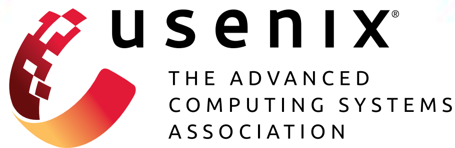


TVStore: Automatically Bounding Time Series Storage via Time-Varying Compression

Yanzhe An, *Tsinghua University;* Yue Su, *Huawei Technologies Co., Ltd.;*
Yuqing Zhu and Jianmin Wang, *Tsinghua University*

https://www.usenix.org/conference/fast22/presentation/an

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

TVStore: Automatically Bounding Time Series Storage via Time-Varying Compression

Yanzhe An[1], Yue Su[2]†, Yuqing Zhu[✉1]‡, Jianmin Wang[1]
[1]Tsinghua University, [2]Huawei Technologies Co., Ltd.

Abstract

A pressing demand emerges for storing extreme-scale time series data, which are widely generated by industry and research at an increasing speed. Automatically constraining data storage can lower expenses and improve performance, as well as saving storage maintenance efforts at the resource-constrained conditions. However, two challenges exist: 1) how to preserve data as much and as long as possible within the storage bound; and, 2) how to respect the importance of data that generally changes with data age.

To address the above challenges, we propose time-varying compression that respects data values by compressing data to functions with time as input. Based on time-varying compression, we prove the fundamental design choices regarding when compression must be initiated to guarantee bounded storage. We implement a storage-bounded time series store TVStore based on an open-source time series database. Extensive evaluation results validate the storage-boundedness of TVStore and its time-varying pattern of compression on both synthetic and real-world data, as well as demonstrating its efficiency in writes and queries.

1 Introduction

Time series databases are becoming the most popular type of databases in recent years [2]. We are witnessing a growing demand for time-series-specific storage and processing from many fields such as cluster monitoring [91], Internet of Things [6], finances [80], medicine [51], and scientific research [63]. In fact, the fast increasing volume of time series data has placed an unprecedented requirement on computing resources, especially storage space [6, 79].

An effective storage management strategy that can constrain the storage space is desirable and important for time series databases. While large organizations can afford the storage to hold the ever-growing time series data, small or medium-sized entities prefer to strike a good balance between data volume and storage cost [45]. Besides, storage space is restricted in some specific deployments, e.g., real-time monitoring at far remote sites [8, 19, 78]. On the other hand, as the significance of time series data is highly correlated with the age of the data [22, 37, 89], it is desirable to have a storage management strategy that takes data ages into account [3, 7].

Significant prior work has addressed the storage-control problem by compression, which can be lossless or lossy. Lossless compression [10, 33, 57, 71, 73] preserves the complete data, but its achievable upper bound on compression ratio [93] might not be satisfactory for applications. Hence, time series databases commonly control storage consumption by directly discarding data older than a given time [43] or exceeding a storage threshold [67]. But discarding historical data causes a loss [94]. For example, historical data are crucial for long-term observations and enabling new scientific knowledge creation in the future [63]. Besides, time-based retention policy might not bound the data volume in case of unevenly spaced time series with unknown arrival intervals. Another common approach is to exploit lossy compression [15, 41, 65], which preserves partial data and trades off precision for space. But existent approaches to lossless and lossy compression are only best-effort about the final size of compressed data size [13, 24, 99].

In this paper, we take a new approach towards controlled storage space for time series stores. We consider the problem of automatically bounding the storage of a time series store by compression. To enable this, our key insight is that time series data can be compressed *losslessly* or *lossily* according to its *importance,* which is in turn related to its age, as users commonly accept information loss on less important old data [12, 14, 23, 38, 40]. We control the storage space by time-varying compression, which compresses data in a sequence of ratios defined by a time-dependent function. Inspired by time-decayed windowing of stream processing [9, 22], our design of time-varying compression takes the chunking-and-varied-segmentation approach, accepting user-defined time-dependent functions and fixed-ratio compressors.

To automatically bound time series storage by compression, three fundamental challenges exist. The first is deciding when to start the time-varying compression, i.e., the proper moment when 1) it is not too late that the storage space is exceeded during compression; and, 2) it is not too early that unnecessary compression is applied to some recent data, for preserving as much information as possible. The second challenge is computing the proper compression ratio r, given which the

†Co-first author. Work done at Tsinghua University.
‡The corresponding author (zhuyuqing@tsinghua.edu.cn).

sequence of compression ratios can be deduced using a time-dependent function. r should not be too large to prevent discarding information unnecessarily and meantime not be too small to exceed the storage bound. The third challenge is finding out how to run the time-varying compression, i.e., whether to compress data in an online stream processing manner or in a batch processing manner. The goal is to reduce computing resource consumption and improve performance.

To address these challenges, we propose TVStore, a storage-bounded time series store built upon time-varying compression. TVStore can automatically and effectively bound the time series storage even if data keep being ingested. We implement TVStore by extending the storage engine of an open-source time series database named Apache IoTDB [96]. Hence, all the database functions and operations remain supported in TVStore. We evaluate TVStore in extensive experiments based on synthetic data and real-world data. Results validate the storage-boundedness of TVStore and its time-varying pattern of compression. The compression technique employed TVStore incurs low overhead compared to its baseline. It is efficient in writes and reads, 3×(25×) and 35×(8.7×) faster than the state-of-the-art(state-of-the-practice) related works [3, 67] respectively. Under the same conditions, TVStore can respond to queries with much lower error rates in most cases than the related work.

In sum, we make the following contributions in this paper:

- We propose a time-varying compression framework TVC, which can compress data by varied ratios complied with a given time-dependent function that corresponds to the age-varying importance of time series data.
- We design a time series store TVStore that can automatically run the time-varying compression framework TVC at the *proper* time, effectively bounding the storage space to a specific threshold while preserving data according to the time-varying importance for applications. To the best of our knowledge, TVStore is the first time series store that can automatically bound its storage space by time-varying compression patterns.
- We implement TVStore based on an open-source time series database[1], introducing a three-layer data reduction scheme and exploiting a line generalization algorithm as the fixed-ratio compressor for TVC.
- We run extensive experiments using synthetic and real-world data to demonstrate the efficiency and advantage of TVStore in comparison to three related time series stores, as well as to validate its storage-boundedness and time-varying pattern of compression.

2 Background and Motivation

2.1 Why Constrain Storage

Time series databases are gaining an increasing popularity [1]. Rather than processing time series as streams and analyzing

[1] https://github.com/thulab/TVStore

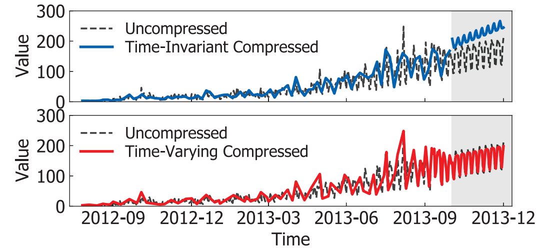


Figure 1: Time series predictions by data compressed in the *time-varying (TV)* vs. *time-invariant (TI)* manner. Predictions lie in the gray area. TV-compressed data have varied compression ratios for data at different ages, while TI-compressed data have the same compression ratio at all times. Both cases have the same overall compression ratio.

only once, mounting demands have emerged for keeping time series data for future analysis [94]. But time series data are generated at a growing speed that is outpacing the increase of computing capabilities [17, 79]. Many application scenarios cannot afford enough computing resources such as storage and network bandwidth to accommodate the processing needs for time series data. Storage-bounding compression can enable the control of storage cost.

Limited storage expense. Many medium or small entities have to limit their expense on storage in their daily operations, even though the public clouds have the capacity to keep all their data [94]. As value is yet to be extracted from the huge volume of time series data, it is desirable to automatically keep as much data as possible within the storage constraint.

Sensors of a connected car can generate about 30 terabytes (TB) of data per day [62, 77]. Time series data is among the major components of the generated data. To hold all the data on such moving vehicles, large disks are installed. Since a 30TB disk can cost around $1200, a month's worth of data can fill up a 960TB disk, causing a cost of $30,000. This adds an unrealistic amount to a vehicle's price, but keeping as much data as possible can enable valuable data analytics [77, 83].

Limited computing resources. In the oil and gas industry, a typical offshore oil platform generates more than 1TB of data [19] daily. But common data transmission via satellite connection allows only a speed from 64 Kbps to 2Mbps for these offshore oil platforms. If all data are transmitted back for processing, it would take more than 12 days to move 1 day's worth of data to the processing backend [8]. Data compression is demanded for reducing data in both transmission and storage.

