ByteSeries : An In-Memory Time Series Database for Large-Scale Monitoring Systems

Xuanhua Shi, Zezhao Feng, Kaixi Li
Huazhong University of Science and Technology, China
{xhshi,lyfone,kaixili}@hust.edu.cn

Yongluan Zhou
Department of Computer Science, University of Copenhagen, Denmark
zhou@di.ku.dk

Hai Jin, Yan Jiang
Huazhong University of Science and Technology, China
{hjin,harryjy_hust}@hust.edu.cn

Bingsheng He
National University of Singapore, Singapore
hebs@comp.nus.edu.sg

Zhijun Ling, Xin Li
Bytedance Inc., China
{lingzhijun,lixin.sin}@bytedance.com

ABSTRACT

Monitoring large-scale and complex systems often generates high -dimensional and highly dynamic time series data. In such a scenario, massive metadata has to be maintained to support efficient querying, whose large footprint poses great challenges to in-memory databases. In this paper, we present ByteSeries, an in-memory time series database that is designed specifically for large-scale monitoring systems to manage high-dimensional time series. We start with an analysis of the production data and workload at ByteDance's metric monitoring system, which contains over 10 billion time series dimensions. The observation of high overhead of metadata management in high-dimensional time series data calls for a rethink of time series database systems. ByteSeries's memory structure employs the novel Compressed Inverted Index to effectively compress metadata while maintaining high efficiency for multi-dimensional queries. In addition, an algorithm is proposed to effectively convert data into compressed form without sacrificing the data ingestion throughput. We experimentally evaluate ByteSeries by comparing it with ByteDance's original production system, tsdc, as well as two open-source systems, namely Gorilla and Prometheus. We show that ByteSeries significantly improves over ByteDance's original production system by 1) reducing the memory footprint of metadata by 60% and the whole memory consumption by 50%, and 2) speeding up multi-dimensional queries by 1.8x-10.7x.

CCS CONCEPTS

• **Information systems → Data layout**; **Record and buffer management**.

KEYWORDS

monitoring system; time series database; metadata compress

ACM Reference Format:
Xuanhua Shi, Zezhao Feng, Kaixi Li, Yongluan Zhou, Hai Jin, Yan Jiang, Bingsheng He, and Zhijun Ling, Xin Li. 2020. ByteSeries : An In-Memory Time Series Database for Large-Scale Monitoring Systems. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421289

1 INTRODUCTION

With the ever-growing scale and complexity of internet services, systems have scaled beyond hundreds of machines to thousands of machines. Monitoring large-scale machines, systems and services have become a great challenge. It needs to continuously collect many different data, for example the CPU usage of a particular host, or the delay of an API call. Efficiently managing such monitoring data and providing real-time query services is a critical requirement.

Data in a large-scale metric monitoring system can typically be modeled as high-dimensional time series, with each dimension representing a unique sequence of measurements. For example, the monitoring system of ByteDance's internet

*Shi, Feng, Li, Jin and Jiang are from National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab. This work is partially supported by NSFC under grant No. 61772218.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421289

services has more than ten billion of active dimensions on every day, which is doubling every year. Furthermore, with the technological trends such as container and microservices, massive new time series tend to be generated and disappeared dynamically at runtime. For instance, dynamically starting new containers or microservices would produce new time series dimensions identified by new keys. This can be triggered frequently by short-term tasks, such as running one MapReduce job, or reconfiguration of system deployments. Such dynamic behaviors of systems further inflate the number of time series dimensions.

The high dimensionality and dynamicity of time series data in a large-scale monitoring system poses great challenges to time series database systems. To support high-throughput data ingestion and real-time querying, a viable option is to use an in-memory time series database to cache the recent data and to execute real-time queries. Memory storage efficiency is critical to such a solution. However, in a monitoring system, the key of a dimension is typically verbose, consisting of, for instance, a metric name and a set of tags in the forms of key-value pairs. Memory consumption of such metadata could be even higher than time series data points per se, especially when there are a large number of dynamic dimensions, which appear and disappear within a relatively short period of time. Moreover, efficient indexing on metadata is needed to support efficient querying in high-dimensional and dynamic time series, which, however, could further increase memory consumption of metadata. In ByteDance's production system, we observe that metadata occupies more than 80% of the total data size. Designing efficient approaches to compress the metadata without sacrificing ingestion throughput and query efficiency is yet to be addressed by existing time series database systems.

In this paper, we present ByteSeries, an in-memory time series database system specifically designed for large-scale monitoring systems. It provides in-memory caching and querying of high-dimensional and dynamic time series. The salient features of ByteSeries include aggressively compressing metadata to minimize the memory footprint, while at the same time supporting efficient multi-dimensional group-by aggregate queries, which are the most important types of queries in many time series data applications, including ByteDance's metric monitoring system. The contributions of this paper include the following.

First of all, we conduct an analysis of data and query workload on ByteDance's production system. Our analysis shows the data and query characteristics in industrial production systems that can be applied to many scenarios. Based on the analysis, we identify challenges that have not been sufficiently addressed by existing time series database systems.

Second, we propose efficient memory data structures for ByteSeries. ByteSeries organizes memory into two parts, namely *Active Buffer* and *Static Buffer* respectively. *Active Buffer* is designed to efficiently ingest high-throughput time series with dynamic data structures. No compression is carried out on the data in *Active Buffer*. *Static Buffer* is the main storage area. ByteSeries adopts a novel Compressed Inverted Index to compress metadata and to support efficient multi-dimensional aggregate queries at the same time. And in order to maximize data ingestion throughput while keeping efficiency of memory usage, we design an algorithm to schedule the conversion and movement of data from *Active Buffer* to *Static Buffer*.

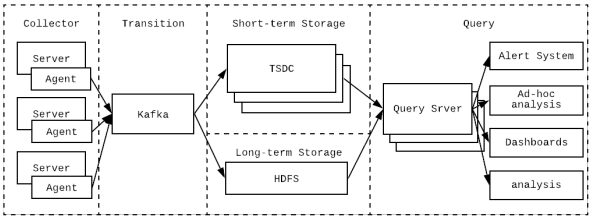


Figure 1: High level overview of the ByteDance monitoring system

Finally, we conduct extensive experiments to verify the performance of ByteSeries. We first examine the performance of different components in ByteSeries, and then compare ByteSeries with not only ByteDance's own production system, but also two open-source in-memory time series databases, namely, Gorilla and Prometheus. We also evaluate the performance of ByteSeries over ByteDance's production workload. The results show that ByteSeries significantly outperforms the existing solutions by achieving smaller memory footprint while maintaining higher data ingestion throughput, as well as lower query response time.

