Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries

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

Jin Xue
The Chinese University of Hong Kong
jinxue@cse.cuhk.edu.hk

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

ABSTRACT

Performance-monitoring timeseries systems such as Prometheus and InfluxDB play a critical role in assuring reliability and operationality. These systems commonly adopt a column-oriented storage model, by which timeseries samples from different timeseries are separated, and all samples (with both numeric values and timestamps) in one timeseries are grouped into chunks and stored together. As a group of timeseries are often collected from the same source with the same timestamps, managing timestamps and metrics in a group manner provides more opportunities for query and insertion optimization but posts new challenges as well. Besides, for performance monitoring systems, to support better compression and efficient queries for most recent data that are most likely accessed by users, huge volumes of data are first cached in memory and then periodically flushed to disks. Periodic data flushing incurs high IO overhead, and simply discarding flushed data, which can still serve queries, not only is a waste but also brings huge memory reclamation cost. In this paper, we propose Heracles which integrates two techniques - (1) a new storage model, which enables efficient queries on compressed data by utilizing the shared timestamp column to easily locate corresponding metric values; (2) a novel two-level epoch-based memory manager, which allows the system to gradually flush and reclaim in-memory data while unreclaimed data can still serve queries. Heracles is implemented as a standalone module that can be easily integrated into existing performance monitoring timeseries systems. We have implemented a fully functional prototype with Heracles based on Prometheus tsdb, a representative open-source performance monitoring system, and conducted extensive experiments with real and synthetic timeseries data. Experimental results show that, compared with Prometheus, Heracles can improve the insertion throughput by 171%, and reduce the query latency and space usage by 32% and 30%, respectively, on average. Besides, to compare with other state-of-the-art storage techniques, we have integrated LevelDB (for LSM-tree-based structure) and Parquet (for column stores) into Prometheus tsdb, respectively, and experimental results show Heracles outperform these two integrations. We have released the open-source code of Heracles for public access.

PVLDB Reference Format:
Zhiqi Wang, Jin Xue, and Zili Shao. Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries. PVLDB, 14(6): 1080 - 1092, 2021.
doi:10.14778/3447689.3447710

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/naivewong/heracles.

1 INTRODUCTION

Performance-monitoring timeseries systems like Prometheus [38] and InfluxDB [25] are used to collect real-time performance metrics such as CPU utilization and memory usage, and play a critical role in assuring reliability and operationality with the support of system alerts and problem diagnosis. In this kind of systems, timeseries samples mainly consist of numeric values (representing by double-precision floating-point number) and timestamps (representing by 64-bit integer); each timeseries is uniquely identified by a set of predefined labels that are used for data queries. By handling millions of samples coming from different sources at the same time, these systems need to provide high insertion throughput (several tens millions of samples per second) and low query latency (millisecond level involving large-scale data aggregation) with efficient storage based on compression. With these characteristics, performance-monitoring timeseries data cannot be efficiently handled by SQL and NoSQL databases [27], and require specially-tailored data management.

Specifically, performance-monitoring timeseries systems commonly adopt a column-oriented storage model [25, 38, 44], by which timeseries samples from different timeseries are separated, and all samples (with both timestamps and numeric values) in one timeseries are grouped into chunks (partition in column store) and stored together in a single column. By combining and storing timestamps and values together, this storage model enables easy data management. Particularly, queries can be directly served once temporal information of a timeseries has been located inside one column. Moreover, in performance monitoring systems, to support better compression and efficient queries for most recent data that are most likely accessed by users, huge volumes of data are first cached and compressed in memory and then periodically flushed to disks. Storing timestamps and values together makes data flushing simpler; otherwise, for each timeseries, timestamps and values need to be handled separately.

This storage model, however, hinders insertion and query optimization from exploring group behaviors. That is, performance monitoring timeseries data are commonly collected in a group manner so a group of timeseries from the same source are with the same timestamps. For instance, all metrics from a Docker container [19, 22], a MySQL server [35], or a device, share the same temporal information. With this storage model, nonetheless, the temporal information (i.e. timestamps) is duplicated and stored

doi:10.14778/3447689.3447710

with each metric column when forming one timeseries. This not only introduces unnecessary memory/storage space overhead but also raises two more issues as follows.

First, this will add more latency to query execution. To serve a query with a time range on a timeseries, as timestamps are stored with values in compressed chunks, we need to first locate corresponding chunks, decompress them and then obtain specific samples based on the timestamps inside. Second, this will degrade the insertion throughput. For a group of data from different timeseries with the same timestamps, by duplicating timestamps and storing with values in each timeseries, during insertions, we have to lock each timeseries individually instead of utilizing a group-based insertion with only one lock. When the number of timeseries in a group is big (e.g. one single Linux machine can have 500 performance metrics [36]), this overhead is not negligible.

To address these issues, we propose a new group-based storage model to manage all timeseries with the same timestamps in a group manner for performance monitoring timeseries. There are two challenges for implementing such a group-based storage model. First, given the continuously-growing large volume of timeseries data, timestamps and metric values need to be compressed with less memory/storage footprints. However, when they are stored separately and compressed with different techniques, it is more difficult to map timestamps to metric values. Without a proper mapping, we cannot directly read corresponding metric values in metric value columns after we locate the timestamp from the timestamp column. While this is similar to column mapping in column-oriented databases [15, 44, 45], fixed data formats in timestamps and metric values open new optimization opportunities for efficient queries on compressed floating-point numbers based on compressed timestamps.

Such a group-based storage model also poses a challenge on data flushing. For performance monitoring systems, while efficient queries for most recent data are extremely important as they are most likely accessed by users, in many applications, such as sensor monitoring [48], network anomaly troubleshooting [9], and IT infrastructure monitoring [24], long-term trends of metrics with days/months/years of data are needed as well, thus requiring both in-memory and on-disk data management [24, 25, 34, 38]. To address this, one effective solution is to cache and compress data in memory and then periodically flush them to disks. However, periodic data flushing incurs high IO overhead, and reclaiming flushed data degrades system performance as well. A group-based storage model can bring further system deterioration in data flushing, when timestamps and values are handled separately with different compression schemes and different in-memory and on-disk formats, thus demanding more computation resources and conversion time.

In this paper, we propose Heracles that integrates two techniques. The first one is a storage model with a new compression and mapping method to jointly optimize data compression and direct queries on compressed data considering both in-memory and on-disk timeseries data. The key issue is to strike a balance between the storage and query efficiency by compressing one shared timestamp column (with 64-bit integer) and multiple metric columns (with floating-point number) with a proper time-to-metric mapping. In our method, a fine-grained compression and mapping technique

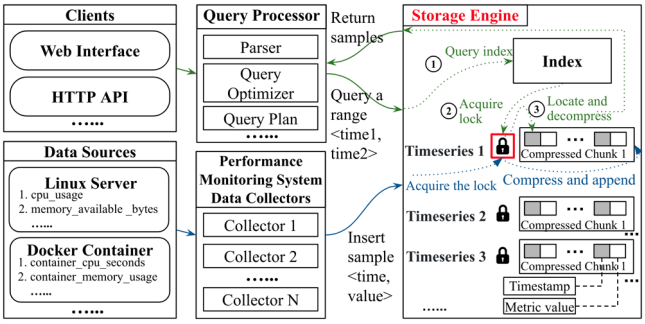


Figure 1: The architecture of a performance monitoring system.

is proposed, by which samples in a group will be continuously inserted and compressed into tuples in one shared timestamp column and multiple metric columns, respectively. A tuple contains a fixed number of compressed values or timestamps, and is the unit for the alignment of timestamps and metric values. Specifically, different from Gorilla [33], a state-of-the-art compression technique, a scaling method and fine-grained classification are proposed to further optimize timestamp and value compression, respectively, by fully exploring more regular data patterns in performance monitoring timeseries.

Second, we introduce a novel two-level time-partitioned epoch-based memory manager which can serve data caching and memory allocation for performance monitoring timeseries. Through it, we can gradually flush and reclaim outdated in-memory data blocks while unflushed data are still cached and can serve queries. Different from the previous works [7, 16, 20, 30], we judiciously exploit the temporal information, a special characteristic in timeseries data, in the epoch-based memory reclamation algorithm. Specifically, every data flushing period is associated with a unique version that will be returned together with memory blocks when they are allocated. This version enables our two-level epoch mechanism, in which memory blocks with the first-level epoch will be put into a hot data area partition during data flushing (that can continue to serve queries), and after a hard release time, all data blocks in the hot area partition will be pushed to a cooling area and attached with the second-level epoch (that become invisible for query threads and can be allocated as free memory blocks).

