is a contiguous space in memory or SSDs. The size of each slab is 4 KB, the smallest read/write unit in SSDs [26].

Slab structure. As shown in Fig. 9, each slab is evenly divided into multiple items. Each item consists of a *MetricID* (2 bytes), a *SourceID* (8 bytes), and a *chunk*, all of which are from a time series. Since queries need to decompress chunks to locate samples in the given time range, a larger chunk size can degrade query performance. Conversely, a smaller chunk size can degrade compression efficiency. To balance these factors, we set the item size to 256 bytes, *i.e.,* each chunk is

Table 2: Metadata Storage Costs

TSDB	Storage Cost
tsdb [11]	$N \cdot (T \cdot S_l + 2S_h + T \cdot S_p)$
TU [37]	$(N_s \cdot (T-1) + N) \cdot S_l + (2N + N_s) \cdot S_h + (N_s \cdot T_u + N) \cdot S_p$
TCS	$N_m \cdot S_l + (N + N_m + N_s) \cdot S_h + N_s \cdot (T-1) \cdot (S_l + S_p)$

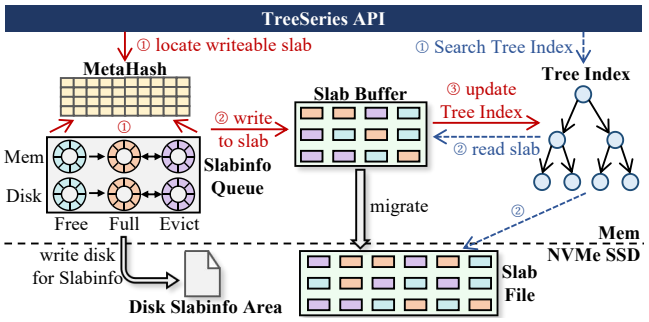


Figure 8: The design of TreeSeries.

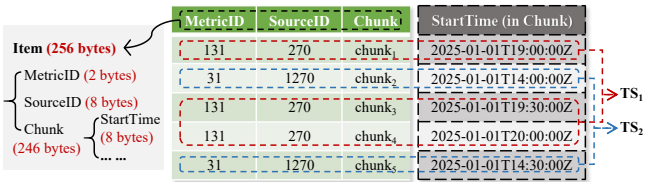


Figure 9: Illustration of slab structure and relaxed ordering.

246 bytes (the extensive results for other item/chunk sizes can be found in the technical report [12]).

Slab buffer. A substantial contiguous block of memory is allocated as the slab buffer, which is subsequently partitioned into multiple slabs. When a chunk is to be stored in TreeSeries, it is first written to a slab buffer rather than directly to the SSD, ensuring a high throughput of insertion.

Slab file. We create a large SSD file using 'truncate' [17], referred to as the slab file, and divide it into multiple slabs. When the slab buffer reaches a threshold, a background thread flushes data to the slab file. In our experiments, the slab buffer and slab file size are set to 8 GB and 24 GB, respectively, to balance cost and performance; additional settings are evaluated in the technical report [12].

Slabinfo lock-free queue. We bind a *slabinfo* to each slab, recording metadata such as *SlabID* (slab ID), *Status* (usage status), and *Mem* (in-memory flag). We use an *atomic_queue* [2], a lock-free circular-buffer queue supporting multiple producers/consumers, to manage concurrent access to *slabinfo*. PromTCS maintains six queues to track slabs in three statuses (free, full, evict) across memory (*MemFreeQueue*, *MemFullQueue*, *MemEvictQueue*) and disk (*DiskFreeQueue*, *DiskFullQueue*, *DiskEvictQueue*). Fig. 8 illustrates the *slabinfo* migration process: on insertion, a *slabinfo* is dequeued from a free queue to locate a writable slab; when full, it enters the full queue; after migration, it moves to an evict queue. If the free queue is empty, TreeSeries reuses an evict-queue *slabinfo*, overwriting its slab in-place and returning it to the full queue. The detailed operations appear in Alg. 2 and Alg. 3.

3.3.2 Concurrent Index Structures

We introduce a pair of indexes into TreeSeries: a *tree index* for concurrent queries and a *MetaHash* for locating the writable slab on concurrent insertion.

Tree index. To enable fast data retrieval, we build a *lockless cache-aware B+ tree* [29] on a per-slab basis. The index key consists of the SourceID, MetricID, and time range of Items belonging to the same time series within a slab. Therefore, a slab corresponds to multiple index entries, since a slab stores ≥ 1 chunks of ≥ 1 time series.

MetaHash & Relaxed ordering. *MetaHash* is a two-dimensional array, where each entry stores a mutex lock *mtx* for fine-grained synchronization and a *SlabID* to locate the slab in the slab buffer. When writing massive data, TreeSeries uses MetaHash to distribute it across multiple slabs. The detailed write process is shown in Alg. 1. When a chunk is written, the *MetricID* and *SourceID* of the time series are modded by the row and column counts of *MetaHash* to obtain indices i and j (line 2–3, where *GetRowNum* and *GetColNum* return the dimensions of *MetaHash*). The $MetaHash[i][j]$ is then accessed and locked to locate a writable slab. If the *SlabID* is IN_VALID, a usable slab is fetched from *MemFreeQueue* or *MemEvictQueue*, and the slab information in $MetaHash[i][j]$ is updated (line 4-11). When a slab is fully written, the corresponding *slabinfo* is stored in *MemFullQueue*, and $MetaHash[i][j]$ is reset to IN_VALID (line 12-14). Finally, TreeSeries unlocks $MetaHash[i][j]$ (line 15).

Mapping writable slabs via *MetaHash* brings three advantages. ❶ Distributing data from a large number of time series across multiple slabs reduces lock contention among worker threads during insertion. ❷ Each slab contains samples from as few time series as possible, which effectively reduces the number of index entries in the *tree index* and the I/O overhead for large time range queries. ❸ TreeSeries implements a **relaxed ordering**, and thus avoids the costly compaction for sorting, which is the main cause of WA and write stalls in LSM-trees [41]. In particular, within TreeSeries, items are unordered across different slabs (*resp.*, time series), but within each slab, items belonging to the same time series are stored either contiguously or discretely in time order. As illustrated in Fig. 9, this slab contains data from two time series: $TS_1(MetricID = 131, SourceID = 270)$ and $TS_2(MetricID = 31, SourceID = 1270)$. There is no order between the items of TS_1 and TS_2. However, the items within each time series are sorted by StartTime (*e.g.*, $chunk_2.StartTime < chunk_5.StartTime$).