Scientific research applications nowadays are producing too much data to be stored or processed efficiently. For example, cosmological simulations generate petabytes of data per simulation run [34] and climate simulations generate tens of terabytes per second [29]. Such large volumes of data are imposing an unprecedented burden on storage and computation. Data reduction is necessary to enable data processing and analytics within a reasonable amount of resource and time [92].

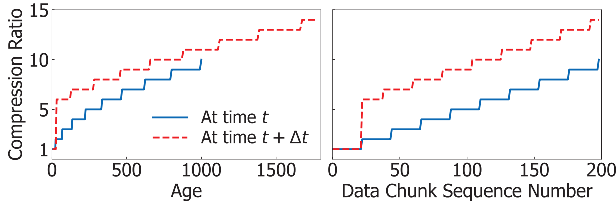


Figure 2: Compression ratio sequences generated by a function of the data age at time t and $t+\Delta t$, for reducing different data volumes to the same size, i.e., 200 data chunks. Data volumes increase with time.

2.2 Time-Varying Importance of Data

The importance of time series data changes along with time, as reflected by applications' favoring recent data over old data [5, 18, 31], or favoring some events at certain moments over others [49,83]. Time-changing importance of data in fact commonly exists in natural and scientific phenomena [75, 86, 98]. As a result, we have seen a plethora of research on data series analysis and prediction considering the time-changing pattern [9,22,36,37,89]. Figure 1 illustrates how the importance of data varies with time in time series prediction, which is widely used in applications [39,48,59,76,88]. Recent data have dominant impact on the result of prediction, making the time-varying compression outperform the common time-invariant compression.

Time-changing importance of time series data can be exploited to form time-varying compression. For important data, we compress them *losslessly* or with a low ratio by *lossy* compression. For unimportant data, we compress them by a high compression ratio. As time series data can be identified by timestamps, we use a time-dependent function to denote the changing importance of data. Hence, the compression ratios can also be deduced from the function. Time-varying compression can suit users' requirements on data analysis well and save storage space to the most extent. As shown in Figure 2, the power-law function t^{β} with $\beta = 1$ is used for depicting time-changing importance of data and exploited to define time-varying compression ratios in both graphs. As data keep arriving, the compression at a later time reduces more data to the same volume as that at an early time, but by higher ratios as generated according to function t^{β}.

2.3 Automatic Compression and Bounding

To meet the above requirements of time series applications, we propose time-varying compression that respects the time-varying importance of time series data. Furthermore, we propose the design of a time series store that automatically bounds the total storage space to effectively control costs. To this end, compression must be initiated at proper moments to cap the overall storage space, as data increase. The compression ratios must be computed automatically, with compressions initiated at proper moments. These moments must be computed carefully such that users can keep data to its highest precision as long as possible. We must deduce the proper moments for compression initiation when 1) it is not too late that the storage space is exceeded during compression; and, 2) it is not too early that unnecessary compression is applied to some recent data or that an improperly high compression ratio is used. Besides, when lossy compression is used, users would need the overall error rates for understanding the data analysis results they could expect. The error bound computation must evolve along with time-varying compression. And, users are allowed to request the removal of data with high error rates.

Challenges: As a result, two main challenges exist in automatically bounding the time series storage by time-varying compression: 1) how time-varying compression can be executed on an ever-increasing volume of data and with error bounds computed, when the compression ratios keep changing as shown in Figure 2 (§3); and, 2) how to automatically decide the conditions for running time-varying compression such that storage space is always bounded but not too much (§4).

3 Time-Varying Compression

Given a time series, time-varying compression (TVC) compresses it to an overall compression ratio no smaller than a user-specified threshold r. TVC compresses data by the unit of chunk, which is time series data within a time interval. The compression ratios vary for different chunks according to a time-dependent function $r(t)$'s definition, where the input t is a data chunk's age as relative to the most recent timestamp of the time series. TVC enforces the compliance to different compression ratios defined by $r(t)$. The benefits of this compliance is that different $r(t)$ can be used for various use cases [22]. Provided with properly designed $r(t)$ and compressor, TVC can even achieve functionally lossless compression [54] for a long range of data.

The key challenge of time-varying compression is *how to continuously preserve the compliance with any $r(t)$ definition, when a data chunk's age increases along with the data volume*. To address this challenge, TVC initiates rounds of compression on data chunks iteratively. Figure 3 overviews time-varying compression in rounds. Later rounds of compression must execute on differently compressed data. **Two problems must be tackled**: 1) how to compute the correct sequence of compression ratios to enforce the compliance to a given $r(t)$ (§3.1, §3.2); and, 2) what properties a compressor must have to guarantee a feasible time-varying compression process, besides the fixed-ratio requirement(§3.3).

3.1 Ratio Sequencing and Data Chunking

For an overall compression ratio $\bar{r}$, TVC first finds a sequence of compression ratios $r_1, r_2, ..., r_k$ defined by the time-dependent function $r(t)$. The average of the compression ratio sequence $r_1, r_2, ..., r_k$ should approximate $\bar{r}$. The time-

dependent function $r(t)$ produces a compression ratio when given an integer t. Here, a smaller t is a time interval closer to the most recent time of a time series. Decay functions [3, 22, 75] commonly used in time series analysis can be used for $r(t)$, e.g., exponential function ($e^{\alpha t}$) and polynomial/powerlaw function (t^{β}). $r(t)$ can also be a constant function (C), but then TVC degrades to a common lossless/lossy compressor.

We assume the data for compression is kept in the unit of chunks, as time series stores commonly keep data in units like chunk [96] or block [4, 43]. TVC executes compression by the unit of chunk. To guarantee that data are compressed to the compression ratio sequence $r_1, r_2, ..., r_k$, TVC groups r_i data **chunks** into the ith **segment**. Each segment is then compressed to an output chunk. Hence, the ith chunk of the compression output has a compression ratio r_i, complying to the definition of $r(t)$.

Moreover, the compression ratio sequence must guarantee that 1) all data chunks to be compressed are actually processed; and, 2) the actual compression ratio is no smaller than $\bar{r}$ to avoid exceeding the storage bound. Hence, the sum of compression ratios must be no smaller than the number m of raw data chunks to be compressed. Besides, the average of compression ratios must be no smaller than $\bar{r}$. We then approximate $\bar{r}$ by the average of the smallest sequence of $r_1, r_2, ..., r_k$ that satisfy the following equations:

$$\Sigma_i^k r_i \geq m \quad (1)$$

$$m/k \geq \bar{r} \quad (2)$$

It is possible that no such sequence complied with $r(t)$ is found to satisfy both of the two equations, if $r_1 = r(1)$. Hence, TVC allows the compression ratio sequence r_1 to be $r(i)$, with $r_k = r(k+i-1)$. But TVC requires that the sequence $r_1, r_2, ..., r_k$ is non-decreasing, which means that $r(t)$ must be a non-decreasing function. The condition is necessary to avoid that some data chunks have a lower compression ratio in later rounds, while they are compressed in a higher ratio in a previous round. In fact, this condition naturally follows from the fact that data are aging and must be compressed with no lower ratios in later compression rounds.

3.2 Virtual Decompression and Compressions

TVC initiates a new round of compression when necessary, e.g., when conditions for constraining storage are met (discussed in Section 4). In rounds other than the first, the compression is executed on differently compressed data. It is difficult to compute the actual compression ratios based on data chunks compressed to different ratios. But the actual compression ratios are needed in enforcing the compliance to the time-dependent function $r(t)$, according to equation (2).

To compute actual compression ratios, TVC adopts the technique of *virtual decompression*. For the compression round n, TVC does not compute the compression ratios based on the compressed data from the last round. Rather, given the data chunks to be compressed in round n, TVC

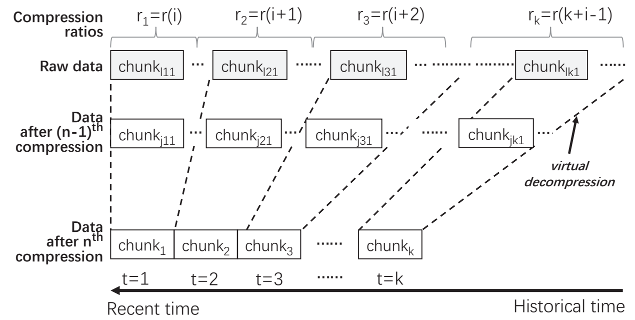


Figure 3: Time-varying compression in rounds. In each round, data of different ages are compressed to ratios that change according to a time-dependent function.

virtually decompresses them, by *mathematical mapping*, to the original raw data chunks for computing the sequence of compression ratios. Then, the conditions for ratio sequencing are considered. Thanks to the chunk-based data unit, virtual decompression can be supported by recording the number of original raw data chunks in every compression round.