2 BACKGROUND AND MOTIVATION

2.1 Performance Monitoring System

A performance monitoring system can collect, index, store and query monitoring metric data from targeted data sources. Figure 1 shows a general architecture of a performance monitoring system. Data collectors automatically collect monitoring metric data from targets and insert them into the storage engine. Users can query

Timeseries Labels		Inverted Index	
TS 1	{name="net_receive", job="prometheus"}	{name="net_receive"}	= [TS1]
TS 2	{name="node_cpu", job="prometheus"}	{name="node_cpu"}	= [TS2]
		{job="prometheus"}	= [TS1, TS2]

Figure 2: An example of inverted indexes for timeseries.

data from the storage engine through the query processor. For the storage engine, it generally handles two kinds of data - temporary data (in-memory data manager) and persistent data (on-disk data manager).

2.2 Storage Model

Timeseries identifiers and metric values are stored separately. For timeseries identifiers, they are semi-structured data like JSON labels and an inverted index is often applied on them to locate timeseries. Figure 2 shows an example, in which for each label pair (e.g. name="net_receive") of a timeseries, an inverted index is built (e.g. name="net_receive" -> [TS1]) accordingly. For timeseries data, each timeseries logically has two columns - one for timestamps (64-bit integer) and the other for timeseries metric values (64-bit double-precision floating-point number). As a result, the compression methods in column-oriented database systems [1] can be applied to the two columns of each timeseries. For convenient management, timestamps and metric values of one timeseries are combined and stored together into chunks. To achieve better compression ratios, compression methods are normally based on relatively large chunks, and thus, in order to get a sample in the middle of a compressed chunk, decompression from the beginning is needed.

For each timeseries, an index is provided so the time interval of one chunk and its in-memory or on-disk chunk location can be recorded. The index contains two types of information: one is label pairs that are the identifier of a timeseries, and the other is chunks metadata that records the minimum and maximum timestamps and the location of each chunk. To serve queries with time ranges, the index needs to be searched first so as to locate chunks; after that, each chunk in the range needs to be decompressed so as to obtain timestamps; finally, based on the timestamps, data can be returned to users.

A typical workflow in a performance monitoring system is illustrated in Figure 1. Data collectors fetch monitoring data from data sources and insert them into the storage engine periodically. Users can issue queries through different clients, and the query processor will send parsed queries to the storage engine. Although many timeseries (e.g. 500) may come from one data source, the storage engine still stores them separately. The timestamps and metric values are stored together in the chunks where the gray square represents timestamp and the white square represents timeseries metric value. The compression method such as Gorilla [33] is applied to compress timestamps with delta-delta and metric values with the XOR-based algorithm. One chunk is kept relatively large (more than 100 samples) for high compression ratios. Note that even timeseries are from the same device, their timestamps are stored in separate chunks redundantly for each timeseries.

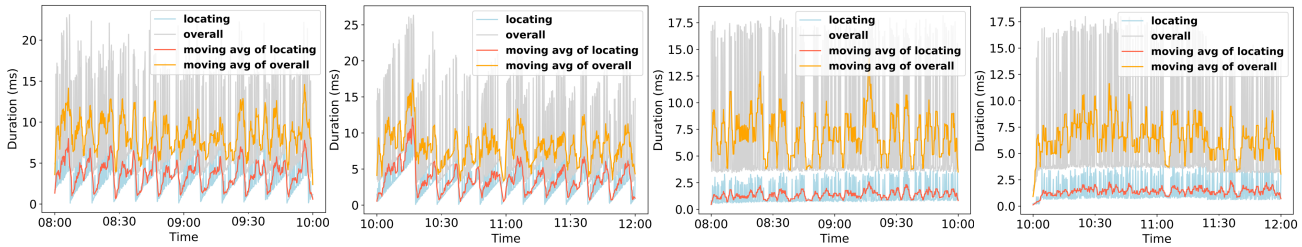
2.3 Motivation

2.3.1 Issues In the Storage Model. State-of-the-art performance monitoring timeseries systems like Prometheus and InfluxDB handle and store timeseries independently although they may come from the same source (e.g. same host/application). There are several problems caused by this model.

Problem 1: Inefficient Query. This storage model introduces extra query overheads. As aforementioned, during a query with a time range, we need to first traverse chunks metadata in the index to locate chunks before we read chunk data. With a separate timestamp column, for each group of timeseries, metric values in chunks can be directly located based on the timestamp column; thus, chucks metadata in the index are not needed and the search overhead for chunks metadata can be eliminated. Furthermore, by storing timestamps and values together in chunks, when reading data from a chunk, we have to decompress it in order to get temporal information. This is particularly inefficient when we need to get data samples in the middle of a chunk, in which we have to decompress the timestamps and metric values from the beginning of the chunk until the queried timestamp is found. Instead, a separate timestamp column enables queries on compressed data, by which the chunk-decompression overhead can be mitigated.

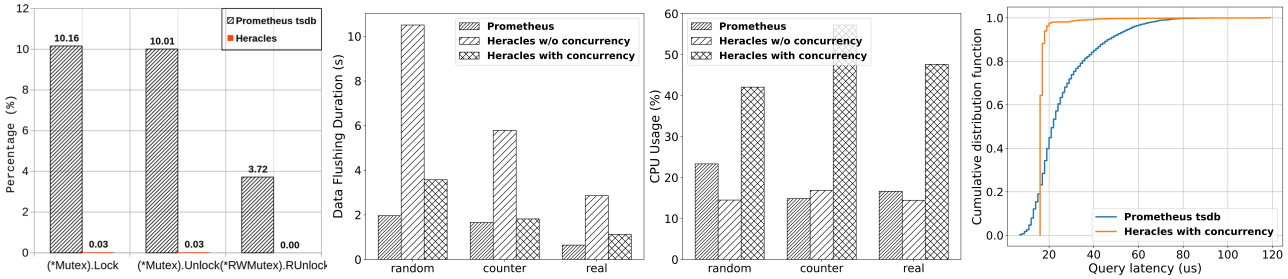
We have conducted experiments to reveal the overheads with the experimental setup in §4.3. Specifically, we run Prometheus and Heracles to monitor 10 Linux servers, respectively, and query the average CPU usage in the last 5 minutes every 10 seconds (See §4.3 for details), which only accesses the in-memory data. Figure 3a and Figure 3c show the results of locating the first sample and the overall query time from Prometheus and Heracles, respectively. The red curve is the moving average time of locating the valid position in the chunks, which can be reduced with a separate timestamp column in Figure 3a. For Prometheus, it can be observed this unnecessary cost is slightly more than a half of the execution time for the in-memory queries. Second, we move this query window two hours before the current time, which only accesses the on-disk data, and similar results can be found in Figure 3b and Figure 3d as well.

Problem 2: Redundant Timestamps. For performance monitoring systems, timeseries are usually collected and sent as groups to the storage engine periodically (e.g. timeseries from the same hosts for system-level performance metrics or timeseries from the same services for application-level performance metrics). If we store timestamps and metric values separately, the timestamps of the



(a) Prometheus memory chunks (b) Prometheus disk chunks (c) Heracles memory chunks (d) Heracles disk chunks

Figure 3: Cost of wasted decompression in a two-hour Prometheus recording rule tracing.



(a) Lock operation percentage in the profiling **(b) Duration of flushing in-memory data to disk and CPU usage** **(c) CDF of query latency during data flushing**

Figure 4: Issues in the storage model.

timeseries from the same source can be stored together; without redundant timestamps, the storage footprint can be reduced.

Problem 3: Limited Write Throughput. As each timeseries is tracked and handled independently, a cache-line-aligned [43] lock is needed for each operation on a single timeseries. As a result, when inserting a large scale of data from different timeseries, the overhead of locking operations can be huge. Figure 4a shows the cost of lock operations under a 200Hz sampling-rate profiling of an insertion micro-benchmark of Prometheus tsdb (the storage engine of Prometheus), from which it can be observed that lock operations occupy around 24% of the total time. If we manage timeseries in a group manner, this problem can be naturally mitigated since lock operations will operate in a coarse-grained manner.

2.3.2 Issues In Data Flushing. Although separating timestamps and metric values unleashes the potentials to reduce query latencies and storage footprint and enhance write throughput, a group-based storage model can bring further system deterioration in data flushing as follows.