2 PRELIMINARIES

2.1 ByteDance Metrics Monitoring System

ByteDance's business services are supported by a large infrastructure comprised of thousands of node instances distributed across multiple data centers. ByteDance's metrics monitoring system is used to monitor not only the system infrastructure, but also various business services and web back-end services. Figure 1 shows a high-level overview of ByteDance's metric monitoring system that comprises of multiple data collectors, a time series database for short-term storage, a long-term storage and a query service (for an alert system and so on). In particular, the in-memory storage, called *tsdc*, is the core component, which stores the most recently collected measurements in main memory, and regularly checkpoints them to the local storage. It also provides

series key		data point	
series name	tags	timestamp(ms)	value
mem_usage	local=wuhan,os=linux,id=1	1580540293000	0.3
mem_usage	local=shanghai,os=win,id=2	1580540293000	0.4
cpu_usage	local=wuhan,os=linux,id=1	1580540293000	0.7
cpu_usage	local=shanghai,os=win,id=2	1580540293000	0.6

Figure 2: Series data format

high bandwidth and low-latency querying. In this paper, we identify the efficiency problems of *tsdc* and design ByteSeries with much higher performance to meet the business requirements of ByteDance.

The data in ByteDance's metric monitoring system are modeled as high-dimensional time series as described below.

Data Point: A data point is a pair of *timestamp* and *value* in the form of (t, v).

Series Key: A series key consists of a metric name and a list of tags in the form of $\langle name, tags \rangle$. Each item in the tag list is a key-value pair. A series key uniquely identifies a time series, and we call it a dimension.

Figure 2 shows some time series data. In the first line, *mem_usage* is the metric name, $\langle$*(local, wuhan), (os, linux), (id, 1)*$\rangle$ is a list of tags, and (158054029300, 0.3) is the data point (*dp*).

Time Series: A dimension of time series is a sequence of data points (*dps*) with a particular series key (*SK*), which are ordered by time in ascending order. Formally, it can be denoted by $\langle SK, \langle (t_1, v_1), (t_2, v_2), ..., (t_n, v_n) \rangle \rangle$, where $t_{i+1} > t_i$.

A single-dimension query (or exact query) is defined as a query whose selection conditions match all the tags of a single time series. On the other hand, a multi-dimension query is defined as a query whose selection conditions contain only some but not all of the tags in any dimension.

We use the data shown in Figure 2 to show some example queries. *SELECT mem_usage FROM mydb WHERE local = wuhan AND os = linux AND id = 1 AND time ≥ '2020-2-1 12:00:00'* is a single-dimension query, and *SELECT mem_usage FROM mydb WHERE local = wuhan AND time ≥ '2020-2-1 12:00:00'* is a multi-dimension query. Typically, most queries have predicates specifying time ranges.

Because time series have high degree of temporal correlation, we usually adopt time slicing, e.g., two hours per slice, to store and index them.

2.2 Analysis of ByteDance's Workload

In order to clarify the requirements and challenges of ByteDance's monitoring system, we study the workload in a cluster, consisting of about 1,200 nodes. The cluster is used to manage time series data with over 10 billion dimensions. These time series data are mainly monitoring data of ByteDance's internal cluster and user business activities. The data generated by this workload can reach tens of terabytes

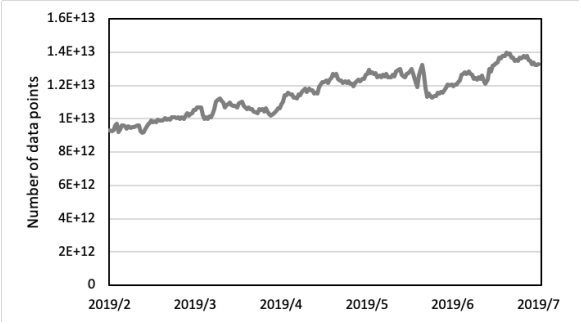


Figure 3: Statistical graph of the total number of data points per day from February to July 2019

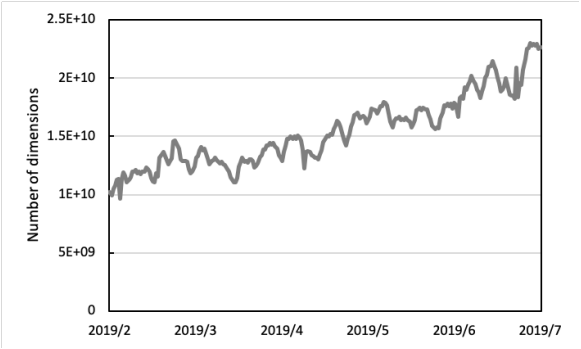


Figure 4: Statistical graph of the total number of dimensions per day from February to July 2019

per day. Figure 3 and Figure 4 show a complete data set on one randomly selected node instance of the monitoring system, which contains nearly 40 million dimensions of time series. From the analysis, we have the following interesting observations.

1). The dimensions are growing rapidly and have high dynamicity. Figure 3 and Figure 4 illustrate the growth trends of data points and dimension number from February to July 2019. First of all, with the development and expansion of online services, most of the newly added time series are recorded as irregular events without a fixed time interval. Only a small number of dimensions have data points with a fixed time interval, e.g. CPU usage of a host. In the figure, the number of dimensions has grown more than twice, while the number of data points has only increased by about 40%. Second, almost 1/3 of the dimensions produced per day are new, which is due to, for example, frequently changes of container deployment of microservices, and short-lived tasks. Since such dynamic dimensions often have very few data points, maintaining hash indices, compression contexts and other metadata for them in main memory is resource inefficient.

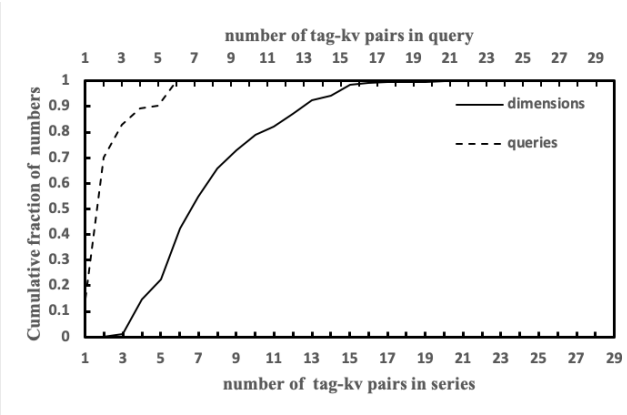


Figure 5: Cumulative statistic graph of query tag conditions and overall time series with tag quantity.

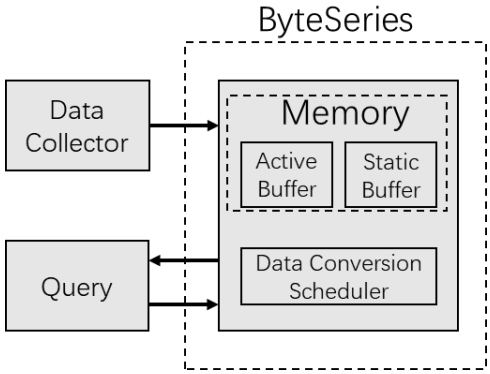


Figure 6: The overview of architecture.

2). Queries are mostly multi-dimensional with filter and aggregation conditions. In real-time business, we observed that users of monitoring systems tend to be more interested in multi-dimension aggregate analysis rather than querying individual data points and individual dimensions. Most workloads are for alarm monitoring and updating real-time dashboards. These queries often only query the data within the latest few hours, and usually involve aggregate on multiple dimensions. Drilling down to a specific piece of abnormal data only happens when the aggregate query results show some abnormality. Therefore, supporting efficient multi-dimension query for the most recent data is important for enhancing the system's performance. As shown in Figure 5, there are no more than 3 *tag-kv* pair condition filters in more than 80% of the queries, however, almost all of the dimensions have more than 3 different *tag-kv* pairs. This means that there are often more than one dimension that can satisfy the corresponding query conditions.