Virtual decompression enables the generation of a compression ratio sequence based on the original raw data even after rounds of compression. Only by virtual decompression could the compressed data always follow the time-dependent function $r(t)$'s definition. Otherwise, data can only be compressed following the exponentially decaying pattern, as compression on compressed data leads to the multiplication of compression ratios. This would limit the applications of TVC, as $r(t)$ can only be an exponential function.

Algorithm 1 presents the main algorithm of time-varying compression for a time series. The input to the algorithm includes the number of actual chunks to be compressed and the target overall compression ratio $\bar{r}$. The algorithm consists of three parts. The first two parts guarantee the two conditions as specified by Eq. (1) and (2) while approximating $\bar{r}$ by $r_1, r_2, ..., r_k$. The third part actually compresses the data chunks by the ratio sequence.

In the first part of Algorithm 1, virtual compression is applied to the actual data chunks such that the corresponding number m of raw data chunks is obtained (line 2). According to Eq. (1), an initial sequence of compression ratios is obtained (line 4-8). As discussed above, the condition specified by Eq. (2) is not necessarily satisfied, even if Eq. (1) is met; and vice versa. Therefore, we refine the sequence to satisfy both Eq. (2) (line 9-12) and Eq. (1) (line 13-17) by approximation.

3.3 Compressor and Error Bounding

In a compression round, TVC compresses r_i data chunks by varied compression ratios into a single data chunk. To guarantee that data in an output chunk are actually compressed by the same compression ratio, we can have TVC decompress all the input chunks and then apply the same compressor by the same compression ratio. But two problems exist. First,

Algorithm 1: Time-varying compression.

```
Input: m_a: number of data chunks to be compressed;
       r̄: overall compression ratio
1  ratioSeqQueue← ∅;
   /* To ensure the condition of Eq.(1)                        */
2  m ←virtualDecompress(m_a);
3  seqSum←0, j←0,i←0;
4  while seqSum< m do
5      j+=1;
6      seqSum+=r(j);
7      ratioSeqQueue.enqueue(r(j))
8  end
   /* To guarantee the condition of Eq.(2)                     */
9  while j-i+1> m/r̄ do                   // j−i+1 = k for Eq.(2)
10     j+=1;
11     seqSum+=r(j);
12     ratioSeqQueue.enqueue(r(j));
       /* To approach Eq.(1)'s equality condition             */
13     while seqSum≥ m do
14         i+=1;
15         seqSum-=r(j);
16         ratioSeqQueue.dequeue();
17     end
18 end
   /* To compress chunks by the ratio sequence                 */
19 while ratioSeqQueue.size> 0 do
20     compressOneChunk(ratioSeqQueue.dequeue());
21 end
```

as TVC takes iterative compression rounds, decompression before compression is highly inefficient. Second, if lossy compression is used, the decompressed data is imprecise. Rounds of decompression and compression can lead to a high deviation from the original data. A proper error bound on the lossily compressed data is desirable to users.

To avoid the above two problems, TVC requires the compressor to have the following three properties. First, compression on previously compressed data does not require decompression. Second, decompression on data compressed multiple times works the same way as on data compressed once. Third, the error bounds must be easily computed for the rounds of compression. While these properties seem to be restricted, proper approximation or representation models for time series data [26, 44, 69] are feasible choices, e.g., piecewise linear approximation (PLA) [27, 60, 87].

Among the various lossy compressors, PLA-based compressors compress a time series by approximating it using line segments. According to the related work [68], a line segment built from two line segments is the same as the line segment built from the original time series data, if line segments are properly constructed. Decompression on data at any round only needs to compute the linear function for a given time. Moreover, the mean bias error (MBE) of PLA can be computed easily even after rounds of compression. MBE is a commonly used metric for evaluating approximations [64, 81, 82]. For the ith compression round, MBE_i is the sum of round-relative error $MBE_{j-1,j}$ in previous rounds, i.e., $MBE_i = \Sigma_{j=1}^{i} MBE_{j-1,j}$, where $MBE_{j-1,j} = \frac{1}{n}\Sigma_{k=1}^{n} x_{j-1,k} - x_{j,k}$. Here, $x_{j,k}$ represents the decompressed value.

TVC accepts the specification of PLA compressors currently. TVC records the compression ratio and the error rate for every data chunk. After rounds of compression, there would be a time when some old data chunk has a high compression ratio and thus a high error rate. Keeping data at an extremely high error rate is no better than discarding it. Therefore, TVC allows users to specify a compression ratio r_{max} or an error rate e_{max}. TVC automatically discards data compressed at a ratio higher than r_{max} or at an error rate larger than e_{max}. If the compressor and the compression ratio-defining function $r(t)$ are properly chosen, TVC can achieve functionally lossless compression [54] for a long range of data, as well as supporting advanced analytical workloads [70].

4 TVStore: Automatic Storage Bounding

We propose TVStore that automatically bounds time series storage to a user-provided size using TVC as data keep being ingested. It allows users to set a recent data volume D_o that is not to be compressed. After reaching D_o, TVStore starts the compression at a *proper* time to avoid overrunning the storage bound or losing too much information. It monitors the storage consumption and initiates a process of time-varying compression when needed. Hence, three key design choices are made here:

How to compress: Shall compression be applied continuously to cold data in a batch-processing manner or hot data in a stream processing way [3]?

What ratio to compress: Will all compression ratios be feasible for storage bounding? If not, what is the proper compression ratio interval?

When to compress: When would be the proper time to start a TVC process that is neither too early to lose too much data nor too late to exceed the storage bound?

4.1 Compression on Hot Data or Cold Data?

TVStore exploits time-varying compression to bound the storage space. Compression can be applied to hot data as stream processing does [3]. It can also be applied to cold data in a batch processing mode. As TVStore targets resource limited environments, it is desirable to reduce the number of compression times and I/O accesses such that power consumption, memory utilization, and processor utilization can be reduced.

Consider the procedure of time-varying compression presented in Section 3. The compression ratios are equal to the segment sizes, in terms of data chunk numbers. As data in the largest segment is compressed the most times, it can be shown that such a segment after multiple compression rounds of TVC *has a smaller number of compression times* by cold-data compression than by hot-data compression.

In a compression round, a sequence of k segment sizes of $r(t)$, $t = i,...,k+i-1$ is generated, as shown in Figure 3. Let

$F(k)$ be the compression times of the kth segment after this compression. For the first round of compression on cold data, k segments are compressed into k chunks, with $F_c(k) = 1$.

As for hot data, in the compression round with the same data volume, the kth segment must be compressed from multiple smaller segment of chunks, since it continuously compresses smaller chunks into larger chunks whenever possible. To obtain the kth segment, we need segments with sizes summarized to $r_k = r(k+i-1)$, i.e.,

$$r_k = \sum_{t=j}^{k+i-2} a_t r(t) \tag{3}$$

Following Eq. (3), the compression times $F_h(k)$ of the kth segment is represented as follows:

$$F_h(k) = 1 + \sum_{t=j}^{k+i-2} a_t F_h(t) \tag{4}$$

As a result, $F_c(k) < F_h(k)$, i.e., $F(k)$ has a smaller value in cold-data compression than in hot-data compression.

For the latter rounds of cold-data compression, the kth segment will also be compressed from segments with smaller sizes. That is, Eq. (4) also applies to the latter rounds of the cold-data compression case. However, when the nth compression round is triggered, the largest segment k_n will be compressed from much smaller segments, the largest of which is k_{n-1}. Segments from $k_{n-1}+1$ to k_n do not exist until the nth compression.

In comparison, the k_n segment of the hot-data compression method must be compressed from segments having the largest one equal to $k_n - 1$. It can be shown that $k_{n-1} < k_n - 1 < k_n$. Considering Eq. (4), it follows that *the cold-data compression has a smaller number of compression times than the hot-data compression*. Hence, we have the following design principle.

Principle 1. *For a given range of time series data and a sequence of compression ratios, iterative compressions over cold data can reduce the compression rounds as compared to the continuous compression method on hot data.*

The result of Principle 1 has two indications for the design of TVStore: 1) TVStore should employ the cold-data compression rather than the hot-data compression to have a smaller number of disk I/Os; and, 2) TVStore can have higher performance using the cold-data compression, as the duration of and the cost of compression are smaller (§6.2 and §6.4).

4.2 Proper Compression Ratio Interval

A proper compression ratio is required to guarantee that the storage bound will never be violated. To compute the overall compression ratio, TVStore monitors the average read throughput v_r from the disk and the average write throughput v_w to the disk, as well as the ingestion throughput v_i by applications. Next, we describe how the proper compression ratio interval can be deduced as a design choice.