Problem 1: Computation Overhead. Since data are first accumulated in memory, data flushing can occupy a huge amount of CPU and I/O resources especially when the formats of in-memory and on-disk data are different. Figure 4b shows the data flushing durations of 2 hours' data (random, increasing counter, and real monitoring data, respectively) from 10000 timeseries with Prometheus and Heracles, respectively (See §4.5 for details), where Heracles (Heracles w/o concurrency) is not integrated with our two-level epoch-based memory manager for data flushing optimization. It can be observed that data flushing with Heracles is much longer than that with Prometheus. Since Prometheus utilizes a uniform data format for both in-memory and on-disk data, data flushing simply writes the data to disks without extra computation, thus enabling more efficient data flushing.

In Heracles, we can reduce the flushing duration by increasing the degree of parallelism. For instance, as shown in Figure 4b, for Heracles with concurrency, for each timeseries in the same group, we launch a Goroutine (lightweight thread in Golang) [28] to handle the data conversion and flushing (around 100 Goroutines running concurrently). However, this will compete with query threads for CPU resources and incur query tail latencies. Figure 4c shows the CDF of query latencies during the data flushing and the P99.99 latencies of Heracles is 25% higher than that of Prometheus. Thus, optimizing data flushing becomes critical for Heracles.

Problem 2: Memory Reutilization. In current systems like Prometheus and InfluxDB, flushed data are directly discarded and are immediately invisible to query threads. However, since flushed in-memory blocks are still valid and can serve queries, simply discarding them not only is a waste but also causes high garbage collection overhead. If we manage in-memory data by ourselves, we can gradually reclaim and reuse flushed data blocks while unreclaimed data blocks can continue to serve queries.

3 DESIGN

The general data flow of Heracles is presented in Figure 5. Data collected by data collectors are converted to group-based samples (containing a timestamp and values of all timeseries in the group) by data processors that will perform the data insertion through the group insertion primitive. Queries are converted to *<group id, timeseries id, time range>* by the query processor and it will call the group querying primitive accordingly.

The physical format of Heracles is illustrated in Figure 6. Similar to Prometheus and InfluxDB, data are first cached in memory and then are flushed to disks where they are compressed differently.

In the following sections, we will first introduce Heracles storage model (§3.1). Then, to solve the problems in the data flushing of Heracles storage model mentioned in §2.3, we introduce our two-level EBR memory manager (§3.2). Finally, we introduce the query procedure of Heracles (§3.3).

3.1 Heracles Storage Model

In this section, we first introduce two different tuple mappings for Heracles storage model. Then, we introduce the in-memory and on-disk storage models, respectively.

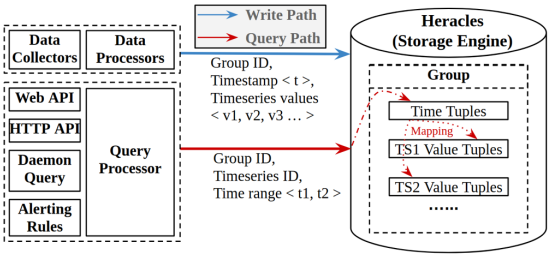


Figure 5: The data flow of Heracles.

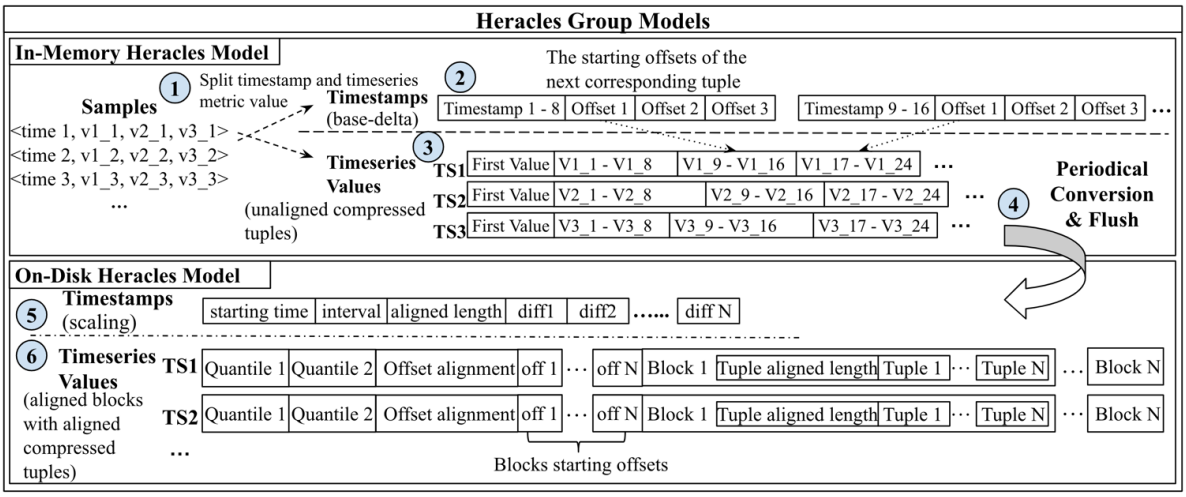


Figure 6: Heracles in-memory/on-disk models.

3.1.1 Tuple-Based Mapping. To strike a balance between compression ratios and query efficiencies, timestamps and metric values are managed based on tuples and compressed with different techniques. As a result, we need to carefully design the mapping from timestamp tuples to timeseries value tuples through which we can quickly obtain the corresponding timeseries value tuple after we locate the timestamp tuple. We utilize two methods to implement the mapping: (1) Tuples with index - We utilize an extra index (i.e. containing the location information of metric value tuples) to help us track the tuples and indexes can be stored together with timestamp tuples; (2) Padded tuples - Because the size of a tuple can be irregular after compression, we pad the compressed tuples to make them aligned.

In our experiments, we select eight as the tuple size because we find it can achieve good results on both compression ratios and query efficiencies. We will show the trade-off of different tuple sizes in §4.4.5.

3.1.2 In-Memory Group Model. In-memory Heracles accepts incoming group samples, compresses them on the fly, and then flushes them to disks periodically. As shown in Step (2) and Step (3) in Figure 6, in-memory Heracles separates and compresses timestamps (that will be stored in the time tuple) and metric values (that will be stored in the series slots) differently.

Because we compress data on the fly and in-memory Heracles is continuously absorbing incoming samples, the size of padded tuples cannot be determined. As a result, we choose *Method 1* (i.e. Tuples with index) where we generate an index for each timestamp tuple to track the location of its metric-value tuple and we store this index together with the timestamp tuple as shown in Step (2) in Figure 6.

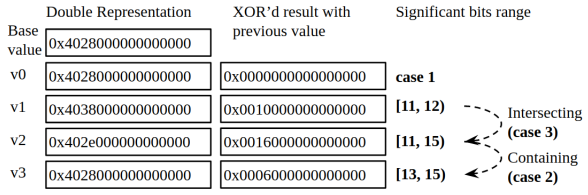


Figure 7: An example of the timeseries metric compression.

Timestamp Compression. We use the base-delta method to compress timestamps in the Varint format [11] because samples are continuously coming and the ending time cannot be determined. At the end of a timestamp tuple, we store the location information of each series slot as aligned-bits with the same sizes. In our implementation, we manage all time tuples as an object array. As a result, to query a timestamp, we can easily perform a binary search on this time tuple array by comparing the first timestamp of the tuple and locate the target tuple.

Timeseries Metric Value Compression. Metric values from different series are separated and compressed into different series slots. For each series, we continuously compress the values into tuples as shown in Step (3) in Figure 6.

To insert a new metric value into a tuple, the basic idea is to use the XOR operation to compare it with a base value (if the tuple is empty, the base value will be the first value of this timeseries in the current in-memory group; otherwise, the base value will be the previous value in this tuple) and then store the comparison result. Similar to the XOR'd compression in Gorilla [33], there are three cases as follows:

- Case 1: If the XOR'd value is 0 (the new value is the same as the base value), we only store one control bit, namely '0', in the tuple.
- Cases 2 and 3: Otherwise, we will first store the control bit, namely '1'. Let *LZ* be the number of leading zeros and *L*

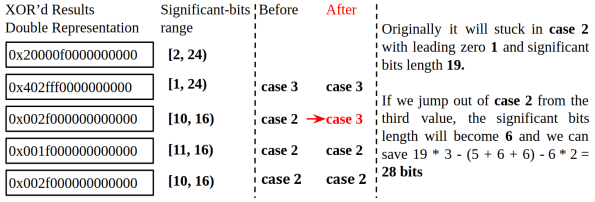


Figure 8: An example of the local optimality with useless significant-bits in XOR'd compression.

