Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics

Wei Cao[†,‡], Yusong Gao[†], Feifei Li[†], Sheng Wang[†], Bingchen Lin[†], Ke Xu[†], Xiaojie Feng[†], Yucong Wang[†], Zhenjun Liu[†], Gejin Zhang[†]
{mingsong.cw, jianchuan.gys, lifeifei, sh.wang, bingchen.lbc, ted.xk, xiaojie.fxj, yucong.wyc, zhenjun.lzj, gejin.zgj}@alibaba-inc.com
[†]Alibaba Group and [‡]Zhejiang University

ABSTRACT

With the increasing demand for real-time system monitoring and tracking in various contexts, the amount of time-stamped event data grows at an astonishing rate. Analytics on time-stamped events must be real time and the aggregated results need to be accurate even when data arrives out of order. Unfortunately, frequent occurrences of out-of-order data will significantly slow down the processing, and cause a large delay in the query response.

Timon is a timestamped event database that aims to support aggregations and handle late arrivals both correctly (i.e., upholding the exactly-once semantics) and efficiently. Our insight is that a broad range of applications can be implemented with data structures and corresponding operators that satisfy *associative* and *commutative* properties. Records arriving after the low watermark are appended to *Timon* directly, allowing aggregations to be performed lazily. To improve query efficiency, *Timon* maintains a *TS-LSM-Tree*, which keeps the most recent data in memory and contains a time-partitioning tree on disk for high-volume data accumulated over long time span. Besides, *Timon* supports materialized aggregation views and correlation analysis across multiple streams. *Timon* has been successfully deployed at Alibaba Cloud and is a critical building block for Alibaba cloud's continuous monitoring and anomaly analysis infrastructure.

CCS CONCEPTS

• **Information systems** → **Data management systems**;
• **Networks** → *Cloud computing*.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3386136

KEYWORDS

time series database; cloud computing; data processing system; real-time data analytics; out-of-order events

ACM Reference Format:
Wei Cao[†,‡], Yusong Gao[†], Feifei Li[†], Sheng Wang[†], Bingchen Lin[†], Ke Xu[†], and Xiaojie Feng[†], Yucong Wang[†], Zhenjun Liu[†], 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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3386136

1 INTRODUCTION

Large volumes of timestamped event data are found in increasingly more application domains, ranging from online system monitoring to real-time analysis in IoT projects. Many of these applications require time series analysis and aggregations on both new data and historical data. In most scenarios, aggregation correctness (i.e., count-exactly-once) and low latency are necessities. For example, our TcpRT [14] system is a monitoring and diagnosis system for the cloud database platform, which collects a large amount of event data from various sources, such as request tracing logs, system performance metrics, and network device logs. This information is gathered and analyzed in real time to detect anomaly events and derive the root cause, with which the system can recover from failures as soon as possible by taking actions (like migrating database instances to healthy machines or isolating the traffic of an overloaded server). If faults are not detected due to data missing, the availability of cloud service will be impaired. On the other hand, duplicated events may cause fake spikes that trigger repair operations like migrations, wasting huge amounts of system resources.

From the recent emergence of the lambda architecture, stream processing systems like Storm [11] explore solutions to address the issues above and have iterated over several variants (detailed in Section 7). The latest Dataflow Model [2] supports stream data analytics via the concepts of window,

trigger and incremental processing model. Events are accumulated to windows when they arrive, and the windows are flushed into external storage when they reach trigger conditions. Large scale databases, such as HBase [8] and Cassandra [7], are usually chosen as the external storage. Though this architecture can handle and aggregate large amounts of timestamped events, it exposes three critical problems.

Delayed visibility. The data is aggregated in the streaming system, and newly arrived data cannot be queried until it is flushed from the streaming system, which significantly increases the delay of data visibility.

Read-Modify-Write. When late data arrives, a Read-Modify-Write operation is required to read and update the previously aggregated value on external storage. Since the used external storage is often write-optimized (i.e., using LSM-trees [23]), this Read-Modify-Write operation is costly due to read amplification, significantly affecting the throughput when the ratio of late events increases. However, the late arrival of events is a norm in a distributed system due to various reasons, such as clock skew, network delay, and node failover. Even worse, in our scenario, out-of-order events are born in nature. Consider a cloud database dashboard that shows average query execution time grouped by their issue time. This execution time event cannot be generated until the query completes. For OLAP-type analytical queries, execution time may take hours and vary drastically, and hence such events are disordered in nature.

Analysis on massive and long-term events. In many scenarios, a massive number of event streams need to be managed, and at the same time, historical events need to be diagnosed on purpose, which is challenging to existing database systems. First, high-rate concurrent writes from a large number of streams (i.e., sequences of timestamp-value pairs) exist. For example, TcpRT monitors the machine-to-machine link channels from tenants, leading to hundreds of millions of channels in our product system. Each link channel further contains ten metrics, such as round-trip time and traffic flow. That results in more than one billion streams. Second, the fast exploration of long-term event streams is required. When troubleshooting an anomaly in TcpRT, we often need to obtain long-term trends of metrics with different statistical operators (e.g., `avg`, `max/min`, and `quantile`) and compare them with the problematic period. Similarly, when an outlier is observed, we need to retrieve historically similar behaviors for further investigation. However, existing databases cannot meet both requirements at the same time. InfluxDB [20] supports massive-stream writes but lacks fast long-term time-series exploration. BtrDB [6] supports long-term time-series exploration through an innovative time-partitioned tree but does not consider the massive-stream scenario. More discussions are covered in Section 7.

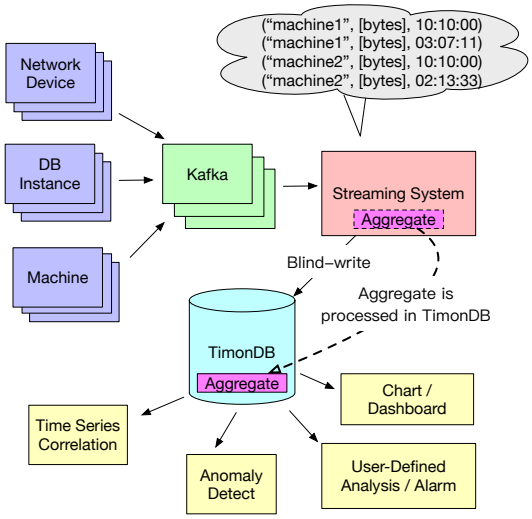


Figure 1: Inputs are collected from various sources and processed in Streaming System such as flink by event-time, and then results are appended to *Timon*. Outputs are consumed by dashboard and anomaly analysis agents. In our architecture, we offload window operations such as aggregate and out-of-order processing (blind-write) down to the storage layer.

Our contributions. We have designed and implemented *Timon*, a persistent timestamped event database that supports high-throughput writes and fast long-term time-series queries. In particular, it gracefully handles out-of-order events with low overhead and provides high query efficiency. The design of Timon takes into account the effective integration with the Dataflow Model. The late arrivals can be appended to *Timon* directly as blind writes without an upper-layer buffer. Records written to *Timon* are kept in memory and periodically merged to special data structures on disk, which is much more efficient than Read-Modify-Write implementations. The correctness is theoretically guaranteed by the *associative* and *commutative* properties of operators on these structures (e.g., statistical structures like Histogram and probabilistic structures like Hyperloglog counter [19]). We observe that a broad range of applications can be represented using these data structures. Besides, a monotonically increasing offset attaches to each record, so that duplicated records can be detected and dropped. This makes insertions idempotent in *Timon* and eases the achievement of exactly-once semantics in the computation layer.

As shown in Figure 1, with *Timon*, we re-implement monitoring and diagnostic systems like TcpRT [14] in Alibaba Cloud. There are thousand millions of records written to the *Timon* cluster every second. Though sustaining such a high ingestion rate, the cluster only consists of 16 machines. In this scenario, read operations come from automated tasks that

detect suspicious machines and instances, as well as from data visualization systems that display dashboards refreshed every second. With *Timon*, advanced analytical functions can be easily implemented, such as percentile watermarks, long time-span queries, multiple stream aggregations, and anomaly detection algorithms (like correlation search). These functions can be invoked using TQL, an intuitive and expressive query language provided by *Timon* to ease application development.