Consider when compression is started for the first time. The saved storage size ΔD by compression must be larger than the ingested data volume D_i in the whole compression process. Let D_r be the data volume to be compressed and read from the disk. Let D_w be the data volume after compression and written to the disk. ΔD is equal to the difference of D_r and D_w. Hence, we have the following equations:

$$\Delta D = D_r - D_w \geq D_i \tag{5}$$

Here, D_w is decided by the original data volume D_r and the relative compression ratio r_c, i.e.,:

$$D_w = \frac{1}{r_c} D_r \tag{6}$$

We assume that compression, reads and writes run concurrently for different time series. Reads take the most time. As a result, the time to generate data volume D_i is about the same as that for reading D_r. Thus, we have:

$$\frac{D_i}{v_i} = \frac{D_r}{v_r} \Rightarrow D_i = \frac{v_i}{v_r} D_r \tag{7}$$

Combining the above three equations, we have the following:

$$r_c \geq \frac{v_r}{v_r - v_i} \tag{8}$$

Eq. (8) points to the following two rules. First, the application ingestion throughput must be lower than the disk read throughput to enable the initiation of compressions. Second, the difference $v_r - v_i$ between the disk read throughput and the application ingestion throughput is the throughput that the disk allows for filling more data besides v_i. The ratio between v_r and $v_r - v_i$ is the lower bound on the ratio for compressing the data read from the disk. That is, the following design principle exists.

Principle 2. *To avoid overrunning a storage bound, the compression ratio r_c for each round of compression must be no smaller than $\frac{v_r}{v_r - v_i}$, where v_r is the average read throughput from the disk and v_i is the ingestion throughput by applications.*

Hence, we make the design choice in TVStore regarding the compression ratio r_c for each round of the iterative compression by Principle 2.

The overall compression ratio $\bar{r}$ can be deduced based on the round compression ratio r_c. Since r_c is greater than 1, $\bar{r}$ increases as compression rounds increase. If the user-specified max compression ratio r_{max} is reached and overly-compressed data began to be deleted, then data will need to be deleted in every later round. If deletion exists from the first round, it will greatly reduce the efficacy of TVC. Hence, r_c should be at least smaller than r_{max} to avoid this case. Thus, we have a loosely feasible range for r_c, i.e., $[\frac{v_r}{v_r - v_i}, r_{max}]$.

When $r_c = \frac{v_r}{v_r - v_i}$ and $\Delta D = D_i$, compression will be initiated consecutively. This will not only reduce the system performance but also wear out the storage device. If $r_c = r_{max}$, TVStore is not storing data with as much information as possible. As a general rule, TVStore sets r to the average of the two extremes, i.e., $r_c = \frac{1}{2}(\frac{v_r}{v_r - v_i} + r_{max})$.

4.3 Compression Initiation Time

TVStore initiates compression based on the monitored data storage. Compression is initiated when the data volume reaches a threshold D_c. For a given bound D_u on the storage space, TVStore must guarantee that D_u is not exceeded at any time during any of the compression rounds. The maximum storage consumption in all compression rounds is the key to decide the threshold D_c. We first find out when this maximum storage consumption is reached.

Figure 4 illustrates two compression rounds of TVStore. Consider the first round of compression. The threshold D_c is the data volume that triggers the first compression round. $\bar{r}_1$ is the target compression ratio of this first round. The meanings of D_u, D_o, D_r, D_w, and D_i are given and illustrated in Section 4.2 and Figure 4. v_i and v_r are the ingestion throughput by applications and the average read throughput from the disk respectively. The data D_r to be compressed is the difference between D_c and D_o, while D_r and $\bar{r}_1$ decides the written data D_w after compression, i.e.,:

$$D_r = D_c - D_o \tag{9}$$

$$D_w = D_r/\bar{r}_1 \tag{10}$$

A peak of storage consumption occurs at the time right before a compression round finishes, e.g., before $t2$ in Figure 4. At that time, the original data for compression is not deleted and the compressed data is written to the disk. Let the first peak storage consumption be D_1, we have:

$$D_1 = D_o + D_i + D_w + D_r \tag{11}$$

According to Section 4.2, when compression rounds follow one another consecutively, data is kept with the most information, i.e., taking up the most storage space. Then, we can deduce from the first compression round to the kth compression round. Due to the limit of space, we leave out the straight-forward deduction process. For the kth compression round with the target compression ratio $\bar{r}_k$, the peak storage consumption D_k is:

$$D_k = D_o + (1 + \frac{1}{\bar{r}_k} + \frac{v_i}{v_r})(D_w + D_i)\prod_{x=2}^{k-1}(\frac{1}{\bar{r}_x} + \frac{v_i}{v_r}) \tag{12}$$

From Eq.(12), we can deduce two possible cases for the maximum storage consumption. If $\frac{1}{\bar{r}_x} + \frac{v_i}{v_r}$ is no greater than 1, the maximum storage consumption is D_1; otherwise, it is D_k. From Section 4.2, we can deduce that $\frac{1}{\bar{r}_x} + \frac{v_i}{v_r} \leq 1$. As a result, **the maximum storage consumption occurs at the first round of compression**.

Thereupon, TVStore decides the compression initiation time based on the maximum space consumption. That is, we only need to guarantee that $D_1 \leq D_u$. With Eq.(11), we have:

$$D_1 = D_o + D_i + D_w + D_r \leq D_u \tag{13}$$

Combining Eq.(7), Eq.(9), and Eq.(10), we deduce that:

$$D_o + (\frac{v_i}{v_r} + \frac{1}{r} + 1)(D_c - D_o) \leq D_u \tag{14}$$

Hence, with Eq.(15) deduced from Eq.(14), the following design principle stands.

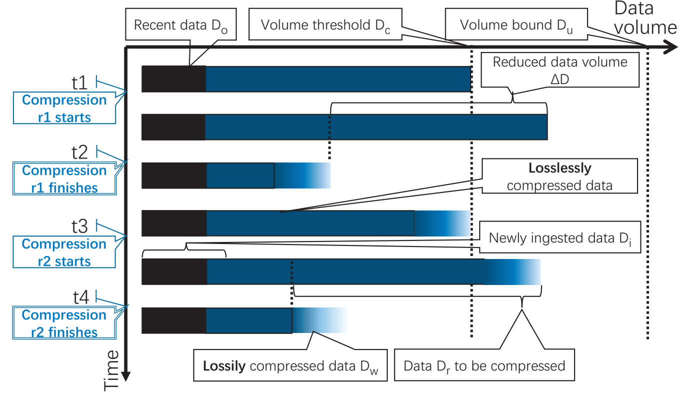


Figure 4: Storage bounding processes of TVStore: 1) at $t1$, compression round $r1$ starts when data volume reaches D_c; 2) when compression round $r1$ finishes at $t2$, storage space ΔD is saved through compression; and, 3) compression round $r2$ starts at $t3$, when data volume reaches D_c again. Data ingested during compression is D_i. Recent data D_o is not compressed lossily. The upper bound D_u of data volume is never exceeded at any time.

Principle 3. *Let D_u be the bound on the storage space and D_o be the recent data not to be compressed. Let v_r be the average read throughput from the disk and v_i the ingestion throughput by applications. Given the compression ratio $\bar{r}$ for a compression round, the threshold D_c of data volume to start a compression must satisfy the following condition.*

$$D_c \leq (D_u - D_o)/(\frac{v_i}{v_r} + \frac{1}{r} + 1) + D_o \tag{15}$$

TVStore initiates compression rounds based on results of Principle 3. Storage size and data volume monitoring is needed in the implementation of TVStore such that compression rounds can be initiated on time. Besides, the proper collection of the average metrics v_i and v_r is equally important to compute the initiation time.

5 Implementation of TVStore

We implement TVStore by extending an open-source time series database (TSDB), Apache IoTDB. The system architecture for TVStore is presented in Figure 5. TVStore replaces the TSDB storage engine by the time-varying compression/decompression storage engine. Ingested data directly go to the underlying TSDB storage. A monitoring thread runs in the background to automatically initiate time-varying compression (§3) on data in the storage when conditions are met (§4). Data are decompressed before being returned to the query engine. Hence, all the database functions originally supported by the TSDB remain supported. This architecture also allows complex analysis functions, which might be implemented in the query engine in the future, to be supported directly.