Algorithm 1: On-disk Timestamp Scaling Compression

Input: Timestamp array: Ts

```
// Compute interval between two samples
interval=(Ts.Back()-Ts.Front())/Ts.Size();
for i = 0; i < Ts.Size(); i++ do
    diff = Ts[i] - (Ts.Front() + i * interval);
    WriteAlignedBits(diff);
end
```

the length of significant-bits in the XOR'd value. To further optimize this, one more controlled bit is used to differentiate the following two cases:

- Case 2: If the significant-bits of the current XOR'd result are in the range of the significant-bits of the previous XOR'd result (e.g. as shown in the third to fifth numbers in Figure 7), the second control bit will be '0' and after that, we will directly store the significant-bits of the XOR'd value based on the previous LZ and L (i.e. reusing the previous significant-bits range).
- Case 3: If the significant-bits of the current XOR'd result are outside the range of the significant-bits of the previous XOR'd result (e.g. as shown in the second number in Figure 7), the second control bit will be '1' and after that, we will store LZ (5 bits), L (6 bits), and then the significant-bits of the XOR'd value.

In the above algorithm, for Case 2, we find that we may fall into the local optimality with big significant-bits range (e.g. as shown in Figure 8); thus, when a tuple is finished, we will conduct an overall comparison with all the values in it and may switch from Case 2 to Case 3 to further save space for the final result of the tuple.

3.1.3 On-Disk Group Model. With on-disk Heracles, data stored in memory are flushed to disks periodically as shown in Step (4) in Figure 6. Since on-disk data blocks are immutable, we implement the mapping based on *Method 2* (i.e. Padded tuples) and compress data differently from in-memory Heracles.

Timestamp Compression. For timestamps, because performance metrics are collected periodically and the interval between two samples is normally stable, we resort to a scaling method instead of using the base-delta method. As shown in Algorithm 1, we first compute the average value of all the timestamps and for each timestamp, we only store the difference (the scaling value) by subtracting it from the average value. Compared to the base-delta method, this scaling method exhibits a better compression ratio because the scaling differences are commonly smaller than the increasing deltas.

Timeseries Metric Value Compression. In XOR'd compression, an appropriate base value can significantly improve the compression ratio and since the data are fixed in on-disk Heracles, we can choose the base value for the XOR operation more efficiently. In our preliminary experiments, using the median value can achieve a better compression ratio than using mean/min/max values. But when the timeseries values fluctuate intensively, simply choosing one median value is not enough. As a result, we choose two quantiles (the $(n/4)^{th}$ largest value and the $(3^*n/4)^{th}$ largest value) as the base values. For the first value in a tuple, we perform the XOR operation with the closer quantile. For the following value (i.e. the second one) in a tuple, we perform the XOR operation with the previous value.

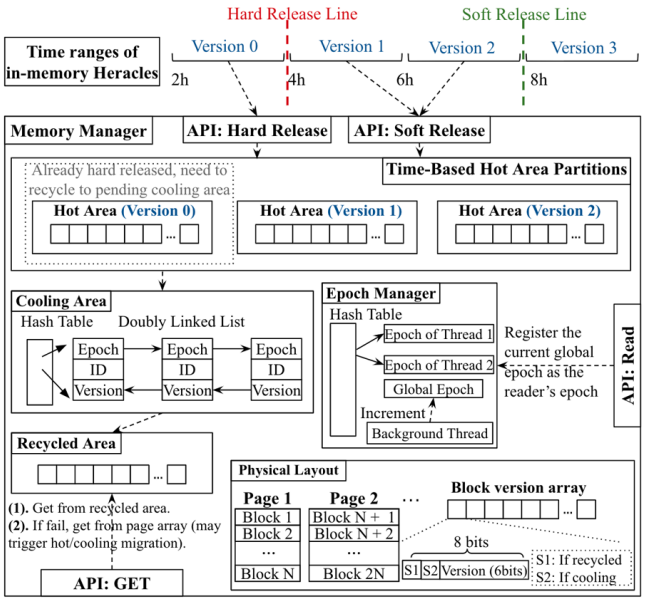


Figure 9: The memory manager in Heracles.

To track metric values efficiently, we leverage a 2-level alignment structure (block and tuple) as shown in Figure 6. A block contains multiple tuples whose sizes are aligned with the largest tuple in the same block and since block sizes are not aligned, we need to store the starting offset of each block.

Currently, we use two quantiles as the base values for XOR operations, which requires one indicator bit for each tuple. Theoretically, more quantiles can reduce the XOR bits of the first value of the tuple. But it will also introduce more indicator bits. Based on our experiments, two base values have the best compression ratio compared to using more base values.

3.2 Two-Level Epoch-Based Memory Manager

For convenient management, current performance monitoring systems directly discard flushed data after periodic data flushing and then the flushed data are either freed or handled by the GC (Garbage collection) of the programming language (e.g. Golang, Java). However, as mentioned in section 2.3.2, the data flushing procedure is resource consuming and simply discarding the flushed data that can still serve queries is wasteful.

In Heracles, we control the rate of data flushing and manage flushed data including reclaiming the flushed data. To avoid read/reclamation races, an efficient memory reclamation algorithm is needed [16, 18, 31, 42, 47]. Compared to lock-free reference counting, epoch-based reclamation (EBR) has a lower overhead because it does not require per-element atomic instructions. Besides, compared to hazard-pointer-based reclamation, EBR scales better when many elements must be traversed. Thus, in Heracles, based on EBR, we propose a novel two-level epoch-based memory manager.

Physical Layout. The bottom of Figure 9 shows the physical layout of memory blocks managed by the memory manager. Inside

the big page (e.g. 1MB), space is split into small blocks (e.g. 256 bytes) and the ID of each block is the overall index of that block. Besides, there is a version array that records the version of each block. For each version, we use 8 bits where the first two bits indicate whether the block is currently in the recycled or cooling area and the remaining 6 bits store the actual version. Since we increment the version after a long period (e.g. 2 hours), 6 bits can cover a long time span that is enough for the system to reclaim outdated memory blocks. When writer threads request for new blocks, the memory manager will return block IDs and a version that matches the corresponding time range based on the sample timestamp provided by writer threads.

First-Level Epoch. Because of periodic data flushing (e.g. 2 hours), we assign each time period with a version/epoch as shown at the top of Figure 9 and the memory manager will return the corresponding version when allocating new memory blocks. During data flushing (the soft release limit in the example), the flushed data blocks will be put into the hot area of the memory manager. Since the memory space is limited, we need to explicitly make the oldest time range invisible in the in-memory index and reclaim the corresponding memory blocks at some time point (the hard release line). In in-memory Heracles, we store the versions together with block IDs. Thus, query threads can utilize the version to identify whether a block is still valid before reading it. Specifically, a query can only access memory blocks that have an equal or a higher version.

Second-Level Epoch. We apply an epoch-based reclamation algorithm to prevent memory reclamation when some query threads are currently reading the corresponding blocks. There is a global epoch that is periodically incremented. Query threads need to set their local epochs to the global epoch and register them at the beginning of a query. Memory blocks in the hot area are gradually precipitated into the cooling area and the memory manager will record the global epoch for each block when it enters the cooling area. When the memory manager attempts to reclaim a memory block in the cooling area, it will compare the attached epoch with the smallest epoch among the registered query threads. If the epoch of the memory block is smaller than the minimum epoch of the query threads, then it means no query thread is currently reading this block and it is safe to be moved to the recycled area.

Time Partitioning and Reclamation Strategy. As discussed above, we assign different versions for different time ranges. Inside the memory manager, we partition the hot and cooling areas based on the same time ranges. During a data flushing procedure on the soft release limit, all memory blocks of the same time range are put into the corresponding hot area partition. During a cleaning procedure on the hard release limit, all the memory blocks in the partition are moved to a global cooling area (the cooling area in Figure 9). When the size of allocated memory blocks exceeds the pre-defined threshold, the memory manager will trigger background hot-cooling migration and reclamation of the blocks in the cooling area. Since the recent data are more important than the old data in performance monitoring systems, we preferentially reclaim the blocks in older partitions (e.g. given a number of blocks to be reclaimed, we perform 45% in the first partition, 35% in the second partition, and 20% in the third partition).

3.3 Query Procedure

A query in performance monitoring systems comes with a set of labels and a time range. It consists of the following steps:

(1) Generate Heracles Primitives. For the provided labels, the storage engine first queries the index to find the group IDs and timeseries IDs. Second, by combining those IDs with the time range, we can generate the Heracles query primitives.