2.3 Motivations

The challenge of storing recent data in memory is achieving good trade-off between memory footprint and read/write performance. The analysis in Section 2.2 reveals that it is paramount to address the issues of metadata posed by high dimensionality of time series in memory. With the existence of efficient compression algorithms for time series data points, e.g. the ones proposed in Gorilla [18], the memory footprint of metadata becomes even more prominent. At the same time, in order to ensure high query performance, we need to build in-memory index for meta data, which however would negatively affect memory consumption and ingestion throughput.

Based on the above objectives and challenges, we design an in-memory time series database, ByteSeries. It can be used as an in-memory cache for disk storage solutions and provides high-throughput ingestion and low-latency querying. We propose a multi-level cache design that implements different metadata compression schemes to balance the performance of memory consumption and read/write throughput, which will be described in detail in the following sections.

3 SYSTEM DESIGN

In this section, we present the design of ByteSeries, which is designed to replace ByteDance's *tsdc* to manage ByteDance's tens of billions dimensions of time series.

3.1 Overview of the Architecture

As mentioned earlier, ByteSeries is an in-memory cache that can be placed in front of a disk-based time series database system to provide high-throughput data management. Since ByteSeries solely relies on main memory to store data, memory footprint is of utmost importance. ByteSeries adopts Gorilla's compression algorithm to efficiently compress data points. However, the memory footprint of metadata is very significant with ByteDance's production dataset. Therefore, in order to effectively reduce the memory footprint, we design and implement a whole set of mechanisms to efficiently compresses the metadata while maintaining indices to speedup queries. To avoid the ingestion throughput affected by the overhead of data compression, we adopt a segmented memory approach to smooth out the effect.

Figure 6 depicts ByteSeries's overall architecture. We present two major components of the entire design: 1) memory layout and data structures, 2) a data conversion scheduler that dynamically coordinates the execution of data conversion between different memory segments.

Data collected by the *Data collector* is first written to the *Active Buffer*, which is designed to ensure high ingestion throughput. In *Active Buffer*, a simple data structure, such

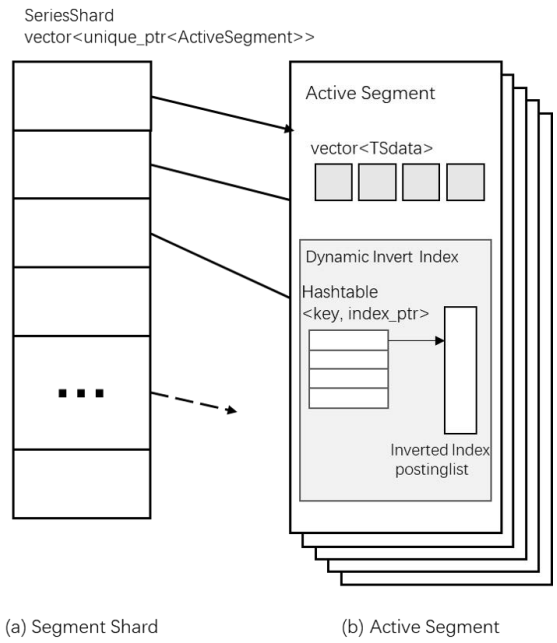


Figure 7: The in-memory data structure of *Active Buffer*. We divide the *Active Buffer* into multiple shards (a), and each shard has its own data segment called *Active Segment* (b).

as hash table and append-only array is used to meet the requirement on high ingestion rate. No data compression is performed within this buffer, so it is desirable that it only occupies a small portion of memory to achieve low memory footprint. *Static Buffer* stores the compressed data to minimize the memory footprint, as well as the novel Compressed Inverted Index to speed up query processing. Each block of data from *Active Buffer* will be first compressed and converted to the data format of *Static Buffer*, and then merged with the existing data stored in *Static Buffer*. To amortize the overhead of compression and merging, this process is done in a batch manner. A scheduler is responsible for coordinating the execution of the process, including the data conversion from *Active Buffer* to *Static Buffer*, the compression of metadata inside *Static Buffer* and so on.

3.2 Memory Layout & Data Structures

3.2.1 Active Buffer. Data arriving at ByteSeries would be first stored in *Active Buffer*. As mentioned before, in order to maintain high-throughput data ingestion, this does not involve any complicated data conversion or compression. Figure 7 shows the data structure of *Active Buffer*. To achieve scalability, *Active Buffer* is sharded into multiple segments, called *Active Segments (AS)*, each of which is treated independently. Data is placed in different segments according to the hash value of the series key. An *AS* consists of an append-only array and a dynamic inverted index. Data are appended into the array and inserted into the inverted index to accelerate query.

The reasons for sharding are twofold. First of all, sharding allows data ingestion be executed in parallel on different segments to achieve higher data throughput. Second, to convert data from *Active Buffer* to *Static Buffer*, it is necessary to pause the ingestion and then re-allocate new segment for *Active Buffer*, after which ingestion can be resumed. The original memory space of the *Active Buffer* can only be released/reused after the data conversion is completed. If we perform this operation over the entire *Active Buffer*, the memory overhead would be at least as large as the size of *Active Buffer*. This overhead could be even higher if data conversion temporarily falls behind data ingestion. With sharding, data conversion is done on the segment level, therefore the memory overhead would only be the size of a few segments, especially during the normal situation where data conversion can keep up with data ingestion.

3.2.2 Static Buffer. *Static Buffer* is designed to achieve both high memory efficiency and high query performance. Through the data analysis in Section 2.2, we observed that although there are over 10 billion dimensions in the whole system, more than 99% dimensions have less than 15 *tag-kv* pairs, and there are a large number of dimensions share the same tag keys. The whole dataset can be represented by a combination of hundreds of thousands of metrics and tens of thousands of tag combinations under each metric. Based on such data characteristics, we propose Compressed Inverted Index, a novel data structure to achieve not only data indexing, but also metadata compression. As shown in Figure 8 (c), Compressed Inverted Index employs a *trie*[4] to store the key to serve as an inverted index. Furthermore, integer compression is applied to compress the array storing the offsets of the data points.

To limit the impact of the compression overhead on ingestion throughput, we further divide *Static Buffer* into two parts, which is shown in Figure 8. The first part consists of the so-called *Static Segment (SS)*. Time-series data points in SS are stored in an append-only array. Newly added data points are directly appended into the array without compression, and their offset are also directly appended to the inverted index. The simple append operations incur only minimum overhead so that converting data from a *AS* to an SS can keep up with the speed of data ingestion to *Active Buffer*. In order to facilitate parallel processing, each *AS* is first converted into a *Temporary Static Segment (TSS)* in parallel. Then one or more TSSs would be merged into *SS*.