In summary, our contributions are as follow:

- We present a better architecture for timestamped event data processing and analytics. It offloads window operations and out-of-order processing down to the storage layer, which brings lower data visibility delay and enhances the analyzing capability for both real-time and historical events.
- We verify the advantages of blind-write over Read-Modify-Write for handling out-of-order events and implement both methods on top of state-of-the-art databases, including HBase, InfluxDB, BtrDB, and Gorilla. Based on this observation, we design and implement *Timon* that improves blind-write with a lazy-merge strategy, relying on associative and commutative data structures.
- We propose a novel *Time-Segment Log-Structured Merge-Tree* (*TS-LSM-Tree*), which combines the advantages of LSM-tree and segment-tree. It builds segment-trees using asynchronous compactions to sustain high throughput for massive streams and support efficient query for on-disk time-series. This structure further benefits our lazy-merge strategy and two typical query types: monitoring (i.e., scan all recent streams) and diagnosis (i.e., retrieve a few long-term streams).
- We enhance *Timon* with user-friendly tools and facilities, such as metric set, materialized view and TQL. We introduce several industrial scenes and illustrate the advantages of *Timon* for monitoring and diagnostic applications.

The rest of the paper is structured as follows. Section 2 explains our motivation, i.e., handling out-of-order events with blind-write, and introduces background on associative and commutative operators. Section 3 and 4 describe the design of *Timon*. Section 5 gives the applications of *Timon* in our production environment. Section 6 presents and discusses our experimental evaluation. Section 7 reviews the related work, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we explain why blind-write is preferred for out-of-order events processing. We then introduce two fundamental concepts that support the feasibility of blind-write, i.e., associative and commutative operators, and idempotence guarantee. We observe that daily-used aggregation operations are directly (or after some transformations) compliant with these operators.

2.1 Out-of-order Events Processing

As shown in Figure 1, input events are collected from machines, network devices and database instances, and then written to message queues. Out-of-order events occur due to various reasons, such as machine failures and application natures. These belated events will eventually reach TcpRT [14] computation nodes, since our infrastructure ensures that a message is delivered at least once. It is up to the application to handle these out-of-order data properly and correctly.

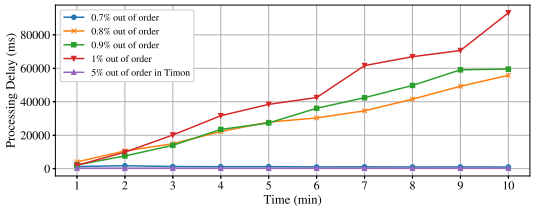


Figure 2: Read-Modify-Write performance on HBase with different out-of-order event ratios (v.s. *Timon*).

In practice, two common approaches are widely used. The first approach is to simply drop out-of-order events. This is easy to implement, however, it compromises the correctness. The other approach is known as *Read-Modify-Write*, in which results of previously processed windows need be fetched from external storage. After merging with newly arrived out-of-order events, the updated results are then written back to the storage. Although this method ensures correctness, it brings in significant overhead, since large amount of random reads and writes are potentially involved. We evaluate this overhead experimentally using a 4-node HBase cluster as the external storage. We observe that the processing delay (i.e., process-time minus event-time) of the stream with a higher degree of disorder increases faster because of more frequent *Read-Modify-Write*, as shown in Figure 2.

Our solution is to adopt a *blind write* strategy, in which applications simply append out-of-order events to *Timon*. We introduce and implement in *Timon* a number of data structures and corresponding operators that satisfy both associative and commutative properties. Events and their aggregations in any window are stored in these data structures. When an out-of-order timestamped record arrives, it will be converted to one or more of these structures, and be merged with previous results periodically. We observe that most application logic can be expressed using these data structures (detailed in Section 2.2).

2.2 Associative and Commutative Operators

There is a collection of associative and commutative operators supported in *Timon*, including `sum`, `max`, `min`, `avg`, `quantile`, `stddev` and `distinct`. Among them all, `sum`, `max`, `min` and `avg` are basic statistical functions used by a wide range of applications in monitoring systems. Besides, real-world applications further desire more sophisticated operators. For example, database administrators need to check the 95th and 99th percentiles of query latency to ensure the quality of their services. Site reliability engineers need to monitor the number of distinct failed machines to ensure the availability of clusters. Hence, advanced operators, such as `quantile`, `stddev` and `distinct`, are demanded.

In *Timon*, above operators are implemented on top of several data structures, in order to obtain the associative and commutative properties, which are the essence for achieving correctness in our system. The formal definitions are provided below.

Definition 1 (Operator). *For a data structure D, an 2-arity operator σ takes two instances x, y of D and produces a new instance z:*

$$\sigma(x, y) \rightarrow z$$

Definition 2 (Associative Property). *A data structure D meets the associative property, if its operator σ follows the associative law:*

$$\sigma(\sigma(x, y), z) = \sigma(x, \sigma(y, z))$$

Definition 3 (Commutative Property). *A data structure D meets the commutative property, if its operator σ follows the commutative law:*

$$\sigma(x, y) = \sigma(y, x)$$

Lemma 1. *For two associative structures D_1, D_2 with respective operators σ_1, σ_2, operator σ_c on the composed structure $[D_1, D_2]$ is also associative:*

$$\sigma_c([x_1, x_2], [y_1, y_2]) = [\sigma 1(x_1, y_1), \sigma 2(x_2, y_2)]$$

where $[x_1, x_2]$, $[y_1, y_2]$ are two instances of the composed structure.

Based on above definitions and lemmas, we discuss how operators are implemented using different data structures:

Sum, Max, Min. Implementations of these operators are straightforward. Computations on value pairs inherently satisfy the associative and commutative properties.

Avg, Stddev. These operators cannot be directly derived from partial results, and require additional state information. For `avg`, we maintain two fields, i.e., sum and count, in the data structure, in which average is computed as *sum*/*count*. During an operation, both fields are accumulated separately

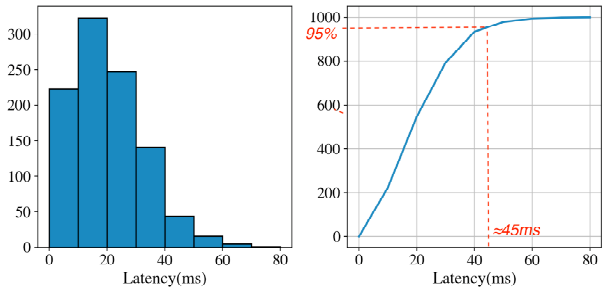


Figure 3: Query Latency Histogram

from two partial results. The case for `stddev` is more complicated, in which three fields are required. To explain, we show here the definition of standard deviation:

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}}$$

where x_i is an individual value, $\overline{x}$ is the mean value, and N is the total number of values. We denote $V = \sum_{i=1}^{N}(x_i - \overline{x})^2$, and hence the required tuple is $[N, \overline{X}, V]$. Suppose there are two partial results $[N_1, \overline{X_1}, V_1]$ and $[N_2, \overline{X_2}, V_2]$, the combined tuple $[N', \overline{X'}, V']$ can be calculated as follows:

$$N' = N_1 + N_2$$

$$\overline{X'} = \frac{N_1 * \overline{X_1} + N_2 * \overline{X_2}}{N'}$$

$$V' = N_1 * (\overline{X_1} - \overline{X'})^2 + V_1 + N_2 * (\overline{X_2} - \overline{X'})^2 + V_2$$

The combined standard deviation is hence $\sqrt{V'/(N'-1)}$.

Quantile. We maintain the estimation of percentiles using a histogram structure. Figure 3 gives an example of query latency histogram in a database, where the x-axis is the query latency in millisecond and the y-axis is the count of queries. This histogram can be converted (and smoothed) to a cumulative distribution function, which is shown on the right side of Figure 3. It is then straightforward to estimate the 95th percentile of the query latency (e.g., 45ms in this case). A simplified Histogram data structure is implemented in *Timon* with four fields: a *low* that represents the lower bound of the value range; a *upper* that represents the upper bound of the value range; a *bars* that contains a vector of counters, each of which represents the number of values in a given sub-range; a *step* that indicates the length of sub-range covered by each bar. The σ operator of Histogram is essentially the sum of two vectors, i.e. *histogram.bars*[1]. Note that Histogram only provides approximate percentiles, and the precision depends on the choices of *low*, *upper* and *step*.