(2) Query In-Memory/On-Disk Heracles. Using the time range, we can filter out unrelated blocks because each block corresponds to a time range. Because of the gradual flushing scheme mentioned above, the time ranges of in-memory and on-disk Heracles may be overlapping so we start with in-memory Heracles and then on-disk Heracles as follows. First, in-memory Heracles will be queried when the given time range in a query overlaps with the time range of in-memory data. Before querying, a query thread needs to register with the current global epoch in the memory manager. To query based on a given time, we can directly perform binary search on the in-memory timestamp tuple array by comparing it with the first timestamp of the tuple. After we locate the tuple, we can sequentially search inside the tuple until we find the corresponding timestamp. Then, we can read the location information of metric values stored in the tuple. Before we read the memory blocks in the series slot, the corresponding version stored in metadata will be validated in the memory manager. If the version is not equal to the version stored in the version array in the memory manager, which means the memory block has been reclaimed, then we need to utilize on-disk Heracles. Second, to query a sample with a specific timestamp of on-disk Heracles, namely t, we first perform binary search on the ***diff*** array as shown in Figure 6. After locating the index inside the array, we can use it to find the corresponding block and the timeseries metric value tuple. Finally, we decode that tuple and obtain the corresponding metric value.

4 EVALUATION

4.1 Implementation

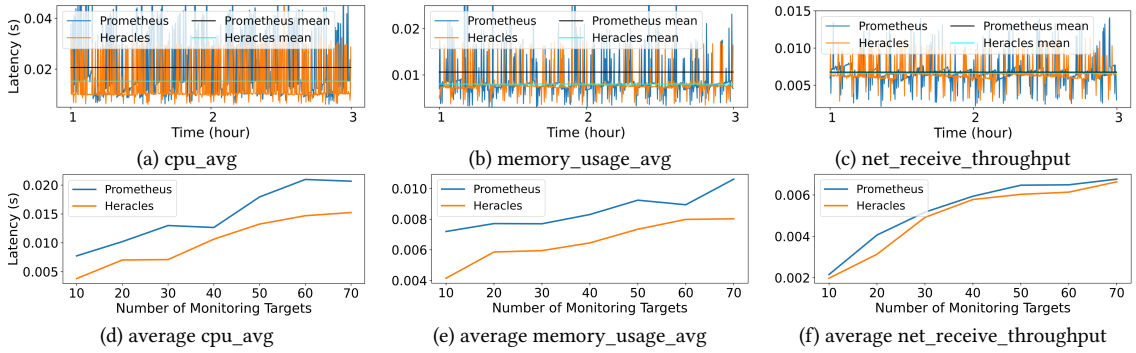
We implement Heracles storage engine based on Prometheus tsdb v0.8.0 (the storage engine of Prometheus) where we replace the storage model and add our memory manager.

To further integrate with Prometheus, two components of Prometheus are involved, namely, the storage engine (Prometheus tsdb) and data scraper (data collector). Specifically, for the storage engine in Prometheus, we replace it with our group-based storage engine and the epoch-based memory manager for data flushing; for the data scraper, we modify it by adding a group id so all data with the same timestamps from the same source can be identified.

4.2 Experimental Setup

Our experiments are conducted on a Ubuntu 18.04 workstation with an 8-core 3.0 GHz Intel Core i7-9700 CPU, 32GB of RAM, and 2TB HDD (Seagate Barracuda). We install Prometheus v2.10.0 as the baseline, our Heracles prototype, and InfluxDB on the workstation.

Monitoring Targets. We choose 10 Debian servers with Linux kernel 3.2.0.5 as the performance monitoring targets. On these targets, we run Prometheus Node Exporter [36], which is a program that can collect and export the metrics (CPU, memory, network, etc.) of the host machine periodically to a web page, so we can



(a) cpu_avg (b) memory_usage_avg (c) net_receive_throughput

(d) average cpu_avg (e) average memory_usage_avg (f) average net_receive_throughput

Figure 10: The real-time end-to-end query latency with Prometheus recording rules.

Table 1: Recording rules used in production environment evaluation.

Recording Rule	Description
cpu_avg	Get the recent average CPU usage of all targets (last 5 minutes)
memory_ usage_avg	Get the recent average memory usage of all targets (last 5 minutes)
net_transmit	Get the recent network receiving throughput of all targets (last 5 minutes)

periodically collect data from those targets. The workstation and the monitoring servers are connected with a 1Gbps local Ethernet network.

Group Setup. For all the benchmarks of the storage engine with our group models, all the timeseries (500) from the same target are identified and processed as a single group during data insertions.

Comparison Systems. For the production environment evaluation (§4.3), we compare Heracles with the Prometheus system. For the storage engine evaluation (§4.4) and the data flushing evaluation (§4.5), we compare the storage engine of Heracles with the storage engine of Prometheus (namely Prometheus tsdb). For the comparison with state-of-the-art storage techniques (§4.6), we modify the on-disk storage of Prometheus tsdb and evaluate the query latencies.

4.3 Production Environment Evaluation

To evaluate end-to-end query latency performance, each of Heracles, Prometheus, and InfluxDB is deployed on our Ubuntu workstation to collect performance metrics of the 10 Debian servers, where we have a totally 70 targets (on each server, 7 Prometheus Node Exporter programs are operated). In our experiments, data are collected and inserted into each system every 5 seconds, and specifically, in Prometheus, we utilize its own data collector to scrape data; in Heracles, the Prometheus data collector is modified to scrape data in a group manner; in InfluxDB, a simple Golang program has been implemented for data scrapping and insertion. For query latency evaluation, we first compare Heracles with Prometheus by utilizing Prometheus user-defined recording rules, and then compare Heracles with Prometheus and InfluxDB via the HTTP API interfaces.

Recording Rules Latency. Prometheus supports user-defined recording rules [37] that defines periodic background query tasks, by which the duration of executing a query defined by a rule can be

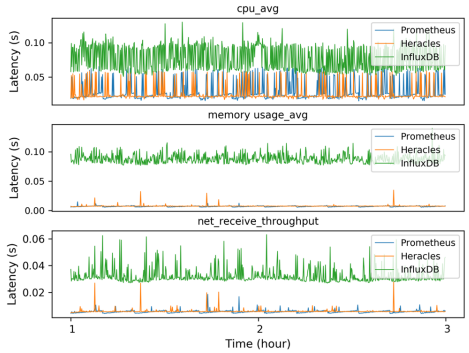


Figure 11: The real-time end-to-end query latency through the HTTP API interface.

recorded automatically. Through this, we can exclude the network latency and obtain the query latency of the system. In the paper, we only show three typical performance monitoring metrics with the recording rules in Table 1. In the experiment, we run Prometheus and Heracles prototype one by one, let them scrape data from the monitoring servers, and meanwhile execute the recording rules every 10 seconds. We record the real-time latency of executing each recording rule. We capture a two-hour time range after the system has been warmed-up for one hour which is shown in Figures 10(a)(b)(c). We can see the real-time rule evaluation latency curve of Heracles is more stable and has lower latency than that of Prometheus. Besides, we also evaluate the average rule evaluation latency under different number of monitoring targets and the experimental results are shown in Figures 10(d)(e)(f), from which a 13% average latency reduction can be observed.

End-to-End Query Latency. An HTTP API query interface is provided by both Prometheus and InfluxDB, by which we further evaluate the end-to-end query latencies of Heracles, Prometheus and InfuxDB. Specifically, Prometheus v1.0 query range HTTP API [39] is utilized for Heracles and Prometheus, and InfluxDB v2.0 query HTTP API [26] is for InfluxDB; to reduce network latencies, all queries are executed via HTTP API directly on our Ubuntu workstation (where Heracles, Prometheus and InfluxDB are deployed) when data collection is performing by each system as described above. InfluxDB does not provide a scrape component like Prometheus, thus we implement a simple Golang program to scrape and insert data of the above-mentioned targets into InfluxDB

Table 2: TSBS query patterns.

TSBS Query	Description
1-8-1	Query data on one timeseries from 8 targets, every 5 minutes window for 1 hour.
5-1-1	Query data on 5 timeseries from 1 target, every 5 minutes window for 1 hour.
5-1-12	Query data on 5 timeseries from 1 target, every 5 minutes window for 12 hours.
5-8-1	Query data on 5 timeseries from 8 targets, every 5 minutes window for 1 hour.
double-group-by-1	Aggregate on across both time and host, giving the average of 1 CPU metric per host per hour for 12 hours.
last-point	The last reading of 1 CPU metric of one host.

every 5 seconds. Similar to the previous experiment, we capture a two-hour time range after the system has been warmed-up for one hour and Figure 11 shows the real-time latency of the periodically HTTP queries. We can see the query latency from Prometheus or from our Heracles prototype is similar because of the influence of transferring data among different system components and the network latency. For InfluxDB, since we do not apply further parameter tuning, it demonstrates inferior performance compared with Prometheus and our Heracles prototype. As InfluxDB utilizes a similar storage model as Prometheus, we expect it can be further optimized by applying our Heracles model.