3.3.3 Disk Slabinfo Area & Crash Recovery

In TreeSeries, crash recovery only needs to restore the slab file and the corresponding items in the tree index. Since the data migration from the slab buffer to the slab file is performed in units of slabs (as discussed in Sec. 3.4), we only need to save the *slabinfo* in the *DiskFullQueue* and its related index items in the tree index to the disk slabinfo area (*i.e.*, a file on the NVMe SSD, shown in the left bottom of Fig. 8) each time

Algorithm 1: The Insertion Process of TreeSeries

```
1  Function InsertTSeries (MetricID, SourceID, Chunk) :
2      i = MetricID % GetRowNum (MetaHash) ;
3      j = SourceID % GetColNum (MetaHash) ;
4      MetaHash[i][j].Lock() ;
5      if MetaHash[i][j].SlabID == IN_VALID then
6          if MemFreeQueue.Empty() then
7              MetaHash[i][j].SlabID=MemEvictQueue.TryPop() ;
8          else
9              MetaHash[i][j].SlabID=MemFreeQueue.Pop() ;
10     msinfo = GetSlabInfo (MetaHash[i][j].SlabID) ;
11     WriteSlab (msinfo,Chunk) ;
12     if IsFullSlab (msinfo) then
13         MemFullQueue.Push(msinfo) ;
14         MetaHash[i][j].SlabID = IN_VALID ;
15     MetaHash[i][j].UnLock() ;
```

Algorithm 2: Data Migration from DRAM to NVMe

```
1  # msinfos: Slabinfos from MemFullQueue ;
2  # dsinfos: Slabinfos from DiskFreeQueue ;
3  # pool: The thread pool for I/O ;
4  Function OptBatchWriteToNSSD (msinfos,dsinfos) :
5      SlabGroups = PartitionSlabGroup (msinfos) ;
6      i = 0 ;
7      foreach sgroup ∈ SlabGroups do
8          flag = CheckSeqSpace (dsinfos[i], sgroup.count);
9          if flag == true then
10             pool.Submit(SeqWrite (dsinfos[i], sgroup));
11         else
12             pool.Submit(RandWrite (dsinfo[i], sgroup)) ;
13         i += sgroup.count ;
14     pool.Wait() ;
15     Move(MemFullQueue, MemEvictQueue, msinfos);
16     Move(DiskFreeQueue, DiskFullQueue, dsinfos) ;
17     WriteDiskInfoArea () ;
18     UpdateCleanTxn () ;
```

data is migrated. During crash recovery, TreeSeries reads the data from the disk slabinfo area to restore the *slabinfo* in the *DiskFullQueue* and rebuild the tree index.

3.3.4 TreeSeries Interface and workflow

We implement the following interface to simplify its usage:

- Put(SourceID, MetricID, chunk). It inserts data (*i.e.,* chunk) into TreeSeries. Fig. 8 (red arrows) illustrates the process: data is written into a writable slab located via the *MetaHash* according to Alg. 1, followed by an update to the tree index.
- Get(SourceID, MetricID, time range) -> vector<slab>. It returns a group of slabs containing samples within the given time range. The scan process is illustrated in Fig. 8 (blue arrows). TreeSeries forms a range search key, $[s_1+m_1+t_{\text{start}}, s_1+m_1+t_{\text{end}}]$, from the SourceID ($s_1$), MetricID ($m_1$), and time range ($[t_{\text{start}}, t_{\text{end}}]$), then looks up the tree index and returns the matching slabs.

3.4 Data Migration

The bandwidth disparity across storage devices can cause data to accumulate on high-bandwidth tiers, degrading write performance. To better exploit heterogeneous bandwidth, we employ device-aware write strategies for NVMe SSDs and HDDs. We further introduce early data migration and lazy flushing to utilize I/O resources more effectively and prevent buildup on fast storage without impacting query performance.

Batch slabs random writing to NVMe SSDs. We next describe migration from DRAM (the slab buffer) to NVMe SSD (the slab file) in TreeSeries. Leveraging NVMe's high concurrency [27, 42], TreeSeries performs batched, slab-granularity parallel I/O using an I/O thread pool to avoid per-migration thread overhead. When the full-slab ratio in the slab buffer reaches threshold k_1 and the slab file has sufficient free space, a background thread triggers migration (Alg. 2). It dequeues *slabinfo*s from *MemFullQueue* (*msinfos*), sorts them by *SlabID*, and groups adjacent slabs into *slab groups* (line 5); the $O(n \log n)$ cost is negligible compared with I/O. For each group, TreeSeries checks *dsinfos* for contiguous free space: if available, it writes the group sequentially; otherwise, it falls back to per-slab random writes (lines 7–13). After I/O completion, it updates the lock-free queues (lines 14–16), persists the *DiskFullQueue slabinfo* and related index entries for crash recovery (line 17, Sec. 3.3.3), and advances *TSObject.cleantxn* to enable log cleanup (line 18, Sec. 3.5). Writing in slab groups ❶ reduces syscall overhead and ❷ exploits NVMe's higher sequential bandwidth than random I/O [42].

Batch slabs sequential writing to HDDs. To migrate data from NVMe SSDs (slab file in TreeSeries) to HDDs (SSTable in time-partitioned LSM-tree [38]), we aim to leverage HDDs' high sequential I/O bandwidth. Thus, it reads slabs from the slab file, materializes them into a large *MemTable*, and flushes it to HDD. Since slabs are ordered by *SlabID*, the thread can read a batch of slabs sequentially with only one or two syscalls (line 3). For each slab, it sorts items and groups them into *ItemGroups* by time series, with each group sorted by timestamp (line 5). Inserting *ItemGroups* into the *MemTable* as units reduces redundant keys composed of IDs and timestamps (lines 6–7). Afterward, it updates the lock-free queues (lines 8–9). To avoid memory buildup, TreeSeries flushes each *MemTable* to HDD during the migration process (line 10).