The TVStore storage engine accepts user-defined compressors for time-varying compression and time-dependent

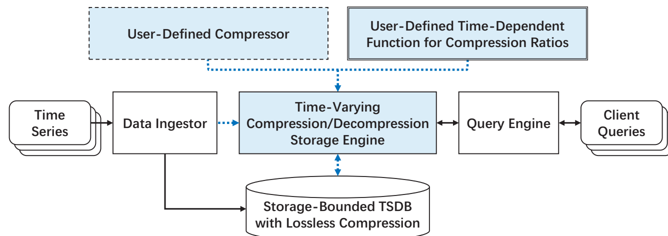


Figure 5: The architecture of TVStore: the filled components are TVStore's extensions over the time series database.

functions for compression ratios, as long as the corresponding Java interfaces are complied with. We have implemented a PLA-based compressor as the default compressor. While exponential, power-law and constant functions are all supported as the time-dependent ratio functions, TVStore uses the power-law function as the default. For time-varying compression, TVStore allows users to set the upper bound of storage space and the largest compression ratio permitted. Data volume monitoring is added to the storage engine to enable automatic storage bounding and to trigger time-varying compression rounds. Besides, we collect average metrics v_i and v_r by periodical monitoring and synopsis [21].

In the following section, we describe how the time-varying compression/decompression storage engine of TVStore integrates with the original TSDB. The choices for the compressor are also discussed as part of the TVStore implementation. The TVStore extension involves about 3000 lines of Java code.

5.1 Storage Engine Integration

In the implementation, the unit of data chunk is a data page in Apache IoTDB, each of whose data files consists of multiple data pages. When TVC compresses pages across multiple files, the involved files will be merged and restructured. Like the original IoTDB, TVStore keeps statistics and metadata on time series, as well as compression ratios of pages.

TVStore adds one layer of lossy compression to the two data-reduction layers of IoTDB. The resulting layers of data reduction are illustrated in Figure 6. Data within a data chunk are first compressed by the user-defined compressor. Then, the encoding techniques are applied to timestamps and values respectively. Encoding techniques include run-length encoding [73], Gorilla encoding [71], and delta encoding [10]. Finally, general compression as LZ4 [20] and snappy [32] is used to further reduce the overall size of stored data. The latter two layers of data reduction are lossless compression.

Although TVStore can be implemented with other TSDB, e.g., BtrDB [4] or InfluxDB [43], we have chosen Apache IoTDB [96] because its storage format enables the co-location of timestamps and values respectively within a data unit such that different encoding methods can be used to reduce data size. Besides, the structure, as well as the statistics and metadata kept within each data file, facilitates the support of TVC's iterative compression procedures.

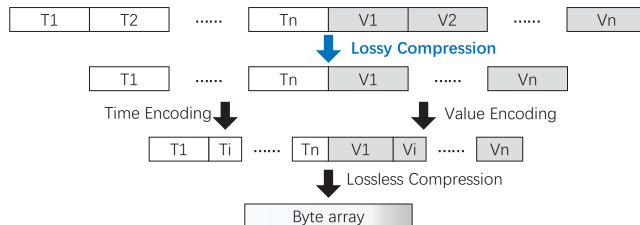


Figure 6: Layers of data reduction for one data chunk.

5.2 Fixed-Ratio Compressor/Decompressor

The time-varying compression of TVStore requires a fixed-ratio compressor to be specified. In the implementation, TVStore adopts a line generalization algorithm as the compressor and uses linear interpolation for decompression. Line generalization algorithms [95] commonly simplify one-dimensional curves by repeated eliminations of visually unimportant points, removing unnecessary details. They are inherently PLA-based compressors [87]. The number of preserved points can be set. Hence, the line generalization algorithm can be used as a fixed-ratio compressor.

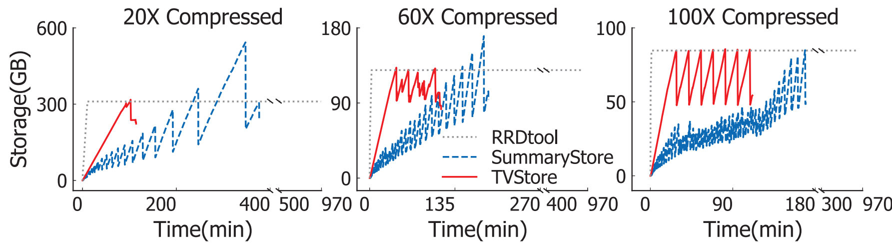
Specifically, TVStore leverages the line generalization method *LTTB* (largest triangle three buckets) [85], which is a variant of the widely accepted and used Visvalingam-Whyatt (VW) algorithm [95]. As compared to other line generalization algorithms, LTTB has much lower complexity. It can compress data in almost a single pass, while preserving visually important points like its counterparts. Simplicity and data preservation are two key features that lead to our choice of LTTB, as many users would naturally prefer storing real data values [13, 99], instead of approximate values.

Decompression exploits linear interpolation. Then, the number of interpolated points between preserved points must be decided. For evenly spaced time series with constant spacing of observation times, the number of interpolated points is computed based on time units. For unevenly spaced time series, we assume important points over a data chunk have a similar distribution as points between two important points. Let ρ be the number of preserved points divided by the number of the original points in a data chunk. As LTTB mainly preserves significant points during compression, we interpolate $\frac{k(1-\rho)}{\rho(k-1)}$ points between every two preserved points.

6 Evaluation

To evaluate TVStore, we consider five questions:

1. Can TVStore bound the storage size as expected? (§6.2)
2. How does TVStore's cold-data compression compare to the hot-data compression?(§6.2)
3. Can time-varying compression compress data according to a given time-dependent function? (§6.3)
4. How does compression influence TVStore's performance? (§6.4)
5. Can TVStore answer common queries within reasonable error bounds? (§6.5)



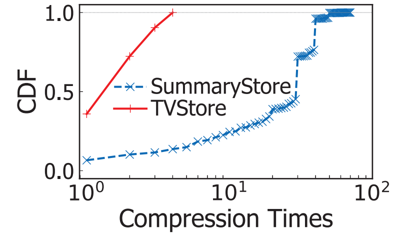


Figure 7: Storage bounding on intensive writes. All time series stores are ingested with 5TB data. TVStore and SummaryStore have the same final overall compression ratios of 20×/60×/100×. RRDtool has the same storage bounds as TVStore.

Figure 8: CDF of compression/merging times for cold data (TVStore) and hot data (SummaryStore).

6.1 Evaluation Setup

Compared Time Series Stores: We compare TVStore with three related time series stores. The first is the closest state-of-the-art work SummaryStore [3], which continuously computes predefined summaries on hot data for reducing data to a target ratio. The second is RRDtool [67], which bounds storage by deleting data when the storage quota is reached. Specially designed for monitoring [61], RRDtool has restrictions on aggregation operations, as well as timestamps and their spacing. We tried our best to circumvent the restrictions to enable comparable evaluations. The last is Apache IoTDB [96], the baseline for the TVStore implementation.

Datasets:[2] We evaluate the time series databases on both synthetic data and real-world data. We generate synthetic data with different patterns, including data with even spacing and that with uneven spacing by the Pareto distribution. We also exploit two real-world datasets, which contain regularity patterns and some random noise. One is the public REDD dataset [50]. The other is a private dataset from one of our users, denoted as the train-load dataset. REDD dataset contains several weeks of low-frequency power data for 6 different homes, and high-frequency current/voltage data for the main power supply of two of these homes. The train-load dataset consists of the train load metrics for months. The private dataset is desensitized for the evaluation purpose.

Workloads and configuration settings[2]**:** We exploit the ingestion and the query workloads included in the open-sourced SummaryStore project when testing synthetic workloads. Like SummaryStore's evaluation, our evaluation uses time series database as an integrated component in the testing client, while using python interfaces for RRDtool. We measure data storage by their final on-disk sizes. We tune the parameters of both systems so that they achieve the highest possible performance. The power-law function is used as the windowing function for SummaryStore and the ratio generation function for TVStore.

Environmental settings: We evaluate TVStore in two different settings. The first is simulating the private cloud environments of medium organizations, while the second is evaluating cases for edge computing scenarios. Hardware setup for the first setting includes 2× 12-core 2.2Hz Intel Xeon E5-2650 CPUs, and 370GB DDR4 memory. The operating system is Ubuntu 16.04.6 and the HotSpot Java runtime version 1.8.0 is used. The second type has 32GB memory and an 8-core CPU, providing a 5TB storage space for the time series database.

[2]Data and workloads – https://github.com/thulab/TVStore-benchmark

6.2 Storage Bounding and Compression Cost

We first evaluate whether TVStore can effectively bound its storage as data keep being ingested at a high speed, in comparison to SummaryStore and RRDtool. We ingest each time series store with 5TB data by 10 evenly-spaced synthetic time series. We have not chosen a larger data volume because SummaryStore cannot process more than this size under the environmental settings. We carefully tuned the evaluations on RRDtool to get the best performance, e.g., using rrdcached. Besides, as RRDtool performs 20× faster given a row larger than 4KB blocks than a row with a single column, we write each time series into one file by putting 1000 consecutive columns in one row.