In *SS*, both the metadata and data points are not stored in a compact form. Data points with the same series key

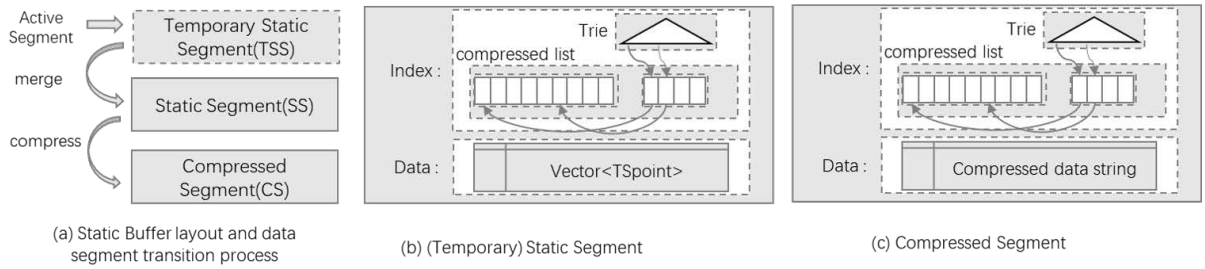


Figure 8: *Static Buffer*'s memory storage design. There are two different data segment structures in *Static Buffer*: (b) *Static Segment (SS)* and (c) *Compressed Segment (CS)*

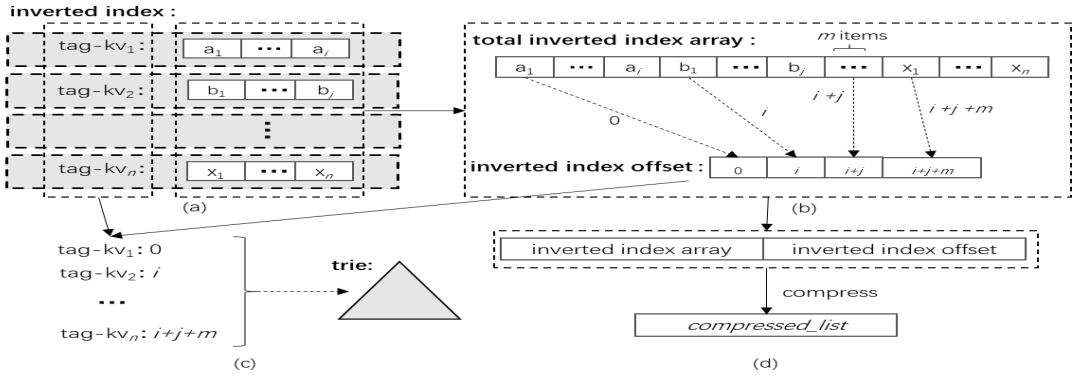


Figure 9: The building process of Compressed Inverted Index

are scattered around in the data point array rather than storing in a contiguous space. This would cause not only that the inverted index has to store more offsets of data points, but also that the compression of data points would be less effective. Therefore, we further convert a *SS* into a so-called *Compressed Segment (CS)* to achieve higher memory efficiency. In *CS*, all the data points with the same series key are stored contiguously, which can achieve more effective compression and faster query operations due to less data lookup. Furthermore, only one data offset is needed for each series key to be stored in the Compressed Inverted Index, so *CS* has much smaller memory footprint.

The purpose of converting data first to *SS* then to *CS* is to batch the expensive compression operations to amortize the overhead. In order to avoid that *SS* takes up a large amount of memory, we also need to limit its size. So when the data in *SS* have accumulated to a certain amount, the data conversion scheduler will schedule the process of compressing and merging data into *CS*.

Listing 1: The data struct of Compressed Inverted Index

```
Struct CompressedInvertedIndex {
    Trie      trie_map;
    char*     compressed_list;
    size_t    compressed_list_size;
    size_t    trie_map_size;
    size_t    index_size;
}
```

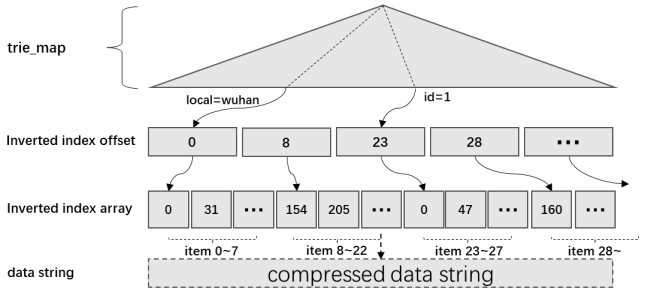


Figure 10: One example of Compressed Inverted Index using in querying

Listing 1 shows the implementation of Compressed Inverted Index, in which *trie_map* is a prefix tree for compressing *tag-kv*s, and *compressed_list* is inverted index arrays after compression consisting of *Inverted Index Array* and *Inverted Index Offset*. The *compressed_list_size* records the compressed

list size, *trie_map_size* is the number of inverted indices, and *index_size* is the total size of the index. We elaborate on the building process of the entire index with an example in Figure 9.

When building the Compressed Inverted Index, the first step is to build inverted indices for all *tag-kv* pairs (as shown in Figure 9 (a)). Then we compress them in two aspects: inverted index keys and inverted index value arrays. For the value arrays, we concatenate them into one array (shown in Figure 9 (b) *total inverted index array*), and then use another array (Figure 9 (b) *inverted index offset*) to record the starting position of the values of each inverted index array in the concatenated array. At the same time, the value corresponding to each key (*tag-kv*) in the original inverted indices is changed to the position recorded (as presented at Figure 9 (c)). Finally, we compress the keys using a prefix tree (get result *trie_map* shown in Listing 1). We implement it using a C++ library called *cedar*[3], which has shown higher compression rate, and faster construction and search speed in comparing with other libraries. Then, we compress the *total inverted index array* and the *inverted index offset* (*compress_list* in Listing 1) using the *p4nzenc64* [25] algorithm and recording their sizes.

3.2.3 Querying over Compressed Inverted Index. We use an example to illustrate how the Compressed Inverted Index can be used in query execution. Figure 10 shows a query: *SELECT mem_usage FROM mydb WHERE local=wuhan AND id=1 AND time≥'2020-2-1'*.

The first step of query processing is to analyze the time range. If there exists no data in the time range, it will return immediately. Otherwise, it proceeds by looking up the index for each *tag-kv* condition. For each *tag-kv* condition of *local=wuhan* and *id=1*, the offset in the inverted index position is retrieved from the *trie_map* (0 and 23 in this case). The query immediate returns with an empty result if the index lookup returns empty. Otherwise, we decompress the *compressed_list* into *inverted index array* and *inverted index offset*, and then obtain the position of the inverted index from *inverted index offset*. We can then read the inverted index from the array, which are ⟨*0,31,...*⟩ *and* ⟨*0,47,...*⟩, and perform the intersection to get the final result, which is ⟨*0*⟩. Finally we read and decompress the data to get the target data with the result ⟨*0*⟩. Time-based group-by operations and/or other aggregate functions can then be performed.

3.3 Data Conversion Scheduler

The *Data Conversion Scheduler (DCS)* controls the data conversions between different data segments to ensure high ingestion throughput. *DCS* consists of three processors (*Compactor*, *Merge Processor* and *Compressor*), which correspond to three major data conversion tasks: 1) *Compactor* compacts an *AS* into a *TSS*; 2) *Merge Processor* merges one or more *TSS* with *SS*; 3) *Compressor* compresses a *Static Segment* into *Compressed Segment*.