Early Migration & Lazy Flushing Mechanism. Due to the I/O bandwidth gap between DRAM, NVMe SSDs, and HDDs, data can accumulate on high-bandwidth storage devices. We set low migration thresholds ($k_1 = \frac{1}{6}$, $k_2 = \frac{1}{10}$) to trigger early data migration, which helps mitigate data piling and improve insertion throughput but may negatively impact query workloads. Thus, we developed a lazy flush mechanism: data migrates early to maximize I/O resources, while old data is kept in the original (fast) storage devices for queries. More

Algorithm 3: Data Migration from NVMe to HDD

```
1  # dsinfos: Slabinfos from DiskFullQueue ;
2  Function OptSeqWriteToHDD (dsinfos) :
3      Slabs = ReadSlabFile (dsinfos) ;
4      foreach slab ∈ Slabs do
5          ItemGroups = GroupByTimeSeries (slab) ;
6          foreach igroup ∈ ItemGroups do
7              WriteToMemTables (igroup) ;
8      DiskFullQueue.Pop(dsinfos) ;
9      DiskEvictQueue.Push(dsinfos) ;
10     Flush (MemTables) ;
```

results for k_1 and k_2 are provided in the technical report [12].

3.5 Meta-driven Parallel WAL

We propose a metadata-driven parallel WAL to cut synchronization and I/O overhead, a log cleanup policy to prevent log accumulation, and parallel recovery for fast crash recovery.

Metadata-based grouped logging. A global WAL limits concurrent writes due to synchronization on the WAL lock (see Fig. 3). Using multiple WALs, however, raises concerns about random disk I/O and transaction dependencies. To address this, we design a metadata-driven parallel WAL.

In PromTCS, logs are split into *series log*s (a *log group*) and *sample log*s (multiple *log group*s): the former records time series metadata and IDs, while the latter records samples and IDs (see Fig. 10). Each log group consists of a ring lock-free log buffer in memory and a log directory on disk. We assign one *log thread* to each log group to perform *group commit* [23, 25, 30], *i.e.,* batch-writing log records to disk. PromTCS dispatches sample records to sample log groups by hashing their *MetricID*s (modulo the number of groups). Since each sample log group contains records from different time series, it does not need to preserve a global order across groups. Notably, we group by *MetricID* rather than *MetricID+SourceID*. Grouping by metric co-locates more sample logs of the same metric within the same WAL group, achieving higher compression efficiency because values of the same metric are more similar. On a real-world dataset (see Sec. 4.1), grouping by *MetricID* achieves up to 36.2% higher compression efficiency than *MetricID+SourceID*. Moreover, we ensure reliability by verifying log persistence per time series. Before *TSObject* inserts a chunk into TreeSeries, it checks whether the log record of the last sample in the chunk has been flushed. This is done by comparing the sample sequence number *txn*, *i.e.,* a monotonically increasing value for samples within a time series. In our experiments, the sample log group size is set to 5 to balance CPU and I/O overhead (see the technical report for additional parameter evaluations).

Log cleanup. To prevent log file buildup, we periodically clean up *sample log record*s, since series logs are much smaller and only store metadata for recovery. A *sample log record* becomes cleanable once its corresponding samples

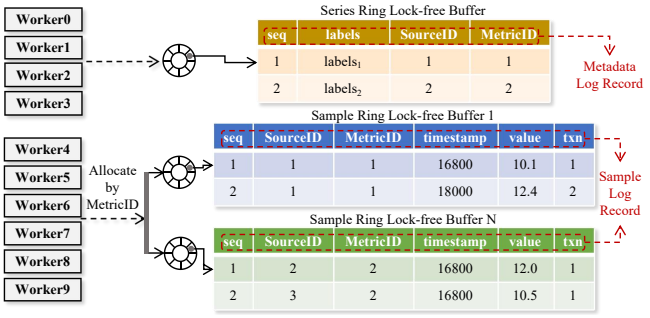


Figure 10: Illustration of meta-driven parallel WAL.

have been persisted to disk. For each *sample log group*, a cleaner thread periodically scans the log files under its *log directory*. When TreeSeries flushes samples to the NVMe SSD, it updates the corresponding *TSObject*'s cleanable sequence number (*clean_txn*, line 18 of Alg. 2). A *sample log record* is removed if its *txn* does not exceed the corresponding *clean_txn*; otherwise, it is rewritten to the log file.

Parallel crash recovery. Since each *log group* contains data from different time series, multiple threads can replay logs from different *log directories* in parallel without synchronization. Recovery first rebuilds the metadata index from the series log using stored *labels*, *MetricID*, and *SourceID*. Each thread then replays one log directory, reinserting data into the corresponding time series and updating sequence numbers. Log recording and cleanup remain suspended until recovery completes. Meanwhile, TreeSeries and the time-partitioned LSM-tree recover by reading only a small amount of metadata (*i.e.,* the *disk slabinfo area* on NVMe SSDs and the *CURRENT* and *MANIFEST* files [7] on HDDs), making their overhead much lower than log replay. Parallel recovery effectively reduces recovery time [12].

4 Experiments

This section presents PromTCS's implementation and experimental setup (Sec. 4.1), followed by evaluations of its end-to-end performance (Sec. 4.2), storage engine performance (Sec. 4.3), and ablation studies on the metadata management and storage architecture (Sec. 4.4–Sec. 4.5).

4.1 Implementation and Experimental Setup

Implementation. We implemented PromTCS [12] on the top of ForestTI [38] with about 12K lines of C++ code. The metadata index builds on Thmap [14] and MergeSet [18]. We develop TreeSeries from scratch, using a lock-free queue [2] and a lock-free cache-aware tree [29] to support concurrency. ForestTI's time-partitioned LSM-tree serves as the HDD layer, while its WAL is replaced by our parallel WAL.

Setup. The test platform consists of one TSDB server and two application servers. Each is equipped with two 2.9 GHz 16-core Intel Xeon Gold 6226R CPUs, 128 GB DDR5 RAM, 1 TB PCIe 3.0 NVMe SSD, 7200 RPM 16-TB HDD, and runs 64-bit Ubuntu 20.04 LTS. We deployed end-to-end benchmarks and generated the real-world dataset on the application servers. All servers are connected via 10-gigabit Ethernet.

Baselines. We compare PromTCS with two types of TSDBs.

Table 3: TSBS query patterns and recording rules.