Distinct. Calculating the exact count of distinct values in a collection requires at least linear space w.r.t. its cardinality (e.g., using a hash table), which is impractical in large-scale

[1]Two histograms have the same low, upper and step.

applications. In *Timon*, we implement distinct count operator using HyperLogLog [19], which is an approximate algorithm for the count-distinct problem. It estimates the number of distinct elements in a multiset using a small number of buckets, i.e. *register sets*. Its σ operator is:

$$RegisterSet'[i] = max(RegisterSet_1[i], RegisterSet_2[i])$$

where i indicates the i-th register. HyperLogLog is widely used in daily tasks like tracking the number of unique visitors (UV) of a website.

2.3 Idempotence

With blind-write and operators satisfying associative and commutative properties, we only need to store aggregated values in *Timon*, and can safely discard original values, which is the fountain of efficiency. However, this brings a problem of correctness for the entire data process pipeline. If failovers occur while the data points are being written, we have to remove the points that have been successfully written. Otherwise, aggregated values will include the same point twice.

In the context of *Timon*, our stream processing task reads events from a message queue system and inserts these events (i.e., records) into *Timon* via SDK. Usually, messages are divided into multiple partitions in message queue systems, such as Apache Kafka, AWS Kinesis [12], and Alibaba Cloud Loghub [3]. Each partition is an ordered, immutable sequence of records that can be continuously appended to. Each record is assigned a local sequential identifier called *offset* that uniquely identifies it within the partition. For each partition, there is a single stream processing worker reads messages in ascending order on offset, and the worker sends its messages along with partition/offset information of message queue system to *Timon* in batches following the stop-and-wait protocol. *Timon* keeps the maximum offset it encountered for each partition. In this way, duplicated messages can be easily detected by comparing its offset with the current maximum value, which makes the whole procedure atomic to ensure consistency.

3 SYSTEM OVERVIEW

In this section, we present the overview of *Timon*, including its data model and design principles.

3.1 Timestamped Data Model

To introduce our data model, we first show in Figure 4 an example that contains link-channel quality information for cloud database tenants, collected from our TcpRT system. In particular, it includes a number of measurements related to requests/responses from different tenants, such as average latency, upstream traffic and downstream traffic. *Timon*'s

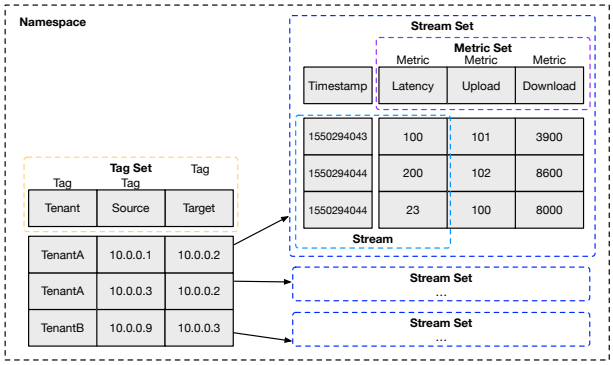


Figure 4: Data Model

data model facilitates the expression of application logic, and has following major concepts:

- **Metric.** A metric is the name of an attribute, e.g., `Latency`, `Upload` and `Download` in our example.
- **Metric set.** A metric set contains a set of metrics that are always collected together. For example, a metric set for link quality has `Latency`, `Upload` and `Download`, while a metric set for memory management has `MemTotal`, `MemUsed`, `Buffer`, `Cached` and `Dirty`.
- **Stream.** A stream is a sequence of numeric data points collected over time for a single metric.
- **Stream set.** It is the collection of streams in a metric set, e.g., memory usage metrics read from `/proc/meminfo`. They are collected and stored as a batch to achieve high write throughput.
- **UUID.** A stream or a stream set can be identified by a unique identifier UUID.
- **Tag.** A UUID can be attached with one or more tags (i.e., tag set). For example, a UUID for a database usually has tags like host IP, host name, instance ID, datacenter ID, etc. Tags are used as filtering conditions, e.g., retrieval of the memory usage on an IP address.
- **Namespace.** Streams and tags that belong to different applications are usually isolated under different namespace.
- **Materialized aggregate view.** A materialized aggregate view is created to pre-compute aggregations of those streams with the same set of tags. For example, if an administrator needs to monitor the traffic of all database instances on a host, he/she may create a materialized aggregate view of the metric `TPS` grouped by `hostname` tag.

Note that this data model in *Timon* can be easily mapped to a relational model, i.e., namespace to table and timestamp/metric/tag to columns. In this case, each row contains a timestamp, values in the metric set, as well as values in the corresponding tag set. As a result, it is easy to perform

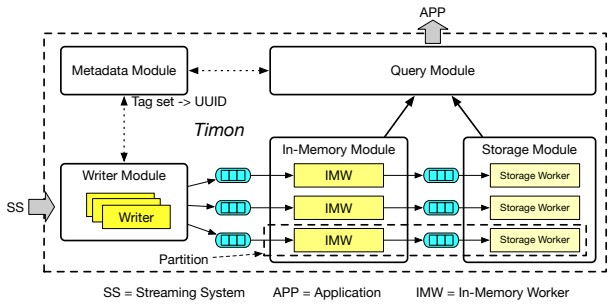


Figure 5: High-level *Timon* Architecture

queries in *Timon* using SQL or SQL-like languages (illustrated in Section 5).

3.2 Design Overview

An overview of *Timon* architecture is shown in Figure 5, where we make three important architectural choices for it. First, we follow the SEDA [25] programming model for the write path design, which is suitable for continuous high-pressure writing. SEDA has good scalability and the ability of load management using backpressure (i.e., by blocking on a full queue). In *Timon*, the write process is split into three sub-stages that communicate with each other via intra-process message queues: 1) the *writer* module receives requests, maps tags to UUID via the *metadata* module, and then sends (UUID, stream set) pairs to stage queues according to their UUID hash values; 2) the *in-memory* module writes data to WALs and MemTables in *TS-LSM-Trees*; 3) when a MemTable is full, the *storage* module flushes it to disk.

Second, writes and read-only queries follow different request paths. We observe that monitoring and telemetry scenarios usually generate continuous write pressure to the underlying storage. However, query requests are of low frequency, some of which require long execution time, e.g., scanning a large number of streams to alarm an exception. These queries will block subsequent write requests if they are mixed together. In addition, queries are more concerned about response latency. Hence, it is desired to be able to read data directly from storage bypassing various stages, which is completely different from the write path design.

Third, tags and stream sets are stored separately. Their relations are managed by the metadata module that maps tags to UUIDs (i.e. stream sets). This design brings in two advantages. One is that the in-memory module and storage module are unaware of those tags, which simplifies the implementation logic. The other one is that this design is also in line with common query patterns of timestamped data, i.e., extract target streams through tags and retrieve all points in these streams within a specific time period.

4 DETAILED DESIGN

Timon divides all data into *partitions*, each of which is managed by an in-memory worker and a storage worker, as shown in Figure 5. Each partition has one *Time-Segment Log-Structured Merge-Tree* (*TS-LSM-Tree*), which contains two MemTables (in in-memory worker) and multiple SSTables (in storage worker). In this section, we first introduce *TS-LSM-Tree*, our core data structure that is designed for efficient event processing and fast exploration of long-term time-series. We then propose a lazy merge strategy to handle out-of-order events with blind writes. Other features, i.e., event idempotence and materialized aggregate view, are also discussed.

4.1 Time-Segment Log-Structured Merge-Tree

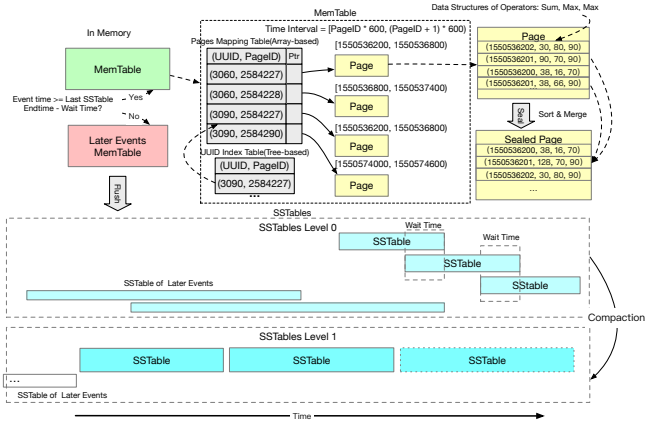


Figure 6: A demonstration of *Time-Segment Log-Structured Merge-Tree*.

