Lindorm TSDB: A Cloud-native Time-series Database for Large-scale Monitoring Systems

Chunhui Shen[‡§*], Qianyu Ouyang[‡†*], Feibo Li, Zhipeng Liu, Longcheng Zhu, Yujie Zou, Qing Su, Tianhuan Yu, Yi Yi, Jianhong Hu, Cen Zheng, Bo Wen, Hanbang Zheng, Lunfan Xu, Sicheng Pan, Bin Wu, Xiao He, Ye Li, Jian Tan, Sheng Wang, Dan Pei[†], Wei Zhang, Feifei Li

Alibaba Group[‡] Zhejiang University[§] Tsinghua University[†]

{tianwu.sch,ouyangqianyu.oyqy,lizi,qingzhi.lzp,longcheng.zlc,yunxing.zyj,suqing.sq}@alibaba-inc.com
{yutianhuan.yth,claude.yy,jianhong.hjh,mingyan.zc,wenbo.wb,zhenghanbang.zhb,xulunfan.xlf}@alibaba-inc.com
{zhikuan.psc,binwu.wb,xiao.hx,liye.li,j.tan,sh.wang}@alibaba-inc.com
peidan@tsinghua.edu.cn,{zwei,lifeifei}@alibaba-inc.com

ABSTRACT

Internet services supported by large-scale distributed systems have become essential for our daily life. To ensure the stability and high quality of services, diverse metric data are constantly collected and managed in a time-series database to monitor the service status. However, when the number of metrics becomes massive, existing time-series databases are inefficient in handling high-rate data ingestion and queries hitting multiple metrics. Besides, they all lack the support of machine learning functions, which are crucial for sophisticated analysis of large-scale time series. In this paper, we present Lindorm TSDB, a distributed time-series database designed for handling monitoring metrics at scale. It sustains high write throughput and low query latency with massive active metrics. It also allows users to analyze data with anomaly detection and time series forecasting algorithms directly through SQL. Furthermore, Lindorm TSDB retains stable performance even during node scaling. We evaluate Lindorm TSDB under different data scales, and the results show that it outperforms two popular open-source time-series databases on both writing and query, while executing time-series machine learning tasks efficiently.

PVLDB Reference Format:
Chunhui Shen, Qianyu Ouyang, Feibo Li, Zhipeng Liu, Longcheng Zhu, Yujie Zou, Qing Su, Tianhuan Yu, Yi Yi, Jianhong Hu, Cen Zheng, Bo Wen, Hanbang Zheng, Lunfan Xu, Sicheng Pan, Bin Wu, Xiao He, Ye Li, Jian Tan, Sheng Wang, Dan Pei, Wei Zhang, Feifei Li. Lindorm TSDB: A Cloud-native Time-series Database for Large-scale Monitoring Systems. PVLDB, 16(12): 3715 - 3727, 2023.
doi:10.14778/3611540.3611559

1 INTRODUCTION

Nowadays, a large-scale service is usually built atop tens of thousands of micro-service applications and physical machines, such that sustaining the reliability of the service becomes extremely challenging. To address this issue, monitoring systems play an indispensable role, which constantly collect massive and diverse metric data to monitor the status of the entire service. They provide real-time analysis on the metric data to identify performance issues (*e.g.*, via diagnosis and alerting) or to prevent such issues by triggering actions in advance (*e.g.*, resource scale-up).

The metric data handled by monitoring systems is inherently a type of time-series data, where a metric (*e.g.*, a machine's CPU usage) is modeled as a *timeseries*. A timeseries consists of a sequence of data points collected over time, and each data point contains a timestamp and a field value. Each timeseries is attached a set of tags, which collectively describe different attributes of a metric. For example, a CPU usage metric usually contains three tags — datacenter, region, hostname. In a large-scale service, massive timeseries are generated from a variety of data sources: performance indicator metrics (*e.g.*, CPU, memory and network usage) are from each host or container; applications-oriented metrics (*e.g.*, request rate and response time) are from each micro-service. Typically, an e-commerce service contains billions of timeseries and generates hundreds of millions of data points every second. At such a scale, it is extremely challenging both to write the data points into the monitoring system and to analyze them in real time.

To help characterize the traits of metric data and understand the difficulty of handling them, we study two application performance monitoring systems from our real-world businesses. Table 1 lists the timeseries workload statistics from these systems, namely Sys-A and Sys-B. Both systems collect metric data at very high rates, 200M and 150M points per second, respectively. Meanwhile, tag cardinalities are large in both systems, while some tags have

Table 1: Performance indicator monitoring workloads in two real-world monitoring systems

	Sys-A	Sys-B
No. of tags per timeseries	14	13
No. of total timeseries	$\geq$ 1 billion	0.5 billion
No. of daily active timeseries	0.6 billion	0.4 billion
data point sampling interval	15s $\sim$ 2h	15s $\sim$ 2h
No. of data points per second	200 million	150 million
time range in a query	80% in [1h, 6h]	80% in [1h, 3h]
No. of timeseries a query hits	0.1 $\sim$ 6 million	1 $\sim$ 20 thousand

*These authors contributed equally to this work.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611559

thousands of distinct values. The combination of the tags (*e.g.*, more than ten tags per timeseries in Table 1) results in billion-scale timeseries. More than 60% of timeseries are daily active (*i.e.*, have newly arrived data points). Consequently, it requires high data ingestion capacity of underlying monitoring system, especially when active timeseries are massive. In addition to data ingestion, the large scale of timeseries also complicates the query processing. For example, in Sys-B, a single query hits more than a thousand timeseries, whose data points are retrieved for aggregative analysis. Even worse, this number reaches a million in Sys-A.

In practice, time-series database (TSDB) is used as the backbone of above monitoring systems to manage metric data and support queries [9]. However, we observe that it is highly inefficient for existing TSDBs to handle data ingestion and queries over massive timeseries. Besides, they all lack the support of machine learning (ML) functions, requiring prohibitive efforts to implement complex ML algorithms on time series (*e.g.*, anomaly detection) and maintain corresponding services externally. In a nutshell, existing TSDBs are unable to fully meet the needs of monitoring systems in large-scale Internet services, facing four major challenges as follows:

C1: Low write throughput for massive timeseries. When writing data points into a TSDB, the set of tags for the target timeseries is given to the database as well. A common approach to dealing with these tags is to create a *forward index*, whose index entry maps the tag set to a timeseries id, *i.e.*, a unique identifier internally used by the TSDB to distinguish timeseries. Since each index entry contains many tags (*e.g.*, over ten in Table 1), the footprint of the forward index will easily be overwhelming when a large number of timeseries are managed. This causes a high-cardinality problem, which makes TSDB unable to accommodate the entire index in memory due to cost constraints, leading to low write throughput from memory swapping during index lookups. Existing TSDBs, such as InfluxDB [18] and TimeUnion [40], use conventional cache mechanisms (*e.g.*, Block Cache, MMap) to accelerate on-disk index accesses. However, these mechanisms do not exploit the traits of time series, hence still achieve unsatisfactory efficiency.

C2: High latency for queries that hit massive timeseries. A TSDB usually processes a query in two steps: first, given tags and time ranges, qualified data points from target timeseries are retrieved from the storage; second, computations are performed on these data points. In a large monitoring system, the first step usually hits a huge number of timeseries (*e.g.*, reaches a million in Table 1). We notice that hit timeseries are usually further grouped by a certain tag for subsequent computation. However, existing TSDBs can not efficiently obtain tags of the hit timeseries from a large number of index entries. For the second step, the computational frameworks in existing TSDBs are not well parallelized. For example, TimescaleDB [23] cannot process data points in different partitions in parallel when asked to group data by a non-primary tag.

C3: Lack of advanced time series analysis capability. In a real-world service, its workload may vary dynamically over time. Hence, for the underlying monitoring system, rule-based analysis on metric data usually fails to recognize performance issues precisely. As a solution, practitioners have turned to machine learning algorithms for time series analysis in order to improve the precision of detecting and localizing performance issues. However, existing TSDBs haven't fully integrated ML-based time series analysis. Consequently, users have to employ an external AI platform to handle tasks such as algorithm development, model training and inference. This not only complicates the overall architecture, but also introduces additional latency and data synchronization problems. Although some databases have supported ML-based data analysis [16, 28, 30], they do not optimize the execution process of ML algorithms for time series data, leading to poor performance.