Type	Name	Description
TSBS Query	1-8-1	Max on 1 metric, 8 hosts, 5-min interval, 1 hour.
	5-1-1	Max on 5 metrics, 1 host, 5-min interval, 1 hour.
	5-8-1	Max on 5 metrics, 8 hosts, 5-min interval, 1 hour.
	5-1-24	Max on 5 metrics, 1 host, 5-min interval, 24 hours.
	5-8-24	Max on 5 metrics, 8 hosts, 5-min interval, 24 hours.
Recording Rule	cpu_avg	Avg CPU usage of all hosts (last 5 min).
	mem_avg	Avg memory usage of all hosts (last 5 min).
	net_transmit	Avg network receiving throughput of all hosts (last 5 min).

- **Hybrid-storage TSDBs.** We compare PromTCS with three LSM-tree-based TSDBs: ❶ Prometheus tsdb [11]; ❷ TimeUnion [37]; and ❸ ForestTI [38]. For a fair comparison, all systems are configured with the same hybrid storage as PromTCS: recent data reside in DRAM and on a 24 GB SSD (matching PromTCS's slab file size), while the remaining data are stored on HDDs. Additional results with different SSD sizes are provided in the technical report [12].
- **Non-hybrid-storage TSDBs.** We select three advanced and popular TSDBs: InfluxDB 3.8.0 [6], TimescaleDB 2.6.0 [16] and Apache IoTDB 2.0.5 [1] (*abbr.,* IoTDB). Their storage layer is tightly coupled with the rest of the system, making it difficult to implement a comparable tiered placement internally. We therefore evaluate them end-to-end under their recommended deployments, and report results for SSD-only and HDD-only setups.

All TSDBs are configured to use 32 worker threads to fully utilize the server while avoiding system overload.

Datasets. In our experiments, we employ both synthetic datasets from a popular benchmark and real-world datasets.

- **Synthetic Dataset.** The DevOps use case is from Time Series Benchmarking Suite (TSBS) [15], simulating 100 metrics from nine systems (*e.g.,* CPU, memory, *etc.*).
- **Real-world Dataset.** We use Prometheus to collect metrics from Docker [5] containers via *Node Exporter* [8] on application servers. Each container exposes about 500 metrics (*e.g., cpu_usage*). We scale the dataset by replication.

Both datasets have a sampling interval of 30 seconds. Unless otherwise specified, experiments use 3 million time series over 20 days (172.8 billion samples in total). Across 1–10 million time series, the total sample volume ranges from 57.6 billion to 576 billion. Experimental results for different numbers of time series are presented in Sec. 4.3 and Sec. 4.4.

Workloads & Metrics. We adopt two workload configurations. For **end-to-end evaluation (Section 4.2)**, we emulate production workloads by issuing concurrent inserts and queries: after ingesting 24 hours of samples, queries are launched every five minutes while inserts continue, using 32 client threads via HTTP APIs. We report insertion throughput (samples/s), query latency (average, P90, and P99), memory usage, disk usage, and total write size. For **storage-engine evaluation (Sec. 4.3–Sec. 4.5)**, we use direct API calls for cleaner ablation studies without HTTP or serialization overhead. All data are ingested before queries to eliminate compaction interference. Only average query latency is reported.

We use query workloads from TSBS [15] and production environments (recording rules [13, 39]), as shown in Tab. 3.

4.2 End-to-end Evaluation

Overall comparison. We evaluate end-to-end performance via HTTP APIs. As shown in Fig. 11, for data insertions, PromTCS performs the best, achieving throughput comparable to the best-performing baseline (*i.e.,* IoTDB-SSD) and outperforming hybrid-storage baselines (*e.g.,* ForestTI and Prometheus tsdb) by 3.2-5.9 ×. PromTCS achieves higher insertion throughput due to four factors: ❶ PromTCS employs a concurrency-friendly metadata index design, which better supports multi-threaded concurrent insertions. ❷ The relaxed ordered of TreeSeries avoids write stalls caused by the expensive compactions. ❸ The meta-driven parallel WAL reduces lock overhead and fully leverages the concurrent I/O capabilities of NVMe SSDs. ❹ PromTCS optimizes its data migration strategy to fully utilize the I/O resources of multiple storage devices, reducing insertion throughput degradation.

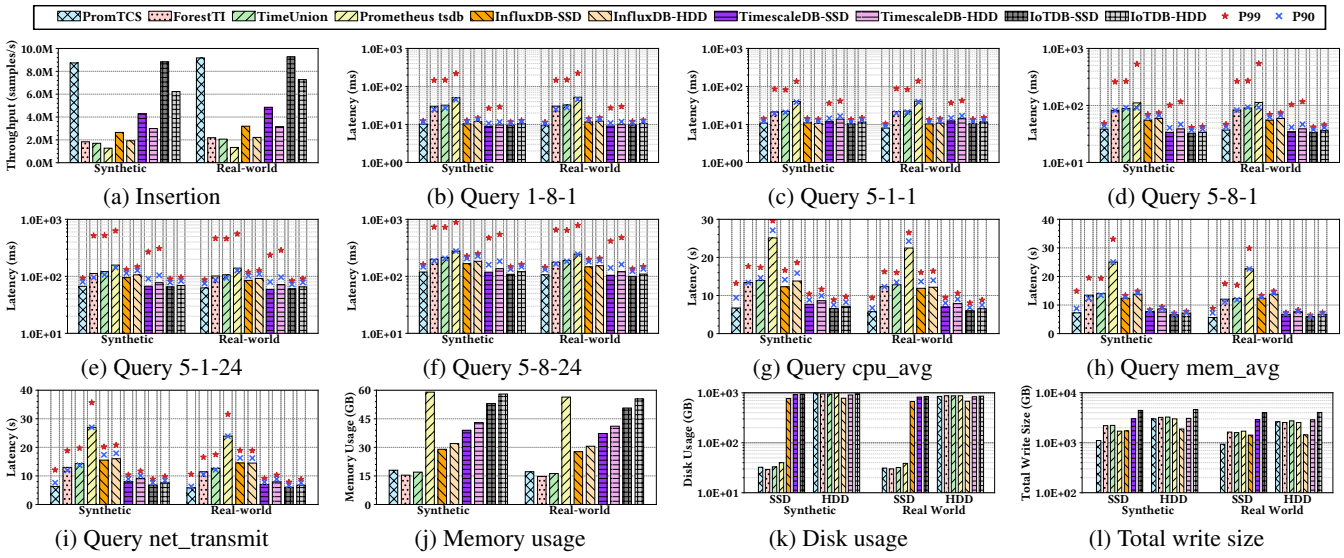
For recent data queries (*i.e.,* 1-8-1, 5-1-1, 5-8-1, and recording rules), PromTCS exhibits 55.3%-82.1% lower average latency than the hybrid-storage baselines. There are two reasons for this advantage: ❶ By storing only source labels and SourceIDs in the inverted index, PromTCS loads fewer, smaller posting lists and reduces the cost of intersecting multiple lists. ❷ PromTCS adopts a concurrency-friendly index design that enables faster access to TSObjects. For long-term queries (*i.e.,* 5-1-24 and 5-8-24), PromTCS reduces average latency by 38.1%-57.5% compared with most baselines, excluding TimescaleDB and IoTDB. The main overhead of long-term queries is disk I/O and decoding during sample retrieval. PromTCS's advantage stems from: ❶ Lazy flushing keeps recent data in high-bandwidth storage devices (*i.e.,* DRAM and NVMe SSD) as much as possible. ❷ PromTCS uses device-aware storage engines: TreeSeries on NVMe SSDs to reduce read amplification and leverage parallel I/O, and a time-partitioned LSM-tree on HDDs to better utilize sequential I/O bandwidth. TimescaleDB and IoTDB also perform well on queries due to effective caching.