The structure of *TS-LSM-Tree* is shown in Figure 6. There are two MemTables in it, one for late events and one for the rest. When an event arrives, *TS-LSM-Tree* first check its event time, and then insert into the corresponding MemTable. If the event time is at least a prescribed time earlier than the end time of the last flushed SSTable, it will be put into the late-event MemTable. This prescribed time is called *wait time*, and is set to 5 minutes in our production environment. In this manner, the number of MemTables (and SSTables) with long time span can be reduced, so that overlaps between different SSTables can be minimized. It improves read performance (detailed in Section 4.2).

4.1.1 MemTable. In *Timon*, all newly arrived time-stamped events (i.e., records), including duplicate, dense and sparse records, are first stored in an in-memory structure called *MemTable*, The goal of MemTable is to execute hot-data aggregation with low cost. The *Page* is the basic memory

management unit in MemTable, and can store up to 600 records[2]. Each active UUID in the partition has at least one page, and records contained in a page are all from a same UUID. Newly arrived records are appended to corresponding pages, leaving them unsorted. When a page is full, it will be *sealed*, which means that all records in that page are sorted by timestamp and those with the same timestamp are aggregated. The sealed page is implemented using an array with event timestamp as the index. For example, in a page starting from time T_0 with 1-second resolution, the element x in the array refers to the record timestamped at $T_0 + x$. In this way, locating a specific timestamp can be simply achieved by calculating its offset in the array, while fetching a time range only involves accessing a contiguous memory space. This structure also makes the insert (and aggregate) of an out-of-order record fast, due to its instant timestamp-based lookup.

In the case that data points are sparse, a page may end up having many empty slots. To avoid the waste of space, we apply an adaptive approach for memory allocation inside page. Initially, a page is unsealed, where new data are appended to the end of it, so that space can be dynamically enlarged according to the current size. In this stage, as elements are not ordered, the whole page needs be scanned for answering a query. Since the length of a page is fixed in *Timon*, the scan cost in unsealed page is still bounded (though with high latency). Before sealing a page, we use a heuristic algorithm to check whether it is dense enough. For a new dense page, the entire memory space of the time-indexed array will be allocated and all elements will be copied to corresponding slots.

4.1.2 SSTable. For a regular time period, (e.g., every 10 minutes), a background task will pack all pages into a SSTable file and flush it to the external storage. In order to avoid having too many small SSTable files, they will be merged periodically, known as the compaction process. A SSTable is composed of multiple *Blocks*, each of which stores up to 600 serialized records with the same UUID, similar to a in-memory page. A UUID's records can span across multiple blocks. These blocks are ordered by timestamp and do not overlap. A *time-partitioning tree* index is built for each UUID in a SSTable to accelerate processing queries over a long time span (discussed in Section 4.1.3).

A SSTable consists of four components: MetaZone, BucketZone, CollisionZone, and DataZone. The *MetaZone* records meta information of the SSTable, such as the minimum and maximum timestamps from contained records. Each MetaZone takes about 0.5KB space, which means that only 50MB is required for 100,000 SSTables. Therefore, *Timon* is able to cache all MetaZone in memory to efficiently serve queries on cold data. The *BucketsZone* together with the *CollisionZone* is analogous to a persistent hash map. This structure maps a UUID to its Time Series Description (TSD), which contains the summary information of the corresponding time series (e.g., minimum time/maximum time and offset). For each UUID, its time series data is stored in *DataZone*, which is made up of a time-partitioning tree (Section 4.1.3) and multiple blocks. In terms of the physical layout, both BucketsZone and CollisionZone are an array of buckets, and each bucket holds a certain TSD and an optional 64-byte field to address other collided entries. The primary purpose of this design is to lower the cost of locating a specific data point, since a SSTable may contain a huge number of UUIDs where even binary search can be expensive. With the help of persistent hash map, UUID lookup in SSTable can be done in constant time.

Note that, in MemTable, the timestamp and all metric values that belong to the same metric set are placed consecutively in one row, so that these data items can be updated in one atomic write operation. There are mainly two advantages for this design. First, updates to a metric set are either visible or invisible as a whole, which guarantees the atomicity. Second, data locality can be utilized, since streams of a metric set tend to be read together. In contrast, in SSTable, the timestamp and metric values are stored separately in a column oriented fashion. Since data in SSTable is less frequently accessed, achieving high compression ratio is more critical (i.e., to reduce amount of disk IOs) than having better locality.

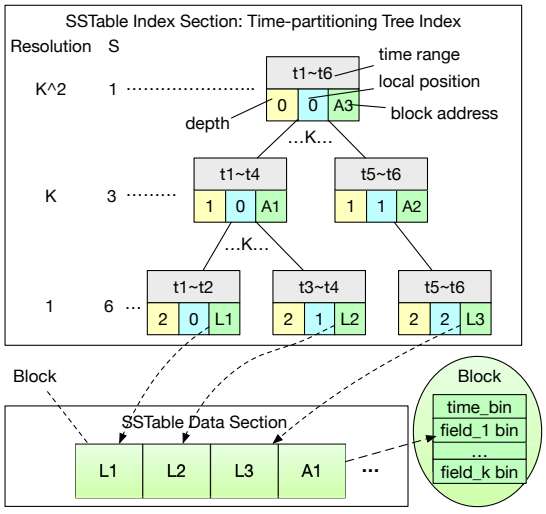


Figure 7: Time-partitioning Tree Index

4.1.3 Time-partitioning Tree. To support efficient query processing in different resolution and time granularity, we design a time-partitioning tree (Figure 7), which indexes time

[2]If events are collected at the fixed frequency of one second, the capacity of a page can hold records received for approximate 10 minutes.

intervals as a decomposition of the entire timespan in a hierarchy. In each level of this hierarchy, time intervals are discrete and disjoint, but *jointly cover the entire timespan*. Each level down the hierarchy provides a decomposition of the timespan in finer granularity and also a finer resolution for its data items. The entire hierarchical decomposition is then indexed by a k-ary tree.

When walking down the tree to look for data of desired granularity and time range, we avoid reading the entire set of data items, as fetching all detailed data items along the path introduces unnecessary data access. Instead, we take an approach where data items and index are separated. In particular, detailed data items exist in the *Blocks*, and only the index and summary information (e.g., min and max time) are stored in time-partitioning tree. We also ensure that each node in the tree is of same size, which enables the easy random access to any node in the tree by calculating appropriate offset values. This also means that we no longer need to pay the overhead to link (i.e., store addresses or node ids) to parent, child, and sibling nodes in any node. Their addresses can be directly calculated through the tree structure itself and the constant size of each node. This design helps reduce the memory footprint of the index.

Once reaching the level with desired granularity, fetching next data item in this level should try to avoid random disk access. Therefore, data items from the same granularity are placed consecutively in the Block. This brings cache-friendly access when retrieving data items from the Block. Furthermore, to achieve better compression performance, raw data items and aggregated values are split into separated compressed column partitions.

Detailed design. The position of a node at a certain depth in the tree is called its *local position*. A node in depth d ($d >= 0$) with local position p ($p >= 0$) is formed by merging nodes in depth $d+1$ from positions $p*K$ to $\min(MaxSize_{d+1}, (p+1)*K-1)$, where K is the fanout of the tree. Assume that the depth of the tree is D ($D > 0$) and the base resolution is b. The resolution of nodes in depth d ($d >= 0$) is $b * K^{D-1-d}$. Time Index Tree is essentially a left-complete k-ary tree, where every node except the rightest one in each level has K children. All nodes are placed into an array sorted by their depths in the tree and then by their local positions in that level. To distinguish node's offset in the array from its local position, we denote it *global position*. Since each node has the same size and they form a left-complete tree, we can easily calculate the global position of a node. Let S_d be the global position of the leftest node in depth d, $Child(d, p, i)$ be the global position of the i-th child of the node whose depth is d and its local position is p. We have $Child(d, p, i) = S_d + K * p + i$. Hence, we do not need to maintain address information of any node; all we need is an array of S which contains $S_0, S_1, \ldots, S_{D-1}$ which allows us to calculate the address of any node.