Algorithm 1 outlines the execution strategy of *DCS*. *DCS* runs in a separate thread and schedules the three tasks (*Compactor*, *Merge Processor* and *Compressor*) according to predefined rules. *Scheduler* is the core function that performs the scheduling. A Boolean variable, *stop*, is used by the system to control the shutdown of the process. *DCS* can be shut down by setting *stop* to be *True* instead of being killed abruptly, so as to ensure its executing task is completed.

As mentioned earlier, we limit the size of *Active Buffer* to reduce the memory footprint. Once the amount of data in an *AS* in the *Active Buffer* exceeds a threshold, it would be converted into a *TSS* and transferred to *Static Buffer*. Since we target the scenario with high-dimensional time series data, where the metadata in *AS* occupies most of the space, we use the relative number of dimensions stored in an *AS* as an estimation of the size of *AS*. More specifically, the threshold *dimen_limit* (line 6) represents the percentage of the total number of dimensions that are stored in *AS* (line 12). This is a tunable parameter, and its effect is explored in the experimental section. In Algorithm 1, we use *CheckProcessor* to check whether the conditions are met (lines 27-28), and this value is updated (lines 31-34) whenever *compressor* is executed. In the meantime, there might be an issue: even though the number of dimensions in *AS* is low, the number of data points is too high so that the size of *AS* is large. In order to prevent this issue, we set the second threshold *max_size*, i.e. the maximum size of *AS*, in line 8. We check this condition once for each round of task execution (lines 18-21).

In order to amortize the overhead of compressing *SS*, to avoid redundant metadata storage, and to speed up the execution of queries, we exploit opportunities to merge multiple *AS* into *SS* and compress them all at once. To do so, we first convert each *AS* into a *TSS*. When the condition of converting *AS* is not met, *CheckProcessor* checks whether there is at least one *TSS* that can be merged into the existing *SS* (lines 29-30). If so, merging would be executed.

Whenever neither of the conditions for converting *AS* and merging *TSS* to *SS* are met, we would check if we should further compress the current *SS* and incorporate it into *CS*. This would happen if *SS* exceeds a given threshold (line 9). Here we use the average number of data points per dimension in *SS* as the condition (lines 31-35). We further discuss the effect of this threshold on system performance in the experimental section.

4 BENCHMARK EXPERIMENTS

We implemented ByteSeries in C++. In this section, we evaluate the data compression and conversion methods using microbenchmarks. Specifically, we first evaluate the memory

usage and performance of the data structures of the three types of memory segments that we proposed. We then evaluate the performance trade-off and parameter choices made by the proposed data compaction and merging strategies. Lastly, we compare the full ByteSeries implementation performance to other existing TSDB to verify the performance.

In next Section (Section 5), we examine the performance of ByteSeries as the time series database service with the production workload of ByteDance's metric monitoring system. We replace the original tsdc with ByteSeries and show the performance improvement.

4.1 Experiment Setup & Benchmark

The experiments were ran on a machine with two eight-core Intel(R) Xeon(R) Platinum 8269CY CPUs at 2.5GHz, 64GB memory, and a 512GB HDD disk. The operating system is Ubuntu 18.04 with Linux kernel version 4.15.

To simulate time series workload, we use Time Series Benchmark Suite (TSBS) [24], a collection of tools to generate data sets and benchmark read and write workloads. We modified TSBS to generate microbenchmarks to measure memory usage, ingestion performance and query performance. We focus on the DevOps use cases that simulate what a system administrator would see when operating a cluster of hundreds or thousands of virtual machines. We scale the number of metrics and hosts to generate different scales of time series. The length of each time series name is about 197 bytes with 10 labels on average, and 20 bytes per label pair on average. We use only the data set and query model of TSBS, without using its data loading and query method.

In all the experiments, we write data first and then execute different queries. We mainly evaluate two types of queries: Single Groupby and Double Groupby. Each Simple Groupby query returns one metric of 1 host, and aggregate the time series for every 5 mins over 1 hour of data. Double Groupby returns aggregattion across both time and host, giving the average of 1 CPU metric per host per hour for 24 hours. Each query will be executed 3 times and the average execution time is reported. All experiments are single-threaded without any network activity unless otherwise specified. Because different databases have very different data structures and write interfaces, to ensure a fair comparison, we skip the upper layer interface and directly measure the write rate of these database storage engines in memory.

4.2 Compression Effects

We analyze our data compression and merging method by first evaluating the three data structures *Active Segment* (AS), *Static Segment* (SS) and *Compressed Segment* (CS). We first compare the data structures by storing the same data and executing the same queries over them. We also evaluate the ingestion and building speed of the three data structures. This is an evaluation of the data structures alone without considering the multi-segment merging and compression process, whose effect are evaluated in the subsequent sections.

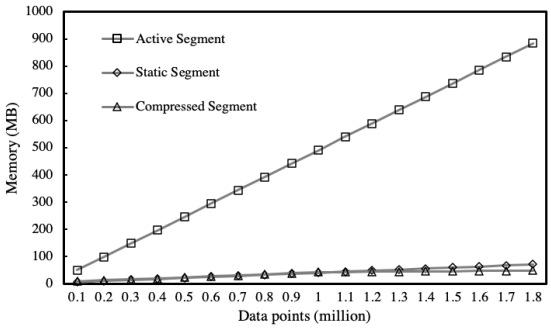
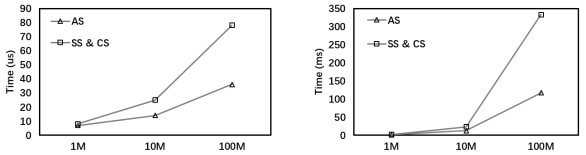


Figure 11: Memory usage of AS, SS, and CS.



(a) Single-Groupby read latency (b) Double-Groupby read latency

Figure 12: Read latency of AS, SS, and CS in different scale of workload

Figure 11 shows the memory usage of metadata in the three types of segments. The memory footprint of AS is up to 9x larger than the metadata of the original data set. This is because AS is designed for high-throughput insertion and do not consider metadata compression. Moreover, it requires additional space to store data structures for building an inverted index. Both SS and CS can effectively compress the metadata, and therefore both can achieve a much lower memory footprint. Unlike AS, SS and CS do not support support insertion, so we do not need to store the forward index and metric names. CS stores data in consecutive blocks and in a compressed form to further reduce memory usage, and does not need to reserve space or maintains locks to support modification. Hence it is even more space efficient than SS.

Figure 12 shows the latency of the two types of queries on the three data structures. The query latency for SS and CS is slightly higher than that of AS, because of the overhead of decompressing the posting lists. However, the query latency in all the cases is still quite low because queries only involve

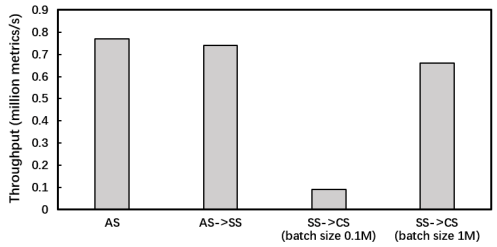
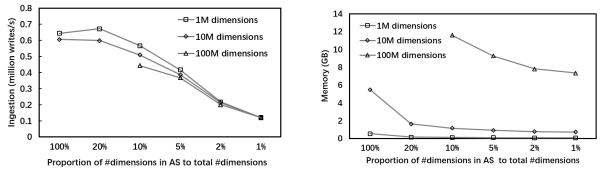


Figure 13: Ingestion speed of AS, SS, and CS.