Tail latency. Across all query types, PromTCS consistently achieves lower P90 and P99 tail latencies, with 78.0%–91.4% lower P99 latency than ForestTI, the best-performing hybrid-storage baseline. This improvement is attributed to its concurrency-friendly index design and efficient data I/O.

Cost-effectiveness. PromTCS achieves lower deployment cost through reduced memory and SSD usage, as well as fewer SSD writes that extend its lifespan. Compared with IoTDB-SSD at comparable performance, it uses 2.9× less memory, 29.1× less SSD space, and writes 4.0× less data.

For ease of discussion and due to space limitations, the remaining experiments report only 5-8-1 to represent recent data queries and 5-1-24 to represent long-term queries.

Larger scale. We scale the experiment to 10 million time

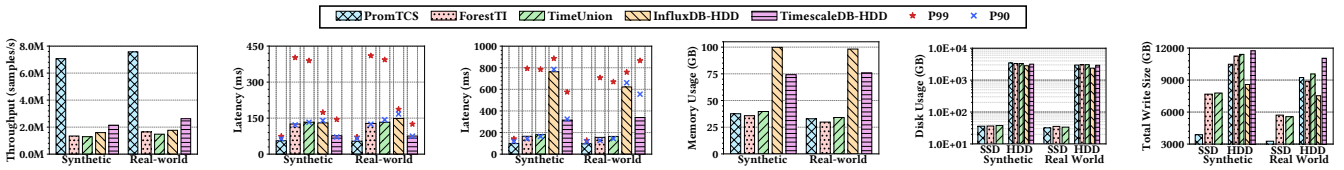


(a) Insertion (b) Query 1-8-1 (c) Query 5-1-1 (d) Query 5-8-1

(e) Query 5-1-24 (f) Query 5-8-24 (g) Query cpu_avg (h) Query mem_avg

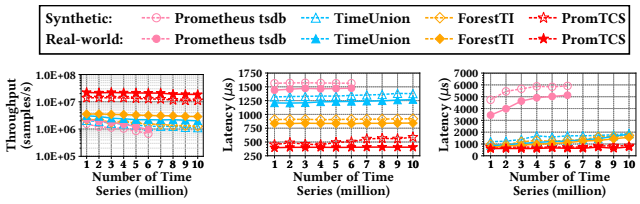
(i) Query net_transmit (j) Memory usage (k) Disk usage (l) Total write size

Figure 11: End-to-end evaluation.



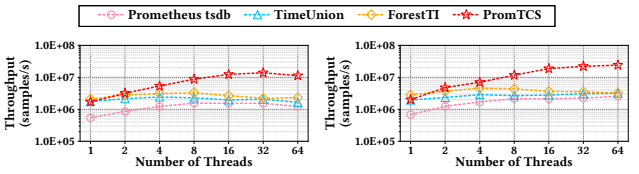
(a) Insertion (b) 5-8-1 (c) 5-1-24 (d) Memory usage (e) Disk usage (f) Total write size

Figure 12: End-to-end evaluation (10 million time series).



(a) Insertion. (b) Query 5-8-1. (c) Query 5-1-24.

Figure 13: Storage Engine Evaluation.



(a) Synthetic dataset (b) Real-world dataset

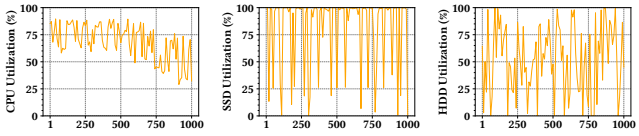
Figure 14: Impact of #worker threads.

series. Due to limited SSD capacity, InfluxDB-SSD and TimescaleDB-SSD cannot store the dataset, while IoTDB and Prometheus tsdb run out of memory (OOM). Hence, we report only PromTCS and the remaining baselines. Fig. 12 shows that PromTCS achieves the best insertion and query performance while maintaining low memory and SSD usage. Specifically, its total SSD writes amount to only 49.8%–58.7% of ForestTI and TimeUnion, thanks to TreeSeries's compaction-free design. PromTCS improves insertion throughput by 2.9×–3.3× compared with TimescaleDB-HDD, the best-performing surviving baseline. For queries, it reduces latency by 10.6%–48.4% for recent-data queries (5-8-1) and by 63.4%–85.3% for long-term queries (5-1-24). These results show that PromTCS sustains high-performance storage at scale with substantially lower deployment cost.

4.3 Storage Engine Evaluation

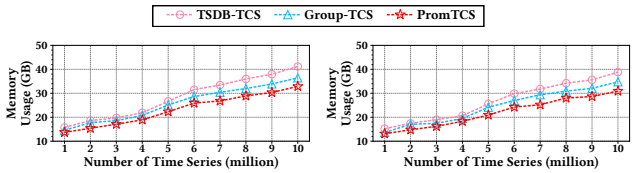
To measure PromTCS's peak performance, we benchmark its storage engine via direct API calls, bypassing HTTP and serialization overhead (all remaining experiments follow this approach). As InfluxDB, IoTDB, and TimescaleDB expose no storage-engine APIs, we exclude them from this comparison.