4.2 Lazy Merge

Recall that *Timon* stores data based on its event timestamp. In our production environment, many services may suffer from significant out-of-order events due to their business nature. For example, data generated by TcpRT [14] takes the start time of the database query being monitored as the event timestamp, and some queries may last for a long time, which results in disordered arrival of query completion events.

These latecomers, small in proportion but wide in time range, expose a great challenge to existing databases. For persistent tree-structured databases (e.g. MySQL InnoDB, BtrDB), such out-of-order records (i.e., fragmented and not compact) is hard to be cached, causing performance degradation from random disk IOs. Some other databases like Gorrila, simply discard them, which renders data to be incomplete. Since *Timon* follows LSM-like storage layout, the out-of-order records do not affect the insertion performance. The major downside is that a query may need to touch many qualified shards and combine qualified records from many shards to obtain the final query result at runtime. As illustrated in Figure 8, if the late-arrival records are mixed with other newly arrived records in the active SSTable, the time span of this SSTable will be stretched extremely wide, while most of the covered time range contains no actual records. As a result, a query needs to access many false-positive SSTables, leading to larger query latency.

In order to solve this problem, we design a delayed (i.e., late-arrival) record detection module, which identifies the delayed record as outliers in the SSTable. During a flush operation, the outliers will be transferred to a separate *Late SSTable*, so that the time range in a standard SSTables is small and compact. Although, for each *Late SSTable*, its time range is large, the total number of *Late SSTable*s is small due to the rareness of long-delayed records. As a result, the number of qualified SSTables for a query is significantly reduced, as shown in Figure 8.

Note that not all out-of-order events are placed in *Late SSTable*. As shown in Figure 6, only events earlier than the end time of the last flushed SSTable for at least a wait time will be placed in *Late MemTable*, which is flushed to *Late SSTable*. This optimization comes from our observation that the delay of most out-of-order events is not long, so that events arrive within the wait time can still go to the normal SSTable, avoiding the rapid growth of the *Late SSTable*. Usually, the setting of the wait time is less than half of the time between two MemTable flushes, thus avoiding the overlap of more than two normal SSTables for any specific timestamp. During the compaction process, the overlapping parts of

Late SSTables and normal SSTables are merged, and the rest remain as *Late SSTables* in the next level.

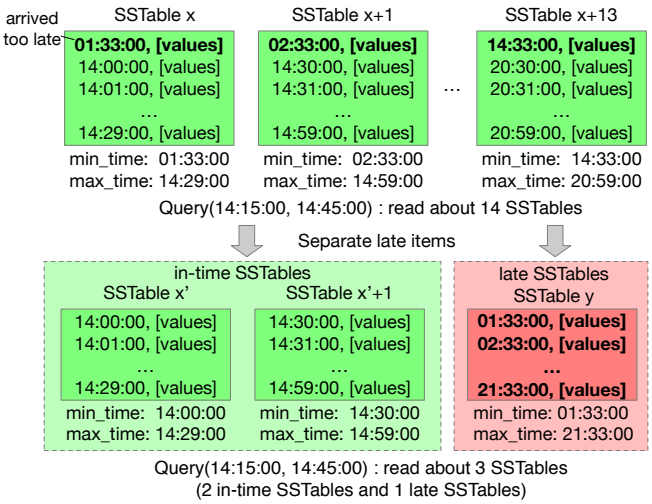


Figure 8: Processing late-arrival records. With *late SSTable*, query execution will choose just 3 *SSTables* in contrast to 14 *SSTables* without *late SSTable* when query from "14:15:00" to "14:45:00".

4.3 Materialized Aggregate View

The design of *Timon* makes it easier to implement materialized aggregate views using the metadata module. Once a materialized view is created, the metadata will assign a derived UUID to it. When there is a request to insert the relevant stream set, the writer will request the metadata to return its UUID. At this time, the metadata can additionally attach the UUID of materialized view in the response, and then the writer will update the view accordingly. Whenever a new sequence of timestamped events arrives, all related views on that partition will perform real-time in-place updates. Such in-place updates are implemented using block array techniques described in Section 4.1.1. To query a view, all sub-sequences from relevant partitions will be fetched and merged. The consistency issue for materialized aggregate views is covered in Section 2.3. By invoking *Timon* APIs, users can manage their aggregation tasks effectively.

5 TQL AND CASE STUDIES

In *Timon*, we provide a declarative query language called TQL, which allows users to retrieve and analyze the underlying timestamped event data with rich semantics. Most of the definitions of TQL are similar to other SQL-like query languages. The main difference is that TQL supports to connect many functions and SQL clauses as a pipeline by a pipe character '|'. Among them, the output of the previous SQL come as the input of the next SQL. Due to the space limitation, we omit its detailed definitions here. With the help of TQL, it is easy for users to express their application needs in *Timon*. In this section, we demonstrate how to use *Timon* to resolve practical problems that we encountered in Alibaba Cloud products. In practice, *Timon* manages various timestamped data, including RDS [5] performance data, TcpRT data, network monitoring data, etc, on which we have developed many application cases as follows.

5.1 Real-Time Dashboard

```
select sum(QPS) as total_qps  from tcprt_view_cluster where role = 'DB'
    group by cluster
    window by 1s period 1s interval 360s
    with confidence 99% on cluster
```

Figure 9: Low-latency aggregate view by cluster dimension using 99th percentile watermark.

To monitor the health of clusters, real-time dashboard under extreme scale (e.g., real-time health heat map of over tens of thousands of machines) is required, refer to Figure 9, so that any faulty behavior can be quickly detected, located and fixed.

Low latency is the key to real-time diagnosis and recovery. We need to collect metrics from tens of thousands of hosts and hundreds of thousands of instances, and then perform aggregation and display. This process was used to be expensive that takes up to tens of seconds, but now its cost has reduced to around just 2 seconds by using the materialized view in *Timon*. The views are updated as soon as raw data newly arrives. Besides, *Timon* also addresses the out-of-order problem where other systems suffer a significant penalty.

5.2 Aggregation Granularity Adaptation

```
select * from tcprt_node where ip in ('192.168.1.1', '192.168.1.2')
      group by ip
      when '2018-08-10 16:30:00','2018-08-17 16:45:00'
      with auto-resolution
```

Figure 10: Read time series with automatic resolution.

Our platform needs to present status information for all machines and instances in Alibaba Cloud service and allow users to navigate in an arbitrary time-range. The system will automatically select the appropriate time granularity to display based on the selected time range, refer to Figure 10. e.g., use hour-granularity in week-range to avoid displaying numerous points. Since the time granularity of raw samples in our production system is one second, we need to pre-aggregate results with different granularities in streaming or batch systems to improve query performance.

However, this method is very troublesome and is difficult to ensure the consistency of the aggregated results. With

the help of time-partitioning tree index, *Timon* efficiently supports multi-granularity aggregation, which is a more nature for this scenario with consistency guarantee.

5.3 Correlation Analysis

```
select * from tcprt_dst where ip in ('192.168.1.1', '192.168.1.2') group by ip
      window by 5s when '2018-08-17 16:30:00','2018-08-17 16:45:00'
| correlation
| metric_filter (_value > 0.7)
```

Figure 11: Correlation Analysis

As shown in Figure 11, this query analyzes the correlation between metrics over a specified time range among a group of machines. In this example, it obtains sub-sequences from timestamped data of targeted machines between "2018-08-17 16:30:00" and "2018-08-17 16:45:00", then applies "correlation" operator to compute the Pearson correlation. If the number of input metrics is n, the number of output metrics will be n^2. Each output metric is a Pearson correlation coefficient, whose tag is the concatenation of a input metric pair. Lastly, "metric filter" operator is applied to filter metrics whose Pearson correlation coefficient are above 0.7.

5.4 Machine Anomaly Detection

Machine anomaly detection is used to detect abnormal machines in real time. For each DB instance, it maintains a time series of query latency in recent hours, builds a Cauchy distribution, and detects abnormal DB instances that suffer from long-running queries. Since DB instances are distributed across multiple machines, if the ratio of abnormal instances on a machine has exceeded a pre-defined threshold, we declare that machine to be an anomaly.