C4: Inefficient adaptability to scale time series management. The number of timeseries in the monitoring system is continuously increasing as the business grows, where the quantities of both micro-services and machines expand along with more fine-grained metrics being monitored [35]. The underlying TSDB is required to continuously scale up to cope with such demands. However, existing TSDBs usually need to redistribute data when scaling out a new node, which is prohibitive on the consumption of both resources and time. One major reason behind this is that the compute and storage resources are tightly coupled. Currently, distributed TSDBs [4, 37] often have a shared-nothing architecture, where each node excursively manages its own memory and disk space. When adding nodes to the cluster, they all suffer from high I/O pressure due to massive data migration. Although some TSDBs [15, 40] deploy a shared storage, this shared storage acts more as a cold storage layer to reduce storage costs, rather than improve scaling efficiency.

To address above challenges, we present Lindorm TSDB, a distributed time-series database that is designed as a powerful backbone for large-scale monitoring systems with massive monitoring metrics. It sustains high write throughput when massive active timeseries exist. It also supports fast queries and ML-based analysis over massive timeseries. In addition, Lindorm TSDB is able to retain stable performance even when it encounters node failures or scaling. Our major contributions are summarized as follows:

- We design Lindorm TSDB, a distributed TSDB combining shared-nothing architecture and shared storage. It contains a cluster of compute nodes and a reliable shared storage, which are logically separated from each other. It partitions data into shards according to their time and tags, facilitating parallel data query and write. In a single shard, the optimized index structure and cache strategy further improves performance. (Target challenges C1/C2/C4; detailed in Section 4.)
- We design an efficient pipelined execution engine for Lindorm TSDB to support common and important types of queries on time series data. The execution engine not only parallelizes the computation into different shards, but also optimizes the computation across multiple timeseries within one shard. On top of that, users can directly use SQL to perform a variety of queries. (Target challenge C2; detailed in Section 4.4.)
- We design Lindorm ML, an integrated machine learning component inside Lindorm TSDB. It enables users to analyze data with anomaly detection and time series forecasting algorithms through SQL, eliminating the effort of operating data and models externally. More importantly, it takes advantage of Lindorm TSDB's data processing capability to achieve higher performance. (Target challenge C3; detailed in Section 5.)

Table 2: Example of Lindorm TSDB's data model

Tags				Fields	
hostname	region	datacenter	timestamp	cpu_user	cpu_sys
host-a	ap-1	ap-1a	1670398200	10	4
host-b	ap-1	ap-1a	1670398200	20	11
host-a	ap-1	ap-1a	1670398210	11	5
host-b	ap-1	ap-1a	1670398210	21	12

- We conduct extensive experiments on a popular benchmark to verify the effectiveness of Lindorm TSDB and its major components. We compare it with two widely-used open-source TSDBs, InfluxDB and TimescaleDB. The results show that Lindorm TSDB is able to achieve higher write throughput as well as lower query latency compared to these baselines. (Detailed in Section 6.)

2 PRELIMINARIES

2.1 Data Model

Lindorm TSDB models metric data as time-series data in schematized tables. We make the data model consistent with the relational data model so that users can easily understand it and fit it into existing systems. There are three types of columns in each table: *tags*, *fields* and timestamp, as illustrated in Table 2. *Tags* describe different attributes of the data source that generates the metric data. A tag is a key-value pair (*e.g.*, ⟨hostname, host-a⟩). At each timestamp (*e.g.*, 1670398200), a data source produces various types of metric data (*e.g.*, `cpu_user` and `cpu_sys`), and we refer them as *fields*. A *timeseries* is uniquely identified by one field and all associated tags, *i.e.*, `cpu_user` and `cpu_sys` above are two timeseries. A timeseries contains a sequence of data points from the same field, where each data point is a pair of ⟨timestamp, field value⟩. For example, in Table 2, the `cpu_user` values, timestamps and tags from the first and third row form a timeseries. Here `cpu_user` is the field, [⟨hostname, host-a⟩, ⟨region, ap-1⟩, ⟨datacenter, ap-1a⟩] is the tag list, and data points contain ⟨1670398200, 10⟩ and ⟨1670398210, 11⟩. When writing data to Lindorm TSDB, the field, tags and the target table name are required. If the combination of the given field and tags is not present in the table, Lindorm TSDB creates a new timeseries.

2.2 Query Patterns

When querying Lindorm TSDB, filtering conditions that consist of target fields, tag selectors and a timerange should be provided:

```
SELECT max(cpu_user), sum(cpu_sys)
WHERE hostname='host-a' AND timestamp >= '2023-1-1 12:00'
```

All timeseries that match the tag selectors will be selected, and the data points in specified time ranges are retrieved for subsequent computations.

In monitoring systems, the vast majority of queries can be divided into three categories according to their computation patterns: latest-value query, downsampling query, and inter-timeseries aggregate query. Figure 1 depicts how these three types of queries are performed, where four timeseries are hit by these queries. For brevity, the figure shows only a subset of results in downsampling query and inter-timeseries aggregate query. A latest-value query refers to retrieving the data point with the latest timestamp for

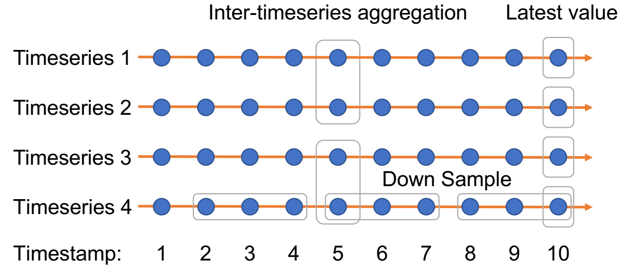


Figure 1: Time series query type

each timeseries, which is important for real-time status monitoring of systems. A downsampling query groups the data points by a given time window in each timeseries, *e.g.*, every three data points in Figure 1, and then the aggregated value such as sum and average for each window is returned. An inter-timeseries aggregate query groups and aggregates data points in all hit timeseries by specified columns, *e.g.*, hostname and timestamp in Table 2, which is similar to the "group by" operation in relational databases.

In practice, downsampling queries and inter-timeseries aggregate queries are often used in combination. Take Table 2 as an example, we may be interested in querying the averages of `cpu_user` in each *region* for every *10 minutes* within the last *24 hours*.

3 LINDORM TSDB OVERVIEW

Recall that Lindorm TSDB is designed to address the four challenges discussed in Section 1. Figure 2 shows Lindorm TSDB's overall architecture, which contains four major components, *i.e.*, TSProxy, TSCore, Lindorm ML and Lindorm DFS. Among them, both TSProxy and TSCore can be scaled horizontally.

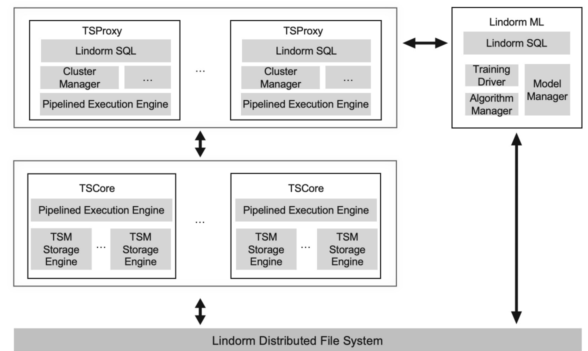


Figure 2: Lindorm TSDB Overview

Lindorm TSDB partitions data on TSProxy into different shards according to two dimensions: timeseries identifier and time. Each shard can be viewed as an independent storage engine exclusively managed by a single TSCore. A TSCore manages multiple shards and is responsible for executing data ingestion and query requests on these shards. Data that belongs to the same timeseries within a period is stored on the same shard, which facilities query push-down optimization (Section 4.4). In a shard, data and corresponding

indexes (*e.g.*, Table 3) are first maintained in the memory of the TSCore they belong to, and later persisted to the shared Lindorm DFS for storage. Lindorm DFS is a distributed file system that provides an HDFS-compatible interface. It leverages Alibaba Cloud's storage infrastructure, *i.e.*, ESSD [12] cloud disk and Object Storage Service [13]. This overall architecture (Section 4.1) combines both shared-nothing and shared-storage designs. It can therefore sustain both horizontal scalability and the ability for each TSCore to access any data, ensuring elasticity and high availability at the same time. In addition, multidimensional sharding strategy (on timeseries identifier and time) avoids data migration when shards change dynamically, which effectively mitigates system performance degradation during node scaling (addressing Challenge C4).