In the ingestion process, we monitor storage consumption by periodically inspecting the on-disk size of database files. TVStore and SummaryStore are set to finally reduce the data by ratios of 20×/60×/100×, while RRDtool and TVStore have the same storage bounds. The changes of database storage sizes and ingestion times are plotted in Figure 7. A curve longer in the x-axis direction means a longer runtime for the corresponding test.

Bounding: TVStore and RRDtool effectively bound their data storage respectively, keeping storage below different thresholds in all cases. The folds of the storage size curves are key to the bounding for TVStore. They occur when the compression process of TVC ends. SummaryStore has also folds in its storage size curves, which occur due to the continuous summarization on hot data for data reduction.

Compression cost: TVStore's cold-data compression involves fewer disk I/Os than SummaryStore's iterative summarization process, i.e., hot-data processing (§4.1), as reflected by the fewer folds in TVStore's storage size curves than in SummaryStore's. This result of Principle 1 is further corroborated by the CDF of compression/merging times for TVStore and SummaryStore in the 60×-compressed case (Figure 8). We record the compression/merging times

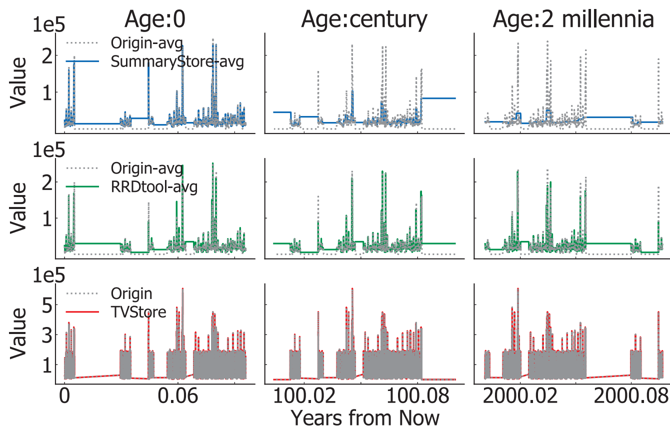


Figure 9: Visualization of data compressed by 100×: TVStore keeps more information than the other systems.

by an extra counter in each window/chunk. The counter is incremented by one for each compression/merging. If multiple windows/chunks with different counter values are merged/compressed, the largest counter is incremented and assigned to the resulting window/chunk. While SummaryStore has windows merged about 70 times in the end, TVStore has only data chunks compressed for 4 times. TVStore has much fewer compression times thanks to its compression based on cold data, instead of hot data, and to its chunking mechanism, instead of point-level windowing.

6.3 Visualization of Compressed Data

We visualize the value patterns of compressed data to see how information is preserved by different systems under the same compression ratio. We experiment with real-world data, the low-frequency REDD data by extending the dataset to 7.5TB. The extended REDD data has a time range of multiple millennia. We visualize data at different ages, i.e., within months, one century, and two millennia. Figure 9 presents the visualization of the dataset.

TVStore and SummaryStore demonstrate time-varying patterns, while RRDtool has the time-invariant curves. SummaryStore and RRDtool keep aggregations only. Thus, only the average values can be visualized for them. In comparison, TVStore enables the visualization of the decompressed data that is compressed by 1×, 11×, and 20× respectively.

TVStore can save storage costs by enabling a high-fidelity overview of the whole range of data using only a storage space as large as 1.5 percentage of the data volume (Figure 9). RRDtool can only support similar visualization on *1.5* percentage of the data for bounding storage by simple deletion, or, have aggregated values too sparse to preserve enough information. SummaryStore has only precise data for recent times and highly different curves for historical times.

Under the same overall data reduction/compression ratio, TVStore can restore data to almost the same as the original, while RRDtool and SummaryStore cannot. The reasons are twofold. First, the implementation of TVStore has exploited

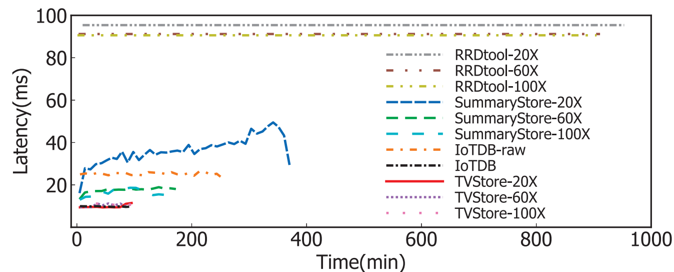


Figure 10: Ingestion latency: much *shorter* ingestion times and *lower* latencies for TVStore than for other stores.

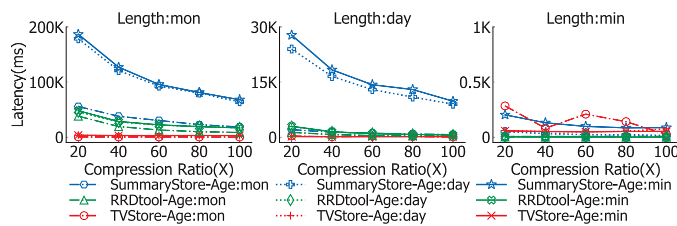


Figure 11: 95-percentile query latencies : TVStore has lower average latencies than SummaryStore and RRDtool in most cases, except for three minute-length cases.

three-layer data reduction, while the other two stores apply only general-purpose compressions. Hence, TVStore can have a smaller ratio for and keep more data by its lossy compression than the other two. Second, TVStore has adopted a line generalization algorithm as the compressor, which performs well at curve visualization. This result implies the importance of choosing a good compressor.

6.4 Ingestion and Query Performance

Ingestion: As shown in Figure 7, TVStore has much higher ingestion throughput than SummaryStore and RRDtool in all cases, leading to shorter curves. In the 20×-Compressed case, TVStore ingests about 3× and 25× faster than SummaryStore and RRDtool respectively, achieving a throughput of *766MB/s* or *47.8 million* time-value points per second. RRDtool has ingestion times about the same length because we have achieved its upper performance bound by writing 4K row blocks in all cases. Exploiting cold-data compression is an important reason for TVStore's advantage over SummaryStore, while insufficient compression and thus longer I/O time is a key reason for RRDtool's disadvantage.

The corresponding average ingestion latencies are demonstrated in Figure 10, with IoTDB storing raw data and IoTDB with two data reduction layers as the baselines. TVStore has a write latency around *10ms* per time-value point. Compared to the baselines, TVStore's compression process has little impact on the normal processing of writes. While RRDtool has stable and long latencies, SummaryStore has fluctuating latencies because of summarizations on hot data. We can conclude from the performance results that *higher compression ratios can effectively improve ingestion performance*.

Query latencies: We evaluate queries on data at different ages, i.e., minutes, days and months. Older data are com-

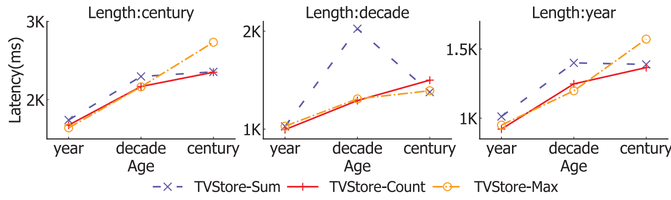


Figure 12: TVStore on the edge: all 95-percentile query latencies are within 2s on 365TB data compressed by 100×.

Length	20X min	20X hr	20X day	20X mon	60X min	60X hr	60X day	60X mon	100X min	100X hr	100X day	100X mon
hr-T	0	0	0	0	0	0	0	0	0	0	0	0
min-T	0	0	0	0.005	0	0	0	0.005	0	0	0	0.005
hr-S	0	0	0	0	0	0	0	0	0	0	0	0
min-S	0.008	0.006	0.007	0.007	0.008	0.006	0.007	0.006	0.008	0.006	0.006	0.007

Figure 13: 95-percentile query errors on *evenly-spaced* random time series reduced by 20/60/100 times: TVStore (top two rows) vs. SummaryStore (bottom two rows). Sum-queries are issued on data at different ages and lengths

Length	Sum min	Sum hr	Sum day	Sum mon	Count min	Count hr	Count day	Count mon	Max min	Max hr	Max day	Max mon
hr-T	0	0	0	0	0	0	0	0	0	0	0	0
min-T	0	0	0	0	0	0	0	0	0	0	0	0
hr-S	0	0.001	0.006	1e+04	0	0	2e+02	3e+02	0	0	1e+02	0
min-S	4e+02	8e+02	3e+03	2e+03	0.008	0.05	3e+01	2e+02	1e+02	1e+02	1e+02	1e+02

Figure 14: 95-percentile query errors on random time series with *Pareto-distributed spacings*: TVStore (top two rows) vs. SummaryStore (bottom two rows). Queries sum/count/max are issued on data at different ages and lengths.

pressed at higher ratios than younger data. Aggregate queries on different time lengths are issued. For each combination of age and length, we issue 100 queries within random time ranges and record the 95-percentile latencies. Our TVStore implementation can answer queries 35× and 8.7× faster than SummaryStore and RRDtool respectively for the best case (5.4s vs. 194s and 47s). Figure 11 presents the results.