Part 1

```
select avg(pt) as pt from tcprt_dst_ins where dst_role = 'dbnode' group by dst, ins
   window by 300s period 300s interval 7200s
   with confidence 99% on machine, instance
| statistics
| reference ref_dst_ins on dst, ins
```

Part 2

```
select avg(pt) as pt from tcprt_dst where dst_role = 'dbnode' group by dst
   window by 15s period 15s interval 7200s
   with confidence 99% on dst
| statistics
| filter cauchy_cdf(mid_pt, MAD_pt, last_pt) > 0.997
| template "select avg(pt) as pt from tcprt_dst_ins
            where dst_role = 'dbnode'
            and (%s) when %d,%d group by dst,ins window by 15s ",
   or_join(str_fmt('dst=\'%s\'', dst)), _endtime -15s, _endtime
| select
   sum(cauchy_cdf(
      ref('ref_dst_ins', str_fmt('%s,%s', dst, ins), 'mid_pt'),
      ref('ref_dst_ins', str_fmt('%s,%s', dst, ins), 'MAD_pt'), pt)
    > 0.997) as anomaly_cnt,
   sum(1) as total group by dst
| filter anomaly_cnt / total > 0.4
| alarm 'An anomaly occured on machine:' + machine + ' at time: ' + _time
        to 'DBA_group' via 'SMS'
```

Figure 12: Machine Anomaly Detection

Figure 12 shows this query composed of two sub-query tasks. The first task maintains the historical statistics of each database instance's query latency, and the other task ingests these statistics as baseline. In particular, the second task collects statistics of each machine's latency in the last 2 hours, constructs Cauchy Distribution base on their median and MAD, and then uses Cauchy CDF to find machines with an anomaly. A machine is abnormal if the ratio of abnormal instances on that machine is significantly large.

5.5 Network Anomaly Detection

The network structure of a large-scale cloud environment is complex and layered. When a network failure occurs, it is challenging to locate exactly where the problem is (e.g., faulty rack interface vs. power failure). Furthermore, the network switches are subdivided into low-level, medium-level, and high-level switches. Therefore, it is necessary to obtain the topology information of the entire network activity.

We collect network activity data through TcpRT [14], and integrate them into a global network topology graph. Organized by instance-to-machine, machine-to-cluster, machine-to-switch, and other mapping relationships, we can perform real-time diagnosis and pinpoint the exact location of the machine/switch/cluster with the most packet-loss ratio. We present the TQL and further explanation in Appendix A.1.

6 EVALUATION

Timon is implemented in C++ from scratch, with about 35,000 lines of code. We have already deployed *Timon* in the production environment at Alibaba that contains data centers distributed in 21 regions around the world. The most typical application is the full-state tracking system of Alibaba Cloud Databases RDS [5] and PolarDB [4, 15, 16], which collects comprehensive metrics that cover database engine, network (monitored by TcpRT), operating system, and even each individual OS process. 97 *Timon* nodes support about 500 million data points writing per second from the system, and the busiest node serves about 18 million data points per second. To thoroughly evaluate the performance of *Timon*, we collected test datasets from the production system and compared test results with InfluxDB [20], HBase [8], Beringei [18] (an open source version of Facebook Gorilla [24]) and BtrDB [6] (more details are discussed in Section 7).

Experimental Setup. The benchmarks ran on machines with 512 GB RAM, one 3.8T NVMe Disk, two 10Gb network cards, and two Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz processors, each with 16 cores.

Datasets. The datasets include RDS performance dataset and TcpRT dataset. Each record in them has an 8 bytes timestamp, several metrics (each is an 8 bytes number), and a few tags, e.g., the host IP of performance record or the instance

name of the TcpRT request. The main difference between these two datasets is the ratio of out-of-order events. In the performance dataset, there are only 0.29% out-of-order events comparing to 53.90% in TcpRT dataset. The reason is that those TcpRT events can't be collected until TCP requests complete, however, performance data can be collected in real time. Hence, in the following experiments, RDS performance dataset is treated as in-order, while TcpRT dataset is out-of-order. To simulate the real production environment, the test sets are unbounded streams read from Apache Kafka [9]. In particular, the RDS performance dataset nearly has 360,000 metrics per second, while the TcpRT dataset has about 1.7 million metrics per second. We set the parallelism of queries to 40 and that of writes to 80 if not otherwise specified.

6.1 Blind Write v.s. Read-Modify-Write

In this section, we focus on revealing the effectiveness of the blind write mechanism in out-of-order scenarios compared with Read-Modify-Write. Generally, we perform time-window based aggregation in a streaming pipeline, and the aggregate results are stored to external databases. To make sure exactly-once semantic, these late events need to be merged back to results in databases, namely Read-Modify-Write. It's easy to implement Read-Modify-Write for all candidates because every database supports to read previous results and write revised results back. The performance of frequent Read-Modify-Write on out-of-order events is poor, as shown in Figure 13. Therefore, we are only affordable to apply this method to in-order scenarios, i.e., the RDS performance dataset. For out-of-order scenarios, i.e., TcpRT dataset, we introduce a naive blind write implementation, i.e., write raw granularity data directly to the database instead of pre-aggregating in the pipeline.

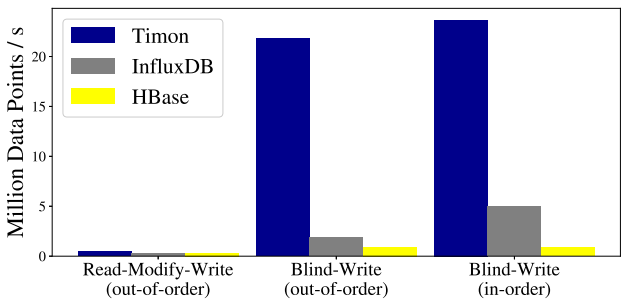


Figure 13: Throughput with different write mode.

Write performance. We compare the write performance of blind write between *Timon*, InfluxDB, and HBase, as shown in Figure 13. In fact, we also tried on Gorilla and BtrDB. However, Gorilla doesn't support blind write of late events due to its local-time windows. The write performance of BtrDB is extremely unacceptable in the dataset where is a large number of streams. Hence, we omit them in this experiment. Besides, since Read-Modify-Write is only valuable in out-of-order scenarios, we skip its test on the in-order dataset. We can see that the write throughput of *Timon* reaches ten times that of InfluxDB in the out-of-order scenario. The main reason includes the following aspects: (a) *Timon* adopts the staged event-driven architecture(SEDA) similar to BtrDB, which can significantly improve system throughput. (b) *Timon*'s MemTable design is also optimized for blind-write(detailed in Section 4.1.1). (c) We observe that InfluxDB spends more than 35% CPU on garbage collection and r/w locks, although we have already optimized relevant parameters.

Query Performance. Blind write can resolve the problem of out-of-order events. However, it hands over the computation of data merging to the query layer, reducing the query performance. Therefore, *Timon* adopts hot-data aggregation and lazy merge mechanism to merge data asynchronously in advance, which help to mitigate the decline of query efficiency. In the experiment, we issue queries to get aggregated results of 100 streams within one hour at 60-second granularity. Because Read-Modify-Write and lazy merge are minor in an in-order dataset, we only focus on the out-of-order dataset. When data is written to the database in the Read-Modify-Write mode, we can read the aggregate result from the database directly, whose query latency is used as the baseline. From Figure 14, we can see that without lazy merge, the query latency of both *Timon* and InfluxDB is much higher than their baselines. *Timon* improves the query performance close to its baseline when the lazy merge is enabled. It can be observed that the query latency of HBase is very high. It is because HBase does not support server-side aggregation semantics, and raw data needs to be sent to clients for aggregation.

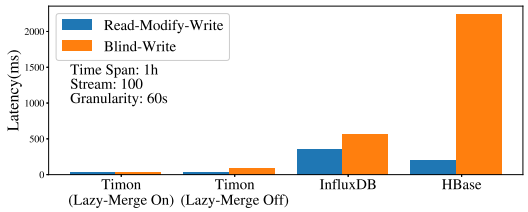


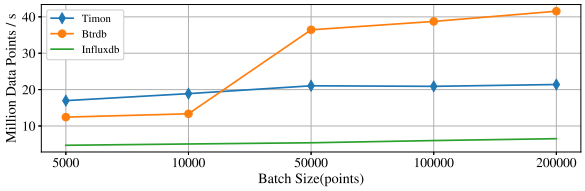
Figure 14: Query latency with different write mode.