4.4 Storage Engine Evaluation

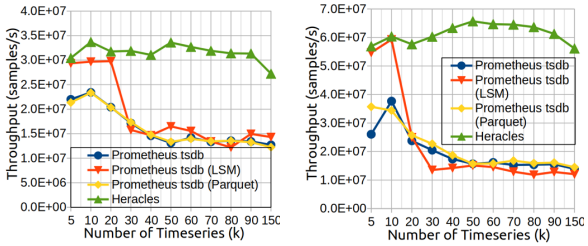
In the production environment, since performance monitoring systems ingest data through periodic data collection, with targets, we cannot saturate the bandwidth so the insertion throughput cannot be effectively evaluated. Similarly, the query performance cannot be effectively compared as well due to the overheads introduced by the network and other system components. Thus, in this section, we directly benchmark the storage engine to evaluate the insertion throughput, query latency, and disk data size under huge workloads.

4.4.1 Dataset. To obtain large datasets, we run 30 node exporters on each of the 10 monitoring servers and totally we have 150K timeseries. For the metadata of the timeseries, we use the timeseries labels collected from the node exporters. For the sample data of the timeseries, we prepare two different kinds of data – real monitoring and random data, respectively.

For real monitoring data, we scrape the above node exporters every 5 seconds for 12 hours to make the real monitoring dataset which contains a total of 1.3 billion samples. For random data, we keep the timestamp and replace the metric value with a random

Table 3: Label Description.

Acronym	Description
P-L	Prometheus tsdb LevelDB integration
P-P	Prometheus tsdb Parquet integration
P	Prometheus tsdb



(a) Random data **(b) Real monitoring data**

Figure 12: Write throughput comparison.

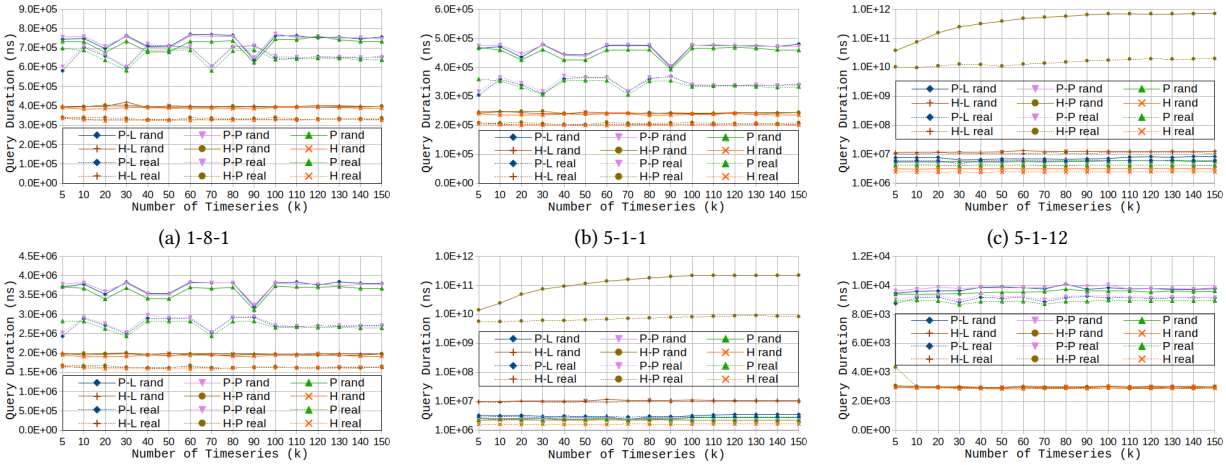
floating-point number between 0 and 1 for each sample in the real monitoring dataset. To insert data into the storage engine with a specific interval, taking 15 seconds as an example, since our collected data has 5 seconds interval, we need to skip 2 samples each time when we insert a new sample of the same timeseries.

4.4.2 Insertion Throughput. To evaluate insertion throughput, we fix the sample interval as 15 seconds and measure the durations of inserting 12-hour data of different numbers of timeseries from both our random and real monitoring datasets. Figure 12 shows the insertion throughput and we can see the insertion throughput of Heracles is much higher and more stable than that of Prometheus tsdb. On average, Heracles has 103% and 239% improvements on the write throughput over Prometheus tsdb for random and real monitoring datasets, respectively, which credits to the group-based locking unit.

4.4.3 TSBS Query Benchmarks. To measure the query latency, we utilize TSBS [46] that is a popular and comprehensive benchmark suite for timeseries databases to generate six query patterns as shown in Table 2. We run the benchmark with preloaded random and real monitoring datasets, respectively. Table 3 shows the descriptions of acronyms we use in the experiment.

(1) Different Numbers of Timeseries. First, we fix the sample interval as 15 seconds and gradually increase the number of timeseries contained in the storage engine and evaluate the query performance. In Figure 13, each point represents one benchmark round for the storage engine containing the data of the corresponding number of timeseries in the x-axis. We can see the query latency curve of Heracles is more stable and lower than that of Prometheus tsdb in Figures 13(a)(b)(c)(d)(f). For Figure 13(e), the difference in query latencies becomes smaller; this is because in ***double-groupby-1***, we query for a long time range of data, the duration of continuous decompression becomes the main influencing factor on the query latency instead of the duration of locating the first sample. On average, from the experiments of Figure 13, Heracles has 43.9% and 44.5% improvements over Prometheus tsdb for the random and real monitoring datasets, respectively.

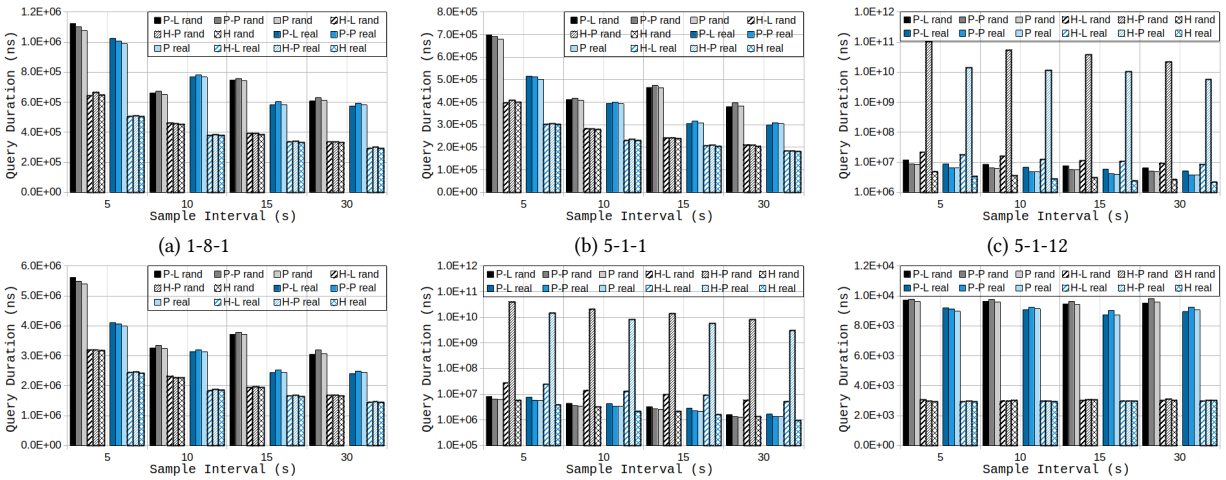
(2) Different Data Densities. Second, we fix the number of timeseries contained in the storage engine as 5000 and benchmark the query performance under different sample intervals (data densities). For the x-axis of Figure 14, different numbers represent different data densities. From Figures 14(a)(b)(c)(d)(f), we can see the query latency of our Heracles prototype under both random and real monitoring datasets is much lower than that of the Prometheus tsdb. For the random dataset, the query latency is slightly larger



(a) 1-8-1 (b) 5-1-1 (c) 5-1-12

(d) 5-8-1 (e) double-groupby-1 (f) lastpoint

Figure 13: Query durations of TSBS patterns under different number of timeseries.



(a) 1-8-1 (b) 5-1-1 (c) 5-1-12

(d) 5-8-1 (e) double-groupby-1 (f) lastpoint

Figure 14: Query durations of TSBS patterns with 5000 timeseries under different data densities.