In a shard, indexes need to be updated whenever a new timeseries is created. For fast lookups and maintenance, keeping all indexes in memory is an ideal choice. However, when the number of timeseries becomes massive, indexes consume a huge space, which is known as the high cardinality problem that makes memory bloat. To solve this problem (addressing Challenge C1), Lindorm TSDB uses a structure similar to Log Structured Merge tree (LSM-tree) to periodically flush in-memory indexes into the shared storage and merge them later. With this hybrid storage scheme, we query an index item by first looking it up in the memory. If it misses in memory, we then access the shared storage. Since the access to shared storage is significantly slower, we apply a tailored cache policy for speeding up (Section 4.3). Moreover, considering many historical timeseries are inactive, we use a time partitioning mechanism to boost the memory utilization (Section 4.3).

To allow users to easily query timeseries, Lindorm TSDB supports SQL syntax. As introduced in Section 2.2, a single SQL statement often involves multiple timeseries and conducts aggregate operations in two dimensions, *i.e.*, by time and by tags. Since data points from the same timeseries resides on the same shard and different timeseries resides on different shards, we propose a pipelined execution engine (Section 4.4) that supports computation push-down (addressing Challenge C2). This pipelined execution engine pushes down the query to all shards where hit timeseries are located, and completes the scanning of multiple timeseries in parallel. It then aggregates values back from the shard to TSCore, and assembles the partial results from each TSCore as the final results. During this process, once aggregated values are computed, we can skip loading and transferring massive original data points, saving considerable memory and network resources. To further speed up the aggreation within one timeseries, we employ a pre-downsampling mechanism (Section 4.4) that reduces retrievals and computations on original timeseries data.

Apart from above designs, we also propose Lindorm ML (Section 5), which integrates machine learning algorithms (*e.g.*, anomaly detection, time series forecasting) inside Lindorm TSDB. Lindorm ML combines the data governance capabilities of a database and the data analysis capabilities of machine learning algorithms. It allows users to directly train machine learning models inside the database via SQL, and using these models to make online inferences. All data and model computations are in the database in both phases. In addition, we utilize Lindorm TSDB's features such as timeseries layout and query push-down to achieve batched, distributed parallel and near-data training and inference optimizations, thus enabling efficient time series data analysis (addressing Challenge C3).

4 SYSTEM DESIGN

4.1 Distributed Architecture

Lindorm TSDB exploits a distributed architecture that combines the best of shared-nothing and shared-storage. In particular, shared-nothing architecture makes the database horizontally scalable, while the cloud-native shared storage gives elasticity and high availability to the database. The time series data is physically stored in the reliable shared storage. When scaling the TSDB (*e.g.*, adding or removing a node), the downtime can be minimized since no data migration is required, improving to the quality of service.

Our logical sharding strategy shards time series data according to two dimensions: time and timeseries identifier (the identifier is uniquely determined by a set of tags and one field). For a data point, we first determine the *shard group* assignment based on its timestamp. A shard group contains multiple shards that all manage data points from the same time range ($t0$, $t1$]. Data points are then routed to shards in the group based on their identifiers' hash values. When the number of database nodes changes (*e.g.*, scales out), the number of shards needs to change as well, *i.e.*, a new shard group will be automatically generated. This design avoids the massive data migration from data redistribution. As shown in Figure 3, when the number of shards increases at time T, a new shard group is created to manage all data generated after T, while all previous shard groups remain unchanged. In this way, the historical data points are still in their original shards, so that they do not need migration. We observe that monitoring systems rarely query and write historical data, and it is worthy to not change distribution of historical data for a stable system performance.

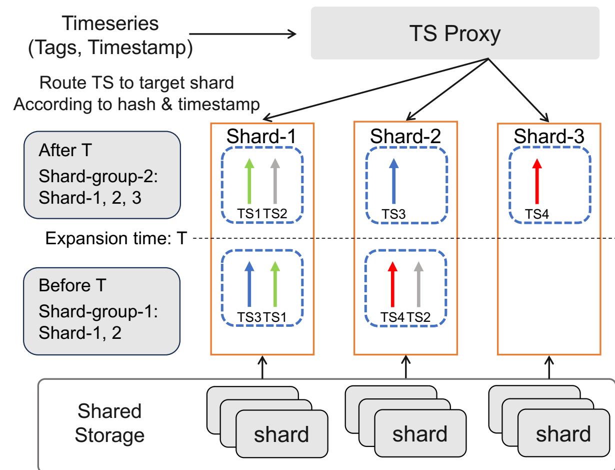


Figure 3: Lindrom TSDB sharding (arrows with different colors mean different timeseries)

Next, we discuss how Lindorm TSDB organizes data physically. The newly ingested data to a shard is first written to the Write Ahead Log (WAL) on the shared storage, and then to the memory on TSCore. Periodically, the data in memory is flushed to the shared

storage. The mapping relations between shards and TSCores are stored in Apache ZooKeeper [2] as metadata. In this manner, compute and storage can be separated. TSCores only need to read from and write to the shared storage, and each node is able to access all timeseries as well as the metadata. If one TSCore fails, other active nodes can instantly take over its requests. In this case, the metadata needs to be updated, and then the unflushed data in the failed node's memory is restored on the active node using WAL.

With the help of the distributed architecture above, both query and write requests can be executed in parallel on multiple TSCores, bringing high efficiency. When ingesting the data, each data point is routed to the corresponding TSCore, and then be written to the shard. In a query, we first determine whether this query can be routed to certain shards according to the query conditions, *e.g.*, the query carries a primary key or a complete tag set. Otherwise, the query is broadcast to all TSCores, each of which executes the query on all shards managed by it.

4.2 TSM Storage Engine

Lindorm TSDB employs an LSM-Tree-like (Log Structured Merge Tree) storage optimized for time series data, which we call TSM (Time Structured Merge Tree) [19], as shown in Figure 4. By taking the characteristics of time series data into account, TSM optimizes the WAL writing, memory organization, compression algorithm, and compaction policy over standard LSM.

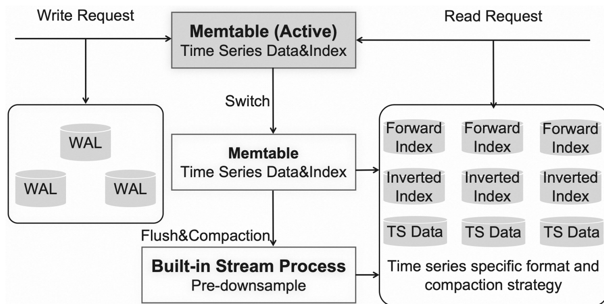


Figure 4: Lindrom TSDB TSM storage engine

Similar to most LSM-based storages, data in TSM is first written to the append-only WAL to ensure durability and high write throughput. Then, it is written to the Memtable in memory, and ready to subsequent accesses. When the Memtable accumulates to a certain threshold, a flush will be triggered to persist the table into storage according to the policy: a forward index file *FwdIdx file*, an inverted index file *InvIdx file* and a time series data file *TSD file*. All the files (*i.e.*, FwdIdx file, InvIdx file and TSD file) will be periodically compacted to new files in the background. A TSD file contains a batch of data chunks (containing timeseries), and it can quickly locate the timeseries in data chunks according to its timeseries ID. When a query arrives, the set of timeseries IDs that meet the query conditions will first be retrieved from the InvIdx files. TSD files will then be fast filtered out according to the query time range, and target data chunks will be located by qualified timeseries IDs.

Compaction. TSD file compaction happens in the background according to certain policies. We use the level compaction strategy to deal with TSD files of different sizes. The compaction ensures that data belonging to the same timeseries and time period only resides in a single TSD file, which reduces the number of TSD files to be scanned during a query. During compaction, the TSD files and indexes will be dropped if their TTLs (Time-To-Live) are set and have expired. In addition, we mark cold TSD files based on their timestamps so that Lindorm DFS can automatically transfer them to cheaper storage medium in the compaction process.

Time series customized compression. We use dictionary encoding, Delta-of-Delta, XOR, ZigZag, RLE and other compression algorithms to compress timeseries, achieving a up to 15× compression ratio. Recall that in our data model (Section 2.1), a timeseries is identified by a combination of field and tags, and a write request may write multiple timeseries with the same tags. Hence, fields and tags of different timeseries will contain a large amount of redundant information. Note that values from the same timeseries often change smoothly over time, which makes compression effective. We use different compression methods for different data. Lock-free compression is applied to in-memory data to improve memory utilization. WAL logs are compressed by dictionary compression in batch way to reduce I/O and improve throughput. In persistent TSD files, data points from the same timeseries over a continuous period is composed into a data chunk, which is internally compressed using Delta-of-delta, XOR, ZigZag, and RLE.