(a) Ingestion throughput (b) Memory footprint

Figure 14: Ingestion throughput and Memory footprint of ByteSeries with different scale data set under different Active Buffer size

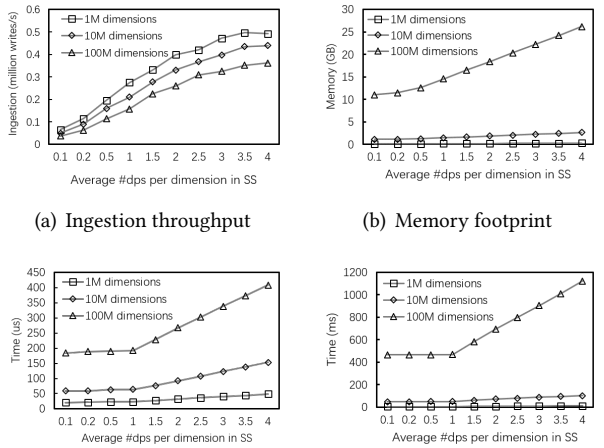
in-memory operations and the decompression algorithm is fast.

We also test the ingestion throughput of AS and the conversion speed from AS to SS and from SS to CS. As shown in Figure 13, the throughputs of AS ingestion and AS-to-SS conversion are comparable. But merging static segment into compressed segment can be the bottleneck for data ingestion if the batch size is set to be too small. With a sufficiently large batch size, the compression throughput can keep up with the AS ingestion throughput.

4.3 Merging Strategies & Overhead

We evaluate the ingestion throughput of AS with different thresholds in terms of the percentage of dimensions (proportional to the total number of dimensions) stored in the AS. Figure 14(a) shows the throughputs with different setups. Setting the AS capacity between 20% and 100% has very little effect on the ingestion rate. When it drops below 10%, the ingestion rate drops significantly. This is because the conversion from AS to SS will cause a certain overhead. Figure 14(b) also verifies that memory usage decreases as the percent decreases. Setting the capacity of AS to 10 can achieve a good balance between compression and write rate. If the demand for throughput is met, then one can reduce the threshold to reduce memory usage so that a smaller amount of resource is needed to support the same scale of data.

We use the average number of data points in every dimension as the threshold to trigger batched compression



(a) Ingestion throughput (b) Memory footprint

(c) Single-GroupBy read latency (d) Double-GroupBy read latency

Figure 15: Benchmark performance of ByteSeries with different merge strategy

and merging of SS into CS. Batching reduces the overhead of index building, compression and merging. However, it could lead to weaker compression, more redundant indices and more segments to look up for query executions. This experiment is to investigate these effects. Figure 15(a) depicts the ingestion throughput as a function of the number of data points in SS. The ingestion throughput achieves peaked at around 3.5 data points per dimension. Figure 15(b) depicts the memory usage as a function of the average number of data points per dimension in an SS. We can observe that the memory size increases with more data points in SS. Figure 15(c) and Figure 15(d) depict the query latency of the two types of queries. Query latency increases as the number of data points increases. With the threshold being 3.5, the query latency is increased by 2 times (still as low as around 100 microseconds).

In the subsequent experiments, we set the AS-to-SS conversion threshold as 0.2 and the SS-to-CS threshold as 3.5 to achieve high ingestion throughput. As a side note, these parameters can be set according the requirements of the system. If the system needs lower ingestion rate, we can reduce AS size to reduce memory requirement, or lower the threshold on the average number data points per dimension in SS to reduce query latency.

4.4 Compare with other TSDBs

In this set of experiments, we compare ByteSeries with ByteDance's original production system, tsdc, and two open-source time series database systems, including 1) Gorilla, an in-memory cache for time-series supporting efficient and effective time series compression, and 2) Prometheus, a TSBS

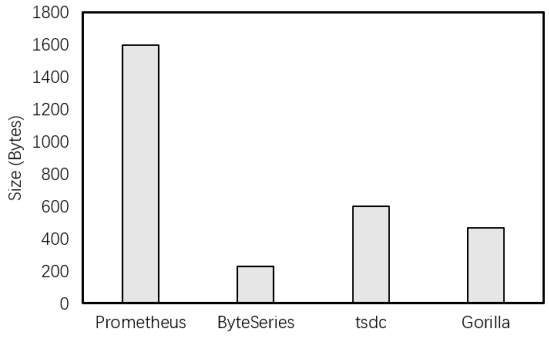


Figure 16: Comparison of memory usage per dimension between ByteSeries, tsdc, Prometheus and Gorilla

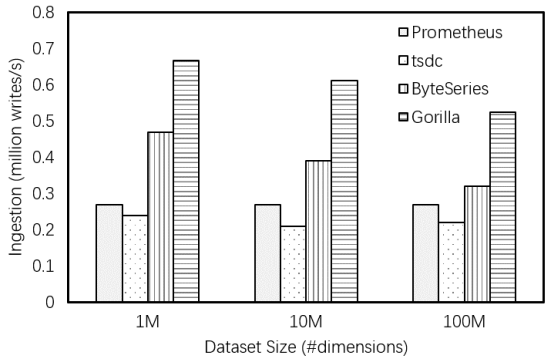


Figure 17: Ingestion speed comparison between ByteSeries, tsdc, Prometheus and gorilla with different size of dimensions

built for monitoring systems managing high-dimensional time series.

Figure 16 shows the memory consumption of the four systems. Prometheus consumes a lot of memory as it needs many data structures regarding the inverted index. Since Gorilla does not build in-memory indexes for the metadata, it has relatively low memory consumption. However, it still needs to maintain dynamic data structures for data ingestion and does not compress the metric name of each dimension. ByteDance's tsdc uses a data structure similar to Gorilla, but has more mapping layers of hashtable, so it consumes more memory. ByteSeries uses significantly less memory than others. This is because ByteSeries only allocate a small portion of memory to Active Buffer to maintain a dynamic data structure for efficient ingestion, and most of the data are stored in a more compressed form in the Static Buffer. These experimental results indicate that ByteSeries may process 7x more distinct time series dimensions comparing to Prometheus with the same amount of memory.