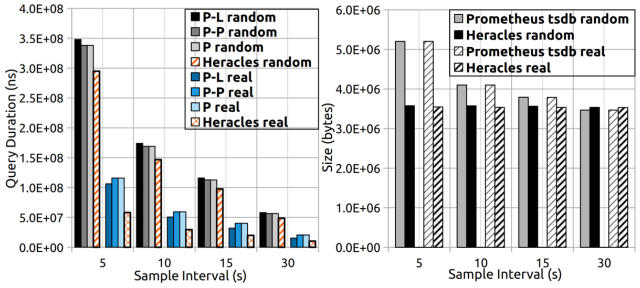
compared with the real monitoring dataset; this is because it takes more compressed bits and more time to write the random dataset after XOR compression. Besides, when the sample interval increases, the query latency becomes lower, which is because the size of the querying data becomes smaller since the data density decreases. For Figure 14(e), similar to the previous experiment, the fraction of the duration of locating the first sample becomes small and the overall query latency of Heracles becomes closer to that of Prometheus tsdb. On average, for the experiments of Figure 14, Heracles has 29% and 33% improvements over Prometheus tsdb for the random and real monitoring datasets, respectively.

4.4.4 Compressed Data Size Comparison. In this section, we compare the compressed data size between Prometheus tsdb and Heracles.

For in-memory data, after the benchmarks in §4.4.3 finish, we observe a 12% data size reduction of Heracles over Prometheus tsdb. For on-disk data, similar to the benchmark in Part 2 of §4.4.3, we insert 12-hour data from both random and real monitoring datasets into Prometheus tsdb and Heracles, respectively. Figure 15 shows the size comparisons for compressed timeseries data and index under different data densities, respectively.

For the block data, on average, Heracles has 13% and 50% of data size reduction over Prometheus tsdb for the random and real monitoring datasets, respectively. For the index, on average, Heracles has a 12% of data size reduction over Prometheus tsdb. Besides, the size of the index of Prometheus tsdb increases with data densities.

4.4.5 Tuple Size Trade-off. As mentioned in §3.1.1, in our implementation, we choose eight as the tuple size because it provides a good balance between the query performance and compression ratios. In this section, we explore the influences of query performance and compression ratios under different tuple sizes. We can see from Figure 16(a) that the query latency slightly increases with the tuple size because of more decompression during querying. Besides, since more samples can be compressed continuously when the tuple size becomes larger, Figure 16(b) shows a slowly decreasing trend of the compressed data size as the tuple size grows.



(a) Size of on-disk data (b) Size of on-disk index

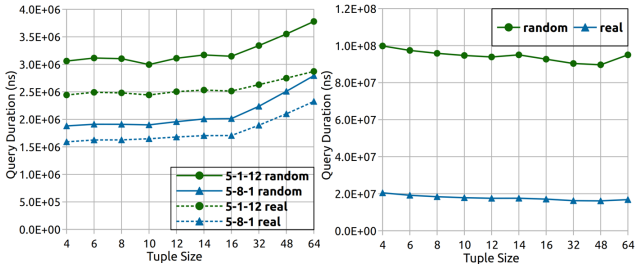
Figure 15: Data size comparison.

4.5 Data Flushing Evaluation

In this section, we evaluate the data flushing duration and query latency during data flushing. Figure 4(b) shows the data flushing durations of Prometheus tsdb and Heracles without the memory manager. We test with three kinds of data – random, increasing counter, and real monitoring data. We flush 2 hours' in-memory data of 10000 timeseries with 15 seconds interval to disks and record the duration. For Heracles, we first evaluate the data flushing duration and the corresponding CPU usage without concurrency where the groups are converted and flushed sequentially. The data flushing duration is 3.4 times higher than that of Prometheus tsdb because the model conversion of Heracles is time-consuming and Prometheus tsdb simply flushes the chunks to disks with the same format for in-memory and on-disk data.

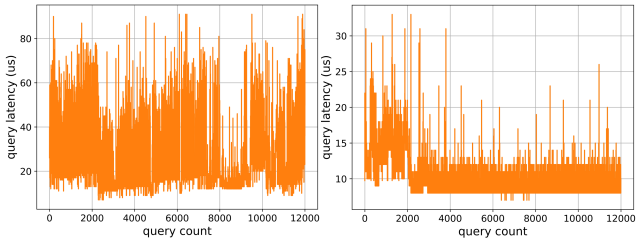
To reduce the data flushing overhead, because the model conversion of each timeseries in the same group is independent, we can launch a Goroutine for model conversion for each timeseries of the same group (around 100 Goroutines running concurrently) and write the converted results to disks sequentially. The data flushing duration is then reduced to 55% higher than that of Prometheus tsdb. However, as we can see from the right part of Figure 4(b), the CPU usage of Heracles is 1.8 times higher than that of Prometheus tsdb which can affect the query performance during the data flushing.

Next, during the data flushing, we continuously query the last 15 minutes' data of one timeseries in a background Goroutine and record the latency of each query for Prometheus tsdb and Heracles with concurrency, respectively. Figure 4(c) shows the CDF of query latency during the data flushing. The P99.99 and maximum query latencies of Heracles with concurrent data flushing are 25% and 28% higher than those of Prometheus tsdb, respectively.



(a) Query performance (b) On-disk data size

Figure 16: Query performance and on-disk data size under different tuple sizes.



(a) Prometheus tsdb (b) Heracles

Figure 17: Query performance during data flushing.

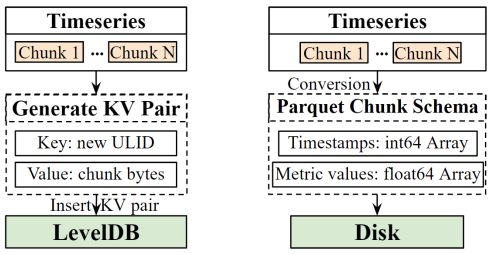
Our memory manager can gradually flush and reclaim in-memory data. In Figure 17(a) and Figure 17(b), after adding the memory manager into Heracles, we repeat the above experiment, in which a significant tail latency reduction can be observed. For instance, the P99.99 and maximum query latencies of Heracles are 70% and 64% lower than those of Prometheus tsdb, respectively, during data flushing.

4.6 Comparison with State-of-the-art Storage Techniques

In this section, we explore the possibility of handling performance monitoring timeseries with state-of-the-art storage techniques. Specifically, we replace the back-end storage of Prometheus tsdb with LevelDB and Parquet and compare the performance with Prometheus tsdb and Heracles.

Insertion throughput. As shown in Figure 12, the insertion throughput of both the LSM integration and the Parquet integration are similar to that of the original tsdb. This is because we manage the in-memory data with the same mechanism of Prometheus tsdb, and we only apply the integration on the on-disk data.

Prometheus tsdb LSM-based KV Store Integration. First, we replace the back-end storage of Prometheus tsdb with LSM-based KV Store that has a write-optimized design. We choose LevelDB [17], a widely used LSM-based KV store for the integration. As shown in Figure 18(a), for each chunk in Prometheus tsdb, we generate a unique ULID [32] as the key of the inserted KV pair, use the chunk contents as the value, and insert the KV pair into LevelDB. Figure 14 shows the query latency under different data densities. In Figure 13, on average, the query latency of the LevelDB integration is 12% higher than that of Prometheus tsdb. This is mainly because of the poor data locality as the chunks of the same timeseries may be scattered in different SSTables.



(a) LevelDB (b) Parquet

Figure 18: Prometheus tsdb integration with State-of-the-art techniques.

Prometheus tsdb Parquet Integration. Second, we explore the possibility of leveraging columnar storage techniques to store performance monitoring timeseries data. We choose Parquet [14], a state-of-the-art columnar storage format for the integration. When flushing in-memory data chunks, we leverage a Parquet schema as shown in Figure 18(b) to store the data chunks into Parquet files [51]. As shown in Figure 13, the query latency of the Parquet integration is similar to that of the original Prometheus tsdb (3% higher). This is mainly because Prometheus tsdb can better compress timeseries data compared to Parquet. On the other hand, similar to Prometheus, Parquet also suffers from the decompression overheads as discussed in §2.3.1.

4.7 Discussion

In this section, we discuss the limitations and possible future directions.

Data Diversity. Currently, we only support 64-bit double-precision floating-point number for timeseries metric values like Prometheus. In the future, we will add support for more data types and design the corresponding compression methods to integrate into our storage model.

SQL Support. State-of-the-art timeseries systems like Prometheus and InfluxDB develop their own query languages (PromQL [40] and Flux [23], respectively). Since Heracles is implemented based on Prometheus tsdb, we follow the interfaces of Prometheus tsdb. In the future, we will add an SQL interface for Heracles to support complex queries and easy integration with existing SQL tools.

Further Comparison. In this paper, we utilize LevelDB for the comparison with LSM-tree key-value stores. However, partitioned LSM-based B+-tree design such as AsterixDB [2–4] may better suit timeseries data management with data of the same timeseries gathered together, which can be a future direction for timeseries management with key-value stores.