Table 3: Forward and inverted indexes (key ⇒ value)

forward index	inverted index
hostname=host-a®ion=ap-1 ⇒ 1	hostname=host-a ⇒ 1
hostname=host-b®ion=ap-1 ⇒ 2	hostname=host-b ⇒ 2
1 ⇒ hostname=host-a®ion=ap-1	region=ap-1 ⇒ 1, 2
2 ⇒ hostname=host-b®ion=ap-1	

4.3 Index Optimization

Recall that for fast data ingestion, Lindorm TSDB creates *forward indexes* in each shard to maintain the mapping between tag sets and timeseries IDs. At the same time, in order to speed up the lookup of timeseries at query time, *inverted indexes* are created to maintain the mapping of each tag to the set of timeseries IDs that contain the tag. Table 3 shows forward and inverted indexes that contain two timeseries, where `hostname` and `region` are tag keys.

For the case of monitoring system, massive short-time-span timeseries are created due to the creation or destruction of containers. These timeseries will soon become inactive and lead to index inflation. To resolve this issue, we partition the data in shard according to time. Hence, each time partition has its own indexes, which only manage those timeseries written within a time period. In addition, we observe that recent timeseries are more favored by queries. When there are too many time partitions, we provide a lazy loading mode to only load the latest time partition in high priority, with the historical ones loaded asynchronously. This significantly reduces the service interruption time caused by partition loading process.

When writing a timeseries, we first search its tag set in the in-memory forward index, and then in the disk index files. If the timeseries does not exist, a new timeseries ID with the tag set will

be created in the Memtable's forward index. After that, each tag in the new timeseries is updated in the Memtable's inverted index. When the flush is triggered, Both forward and inverted indexes in the Memtable will be written to the shared storage, generating new FwdIdx files and InvIdx files, respectively.

To speed up the index lookups on disk, we perform a series of optimizations. First, the index files are merged in the background to reduce the total number of files. Second, we add a bloom filter to each file, through which unrelated files can be filtered out quickly. The bloom filters are cached in memory to further speed up the file filtering. Besides, we use a block cache to cache index files in memory to reduce storage accesses.

Compared to inverted indexes, forward indexes are accessed much more frequently. During the write process, Forward indexes are looked up to determine the existence of timeseries. In the inter-timeseries aggregate query, we also need to obtain the tags of the timeseries from forward indexes. As a result, the efficiency of searching forward indexes is crucial. Hence, in addition to block cache, we design an additional layer of cache for the forward index, called *seriescache*. While the block cache is used to cache file data, the seriescache only stores the mapping between timeseries IDs and tags that are accessed recently, consuming less space. The block cache and seriescache both use the LRU policy. In those cases that the tag lengths vary much or are too long, seriescache may occupy a lot of memory. Fortunately, we can optionally use the MD5 values instead of the original tags to reduce the memory footprint. We observe that in real-world monitoring systems, MD5-encoding seriescache can cache up to 5× of items than the original version.

When looking up inverted indexes, we need to conduct intersection operations on the posting lists. For example, when the query conditions are `hostname='host-a'` and `region='ap-1'`, we first find the posting lists corresponding to these two conditions, which are {2} and {1,2}, respectively. Then, we get the intersection of two lists, which is {2} here. We use RoaringBitmap [26] as the data structure for the posting list. Compared with integer type timeseries IDs, bitmap saves much space and supports fast set operations.

4.4 SQL Execution Engine

Lindorm TSDB supports standard SQL syntax, as well as extended syntax for the downsampling query to simplify the usage. It optimizes the execution of data ingestion by using a fast path based on the characteristics of time series data's write pattern. In order to optimize the efficiency of the downsampling query, it adds a pre-downsampling mechanism in the process of data writing, which aggregates the original timeseries in time dimension in advance. Lindorm TSDB also exploits the fact that timeseries are organized in groups, and proposes a pipelined execution engine that computes in a timeseries-wise manner and supports query push down.

Lindorm SQL. Many time series databases are equipped with dedicated query languages to handle time series data, such as InfluxDB's InfluxQL [20] and OpenTSDB's [8] HTTP API. Compared to these highly customized query languages, SQL has the advantage of ease-to-use and a rich ecology. As a standard language for databases, most developers can use SQL proficiently without extra learning efforts. Hence, Lindorm TSDB chooses to fully support SQL with a relational-like data model. It extends the syntax for time series queries while still being compatible with ANSI SQL standard.

As discussed in Section 2.2, in monitoring systems, users are often less interested in individual data points, but more on the aggregated analysis of multiple data points, *e.g.*, the average metrics within a minute. Lindorm TSDB extends the standard SQL based on Apache Calcite [10] with one new syntax *sample by* for the downsampling query:

```
SELECT max(cpu_user) WHERE hostname='host-a'
AND timestamp >= '2023-1-1 12:00' sample by '10min'
```

Write optimization. The time series data ingestion process can be characterized as a bulk repetition of simple `INSERT` SQL statements. We observe that parsing SQL directly using Calcite results in very low write throughput, because the SQL parser and execution plan generator in Calcite consume a lot of CPU cycles. To optimize the performance of above two parts, we design a fast path for write processing, as depicted in Figure 5. The vast majority of write statements are very simple, containing only three elements: tag set, timestamp, and field value. It is very easy to parse them even without the sophisticated parser in Calcite. Therefore, we have implemented a small parser that only handles simple write statements, and it is only responsible for parsing out the time series related information. This parser is invoked first upon Lindorm TSDB receives a SQL statement. If the parsing is successful, the data points are bypassed Calcite and sent directly to the execution engine, otherwise it will continue to go through Calcite as normal. We observe that the write throughput in fast path mode is 15× higher than that in Calcite path mode. In addition, SQL prepare statement can be used for batch write optimization in clients. Our tests have shown that by combining the fast path and prepare statement execution, we can achieve 20× of write throughput improvement.

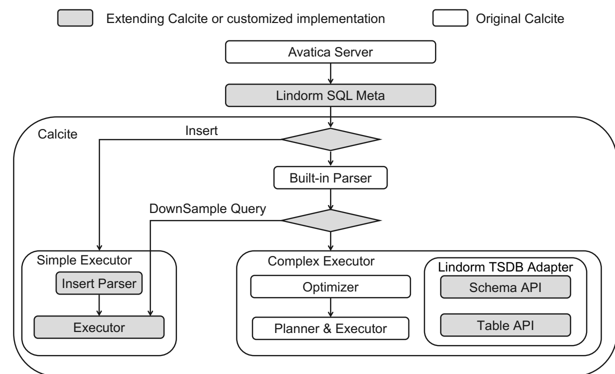


Figure 5: Lindorm TSDB write path optimization

Query optimization. In monitoring systems, the execution process of a typical time-series aggregate query can be divided into three steps:

(1) Find the timeseries that meet the predicates.
(2) Perform a 'sample by' operation on each timeseries to obtain the aggregated values of each timeseries on the time windows.
(3) Perform a 'group by' operation on all the aggregated values.

In order to improve the execution efficiency of the aggregate query, Lindorm TSDB takes advantage of the distributed storage of time series data and optimizes from two aspects: pre-downsampling and pipelined execution engine.

Pre-downsampling. A naive approach for the downsampling query 'sample by t' is to: scan each related data points, divide them into different t-time windows according to timestamp, and then compute the aggregated value for each window. The complexity of this approach is linear to the number of data points. When dealing with high-frequency sampling, it has to scan a considerable number of original data points. To solve this problem, we use a pre-downsampling mechanism when writing data points. Pre-downsampling means that the downsampled values are calculated during writing and then stored in the database. Pre-downsampling allows the aggregated values to be extracted directly without calculation. For example, at write time, the database simultaneously computes the sum of data points every 1, 10, and 60 minutes and stores them. When the user issues a 'sample by $10min$' query, the database can return the result directly without scanning the original data points. If the user performs a 'sample by $30min$' query, which is not within existing sampling rates, the database can also compute the $30min$ aggregated value using three consecutive $10min$ aggregated values. Compared to scanning the raw data, pre-downsampling eliminates the data scanning and computation to a huge extent.

To minimize the impact on write throughput, pre-downsampling is not performed when the data is written to Memtable. It only happens when the Memtable is flushed to shared storage or when TSD files are merged at compaction. Access to the original data is very convenient on these occasions, and the computation can be highly efficient. In addition, the number of pre-downsampled files will be much smaller than the original data files, which further improves the query efficiency. Currently, we support a collection of common operators, *e.g.*, `count`, `first`, `last`, `min`, `max` and `sum`.