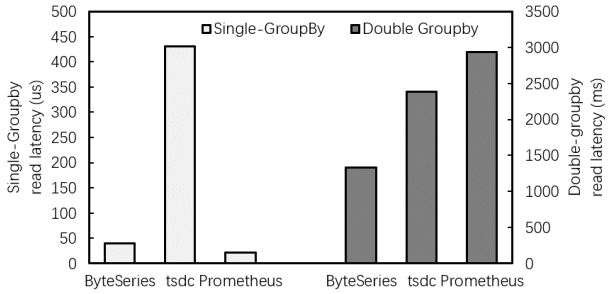


Figure 18: Query latency comparison between ByteSeries, tsdc and Prometheus

Figure 17 shows the ingestion performance. Gorilla has the highest ingestion rate because it writes data using just a simple hash lookup. Prometheus needs to build indices and performs a lot of expiration checks in the background when inserting, which limits the rate of ingestion. ByteDance's tsdc uses dictionary encoding method to compress metadata for saving memory, but requires more encoding calculations. In comparison to Gorilla, ByteSeries has some additional compression and merging operations, hence its ingestion throughput is lower than Gorilla. However, it is significantly higher than Prometheus and tsdc.

Since Gorilla does not natively support multi-dimensional queries, we only compare the query performance of Prometheus, tsdc and ByteSeries. As shown in Figure 18, the query latency of ByteSeries is slightly higher than Prometheus for single-groupby query, but the difference is only a few microseconds which is negligible in comparing to the cost of transmitting the result over the network, while tsdc takes much longer than other two due to the lack of efficient indices. For Double-groupby query, ByteSeries performs much better than Prometheus. This is because ByteSeries and Prometheus have a different query process. The inverted index of Prometheus does not directly return the aggregation group result, but returns the time series name and data points involved, and then performs group-by aggregates on the returned data. On the contrary, ByteSeries returns the time series grouping directly, and hence eliminates the time for running the grouping operations. As for tsdc, due to the fact that the Double-groupby would scan all data of one metric, the full scan schema in tsdc performs better than Prometheus which needs to search index first before scan.

5 EVALUATION ON PRODUCTION DEPLOYMENT

In this section, we perform experiments on ByteDance's production deployment.

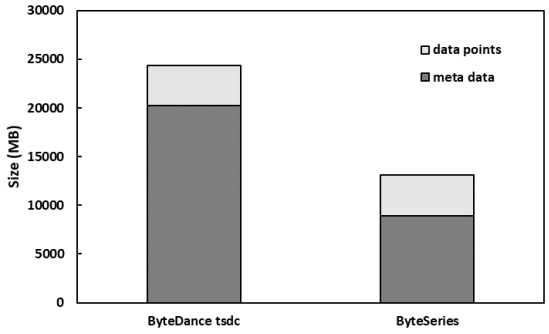


Figure 19: Production workload memory footprint on each node instance.

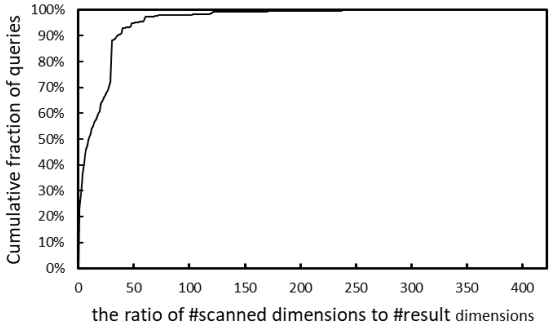


Figure 20: Distribution of #scanned dimensions / #result dimensions in production queries

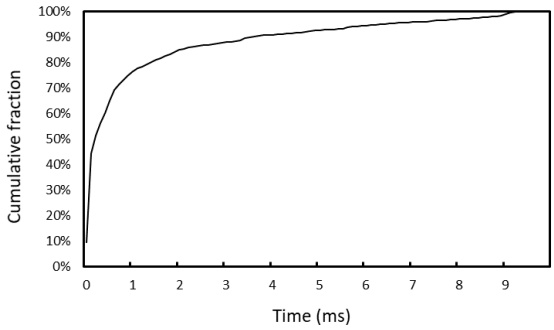


Figure 21: Distribution of query time

5.1 Memory Usage

In this experiment, we compare ByteSeries with tsdc, ByteDance's original system. ByteDance's production system places great demands on memory space. As of the spring of 2020, we need to store close to 40 million dimensions of time series on each node instance. Each dimension contains on average about 40 data points over a two-hour time slice. Without proper compression, it requires about 25GB of memory space for each slice on each instance, of which over 80% is metadata. As shown in Figure 19, ByteSeries successfully reduces about 40%–50% memory space in comparing to tsdc. The improvement is mainly due to ByteSeries' efficient compression of metadata, which shows about 2.3x improvement.

5.2 Ingestion Throughput Requirement

Taking into account the different working conditions during the day and at night, as well as the differences between working days and non-working days, we measured the actual data ingestions throughput in the ByteDance's production environment. We obtained an average of 0.24 million insertion per second per instance. We also observed that the peak insertion rate was 0.29 million per second. According to the benchmark in Section 4.3, it shows that ByteSeries can sustain an ingestion rate that is far exceeding the demand of the production environment.

5.3 Query Efficiency

In this section, we examine the query efficiency of ByteSeries with real production workload. We captured tens of thousands of queries from the real production system in ByteDance and evaluated them on our system. In this experiment, we first examine the efficiency of the Compressed Inverted Index in ByteSeries with varying ratios of the number of scanned dimensions to the result dimensions (called **SR** value). A smaller **SR** value means fewer irrelevant dimensions are scanned, and hence higher higher effectiveness of the index and query execution. As shown in Figure 20, there are about 90% of the queries have an **SR** value not exceeding 50. In comparison to full scan of the *hashtable*, we significantly reduce the scanning overhead of queries. Figure 21 shows the query execution time. It shows that 80% of the queries can complete within one millisecond, and almost all the queries can complete within 10 ms. This meets the requirements of ByteDance.

6 RELATED WORK

A number of systems were designed to support diverse query requirements over time series data [8] and have demonstrated many use cases of time series, such as clustering and classification [5, 19], anomaly detection [12, 14] and time series indexing [11, 13, 21]. We refer the readers for a recent survey [9] on more time-series systems. Among these systems, a lot of efforts focus on developing complex disk-based indices for efficient querying. Most designs were based on the log-structure merge tree (LSM-tree) [17]. Systems in this category include OpenTSDB [16], KairosDB [10], InfluxDB [7] and etc. Although indices based on LSM-tree are designed for efficient disk ingestion, read and write performances are limited by disk I/O. Moreover, LSM-tree lacks secondary indices

and cannot support multi-dimensional time series queries. InfluxDB implements a fully functional index system, called TSI [6], but a query needs to involve more disk I/O. Limited by disk I/O, such disk-based databases cannot provide sufficient ingestion and real-time query performance, and hence they are mainly suitable for offline data analysis.

Some recent efforts investigated how to use main memory to enhance performance on data ingestion and real-time querying. BTrDB [2] organizes data in memory and writes it to the back-end storage in batches. It pre-compute aggregate statistics in memory with different scales. However, BTrDB does not support multi-dimensional time series data model. Akumuli uses a storage structure called *Numerical B^+-tree*[1], a variant of B^+-tree[23], which is somewhat close to BTrDB's design. Akumuli added multi-dimensional data model and index to support multi-dimensional queries. However, its design is optimized to avoid disk I/O being the bottleneck, rather than minimizing memory footprint. RTSI is an in-memory index for searching the live audio in the main memory [26]. However, it does not support meta-data compression.