Performance. As shown in Fig. 13, PromTCS delivers the best performance. At 6 million time series, Prometheus tsdb ran OOM and was killed by the OS. By inheriting ForestTI's efficient memory management and adopting our two-dimensional metadata model, PromTCS scales to 10 million time series. For insertions, PromTCS achieves 4.9–7.8× higher throughput than ForestTI, 5.7–8.9× higher than TimeUnion, and 8.1–20.7× higher than Prometheus tsdb. For recent data queries (*i.e.,* 5-8-1), PromTCS reduces latency by 67.8%–71.3% compared with Prometheus tsdb, 57.3%–67.9% compared with TimeUnion, and 36.1%–52.5% compared with ForestTI. For long-term queries (*i.e.,* 5-1-24), its latency is 82.2%–87.6% lower than Prometheus tsdb, 28.9%–61.1% lower than TimeUnion, and 13.1%–55.3% lower than ForestTI. These gains stem from PromTCS's indexes, WAL, TreeSeries, and data migration (see Sec. 4.2).



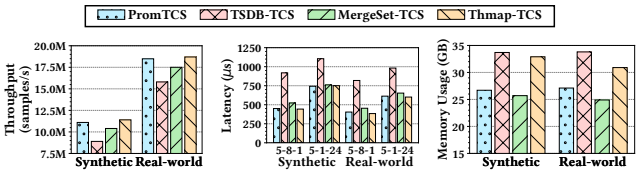
(a) CPU (b) SSD (c) HDD

Figure 15: Resource utilization of PromTCS.



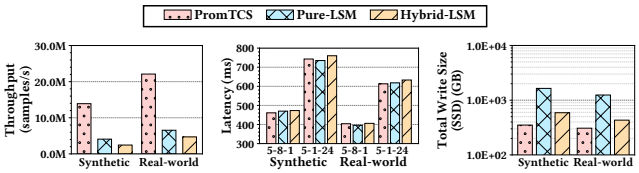
(a) Synthetic dataset (b) Real-world dataset

Figure 16: Memory usage of different metadata models.



(a) Insertion (b) Query (c) Memory usage.

Figure 17: Impact of metadata index design (10M time series).



(a) Insertion (b) Query (c) Total write size

Figure 18: Storage architecture comparison.

Scalability. Fig. 13 shows that PromTCS performs the best as the number of time series increases. In addition, we compare insertion performance changes across TSDBs with varying worker thread counts to evaluate their scalability in concurrent environments. As shown in Fig. 14, increasing worker threads from 1 to 32 boosts PromTCS's insertion throughput by 9.9×. In contrast, the baselines show little improvement beyond 8 threads. This shows that PromTCS scales better and sustains higher insertion throughput with additional threads, enabled by its concurrency-friendly indexes and hybrid storage design.

Hardware utilization analysis. For clarity, Fig. 15 reports only the CPU and disk utilization of PromTCS over a 1,000-second steady-state period on the 10M time-series synthetic dataset. ForestTI, the best-performing baseline, exhibits resource utilization representative of the other baselines, and its utilization is discussed below for comparison.

PromTCS exhibits stable CPU and disk utilization, averaging 68.7% (CPU), 81.2% (SSD), and 53.8% (HDD). This stability comes from two factors. ❶ PromTCS distributes I/O overhead across tiers: SSD handles WAL writes, compaction-free TreeSeries writes, and migration reads, while HDD manages cold data writes and LSM-tree compaction. ❷ Its early-migration strategy proactively flushes data to SSD and HDD, reducing the impact of tier bandwidth differences on insertion throughput. PromTCS's SSD utilization peaks at 100.0%, becoming the bottleneck. ForestTI exhibits lower CPU utilization due to global log contention and compaction-induced write stalls. Its LSM-tree also underutilizes HDD bandwidth: compaction generates bursty writes followed by long idle periods, causing HDD utilization to fluctuate from 60–70% to near zero over the 1,000-second window. Overall, PromTCS utilizes hardware resources more efficiently.

4.4 Metadata Management

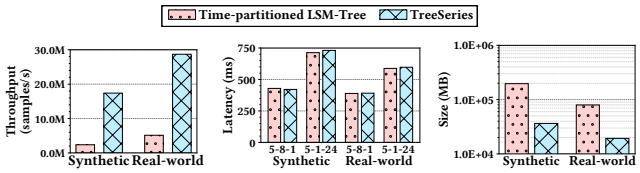
Metadata model. To validate the storage efficiency of our metadata model, we compare PromTCS with two baselines: ❶ *TSDB-TCS*, which adopts the original metadata model of Prometheus tsdb; and ❷ *Group-TCS*, which adopts the group model of TimeUnion. To isolate metadata overhead, index transformations are disabled in all systems (forward index: *Thmap→flattened array*; inverted index: *Thmap→MergeSet*). As shown in Fig. 16, PromTCS consistently uses the least memory, with the advantage increasing as the number of time series grows. At 10 million time series, it uses 25.6% and 11.1% less memory than *TSDB-TCS* and *Group-TCS*, respectively, demonstrating the effectiveness of our metadata model.

Metadata index. To evaluate the metadata indexes, we compare PromTCS with three baselines: ❶ *TSDB-TCS*, which adopts the indexes of Prometheus tsdb (Sec. 2); ❷ *MergeSet-TCS*, which replaces the inverted index with MergeSet; and ❸ *Thmap-TCS*, which replaces it with Thmap.

As shown in Fig. 17, PromTCS achieves insertion throughput comparable to *MergeSet-TCS* and *Thmap-TCS*, and 25.2%–26.7% higher than *TSDB-TCS*, whose metric backward index suffers from severe lock contention. For recent-data queries, PromTCS matches *Thmap-TCS*, outperforming *MergeSet-TCS* by 11.1%–13.9% and *TSDB-TCS* by 49.1%–51.1%. This is because the coarse-grained inverted-index locks in *TSDB-TCS* limit query parallelism, while *MergeSet-TCS* incurs slower inverted-index lookups. For long-term queries, PromTCS remains comparable to *MergeSet-TCS* and *Thmap-TCS*, while reducing latency by 32.8%–37.6% over *TSDB-TCS*. By keeping hot inverted indexes in Thmap and migrating cold ones to MergeSet, PromTCS achieves memory usage comparable to *MergeSet-TCS* and 12.3%–20.8% lower than *Thmap-TCS* and *TSDB-TCS*.