6.2 *Time-Segment Log-Structured Merge-Tree*

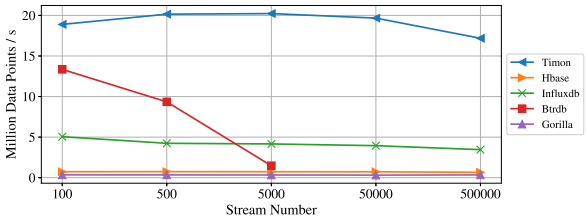
In this section, we compare with other candidates to verify the performance of our *TS-LSM-Tree* design. We create several ordered data sets which consist of 15 billion time-series points, while the number of streams increases from 100 to 500,000.

Write Performance. We can observe from Figure 15(a) that in 100-streams case, with write-batch size changing from 5,000 to 200,000, the throughput of BtrDB rises rapidly, even reaching the double of *Timon*. However, as Figure 15(b) shows, the throughput of BtrDB declines fast when the number of streams exceeds 5,000. Since BtrDB builds time index for each stream in real time, the write cost grows linearly with the number of streams increases. After a closer investigation, we notice that BtrDB is very suitable for small stream-number and large write-batch scenarios, which differ significantly from our production dataset because large write-batch can improve the IO utilization of BtrDB.

As expected, *Timon* maintains high write performance stably when the number of streams increases due to the design of *TS-LSM-Tree*. The main reason is that *Timon* only constructs hash indexes when data is written to MemTable. Time-partitioning Tree Indexes are only built when MemTables are flushed as SSTables.



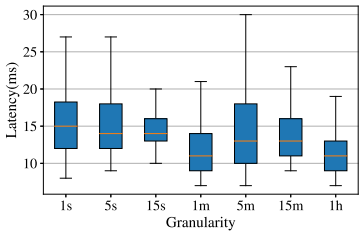
(a) Write Throughput as batch size grow



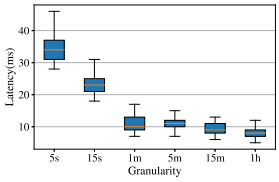
(b) Write Throughput as streams grow

Figure 15: Write Throughput

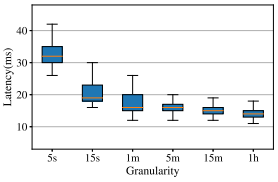
Query Performance. The tree index is multi-layered for various time granularity, in which each tree node caches precomputed aggregate values. We can achieve excellent query performance when the query results can be read directly from tree nodes instead of being computed from scratch. In the experiment, we query on 24-hours (fixed time range) data from *Timon* with different aggregate granularity, e.g., 24 data points on 1-hour granularity, and 1,440 data points on 1-minute granularity. Firstly, we execute the query in MemTables (i.e., warm query). Since there is no pre-aggregated structure in MemTable, The query latency only depends on the number of returned points (i.e., larger granularity means less returned points), as shown in Figure 16(c). Secondly, we execute the query in SSTables (i.e., cold query). Figure 16(b) reveals that the query latency drops dramatically when the query granularity matches the granularity of index-tree nodes, e.g., 1-minute/1-hour queries in the figure. Finally, we query 1,200 points (40 points per stream and 30 streams per query) from each aggregate granularity to further verify the effectiveness of the tree index. Figure 16(a) shows that when we read 1,200 points from 1-second and 1-minute granularity at the same time, the query latency on 1-minute is lower. It is because the aggregate results on 1-minute are cached on tree nodes, while the results on 1-second need to be read from data blocks.



(a) Cold Query (40 Points per Stream, 30 Streams per Query)



(b) Cold Query (24 Hours)



(c) Warm Query (24 Hours)

Figure 16: Query Latency

In addition, we run another set of tests to compare the query performance of all candidates, in case of returning a fixed number of points, e.g., 2,000 in this test. As Figure 17 shows, *Timon* has remarkable query performance as good as that of BtrDB, because both of them have the time segment tree index. They can read aggregation results from tree index so that their performance is not affected by query granularity. However, InfluxDB needs to query raw data and aggregates them on server-side; and HBase even needs to pass raw data to the client for aggregation. Hence, their query latency rises dramatically as the query granularity increases.

6.3 Other Features

In this section, we evaluate the effectiveness and performance of other features in *Timon*. Since these features are independent of the data distribution, we use RDS performance data as the test set solely in these experiments.

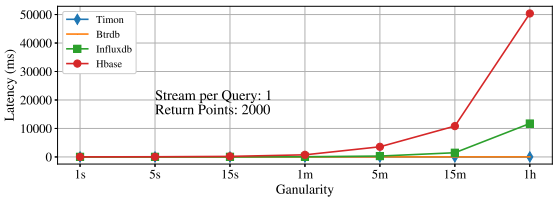


Figure 17: Query latencies between databases as time granularity changes.

Metric Set. For Alibaba's RDS performance data set, an instance/machine record contains dozens of metric values. Usually, time series databases store these metric values separately, which causes performance degradation. *Timon* introduces the concept of Metric Set to represent the set of metrics values placed consecutively in one row. The locality of metrics values makes it easy to read and write them in an atomic operation, lowering amortized cost when the number of metrics increases.

We first evaluate the metric-set mode and single-metric mode (i.e., one metric value per row) on *Timon* and InfluxDB[3], comparing their variations in write throughput. Other databases do not support the metric set feature and thus are omitted. Figure 18 shows that in the single-metric mode, the peak throughput is about 15 million. But when we change to metric-set mode with metric size of 80, the throughput reaches 60 million. It verifies that the metric set significantly improves the throughput in *Timon*. However, the write throughput of InfluxDB does not increase much when we open the field set feature, because it is only a user-friendly interface with no optimization. In the underlying storage, InfluxDB still separates each field.

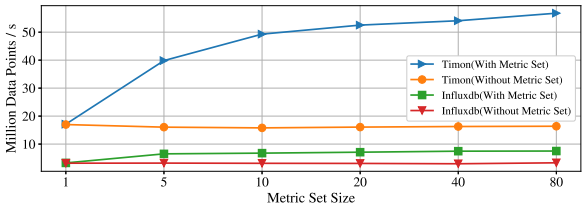


Figure 18: Write throughput as metric set size increases.

Table 1 shows the overhead of metric set in the query phase. We observe that there is not an obvious read amplification when we read a single metric from a metric set, compared with metric-set-off. The latency of the two cases is comparable, and we can infer that the overhead of metric set is negligible.

[3]InfluxDB has the field set feature that seems to be similar to metric set.

Table 1: Overhead of Query with/without Metric-Set when 30 instances' metric were selected for each query. (time range: 1 hour, granularity: 1s, metric set size: 36)

Latency [ms]	Metric-Set On	Metric-Set Off
Query	134.4759	132.4905

Materialized Aggregate View. In the production environment, we often have the requirement of aggregating data on a higher abstract level, e.g., the region level. Most of the time, a region contains thousands of instances, we have to aggregate data from all instances in the query phase, if we only maintain the data at instance level in the database. *Timon* provides the materialized aggregate view to resolve this problem. It pre-computes the aggregated values for levels that users have defined in advance, e.g., one can define *host*, *user*, and *region* views for RDS performance data if wants to query these aggregate results directly from *Timon*.

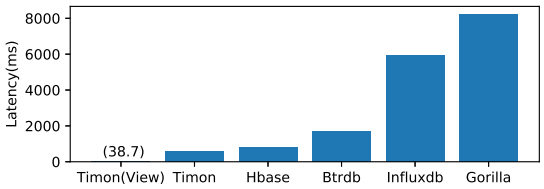


Figure 19: 100-streams query latencies between *Timon* (View On/Off) and other databases.

The first experiment shows how much query performance is improved by the materialized aggregate view. Figure 19 shows that the query latency of *Timon* with a materialized view is much lower than *Timon* without the view and other databases. It is because materialized aggregate views help to read aggregate results directly, instead of reading all data points and aggregating them at runtime.

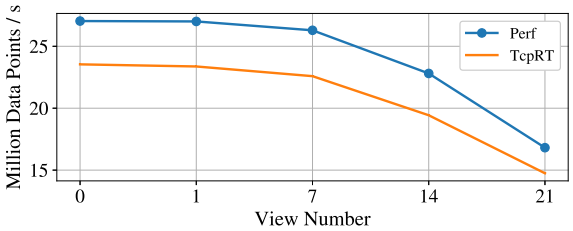


Figure 20: Write throughput as view increases.