Gorilla [18] is a write-through cache of back-end persistent storage designed for large-scale monitoring systems, developed by Facebook. It proposed an efficient time series compression algorithm, which has become the preferred algorithm for many time series databases. Efficient data compression and simplified data structure allows it to store the whole data in memory. However, the simplified data model does not support high-dimensional queries.

Prometheus [20] is another time series database built for monitoring systems. Its idea is similar to Gorilla, providing efficient ingestion and query using main memory and local storage. To reduce memory usage, it partitions the data with time into blocks and only stores the latest block in memory. Even though only a short time slice is stored in memory, it still consumes a lot of memory as it maintains dynamic indices and performs no compression over metadata.

In addition, there are also efforts to use memory to support time series databases on top of conventional database systems. TimescaleDB [22] is a time series storage engine based on PostgresSQL and creates a table of the recent inserted data in memory and writes it to disk with a fixed time interval or a fixed data size. Each table is completely independent, with independent metadata and indices. Due to the lack of effective data compression, the memory efficiency of TimescaleDB is limited. MongoDB [15], a document-oriented database, can support complex queries with rich data models and indices for managing metadata of high-dimensional time series. MongoDB can use in-memory engines to provide higher performance than disk-based storage engines. However, the lack of effective compression makes memory utilization very low. It also requires a lot of additional efforts to support querying features over time.

7 CONCLUSION

In this paper, we identify the challenges of managing metadata for high dimensional time series in large-scale metric monitoring systems through the analysis of BateDance's production data and workload. The analysis showed that the rapid increase of dimensions, and the existence of many dimensions with few data points make the large memory footprint of metadata become a prominent problem to be addressed for data management in large-scale monitoring systems. Furthermore, we showed that multi-dimensional queries are the norm in time series analysis, and hence their efficient execution is of high importance to large-scale monitoring systems.

To address the challenges posed by massive metadata, we developed ByteSeries, that efficiently compresses both time series metadata and data points. The Compressed Inverted Index is not only able to store metadata in a compressed form, but also capable of processing multi-dimensional group-by queries efficiently. Furthermore, our data conversion scheduler is able to efficiently convert the data into the compressed form without sacrificing data ingestion throughput. The experiments showed that, in comparing with ByteDance's tsdc and two existing open-source systems, ByteSeries can effectively reduce memory footprint while having superior data ingestion throughput and query response time. ByteSeries can be used not only as an independent system, but also as an intermediate in-memory cache in front of an on-disk time series database system to achieve high ingestion throughput and real-time query processing.

REFERENCES

[1] Akumuli. 2020. Akumuli Numeric B+tree. https://docs.akumuli.org/. Accessed Feb 12, 2020.

[2] 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.

[3] Cedar. 2020. Cedar - C++ implementation of efficiently-updatable double-array trie. http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/. Accessed Feb 12, 2020.

[4] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. 295–298.

[5] Bing Hu, Yanping Chen, and Eamonn Keogh. 2013. Time series classification under more realistic assumptions. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 578–586.

Algorithm 1: The Execution Strategy of Data Conversion Scheduler

Let *stop* be the flag of whether the *Scheduler* runs ;
Let *dimen_num* be the dimensions in *AS*;
Let *buffer_size* be the size of *AS*;
Let *avg_points* be the dps per dimension in *SS* ;
Let *tss_number* be the number of *TSS*;
Let *dimen_limit* be the the dimensions limit in *AS*;
Let *total_dimensions* be the total dimensions in *CS*;
Let *max_size* be the maximum size of *AS*;
Let *point_limit* be the dps limit per dimen in *SS*;
Let *min_dimen* be the minimal dimensions in *AS*;
Let *max_dimen* be the maximum dimensions in *AS*;
Let *dimen_ratio* be the preset ratio;
stop ← *False*
dimen_limit ← *min_dimen*
p ← *compactor*
Function `Scheduler()`
  while *stop is False* **do**
    if $buffer\_size \geq max\_size$ **then**
      Exec(*compactor*);
      p ← `NextProcessor(compactor)`;
      continue;
    if `CheckProcessor(p)` *is True* **then**
      Exec(*p*);
    p ← `NextProcessor(p)`;
  return *;*
Function `CheckProcessor(p)`
  if *p is compactor And dimen_num* ≥ *dimen_limit* **then**
    return *True*;
  if *p is merge_processor And tss_number* ≥ *1* **then**
    return *True*;
  if *p is compressor And avg_points* ≥ *point_limit* **then**
    update *total_dimensions*;
    tmp_dimen ← *dimen_ratio* * *total_dimensions*;
    dimen_limit ← Min(Max(*min_dimen*, *tmp_dimen*), *max_dimens*);
    return *True*;
  return *False*;
Function `NextProcessor(p)`
  switch *p* **do**
    case *compactor* **do**
      return *merge_processor*;
    case *merge_processor* **do**
      return *compressor*;
    case *compressor* **do**
      return *compactor*;

[6] InfluxDB. 2020. InfluxDB Concepts. https://docs.influxdata.com/influxdb/v1.7/concepts/. Accessed Feb 12, 2020.

[7] InfluxDB. 2020. InfluxDB Home Page. https://www.influxdata.com/. Accessed Feb 12, 2020.

[8] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.

[9] S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.

[10] KairosDB. 2020. KairosDB Home Page. https://github.com/kairosdb/kairosdb/. Accessed Feb 12, 2020.

[11] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 151–162.

[12] Eamonn Keogh, Stefano Lonardi, and Bill'Yuan-chi' Chiu. 2002. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 550–556.

[13] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *Knowledge and information systems* 7, 3 (2005), 358–386.

[14] Jessica Lin, Eamonn Keogh, Stefano Lonardi, Jeffrey P Lankford, and Donna M Nystrom. 2004. Visually mining and monitoring massive time series. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. 460–469.

[15] MongoDB. 2020. MongoDB Home Page. https://www.mongodb.com/. Accessed Feb 12, 2020.

[16] OpenTSDB. 2020. OpenTSDB Home Page. http://opentsdb.net/. Accessed Feb 12, 2020.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[18] 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.

[19] François Petitjean, Germain Forestier, Geoffrey I Webb, Ann E Nicholson, Yanping Chen, and Eamonn Keogh. 2014. Dynamic time warping averaging of time series allows faster and more accurate classification. In *2014*

IEEE international conference on data mining. IEEE, 470–479.

[20] Prometheus. 2020. Prometheus Home Page. https://prometheus.io/. Accessed Feb 12, 2020.

[21] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 262–270.

[22] Timescale. 2020. Timescale Home Page. https://www.timescale.com/. Accessed Feb 12, 2020.

[23] B+ tree. 2020. B+ tree Wiki Page. https://en.wikipedia.org/wiki/B%2B_tree/. Accessed Feb 12, 2020.

[24] TSBS. 2020. TSBS Github home page. https://github.com/timescale/tsbs#devops--cpu-only/. Accessed Feb 12, 2020.

[25] TurboPFor. 2020. TurboPFor Home Page. https://github.com/powturbo/TurboPFor/. Accessed Feb 12, 2020.

[26] Z. Wen, X. Liu, H. Cao, and B. He. 2018. RTSI: An Index Structure for Multi-Modal Real-Time Search on Live Audio Streaming Services. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1495–1506.