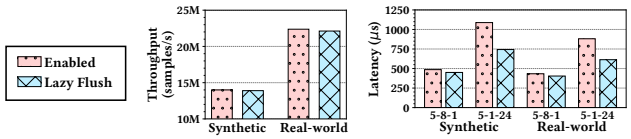
4.5 Storage architecture

Hybrid storage design. We compare PromTCS with two variants: ❶ *Pure-LSM*, which disables TreeSeries and runs the LSM-tree entirely on NVMe SSD; ❷ *Hybrid-LSM*, which also disables TreeSeries and adopts a tiered LSM-tree (upper levels on SSD, lower levels on HDD). Fig. 18 reports the results at the 3-million time-series scale (10 days). PromTCS achieves comparable query performance with both variants. For insertion throughput, PromTCS outperforms Pure-LSM and Hybrid-LSM by 3.37–3.40× and 4.69–5.67×, respectively. For SSD write volume, PromTCS writes only 21.4%–24.8% of Pure-LSM and 59.5%–71.3% of Hybrid-LSM. This



(a) Insertion (b) Query (c) Total data written

Figure 19: Comparison of TreeSeries and LSM-tree.



(a) Insertion (b) Query

Figure 20: Impact of lazy flushing on query.

is because PromTCS employs TreeSeries on SSD to avoid costly compaction and defers cold-data compaction to the HDD tier, thereby preventing compaction from competing for SSD bandwidth with data insertion and WAL. Write stalls on SSD directly degrade insertion throughput, whereas write stalls on HDD only affect data migration, and their impact on insertion throughput is mitigated by our early migration strategy. In contrast, compaction on SSD in both Pure-LSM and Hybrid-LSM causes write stalls and WA.

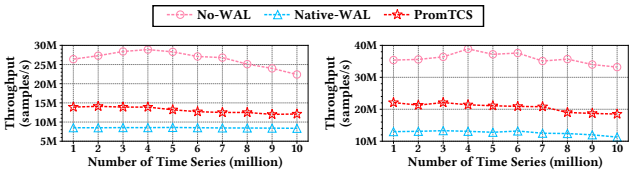
TreeSeries vs. LSM-tree We compare TreeSeries with a time-partitioned LSM-tree on NVMe SSDs. Fig. 19 shows that TreeSeries achieves a 6.3-8.2× higher insertion throughput than the LSM-tree, with comparable query latency and reduces WA by 75.8%-81.5%. This improvement mainly stems from two key factors: ❶ TreeSeries employs MetaHash to maintain relaxed ordering, avoiding expensive compaction. ❷ TreeSeries fully utilizes NVMe SSD I/O parallelism.

Lazy flushing To evaluate lazy flushing, we conduct an ablation study by disabling it (denoted as *Disabled*). As shown in Fig. 20, lazy flushing reduces query latency by 7.2%–10.9% for recent data queries and 27.4%–31.7% for long-term queries. This is because PromTCS initiates data migration once the lower migration threshold is reached, allowing recent data to remain on higher-bandwidth storage. In contrast, *Disabled* retrieves more data from lower-bandwidth storage, particularly for long-term queries.

Meta-driven Parallel WAL To evaluate PromTCS's WAL, we set up two PromTCS's variants: ❶ *Native-WAL*. It employs the same WAL scheme as Prometheus tsdb [11], where all worker threads share a global WAL. ❷ *No-WAL*. It disables WAL. As shown in Fig. 21, the insertion throughput of PromTCS is 42.7% - 69.7% higher than Native-WAL, and 37.6% - 53.4% lower than No-WAL. These results indicate that meta-driven parallel WAL can effectively alleviate the bottleneck caused by global WAL.

5 Related Work

Metadata management in TSDBs. To mitigate the metadata explosion caused by high series cardinality, prior work has



(a) Synthetic dataset (b) Real-world dataset

Figure 21: Impact of WAL designs on insert.

focused on sharing labels and optimizing index structures [35, 40]. Heracles [39] and TimeUnion [37] propose a group model that partitions time series into groups and reduces redundant label storage by sharing common labels within each group. ForestTI [38] introduces a scalable inverted index to control the memory overhead of indexing. Goku [34] reduces index size by deduplicating strings in indexes. Inspired by prior work, we propose a two-dimensional metadata model that further reduces redundant labels and design high-performance, scalable indexes on top of it.

Hybrid storage systems. Hybrid storage leverages heterogeneous devices by keeping hot data on fast, costly media and cold data on slower, cheaper media [33]. Tiered file systems (*e.g.,* NVMFS [32], Strata [22], and Ziggurat [44]) manage data across NVRAM/SSD/HDD to improve I/O efficiency. LSM-tree key–value stores (*e.g.,* HiLSM [28], MatrixKV [41], and SpanDB [21]) place hot SSTables on faster media to accelerate hot-data access. Hybrid-storage TSDBs (*e.g.,* TimeUnion [37] and Goku [34]) tier time-partitioned data across media; for example, Goku stores tier 1–4 (within 80 days) on SSDs. PromTCS targets high performance on low-cost hybrid storage with minimal SSD capacity through *TreeSeries* and optimized WAL and cross-tier data migration.

6 Conclusion

This paper proposes PromTCS, a TSDB that integrates two techniques. (1) two-dimensional metadata management, which combines a two-dimensional metadata model with tailored indexes for storage-efficient, concurrency-friendly metadata management; and (2) a tailored DRAM–SSD–HDD storage architecture with TreeSeries, a bandwidth-aware data migration scheme, and a metadata-driven parallel WAL. Extensive experiments on both synthetic and real-world datasets demonstrated that PromTCS delivers superior performance with more cost-effective deployments.

Acknowledgments

We sincerely thank our shepherd, Rajat Kateja, and the anonymous reviewers for their insightful feedback. This work is supported by the Jing-Jin-Ji Regional Integrated Environmental Improvement-National Science and Technology Major Project of Ministry of Ecology and Environment of China (No. 2025ZD1200600), the National Natural Science Foundation of China (No. 62272369), and the Ant Group Research Intern Program.