The second experiment shows that the write throughput is not much affected unless more than seven (should be enough in most cases) materialized views are created, as shown in Figure 20. The primary reason for performance degradation is the saturation of the machine resource capacity. Hence,

adding additional resources can help if more views are demanded.

7 RELATED WORK

Data Processing Systems. Batch systems such as MapReduce [17] and MicroBatch systems like Spark [26] suffer from high latency problems, because newly arrived records are buffered to be processed at a future time. Streaming processing systems like Storm [11] and Samza [22] lack the ability to provide the exactly-once semantics, i.e., unable to guarantee correctness. Lambda architecture [21] combines the methods of batch processing (the batch layer) and steaming processing (the real-time layer) together as a hybrid approach: the batch layer generates comprehensive, reliable yet dated results, while the real-time layer provides immediately available but less complete and accurate results.

Summingbird [13] is a unified framework for batch and stream processing, which offers a higher level domain-specific language to unify the underlying execution environments. However, similar to the lambda architecture, its correctness is not guaranteed.

MillWheel [1] ensures that records are delivered exactly once between computational nodes through checking and discarding duplicated records against checkpoints. Out-of-order data is handled with low watermark in MillWheel. The skew between wall time and low watermark timestamp determines the latency of overall results, i.e., usually they should be quite close. However in practice, there may be a small ratio of late records that arrive behind the low watermark. Hence, if users want 100% accuracy, they must handle late data in other pipelines and correct aggregates. Although MillWheel implements exactly-once delivery, the programming model itself does not guarantee correctness without performing user-defined actions to handle late data properly.

The Dataflow Model [2], which is based on MillWheel and FlumeJava, further supports incremental processing of out-of-order data in that programming model. It defines triggers as a complement to the windowing model, which allows to trigger output results for a given window and make refinement of previous results when later data arrives. Upon triggering, earlier results are read from external persistent databases, merged with newly arrived data, and overwritten by updated results (i.e., a Read-Modify-Write operation, which is less efficient than a blind write for databases like HBase). When the proportion of out-of-order data increases in the system, database throughput drops and system performance is impacted.

Time Series Databases. InfluxDB [20] is a popular time series database (TSDB) built from scratch. It uses a Time-Structured Merge Tree (TSM) as its core index structure, which is very similar to LSM-tree. The LSM-like structure helps InfluxDB to support massive-stream writes, but unlike *TS-LSM-Tree*, TSM lacks fast long-term time-series exploration and does not optimize for incremental processing, which is very important for efficient blind-write. Although InfluxDB supports the metric set feature, it stores each metric separately.

OpenTSDB [10] is another open source TSDB that is often used in production environments. It uses HBase [8] as the storage backend, and designs the storage model of time series data based on HBase. In many cases, HBase is also used directly to store time series data based on different custom data models. HBase's index structure is LSM-tree, and hence the problems of OpenTSDB and HBase are similar to those of InfluxDB.

Gorilla [24] is an in-memory TSDB. It supports massive streams with high-throughput writes. But it does not permit out-of-order events and lacks fast long-term streams exploration.

BtrDB [6] is a state-of-the-art TSDB that supports long-term time-series exploration through an innovative time-partitioned tree. However, BtrDB is designed for ultra-high frequency data points (i.e., sub-microsecond precision timestamps). In order to ensure high throughput, each write needs to write batches of data points from one stream, and the batch size is 10,000 in the evaluation of BtrDB paper. In our scenario, there are massive streams, but the frequency is one point per second, which cannot be well handled in BtrDB.

In addition, these TSDBs cannot support incremental processing after a data point is written, which is important for efficient implementation of blind write.

8 CONCLUSIONS

Timon is a timestamped event database that has been developed and deployed at Alibaba Cloud. It is optimized for heavy blind writes and analytic queries in an incremental processing system and can handle out-of-order data correctly and efficiently. The correctness is guaranteed by the associative and commutative properties of operators on data structures inside *Timon*. With the design of the architecture and *TS-LSM-Tree*, *Timon* can support low latency queries and fast long-term time series exploration, even in the case of massive stream writes and massive out-of-order arrivals. Besides, *Timon* provides rich features and an expressive query language TQL for analytic tasks, like correlation analysis, and anomaly detection. We have rebuilt the continuous monitoring and anomaly analysis infrastructures and applications like TcpRT (the performance monitoring system of Alibaba RDS and PolarDB) on top of *Timon*, and have gained a lot of benefits from its features.

A APPENDIX

A.1 Network Anomaly Detection

Part 1

```
select sum(sum.pktloss) as src_pktloss from tcprt_graph group by src, dst
   window by 60s period 60s interval 60s
| select sum(src_pktloss > 0) as src_pktloss_node_cnt, sum(1) as src_node_cnt
   group by src as ip window by 60s
| join "select sum(sum.pktloss) as dst_pktloss from tcprt_graph
        group by dst, src window by 60s when %s, %s
      | select sum(dst_pktloss > 0) as dst_pktloss_node_cnt, sum(1) as dst_node_cnt
        group by dst as ip window by 60s ", _start_time, _start_time + 60s on ip
| select sum(src_node_cnt) + sum(dst_node_cnt) as total,
        sum(src_pktloss_node_cnt) + sum(dst_pktloss_node_cnt) as count
  group by ip window by 60s
| send_to_channel "net_analyze_task"
```

Part 2

```
cascade "net_analyze_task"
| joinmeta from external_DB_meta with
  "select ip, tor_switch_pair from meta_table" on ip refresh_period 300s
| select sum(total) as total, sum(count) as count group by tor_switch_pair
| top 10 of pow(count, 1.5) / total
| alarm "An anomaly occurred on switch:" + tor_switch_pair + " at time:" + _time
  to "DBA_group" via "SMS"
```

Figure 21: Network Anomaly Detection

Figure 21 shows the query in two parts. The first part computes the packet-loss extent of each network activity. For each vertex, to count its total connected nodes and ones suffering from packet-loss, a *join* operator is used to merge packet-loss information from the head end and tail end in the directed network graph. The processed data is sent to the middleware named *Channel*, which allows other tasks to consume data from it. The second part aggregates to derive the most abnormal machines/switches. The second query reads data from Channel, joins metadata to expand raw tags, then aggregates based on switch level, and finally computes the top 10 faulty switches. The *cascade* operator reads from a certain Channel; it allows data to be reused for different jobs. In this case, to find troubled machines, switches, and racks, we create three queries, all of which consume data from the same basic-level Channel, for machine-level, switch-level, rack-level aggregation respectively.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. We would also like to thank the monitoring and full-state tracking team (Zuorong Xu, Lingyun Li, Yu Yu, Yunlong Mu, Yintong Ma, Jiabang Pan, Bowen Cai, Zhe Wang, Jing Li, Yongshuai Li, Mengjie Jin), who are our users from the very beginning of *Timon* and gave us much precious advice.

REFERENCES

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[3] AlibabaCloud. Loghub. https://www.alibabacloud.com/product/log-service.

[4] AlibabaCloud. Polardb. https://www.alibabacloud.com/products/apsaradb-for-polardb.

[5] AlibabaCloud. Rds. https://www.alibabacloud.com/product/apsaradb-for-rds-mysql.

[6] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *FAST*, pages 39–52, 2016.

[7] Apache. Cassandra. http://cassandra.apache.org/, 2008.

[8] Apache. Hbase. https://hbase.apache.org/, 2008.

[9] Apache. Kafka. https://kafka.apache.org/, 2011.

[10] Apache. Opentsdb. http://opentsdb.net/, 2011.

[11] Apache. Storm. https://storm.apache.org/, 2017.

[12] AWS. Kinesis. https://aws.amazon.com/kinesis/.

[13] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13):1441–1451, 2014.

[14] W. Cao, Y. Gao, B. Lin, X. Feng, Y. Xie, X. Lou, and P. Wang. Tcprt: Instrument and diagnostic analysis system for service quality of cloud databases at massive scale in real-time. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 615–627, New York, NY, USA, 2018. ACM.

[15] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 29–41, 2020.

[16] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Facebook. Beringei. https://github.com/facebookarchive/beringei, 2017.

[19] P. Flajolet, E. Fusy, O. Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[20] influxdata. Influxdb. https://github.com/influxdata/influxdb, 2013.

[21] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *IEEE Big Data*, pages 2785–2792, 2015.

[22] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[25] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.