TVStore has lower latencies than SummaryStore and RRDtool in most cases, except for the last three cases of minute-length queries. TVStore's implementation co-locates data for a single query. As compared to SummaryStore, fewer data units need to be accessed for the same query by TVStore. SummaryStore has to read data distributed across on-disk storage for processing one query, leading to costly random disk I/Os. As for the minute-length queries, while SummaryStore only needs to retrieve some individual key-value pair, TVStore still has to access and seek a data file for results, leading to slightly higher latencies.

Query on the edge: We also evaluate the query performance of TVStore under the edge-computing condition, which is a typical application scenario for TVStore. The second experimental setting (§6.1) is exploited. Using the REDD dataset, we extend it to 365TB with 365 time series that span century time and then compress it by 100 times. We issue queries on data aging one year, one decade and one century. The resulting query latencies are presented in Figure 12. All queries can be responded within a 95-percentile cold-cache latency of at most 2.7 seconds, even for the longest length and on the oldest data.

6.5 Query Precision on Compressed Data

We evaluate whether TVStore can respond queries on the lossily compressed data within reasonable error bounds, as compared to the state-of-the-art work SummaryStore. RRDtool is not evaluated as it does not support queries approximating any time range that does not align with intervals with precomputed aggregations. Here, we mainly consider the commonly used aggregation queries, which are the basics of many complex analytical operations.

Synthetic data: We first consider the evenly-spaced synthetic data randomly generated at the 1000Hz frequency. Both TVStore and SummaryStore reduce data by 100 times. As they process count, max and min queries with almost zero errors, we present only the 95-percentile error rates of sum queries in Figure 13. TVStore answers queries almost precisely in all cases, except for queries with the smallest length on the oldest data. The minor error rates for such queries are mainly due to the high compression ratio and the high requirements on data details. In comparison, SummaryStore has non-zero error rates for queries with the smallest length at all ages due to the summary-based approach with only two data-reduction layers.

We also experiment on data with timestamps generated according to the Pareto distribution ($\alpha = 1.2$) and values generated uniform randomly. Data are compressed by 100 times before querying. TVStore returns query results almost precisely, with approximately zero 95-percentile error rates, while SummaryStore occasionally has extremely high error rates (e.g., $1e+04$ in Figure 14). In comparison to SummaryStore's incapability in handling unevenly-spaced data, TVStore properly compresses and decompresses the data by its point-oriented compressor based on line generalization algorithms. The results demonstrate the feasibility of error bounding by TVC, if proper compressors are employed.

Real-world data: We test queries on the real-world dataset of train load monitoring. The 6.6TB train-load dataset has 100 individual time series, each of which has about 4.5 billion points. We only evaluate TVStore on this dataset, as SummaryStore cannot support simultaneous ingestions by this number of time series streams. Data is compressed 100 times before evaluation. The results are presented in Figure 15. In most cases, TVStore answers queries with error rates below 2%. The error rates of a few max/min queries are slightly higher, at about 1, due to the high compression ratios of the corresponding old data, as well as the irregular patterns of the real-world data.

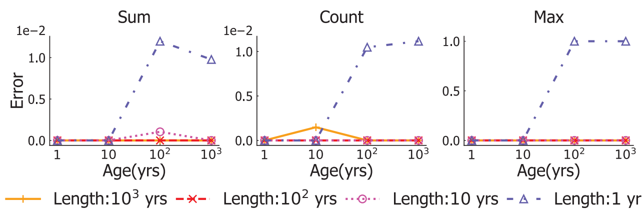


Figure 15: 95-percentile query errors on train-load data by TVStore: queries sum/count/max are issued on data at different ages and lengths.

6.6 Discussion

Our evaluations examine multiple commonly-used statistical operations as supported by the query engine of IoTDB. As high-level analysis operations are mainly implemented in the query engine, TVStore can directly support such operations when the corresponding query engine is used. In comparison, some time series store requires prior knowledge on users analysis requirements for each application [3]. Besides, as TVC can be integrated with different *fixed-ratio compressors* and *time-dependent ratio functions*, better support for learning-based analysis is possible given carefully chosen compressors and ratio functions [7, 70]. We leave the choice and the design of compressors and ratio functions for future work.

7 Related Work

Time series data compression. Lossless and lossy compression methods exist for time series data. Lossless compression can reconstruct the original data accurately. For lossless compression, specialized compressors for integer [10, 33, 55, 73] and for floating-point values [57, 71] outperform general-purpose compressors [20,28,32,58,66]. It is common practice that general-purpose compression is applied along with specialized compressors [43,54,96] in time series stores.

Lossy compression can achieve a much higher compression ratio than lossless compression by giving up partial information. Compression by linear models or PLA (piecewise-linear approximation) [13,60,87,99] has been extensively used in practice for its simplicity. Line generalization algorithms can be considered as variants of the PLA method and used for time series compression [25,46,52,74,85,95]. More complex models based on polynomials [30] or transformations [11,16] are also considered as compressor alternatives in model-based time series stores [44,56], which select from a set of models to compress time series with the least errors. Lossy compression methods commonly optimize the compression ratio towards specified error bounds, but it is difficult for users to set the bounds beforehand for real-world data. Few research work on compressors exists for optimizing error bounds towards a given compression ratio.

Our time-varying compression framework TVC is orthogonal to the above compression methods. As long as a compressor can compress data by a given ratio and satisfy the three properties (§3.3), TVC can take advantage of it to enable time-varying compression.

Time series stores. To manage the ever-increasing volume of time series data, most time series stores either have a native architecture to support a distributed deployment such as InfluxDB [43], or exploit a distributed storage for scalability, e.g., KairosDB [47] on Cassandra [53], TimescaleDB [90] on PostgreSQL database [72], Druid [100] on HDFS [35], BtrDB [4] on Ceph [97], Chronix [54] on Solr [84], and Peregreen on an object store [94]. While data distribution techniques are also applicable to TVStore, TVStore focuses on data reduction and storage control.

Besides lossless and lossy compression, time series databases commonly exploit the data retention policy to reclaim storage space by removing data exceeding a time period [42] or exceeding a storage quota, e.g., RRDtool [67], for further storage reduction. ModelarDB [44] reduces storage and query latency by time series approximation models with user-defined error bounds. Some models can also be exploited by TVC of TVStore. Adopting common stream processing methods [17, 21], SummaryStore [3] can reduce storage to a specified ratio by keeping only data summaries, if data analytical requirements are known beforehand for each application. SummaryStore works only with a limited set of summaries and cannot support data restoration.

In comparison, TVStore automatically bounds time series storage by TVC. It requires neither prior knowledge on exact retention time nor that on query workloads. TVStore enables users to respect varied data significance by integrating a chosen compressor and a time-dependent function for their applications. It can support data restoration given properly designed compressors/decompressors.

8 Conclusion

The fast increasing volume of time series data is outpacing the increase of users' affordable storage space. It is desirable to have a time series database that can automatically control the time series data storage, while preserving as much information as possible and in a manner considering data ages, which are correlated with data importance. To the best of our knowledge, TVStore is the first time series store that achieves this goal. Leveraging the proposed time-varying compression, TVStore bounds the time series database storage by iterative compressions that are initiated at rigorously chosen proper times and ratios. Extensive evaluations based on synthetic and real-world data validate the storage-boundedness of TVStore and its time-varying pattern of compression. Besides, The advantage and efficacy of TVStore are also demonstrated by its superior performance over three state-of-the-art or state-of-the-practice related works of time series store.

Acknowledgements

We sincerely appreciate the shepherding from Deniz Altınbüken. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This work was supported by NSFC Grant (No. 62021002).

References

[1] Db-engines ranking of time series dbms. https://db-engines.com/en/ranking/time+series+dbms, 2020.

[2] Dbms popularity broken down by database model–trend of the last 24 months. https://db-engines.com/en/ranking_categories, 2021.

[3] Nitin Agrawal and Ashish Vulimiri. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 647–664, 2017.

[4] Michael P Andersen and David E Culler. Btrdb: optimizing storage system design for time series processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, 2016.

[5] Florian M Artinger, Nikita Kozodi, Florian Wangenheim, and Gerd Gigerenzer. Recency: prediction with smart data. In *2018 AMA Winter Academic Conference: Integrating paradigms in a world where marketing is everywhere, February 23-25, 2018, New Orleans, LA. Proceedings*, pages L–2. American Marketing Association, 2018.