Pipelined execution engine. To take advantage of our timeseries optimized storage and to optimize queries in monitoring scenarios (*e.g.*, sample by and group by), we propose a pipelined execution engine below the SQL layer and above the storage layer, which is shown in Figure 6. This engine is designed as an operator pipeline, with the lowermost layer being a timeseries scan operator responsible for finding the specified timeseries from the storage engine, and the uppermost layer implementing an adapter for Calcite to provide a row-iterator interface. These timeseries flow between pipeline operators in the form of multiple rows. At query time, the query statement goes through Calcite's syntax parsing in the SQL layer, bypasses the original Calcite executor (into our customized simple executor), and finally goes through the entire pipelined execution engine driven by the row iterator to read data from the storage engine. In the pipeline of the execution engine, various timeseries operators can be extended and defined. The difference between these operators and those in the SQL layer is that they compute the data in the time series dimension rather than in the row dimension, and can therefore serve as optimizations for batch computation. As the name of the pipelined execution engine suggests, the data is streamed through all the operators in the pipeline and released as soon as it is processed by each operator, avoiding data dwell and reducing memory usage. In addition, we have embedded pipelined execution engine in both TSProxy and TSCore, based on which the query push-down feature is implemented, allowing some computations to be distributed and parallelized among multiple TSCore nodes. In addition, the pipelined execution engine can also process queries in parallel between multiple shards within one TSCore and between multiple timeseries to further improve query performance.

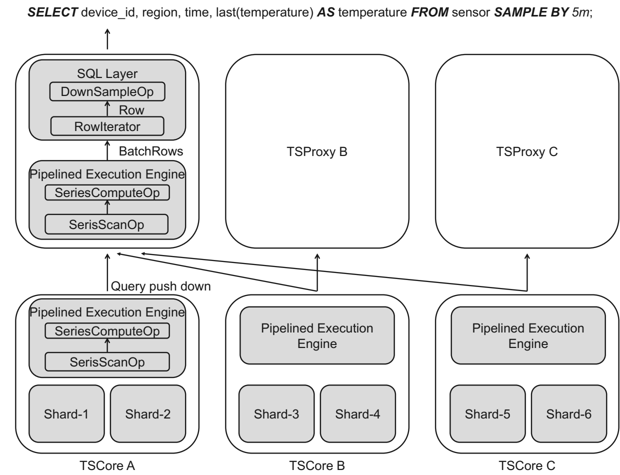


Figure 6: Lindorm TSDB SQL query overview

Figure 7 shows the internals of the pipelined execution engine. As can be seen, the timeseries scan operator, located at the bottom of the pipelined execution engine, takes data input from the lower layers. The data input can either be the network RPC (from TSProxy to TSCore) or the storage IO from the storage engine (due to query push down). The upstream operators of the timeseries scan operator can be divided into two categories according to whether downsampling is required, including the commonly used downsampling type of aggregation (DSAgg) and interpolation (Filling) operators, and non-downsampling type of operators such as obtaining the rate of change (Rate) and obtaining the difference (Delta). In addition, upstream of these operators, other operators that can be used for cross-time series aggregation are implemented to meet the needs of a wide variety of time series processing.

5 LINDORM ML

In this section, we introduce Lindorm ML, a machine learning component integrated into Lindorm TSDB, to enable advanced time series analysis. It leverages SQL syntax extensions to provide Lindorm TSDB with sophisticated algorithms for anomaly detection and forecasting of timeseries. Figure 8 illustrates the simplicity of using Lindorm ML, where users can still interact with Lindorm TSDB through SQL. Firstly, users can train a machine learning model, *e.g.*, an anomaly detector, by specifying an extended CREATE MODEL syntax together with predicates that filter the data from Lindorm TSDB. Then, they can use another extended SQL syntax to perform inference with the trained model.

As an internal service in the database, Lindorm ML accepts the model training request forwarded by the TSDB node and takes on the main control logic to drive the model training process in the database. The inference service is provided directly by the

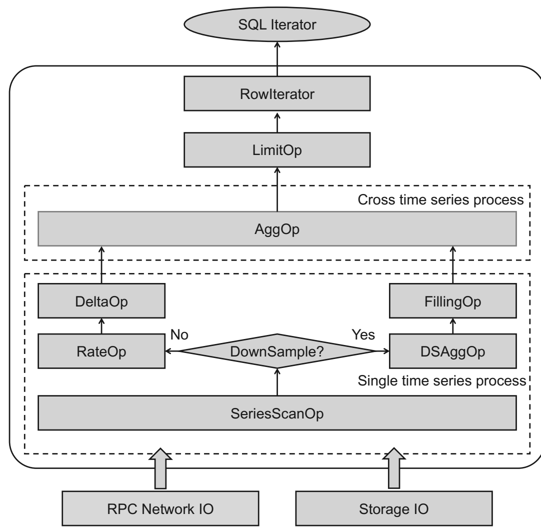


Figure 7: Lindrom TSDB pipelined execution engine

TSDB node, without the participation of Lindorm ML, thus reusing the high availability and scalability of the TSDB services naturally. In addition, the management of model data and metadata is shared between Lindorm ML and TSDB nodes through the underlying distributed file system and ZooKeeper. Lindorm ML utilizes TSDB's distributed storage and query of time series data to propose a model partitioning design and implementation. A user-created logical model actually consists of many physical models, which correspond to different timeseries data. These physical models are divided into model partitions according to the partitioning of the timeseries. This design brings the ability of using the query pushdown technology of TSDB execution engine to enable the model training and inference pushdown, further enabling the distributed parallel, near-data training and inference optimization.

Algorithm support. We support popular statistical and deep learning based time series anomaly detection and forecasting algorithms (*e.g.*, ARIMA[6], DeepAR[33], TFT[27]) provided by open source algorithm packages. Further, we support our in-house online algorithms that support real-time anomaly detection [17]. All these algorithms are uniformly managed by the Lindorm ML plugin on the TSDB node.

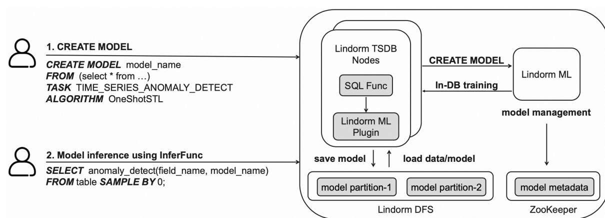


Figure 8: Lindrom ML overview

5.1 In-Database Training

Figure 9 depicts the Lindorm ML training procedure. TSProxy on the TSDB node receives the user's CREATE MODEL command and, after partially decoding the SQL syntax, delivers it to the Lindorm ML node in the same cluster. The SQL query is then fully parsed by Lindorm ML. The CREATE MODEL statement executes in two steps: First, a model management module generates the model's metadata, including the model name, task, algorithm, and so on and persists it in ZooKeeper; then, a train() internal SQL function call is sent back to the TSProxy that performs the model's training process. By design, the training function as the `TrainingOp` operator of the TSDB pipeline can be pushed down to the TSCore nodes for distributed execution. Before entering the training operator `TrainingOp`, the pipelined execution engine processes the data in two steps: the `SeriesScanOp` operator extracts relevant features and the `PreProcessingOp` operator performs the necessary data preprocessing. In Lindorm TSDB, the `SeriesScanOp`, `PreProcessingOp` and `TrainingOp` operator all process each individual timeseries separately, thereby naturally satisfying the requirement of input data for the time series machine learning algorithms (*e.g.*, anomaly detection and forecasting).

The model management module `ModelManager` in the Lindorm ML plugin manages the model partitioning, persists the trained model data to Lindorm DFS, updates the model metadata stored in ZooKeeper (*e.g.*, the training progress, evaluation metrics).

When the training operators are pushed down to multiple TSCore nodes for execution, the physical models trained from the timeseries on one TSCore node naturally forms a partition. The advantage of this way is that it is easy to adapt to the scenarios of adding/deleting and failover of TSDB nodes. If the training operator is not pushed down but executed on TSProxy, the physical models will not be partitioned. In this way, multiple timeseries data are cached in the training operator, thus enabling batch training and improving efficiency. In summary, utilizing the distributed storage of TSDB data and the operator pushed-down technique enables Lindorm ML's batch, distributed parallel and near-data training optimization.