References

[1] Apache iotdb. https://iotdb.apache.org/.

[2] Atomic_queue: C++14 lock-free queue. https://github.com/max0x7ba/atomic_queue.

[3] Cortex. https://cortexmetrics.io/.

[4] The difference between ssd and hdd storage devices explained. https://www.techtarget.com/searchstorage/tip/SSD-vs-HDD-Key-differences-and-when-to-use-each.

[5] Docker. https://www.docker.com/.

[6] Influxdb. https://www.influxdata.com/.

[7] Leveldb. https://github.com/google/leveldb.

[8] Node exporter: Exporter for machine metrics. https://github.com/prometheus/node_exporter.

[9] pprof: A tool for visualization and analysis of profiling data. https://github.com/google/pprof.

[10] Prometheus. https://prometheus.io/.

[11] The prometheus time series database layer. https://github.com/prometheus-junkyard/tsdb.

[12] Promtcs: Source code and technical report. https://github.com/ts-lab1024/PromTCS.

[13] Recording rule. https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/.

[14] thmap: Concurrent trie-hash map library. https://github.com/rmind/thmap.

[15] Time series benchmark suite: A tool for comparing and evaluating databases for time series data. https://github.com/timescale/tsbs.

[16] Timescaledb. https://www.timescale.com/.

[17] Truncate. https://man7.org/linux/man-pages/man1/truncate.1.html.

[18] Victoriametrics. https://victoriametrics.com/.

[19] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L Wiener, and Okay Zed. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, August 2013.

[20] Michael P Andersen and David E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies*, FAST '16, pages 39–52. USENIX Association, 2016.

[21] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies*, FAST '21, pages 17–32. USENIX Association, 2021.

[22] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 17–31. USENIX Association, 2014.

[23] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 1–8. ACM, 1984.

[24] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. Trianglekv: Reducing write stalls and write amplification in lsm-tree based kv stores with triangle container in nvm. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4339–4352, December 2022.

[25] Goetz Graefe. A survey of b-tree logging and recovery techniques. *ACM Transactions on Database Systems*, 37(1), March 2012.

[26] Tao Kong, Hui Li, Yuxuan Zhao, Liping Li, Xiyue Gao, Qilong Wu, and Jiangtao Cui. Stscache: An efficient semantic caching scheme for time-series data workloads based on hybrid storage. *Proceedings of the VLDB Endowment*, 18(9):2964–2977, September 2025.

[27] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering*, ICDE '18, pages 185–196. IEEE, 2018.

[28] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. Hilsm: an lsm-based key-value store for hybrid nvm-ssd storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, page 208–216. ACM, 2020.

[29] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In

Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, page 183–196. ACM, 2012.

[30] Lam Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. Moving on from group commit: Autonomous commit enables high throughput and low latency on nvme ssds. *Proceedings of the ACM on Management of Data*, 3(3), June 2025.

[31] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, August 2015.

[32] Sheng Qiu and A. L. Narasimha Reddy. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies*, MSST '13, pages 1–5. IEEE, 2013.

[33] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. Efficient compactions between storage tiers with prismdb. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, page 179–193. Association for Computing Machinery, 2023.

[34] Monil Mukesh Sanghavi, Ming-May Hu, Zhenxiao Luo, Xiao Li, and Kapil Bajaj. Goku: A schemaless time series database for large scale monitoring at pinterest. *Proceedings of the VLDB Endowment*, 18(2):503–515, October 2024.

[35] Xuanhua Shi, Zezhao Feng, Kaixi Li, Yongluan Zhou, Hai Jin, Yan Jiang, Bingsheng He, Zhijun Ling, and Xin Li. Byteseries: an in-memory time series database for large-scale monitoring systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 60–73. ACM, 2020.

[36] Alexander Visheratin, Alexey Struckov, Semen Yufa, Alexey Muratov, Denis Nasonov, Nikolay Butakov, Yury Kuznetsov, and Michael May. Peregreen - modular database for efficient storage of historical time series in cloud environments. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 589–601. USENIX Association, 2020.

[37] Zhiqi Wang and Zili Shao. 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, page 1418–1432. ACM, 2022.

[38] Zhiqi Wang and Zili Shao. Forestti: A scalable inverted-index-oriented timeseries management system with flexible memory efficiency. *Proceedings of the ACM on Management of Data*, 1(2), June 2023.

[39] Zhiqi Wang, Jin Xue, and Zili Shao. Heracles: an efficient storage model and data flushing for performance monitoring timeseries. *Proceedings of the VLDB Endowment*, 14(6):1080–1092, February 2021.

[40] Jin Xue, Zhiqi Wang, Tianyu Wang, and Zili Shao. Tagtree: Global tagging index with efficient querying for time series databases. In *2022 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '22, pages 1283–1293. IEEE, 2022.

[41] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin 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, pages 17–31. USENIX Association, 2020.

[42] Geoffrey X Yu, Markos Markakis, Andreas Kipf, Per Ake Larson, Umar Farooq Minhas, and Tim Kraska. Treeline: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment*, 16(1):99–112, September 2022.

[43] Kai Zhang, Tianyu Wang, and Zili Shao. An efficient cloud storage model with compacted metadata management for performance monitoring timeseries systems. In *24th USENIX Conference on File and Storage Technologies*, FAST '26, pages 167–181. USENIX Association, 2026.

[44] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies*, FAST '19, pages 207–219. USENIX Association, 2019.

A Artifact Appendix

Abstract

The PromTCS artifact is the implementation of the time series database for performance monitoring presented in this paper. It is designed around a two-dimensional metadata model and a hybrid storage architecture to reduce storage cost while maintaining high performance.

Scope

The artifact is a research-driven prototype that can be used to validate the concepts, designs, and evaluation results of PromTCS presented in the paper.

Contents

The artifact contains:

- *the source code of PromTCS;*
- *a Docker image containing all software dependencies;*
- *a README file providing instructions for obtaining and using the source code and container image;*

Hosting

The artifact is publicly available at the following locations:

- *GitHub repository:* `https://github.com/ts-lab1024/PromTCS`
- *Zenodo archival snapshot:* `https://zenodo.org/records/21207044`

The submitted artifact corresponds to the `main` *branch at commit* `8a650fe`*. The Zenodo archive is an immutable snapshot of this version for artifact evaluation.*