5 RELATED WORK

Columnar Stores. C-Store [44] is a read-optimized relational DBMS which partitions projections horizontally into segments and each segment is associated with sort order. Vertica [29] is implemented based on C-Store, adding super projection and different denormalized projections with different physical layouts to avoid expensive join operations. MonetDB [21] leverages a similar storage model to Vertica which stores each column in a logical table to a separate table. However, similar to Parquet, the compression schemes of the above systems are not optimized for performance monitoring timeseries. Besides, data flushing specially-tailored for timeseries data is critical, which is not provided by these systems.

Timeseries Databases. Gorilla [13, 33] is a fast in-memory timeseries database that proposes a widely used compression scheme for timeseries data. We further optimize Gorilla's compression scheme as discussed in section 3.1.2. InfluxDB is another widely used timeseries database that provides multiple compression schemes for integers [6, 41, 50] and leverages XOR compression for floating-point numbers. However, since InfluxDB manages timeseries independently like Prometheus, it exposes the same problems as discussed in §2. BtrDB (Berkeley Tree Database) [5] focuses on high-precision, high-sample-rate telemetry timeseries. Timon [9] is a timestamped event database. It mainly focuses on the analysis of massive and long-term events and utilizes SEDA programming model [49] which is different from state-of-the-art timeseries databases. Besides, Timon does not have a data re-utilization mechanism as Heracles does.

Main-Memory Databases. Main-memory databases mainly store data in memory. However, when the data size scales up, databases need to evict some data to disks where the memory reclamation algorithms are leveraged [16, 42, 47]. Anti-Caching [10] eliminates the caching layer by leveraging a fine-grained eviction mechanism that gathers the cold tuples and writes them to disks asynchronously. LeanStore [30] focuses on in-memory data management beyond main memory. It introduces three states for a data page (hot, cooling, and cold) where the evicted data is first put into the cooling queue and then gradually released to the cold stage. Besides, to avoid the read/reclamation race for the data in the cooling queue, the original epoch-based reclamation algorithm [16] is modified so the reading threads and the pages in the cooling queue are attached with epochs. The page in the cooling queue can be released to the cold stage only when its epoch is smaller than the minimum epoch among all the current reading threads. Different from LeanStore, we further partition the hot area according to the time ranges and assign different reclamation schemes for different partitions. Besides, the extra epoch attached in each partition helps quick reclamation when the time range reaches the data retention time and also helps query threads differentiate whether the time range is still valid.

Multiversion Concurrency Control. MVCC [8] applied in the state-of-the-art database systems like Microsoft Hekaton [12] commonly leverages a version chain to represent data modifications. Each tuple contains a begin and an end timestamps to represent the version which helps decide the visibility to a transaction. Besides, the version also helps garbage collectors in data reclamation according to the timestamp of the oldest active transaction. MVCC is not adopted by Heracles because data samples are immutable once being compressed into memory blocks. Therefore, we only need to avoid the read/reclamation race so epoch is good enough to decide whether an outdated data block is currently being accessed by active readers.

6 CONCLUSION

In this paper, we present Heracles for performance monitoring timeseries, which contains an efficient group-based storage model and a novel two-level epoch-based memory manager for better data flushing. Heracles not only improves the query performance, but also achieves a higher write throughput and a better compression ratio. In our experiments, compared to Prometheus tsdb, we achieve 171% higher write throughput, 32% lower query latency, and 30% lower space usage on average.

ACKNOWLEDGMENTS

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15273616, GRF 15206617, GRF 15224918), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055096).

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. ACM, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. 2020. An LSM-Based Tuple Compaction Framework for Apache AsterixDB. *Proc. VLDB Endow.* 13, 9 (May 2020), 1388–1400. https://doi.org/10.14778/3397230.3397236

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[4] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (June 2014), 841–852. https://doi.org/10.14778/2732951.2732958

[5] Michael P Andersen and David E. Culler. 2016. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 39–52. https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen

[6] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Software: Practice and Experience* 40, 2 (2010), 131–147. https://doi.org/10.1002/spe.948

[7] Andrea Arcangeli, Mingming Cao, Paul E McKenney, and Dipankar Sarma. 2003. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel.. In *USENIX Annual Technical Conference, FREENIX Track*. 297–309.

[8] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[9] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. 2020. Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 739–753. https://doi.org/10.1145/3318464.3386136

[10] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[11] Google Developers. 2020. *Protocol Buffers - Base 128 Varints.* https://developers.google.com/protocol-buffers/docs/encoding#varints.

[12] Cristian Diaconu, Craig Freedman, Erik Ismert, Paul Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *ACM International Conference on Management of Data 2013* (sigmod 2013 ed.). https://www.microsoft.com/en-us/research/publication/hekaton-sql-servers-memory-optimized-oltp-engine/

[13] Fackbook. 2020. *Beringei - A high performance, in memory time series storage engine.* https://github.com/facebookarchive/beringei.

[14] Apache Software Foundation. 2020. *Apache Parquet.* https://parquet.apache.org/.

[15] The Apache Software Foundation. 2020. *Apache HBase Project.* https://hbase.apache.org/.

[16] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[17] Sanjay Ghemawat and Jeff Dean. 2020. *LevelDB.* https://github.com/google/leveldb.

[18] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2008. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems* 20, 8 (2008), 1173–1187.

[19] M. Großmann and C. Klug. 2017. Monitoring Container Services at the Network Edge. In *2017 29th International Teletraffic Congress (ITC 29)*, Vol. 1. 130–133. https://doi.org/10.23919/ITC.2017.8064348

[20] Thomas Hart, Paul Mckenney, Angela Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67 (12 2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

[21] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35.

[22] Docker Inc. 2020. *Collect Docker metrics with Prometheus.* https://docs.docker.com/config/thirdparty/prometheus/.

[23] InfluxData Inc. 2020. *Flux data scripting language.* https://docs.influxdata.com/influxdb/v2.0/reference/flux/.

[24] LogicMonitor Inc. 2020. *LogicMonitor Case Studies.* https://www.logicmonitor.com/case-studies.

[25] influxdata. 2020. *InfluxDB 1.7 Documentation.* https://docs.influxdata.com/influxdb/.

[26] Influxdata. 2020. *InfluxDB Query.* https://docs.influxdata.com/influxdb/v2.0/api/#operation/PatchDashboardsIDCellsIDView.

[27] Jing Han, Haihong E, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*. 363–366. https://doi.org/10.1109/ICPCA.2011.6106531

[28] William Kennedy. 2020. *Scheduling In Go : Part II - Go Scheduler.* https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html.

[29] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[30] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[31] Maged M Michael. 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. 21–30.

[32] OKLog. 2020. *Universally Unique Lexicographically Sortable Identifier.* https://github.com/oklog/ulid.

[33] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[34] Bartlomiej Plotka. 2020. *Thanos - Highly available Prometheus setup with long term storage capabilities. A CNCF Incubating project.* https://github.com/thanos-io/thanos.

[35] Prometheus. 2020. *Exporter for MySQL server metrics.* https://github.com/prometheus/mysqld_exporter.

[36] Prometheus. 2020. *Node exporter - Exporter for machine metrics.* https://github.com/prometheus/node_exporter.

[37] Prometheus. 2020. *Prometheus - Defining recording rules.* https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/.

[38] Prometheus. 2020. *Prometheus - From metrics to insight, power your metrics and alerting with a leading open-source monitoring solution.* https://prometheus.io/.

[39] Prometheus. 2020. *Prometheus Range Queries.* https://prometheus.io/docs/prometheus/latest/querying/api/#range-queries.

[40] Prometheus. 2020. *PromQL.* https://prometheus.io/docs/prometheus/latest/querying/basics/.

[41] A. H. Robinson and C. Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proc. IEEE* 55, 3 (March 1967), 356–364. https://doi.org/10.1109/PROC.1967.5493

[42] Michael Scott and Maged Michael. 1995. Correction of a Memory Management Method for Lock-Free Data Structures. (1995).

[43] Chris B. Sears. 2000. The Elements of Cache Programming Style. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4* (Atlanta, Georgia) *(ALS'00)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1268379.1268397

[44] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564. http://dl.acm.org/citation.cfm?id=1083592.1083658

[45] Yandex ClickHouse team. 2020. *ClickHouse.* https://clickhouse.tech/.

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

[47] John D Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. 214–222.

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

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

[50] Jason Wilder. 2020. *simple8b Golang implementation.* https://github.com/jwilder/encoding/tree/master/simple8b.

[51] xitongsys. 2020. *Pure golang library for reading/writing parquet file.* https://github.com/xitongsys/parquet-go.