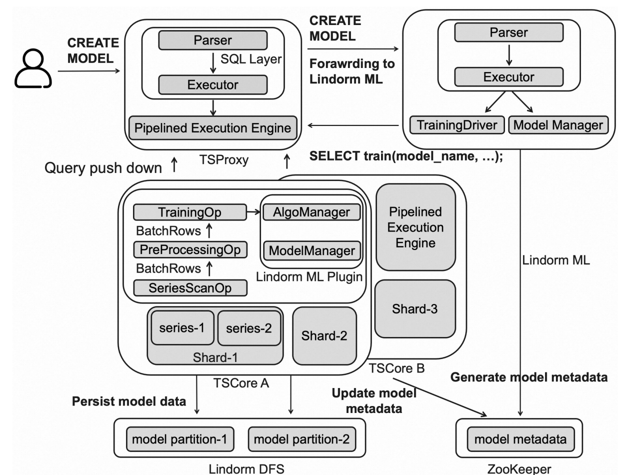


Figure 9: Lindrom ML In-Database Training

5.2 In-Database Inference

Unlike the training process, inference can be done entirely on the TSDB node. The SQL inference function called by the user (*e.g.*, anomaly_detect()) is also a TSDB pipelined execution engine operator that can be pushed down and executed. Similarly, before the inference function `InferenceOp` obtains its input data, the data is first processed by the timeseries scan operator `SeriesScanOp` and the preprocessing operator `PreProcessingOp` at the pipelined execution engine layer. The inference operator `InferenceOp` also calls the Lindorm ML plugin, which finds the corresponding model and algorithm from the model metadata according to a user-specified model name. When loading a model, the model partition corresponding to the input data is found according to the same timeseries routing rules. As with the training process, batch, distributed parallel and near-data inference optimization can be achieved when multiple TSCore nodes are involved in the inference function.

5.3 Model metadata management

The metadata of models is maintained in ZooKeeper, to be consistent with the way that TSDB manages the metadata of tables. We have extended the implementations of the Schema and Table in the SQL layer, so that the model metadata can be queried as if they were tables. We also encapsulate the syntactic sugar "SHOW MODEL(S)" statements to simplify its usage.

6 EVALUATIONS

We evaluate Lindorm TSDB in four aspects. We first compare Lindorm TSDB with two popular open-source TSDBs on write (Section 6.2) and query (Section 6.3) performance. Then, we evaluate the efficiency on time-series machine learning tasks of Lindorm ML (Section 6.4). Finally, we study the contributions of the main components in Lindorm TSDB to the overall performance (Section 6.5).

6.1 Experiment Setup

We conduct experiments on five Alibaba Cloud Elastic Compute Service (ECS) [14] servers, with efficient cloud disk (ESSD) [12] mounted as the disk for each server. We deploy the TSDBs on four servers, each of which has 16 cores and 64GB RAM. The fifth server runs as a client to generate writes and queries, which has 32 cores and 128GB RAM.

Comparison databases. For the end-to-end performance comparison, we choose two representative and open-source TSDBs, InfluxDB and TimescaleDB, as baselines. InfluxDB is a very popular TSDB, ranking first in DB-Engines Ranking [36]. TimescaleDB is an open-source TSDB with both its standalone and distributed version available. Meanwhile, benchmark results show that TimescaleDB has excellent performance [22]. When evaluating the functions of the main components in Lindorm TSDB, we turn off the push-down optimization and the seriescache respectively, and then we study how Lindorm TSDB works. Finally, we also compare the performance of Lindorm TSDB with different number of nodes to verify its horizontal scalability. In summary, there are five TSDBs deployed on the ECS servers:

- **InfluxDB:** single-node InfluxDB.
- **TimescaleDB-1:** single-node TimescaleDB.
- **TimescaleDB-3:** three-node TimescaleDB.
- **Lindorm-1:** single-node Lindorm TSDB.
- **Lindorm-3:** three-node Lindorm TSDB.

Configurations. For InfluxDB, we tune its cache limits to get the best performance. Specifically, we set its cache-max-memory-size to 16g, cache-snapshot-memory-size to 4g, and GOGC to 30. For TimescaleDB, we adjust the configuration for TimescaleDB-3 according to its official guidelines to achieve the best performance. We deploy an additional access node for TimescaleDB-3.

6.2 Writing Performance Evaluation

We evaluate the write throughput, *i.e.*, the number of data points ingested into the database per second, of each database at different timeseries scales. We use the DevOps data generated from Time Series Benchmark Suite, TSBS [3] as the insertion test data. In particular, TSBS generates 101 timeseries for each host to represent different type of system or application metrics, *e.g.*, CPU usage, number of diskio, number of nginx requests, *etc.* Each timeseries contains about 11 tags.

In TSBS, we adjust the number of timeseries generated by changing the number of hosts, *host_scale*. The number of timeseries equals to *host_scale* * 101. To improve the write performance, we set a large write batch for each database, *i.e.*, 10000, and we also set the number of workers to be the number of cores, *i.e.*, 16 for single-node databases and 48 for three-node databases.

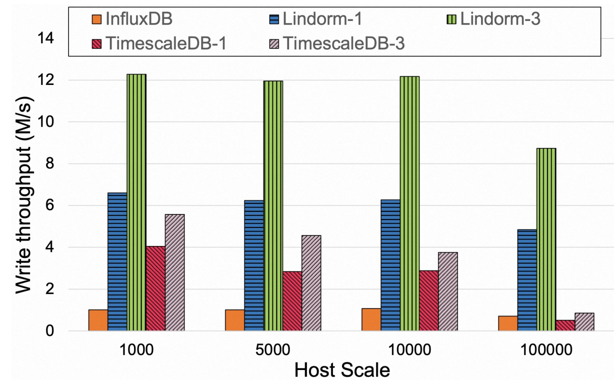


Figure 10: Write throughput at different timeseries scales

Figure 10 shows the write throughput of each TSDB at different timeseries scales, where each timeseries contains 12 hours of data with the data interval as 1 minute. The results show that both single-node and three-node Lindorm TSDB outperform other baselines. At the largest scale (*i.e.*, 100000 hosts and 10M timeseries), three-node Lindorm TSDB has about 10× higher write throughput than other TSDBs. The first reason is that Lindorm TSDB partitions the timeseries according to the tags, allowing multiple timeseries being written in parallel at the same time. Secondly, Lindorm TSDB creates the seriescache for the forward index, which facilitates determining the ID of the timeseries specified by tags in a write command.

As the timeseries scale increases, Lindorm TSDB have much lower performance degradation than TimescaleDB. Because MD5

encoding method for timeseries tags make the seriescache able to cache the tags and IDs of numerous timeseries that have recent data writes. Thus, when host_scale increases to 100000, the number of accesses to the index in disk does not increase much in Lindorm TSDB.

6.3 Query Performance Evaluation

In query evaluation, we adjust the DevOps data generation in TSBS and collect 1 timeseries for each host, where the total number of timeseries equals host_scale. In this way, one query can hit more timeseries at the same host_scale. For each query, we restart the databases, repeat 5 times with different filter conditions, and present the average latency.

Table 4 describes the three query patterns that we have mentioned in Section 2.2. Q1 and Q2 use *region* as the filter tag, *e.g.*, WHERE region=ap-1, and hit $host_scale/9$ timeseries in each query. There is no tag selector in Q3 and thus Q3 queries on all timeseries in the TSDB.

Table 4: Three query patterns

Query	Description
Q1 - Latest value	The last data points of timeseries in 1 region.
Q2 - Downsampling	aggregate on each timeseries in 1 region per 5 minutes for 1 hour.
Q3 - inter-timeseries aggregate	aggregate on all timeseries in each region per 5 minutes for 2 hours.

Table 5: Q1's query latency (ms)

Host Scale	InfluxDB	Lindorm		TimescaleDB	
	1-node	1-node	3-node	1-node	3-node
10000	22	44	53	237	210
100000	145	90	95	1530	1610
1000000	2083	464	284	211689	13452

Table 5 shows the results for Q1, the latest value query. At smaller host scales (10,000 and 100,000), InfluxDB and Lindorm TSDB perform closely. This is because Lindorm TSDB needs to push down the query and collect results from all shards or nodes through RPC. The time consumed by RPC is not negligible when the total latency is low. But at the large scale such as 1M, the query latency of InfluxDB is 4.48× as high as that of single-node Lindorm TSDB and 7.33× as high as that of three-node Lindorm TSDB. Because Lindorm TSDB can push the query down to the storage engine and can scan multiple timeseries parallelly to get their last data points. TimescaleDB is not able to utilize the index on timestamp in the latest value query hitting multiple timeseries [24], resulting in particularly low efficiency.

For the downsampling query whose results are in Table 6, it requires more data points computed than Q1. And for an aggregation query such as Q2 and Q3, the number of returned values is much smaller than the number of data points involved in the query. Therefore, Lindorm TSDB's streaming optimization in the execution engine reduces a lot of memory footprint and data transfer consumption. The larger the timeseries scale, the more significant the advantage of Lindorm TSDB over other TSDBs. Compared with three-node Lindorm TSDB, InfluxDB has 4.5× and 15.3× higher query latencies at the scales of 100000 and 1M hosts respectively. And three-node TimescaleDB has 1.5× and 9.7× higher latencies.

Table 6: Q2's query latency (ms)

Host Scale	InfluxDB	Lindorm		TimescaleDB	
	1-node	1-node	3-node	1-node	3-node
10000	72	89	91	53	67
100000	1046	177	190	502	471
1000000	15261	1165	934	51916	10012

Table 7: Q3's query latency (ms)

Host Scale	InfluxDB	Lindorm		TimescaleDB	
	1-node	1-node	3-node	1-node	3-node
10000	559	175	164	91	427
100000	9437	1390	809	898	4296
1000000	111815	21177	6884	43630	30651

The results of Q3 query are shown in Table 7. In a Q3 query, InfluxDB and Lindorm TSDB need to find *region* values for all hit timeseries in order to group them. Lindorm TSDB has the seriescache to optimize the process of searching tag values in the forward index. In addition, Lindorm TSDB is able to push down the downsampling operator together with the inter-timeseries aggregate operator. This allows the data points to be aggregated by time window and tags in each shard and TsCore node before they are collected by higher level, significantly improving the efficiency. When the query hits 1M timeseries, both single-node and three-node Lindorm TSDB outperform other TSDBs by a large margin. It is worth noting that single-node TimescaleDB outperforms three-node version at small scales. by checking the query execution process in three-node TimescaleDB, we find that the query tasks on partitions are executed serially. It is probably because *region* tag is not set as the partition key, which is *hostname*. We run queries where the data are aggregated by *hostname* and find that computations in partions are parallel, which verifies our hypothesis. When the timeseries scale becomes very large, the memory of single-node TimescaleDB was not enough for such large amount of data, so the performance drops heavily.

6.4 Advanced Time-Series Analysis Evaluation

We evaluate the efficiency of Lindorm ML in performing time-series anomaly detection tasks. We still use the data generation approach in Section 6.3 to prepare data for machine learning tasks. Each timeseries contains two consecutive segments of data for training and inference, both of which are one-day long.

In evaluation, we create training and inference tasks at different timeseries scales (10,000 and 100,000) via SQL provided by Lindorm ML, where we run OneShotSTL [17] as anomaly detection algorithm. Meanwhile, we run the same algorithm **outside Lindorm TSDB** for training and inference as the baseline. Specifically, we first read data from Lindorm TSDB and then apply OneShotSTL to them. We record the time spent in each way respectively.

As shown in Table 8, compared to performing machine learning externally, Lindorm ML consumes about half the time for both

training and inference at different scales. This is because Lindorm ML reduces the time-consuming transmission of the raw data. In addition, various optimizations in the pipelined execution engine also improve the efficiency of machine learning computations.

Table 8: Efficiency of time-series anomaly detection

Host Scale	Training Time(s)		Inference Time (s)	
	Lindorm ML	outside	Lindorm ML	outside
10000	19.69	36.72	19.89	36.37
100000	198.53	431.66	206.11	391.89

6.5 Ablation Study

We study the contributions of the main modules in Lindorm TSDB by evaluating Lindorm TSDBs with different configurations:

(1) Turn off the push-down optimization in the pipeline streaming execution engine.
(2) Turn off the seriescache for the forward index.

In addition to experiments on the above configurations, we also evaluate the write throughput in three cases to investigate the adaptability and scalability: node scaling event, node failure event, and deployment with different cluster size. We use the data generation method in Section 6.3, *i.e.*, one timeseries on each host.

Table 9: Ablation study on the push-down optimization

Host Scale	Q3 query latency (ms)	
	with push-down	w/o push-down
10000	900	2256
100000	7525	25569
1000000	94082	322840

To investigate the effectiveness of push-down optimization in Lindorm TSDB's pipeline streaming execution engine. We perform Q3 query on three-node Lindorm TSDB with and without push-down optimization respectively. The results are in Table 9. The query aggregate on data in all timeseries for 8 hours to guarantee a large computational workload. When push-down optimization is unavailable, Lindorm TSDB have to collect all data and then finish inter-timeseries aggregate operation in the proxy level. This leads to about 2× higher query latency.

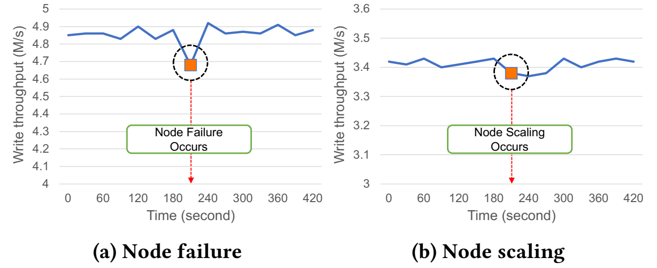
Table 10: Ablation study on the seriescache

Host Scale	Write throughput (M/s)		Q3 query latency (ms)	
	with cache	w/o cache	with cache	w/o cache
1000	5.88	4.75	160	189
10000	5.28	3.6	383	485
100000	4.66	1.4	3549	5235

In Table 10, we explore how the seriescache for forward index improves the performance of Lindorm TSDB. The results show very large improvement in write throughput from seriescache, between 23.8% to 232%. The seriescache also contirbutes to the efficiency of

Table 11: Write throughput (M/s) of Lindorm TSDBs with different nodes

Host Scale	2-node	4-node	6-node
10000	5.05	11.55	19.78
100000	5.08	11.14	19.01
1000000	4.64	10.99	18.06



(a) Node failure **(b) Node scaling**

Figure 11: Write throughput over time when the database cluster status changes

Q3 query where tag values are required for grouping timeseries. With the seriescache, query latencies are reduced by 15.3% to 32.2%.

We compare the write throughputs of distributed Lindorm TSDB deployed in 2, 4, and 6 nodes to study the scalability of data ingestion. The results in Table 11 shows over 100% scalability. At all timeseries scales, the per-node write throughput is higher when there are more nodes in Lindorm TSDB. When the number of nodes increases, a single node manages fewer timeseries, making the data structures in the storage engine more efficient. For example, most of accesses to the index in one node cannot be hit in the cache when data of a large number of timeseries are written to the node. If the Lindorm TSDB cluster has more nodes, it is possible for those index entries that are accessed in the disk to be cached in other nodes.

We study Lindorm TSDB's adaptability to two cases, *i.e.*, single-node failure event and node scaling event. We provide two stable traffic data inputs and then manually close one TsCore node and add two TsCore nodes respectively during data ingestion. Figure 11 displays the write throughput over time before and after manual operations. When a node goes down (Figure 11a), the write throughput of Lindorm TSDB slightly drops by 4%. After that, within 30 seconds, other healthy nodes take over the data of the failure node from the shared storage and the system performance returns to be stable. When adding new nodes to the database (Figure 11b), there is no significant change in the write throughput. Because the data that are written before the scaling does not need to be migrated with the help of the sharding strategy taking time into account.

7 LESSONS LEARNED

Lindorm TSDB serves several large-scale monitoring systems within Alibaba and provides external services in Alibaba Cloud. Lindorm TSDB has undergone many years of iterations of several versions. During the evolution, we accumulate some business observations and system design experience summarized as follows:

- In distributed databases, node failure is common, and when a node crashes in Lindorm TSDB, a healthy node takes over its shards. However, the new node cannot provide service until it

finishes replaying all records in the WAL. To address this, we designed an asynchronous WAL replaying mechanism, which allows the shard to start serving write requests immediately after it is started. The read service is enabled only after the replaying is completed to ensure data consistency. This guarantees high availability of write requests as a priority after shard migration. With this feature enabled, write service interruption time drops from minutes to seconds.

- The adoption of a schematized multi-fields data model and the support of SQL syntax not only helps users understand the time series data model and simplifies its usage, but also facilitates troubleshooting for DBAs. For example, we can use SQL "explain" to see if the entire execution plan meets expectations. Additionally, it allows for easy integration with third-party ecosystems.
- Enabling the pre-downsampling feature effectively reduces query latency by 80% in businesses, at the cost of an 8% increase in storage space. This cost is manageable with the storage tiering feature in Lindorm DFS. Also, since the computation occurs during compaction, the additional CPU usage is minimal, at less than 5%. Compared to instant computing at query time or using features like Continuous Query [21], resource consumption is significantly lower.
- In its early versions, Lindorm TSDB did not have a pipelined execution engine. When a large amount of data was queried, all of the data had to be read out at once and cached in memory for calculation. This led to memory exhaustion with FullGC and affected the service, making it difficult to meet the needs of supporting important business operations, such as the dashboard of Alibaba's Global Shopping Festivals. The newly designed pipelined execution engine solves this problem and improves performance by at least 10×.
- In monitoring scenarios, the latest value query is often used to check the health status of system. This requires high QPS and low latency. To address this, we have designed a cache specifically for this query. The latest value of each timeseries is cached when queried and will be updated when new data points are written to that timeseries. After implementing this cache, query response time was reduced by 85%.

8 RELATED WORK

Time-series database. There are many previous works focusing on time-series databases. OpenTSDB [8] uses HBase [1], a key-value database to store time-series data points, where each data point is an individual data row with rowkey. It leads to low data compression ratio and access efficiency. InfluxDB [18] develops TSM storage architecture based on LSM, greatly improving the write throughput. But it lacks optimization in query execution (*e.g.*, InfluxDB does not parallelly perform computation on multiple timeseries in one data partition). TimescaleDB [23] is a Postgres-based TSDB. It mainly relies on partitioning technology for parallel data ingestion and query. But its performance drops significantly when executing the query hitting multiple timeseries. QuestDB [32] is a column-oriented TSDB showing high single-node write performance, but it does not offer distributed deployment and scalability. Timon [11], BTrDb [5] and Peregreen [38] propose novel data structures storing data points in the same timeseries. They can have fast response time for aggregate query on data of single timeseries across long time range, which is not common in monitoring systems (see in Table 1). Gorilla [31] proposes the delta-of-delta timestamps and XOR'd floating point values, which are widely used in the existing TSDBs for data compression. TimeUnion [40] and Byteseries [35] mitigate the high-cardinality problem by compressing the inverted index in memory. But they ignore the acceleration of access to the index on disk. There are also TSDBs designed for Internet of Things (IoT) scenario, such as DB2 [15] and IoTDB [39]. They are not efficient for complex tags query in monitoring systems. To tackle the increasing timeseries scale in monitoring systems, more and more TSDBs [4, 37] are deployed in distributed way. They use shared-nothing architecture and suffer from performance degradation due to data migration in the case of node scaling.

In-Database Machine Learning. To the best of our knowledge, no existing TSDBs integrate machine learning functions. The existing systems [7, 16, 25, 30, 34] that support in-database machine learning are limited to relational data model. Although Oracle ML [30], Azure Data Explorer [28] and BigQuery ML [16] allows applying time series forecasting and anomaly detection to time-series data, they do not optimize the computation based on characteristics of time-series data. Lindorm ML is inspired by SQL Server's Raven [25]. It reuses the ONNX RUNTIME [29] inference engine for cross-optimization on relational and linear algebra. We also utilize open-source inference engines for specific ML computation.

9 CONCLUSION

In this paper, we first summarize data scales and common query patterns in large-scale monitoring systems. Then we present Lindorm TSDB, a distributed time-series databases that is designed for handling massive timeseries in large-scale monitoring systems. Lindorm TSDB combines shared-nothing architecture and shared storage to scale nodes efficiently as the number of timeseries in systems increases. Lindorm TSDB adopts an optimized index structure with cache and a novel pipelined execution engine to acheive high write throughput and efficient processing of queries hitting a large number of timeseries. For better detection and diagnosis of system performance issues, Lindorm TSDB enables users to analyze data with ready-to-use anomaly detection and time series forecasting algorithms through SQL.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable suggestions and helpful opinions. We would also like to thank Yong Lin, Wei Zou, Songzheng Ma, Dengke He, Yaguang Li, Yuan Cui, Xiang Wang, Wenlong Yang, Yang Liu, Qingyi Meng, Xing Jin and Youdong Zhang who contributed significantly to the development of Lindorm TSDB.

REFERENCES

[1] 2023. Apache HBase. https://hbase.apache.org/. Last accessed: 2023-07-07.
[2] 2023. Apache ZooKeeper. https://zookeeper.apache.org/. Last accessed: 2023-07-07.
[3] 2023. Time Series Benchmark Suite. https://github.com/timescale/tsbs. Last accessed: 2023-07-07.
[4] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. 2020. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3181–3194.

[5] 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)*. 39–52.

[6] Adebiyi A. Ariyo, Adewumi O. Adewumi, and Charles K. Ayo. 2014. Stock Price Prediction Using the ARIMA Model. In *Proceedings of the 2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation (UKSIM '14)*. IEEE Computer Society, USA, 106–112. https://doi.org/10.1109/UKSim.2014.67

[7] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[8] The OpenTSDB Authors. 2021. OpenTSDB. http://opentsdb.net/. Last accessed: 2023-07-07.

[9] The OpenTelemetry Authors. 2023. OpenTelemetry. https://opentelemetry.io/. Last accessed: 2023-07-07.

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[11] 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*. 739–753.

[12] Alibaba Cloud. 2023. Alibaba Cloud ESSDs. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/essds. Last accessed: 2023-07-07.

[13] Alibaba Cloud. 2023. Alibaba Cloud OSS. https://www.alibabacloud.com/product/object-storage-service. Last accessed: 2023-07-07.

[14] Alibaba Cloud. 2023. Alibaba ECS. https://www.alibabacloud.com/product/ecs. Last accessed: 2023-07-07.

[15] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, et al. 2020. Db2 event store: a purpose-built IoT database engine. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3299–3312.

[16] Google. 2023. Bigquery ML. https://cloud.google.com/bigquery/docs/bqml-introduction. Last accessed: 2023-07-07.

[17] Xiao He, Ye Li, Jian Tan, Bin Wu, and Feifei Li. 2023. OneShotSTL: One-Shot Seasonal-Trend Decomposition For Online Time Series Anomaly Detection And Forecasting. *Proc. VLDB Endow.* 16, 6 (2023), 1399–1412.

[18] InfluxData Inc. 2023. InfluxDB. https://docs.influxdata.com/influxdb/v2.6/. Last accessed: 2023-07-07.

[19] InfluxData Inc. 2023. InfluxDB TSM. https://docs.influxdata.com/influxdb/v1.3/concepts/storage_engine/. Last accessed: 2023-07-07.

[20] InfluxData Inc. 2023. InfluxQL. https://docs.influxdata.com/influxdb/v1.8/query_language/. Last accessed: 2023-07-07.

[21] InfluxData Inc. 2023. InfluxQL Continuous Queries. https://docs.influxdata.com/influxdb/v1.8/query_language/continuous_queries/. Last accessed: 2023-07-07.

[22] TimeScale Inc. 2020. TimescaleDB vs InfluxDB. https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/. Last accessed: 2023-07-07.

[23] TimeScale Inc. 2023. TimeScaleDB. https://www.timescale.com. Last accessed: 2023-07-07.

[24] TimeScale Inc. 2023. TimeScaleDB does not use index in the last(). https://docs.timescale.com/api/latest/hyperfunctions/last/. Last accessed: 2023-07-07.

[25] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2019. Extending relational query processing with ML inference. *arXiv preprint arXiv:1911.00231* (2019).

[26] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience* 46, 11 (2016), 1547–1569.

[27] Bryan Lim, Sercan Ö Arık, Nicolas Loeff, and Tomas Pfister. 2021. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting* 37, 4 (2021), 1748–1764.

[28] Microsoft. 2023. Azure Data Explorer. https://azure.microsoft.com/en-us/products/data-explorer. Last accessed: 2023-07-07.

[29] Microsoft. 2023. ONXX Runtime. https://onnxruntime.ai/. Last accessed: 2023-07-07.

[30] Oracle. 2023. Oracle Machine Learning for SQL. https://docs.oracle.com/en/database/oracle/machine-learning/oml4sql/21/dmcon/time-series.html. Last accessed: 2023-07-07.

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

[32] QuestDB. 2023. QuestDB. https://questdb.io/. Last accessed: 2023-07-07.

[33] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. 2020. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* 36, 3 (2020), 1181–1191.

[34] Maximilian Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-database machine learning: Gradient descent and tensor algebra for main memory database systems. *BTW 2019* (2019).

[35] 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*. 60–73.

[36] solid IT. 2023. DB-Engines Ranking of Time Series DBMS. https://db-engines.com/en/ranking/time+series+dbms. Last accessed: 2023-07-07.

[37] TDengine. 2023. TDengine. https://tdengine.com/. Last accessed: 2023-07-07.

[38] Alexander A 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 589–601.

[39] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. 2020. Apache IoTDB: Time-series Database for Internet of Things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904.

[40] 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*. 1418–1432.