[6] Hany Fathy Atlam, Robert Walters, and Gary Wills. Internet of things: state-of-the-art, challenges, applications, and open issues. *International Journal of Intelligent Computing Research (IJICR)*, 9(3):928–938, 2018.

[7] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556, 2017.

[8] Brad Bechtold. Beyond the barrel: how data and analytics will become the new currency in oil and gas. https://gblogs.cisco.com/ca/2018/06/07/beyond-the-barrel-how-data-and-analytics-will-become-the-new-currency-in-oil-and-gas/, 2018.

[9] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

[10] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[11] Peter Bloomfield. *Fourier analysis of time series: an introduction*. John Wiley & Sons, 2004.

[12] A Bremler-Barr, E Cohen, H Kaplan, and Y Mansour. Predicting and bypassing internet end-to-end service degradations. In *Proc. 2nd ACM-SIGCOMM Internet Measurement Workshop. ACM*, volume 10, pages 637201–637248, 2002.

[13] EH Bristol. Swinging door trending: adaptive trend recording? In *ISA National Conf. Proc., 1990*, pages 749–754, 1990.

[14] Peter Burge and John Shawe-Taylor. Frameworks for fraud detection in mobile telecommunications networks. In *Proceedings of the Fourth Annual Mobile and Personal Communications Seminar. University of Limerick*. Citeseer, 1996.

[15] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. Lfzip: lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference (DCC)*, pages 342–351. IEEE, 2020.

[16] Pimwadee Chaovalit, Aryya Gangopadhyay, George Karabatis, and Zhiyuan Chen. Discrete wavelet transform-based time series analysis and mining. *ACM Computing Surveys (CSUR)*, 43(2):1–37, 2011.

[17] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: no silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519, 2017.

[18] Yi-Cheng Chen, Lin Hui, and Tipajin Thaipisutikul. A collaborative filtering recommendation system with dynamic time decay. *The Journal of Supercomputing*, 77(1):244–262, 2021.

[19] Cisco. New realities in oil and oas: data management and analytics. https://www.cisco.com/c/dam/en_-us/solutions/industries/energy/docs/OilGasDigital TransformationWhitePaper.pdf, 2017.

[20] Yann Collet et al. Lz4 - extremely fast compression. https://lz4.github.io/lz4/, 2011.

[21] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[22] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. Forward decay: a practical time decay model for streaming systems. In *2009 IEEE 25th international conference on data engineering*, pages 138–149. IEEE, 2009.

[23] Corinna Cortes and Daryl Pregibon. Giga-mining. In *KDD*, pages 174–178, 1998.

[24] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (IPDPS)*, pages 730–739. IEEE, 2016.

[25] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.

[26] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[27] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proceedings of the VLDB Endowment*, 2(1):145–156, August 2009.

[28] Facebook. Zstandard home page. https://facebook.github.io/zstd/, 2017.

[29] Ian Foster, Mark Ainsworth, Bryce Allen, Julie Bessac, Franck Cappello, Jong Youl Choi, Emil Constantinescu, Philip E Davis, Sheng Di, Wendy Di, et al. Computing just what you need: online data analysis and reduction at extreme scales. In *European conference on parallel processing*, pages 3–19. Springer, 2017.

[30] Erich Fuchs, Thiemo Gruber, Jiri Nitschke, and Bernhard Sick. Online segmentation of time series based on polynomial least-squares approximations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(12):2232–2245, 2010.

[31] Diksha Garg, Priyanka Gupta, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. Sequence and time aware neighborhood for session-based recommendations: Stan. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1069–1072, 2019.

[32] Google. Snappy home page. http://google.github.io/snappy/, 2011.

[33] Network Working Group. Rfc 3229: Delta encoding in http. https://tools.ietf.org/html/rfc3229, 2002.

[34] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. Hacc: extreme scaling and performance across diverse architectures. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2013.

[35] Hadoop. Apache hadoop distributed file system. https://hadoop.apache.org/, 2020.

[36] Peter MC Harrison, Roberta Bianco, Maria Chait, and Marcus T Pearce. Ppm-decay: a computational model of auditory prediction with memory decay. *PLoS computational biology*, 16(11):e1008304, 2020.

[37] Brian Hentschel, Peter J Haas, and Yuanyuan Tian. General temporally biased sampling schemes for online model management. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[38] Brian Hentschel, Peter J Haas, and Yuanyuan Tian. General temporally biased sampling schemes for online model management. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[39] Antony S Higginson, Mihaela Dediu, Octavian Arsene, Norman W Paton, and Suzanne M Embury. Database workload capacity planning using time series analysis and machine learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 769–783, 2020.

[40] Jie Huang, Fengwei Zhu, Zejun Huang, Jian Wan, and Yongjian Ren. Research on real-time anomaly detection of fishing vessels in a marine edge computing environment. *Mobile Information Systems*, 2021, 2021.

[41] Nathanael Hübbe, Al Wegener, Julian Martin Kunkel, Yi Ling, and Thomas Ludwig. Evaluating lossy compression on climate data. In *International Supercomputing Conference*, pages 343–356. Springer, 2013.

[42] InfluxData. Influxdb data retention. https://towardsdatascience.com/influxdb-data-retention-f026496d708f, 2020.

[43] InfluxDB. Influxdb home page. https://www.influxdata.com/, 2020.

[44] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Modelardb: modular model-based time series management with spark and cassandra. *Proceedings of the VLDB Endowment*, 11(11):1688–1701, 2018.

[45] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J Elmore. Good to the last bit: data-driven encoding with codecdb. In *Proceedings of the 2021 International Conference on Management of Data*, pages 843–856, 2021.

[46] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: a visualization-oriented time series data aggregation. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014.

[47] KairosDB. Kairosdb home page. https://kairosdb.github.io/, 2020.

[48] Zahra Karevan and Johan AK Suykens. Transductive lstm for time-series prediction: an application to weather forecasting. *Neural Networks*, 125:1–9, 2020.

[49] Hakkyu Kim and Dong-Wan Choi. Recency-based sequential pattern mining in multiple event sequences. *Data Mining and Knowledge Discovery*, 35(1):127–157, 2021.

[50] J Zico Kolter and Matthew J Johnson. Redd: a public data set for energy disaggregation research. In *Workshop on data mining applications in sustainability (SIGKDD), San Diego, CA*, volume 25, pages 59–62, 2011.

[51] Antti Koski, Martti Juhola, and Merik Meriste. Syntactic recognition of ecg signals by attributed finite automata. *Pattern Recognition*, 28(12):1927–1940, 1995.

[52] Sam Kumar, Michael P Andersen, and David E. Culler. Mr. plotter: unifying data reduction techniques in storage and visualization systems. Technical Report UCB/EECS-2018-85, EECS Department, University of California, Berkeley, May 2018.

[53] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[54] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. Chronix: long term storage and retrieval technology for anomaly detection in operational data. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 229–242, 2017.

[55] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[56] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. Plato: approximate analytics over compressed time series with tight deterministic error guarantees. *Proceedings of the VLDB Endowment*, 13(7):1105–1118, 2020.

[57] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[58] Jean loup Gailly and Mark Adler. Gzip home page. https://www.gzip.org/, 2003.

[59] Sidi Lu, Bing Luo, Tirthak Patel, Yongtao Yao, Devesh Tiwari, and Weisong Shi. Making disk failure predictions smarter! In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 151–167, 2020.

[60] Ge Luo, Ke Yi, Siu-Wing Cheng, Zhenguo Li, Wei Fan, Cheng He, and Yadong Mu. Piecewise linear approximation of streaming time series data with max-error guarantees. In *2015 IEEE 31st International Conference on Data Engineering*, pages 173–184. IEEE, 2015.

[61] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[62] Chris Mellor. Data storage estimates for intelligent vehicles vary widely. https://blocksandfiles.com/2020/01/17/connected-car-data-storage-estimates-vary-widely/, 2020.

[63] Angela P Murillo. Data at risk initiative: examining and facilitating the scientific process in relation to endangered data. *Data Science Journal*, pages 12–048, 2014.

[64] Jan Pawel Musial, Michael M Verstraete, and Nadine Gobron. Comparing the effectiveness of recent algorithms to fill and smooth incomplete and noisy time series. *Atmospheric chemistry and physics*, 11(15):7905–7923, 2011.

[65] Rascha JM Nuijten, Theo Gerrits, Judy Shamoun-Baranes, and Bart A Nolet. Less is more: on-board lossy compression of accelerometer data increases biologging capacity. *Journal of Animal Ecology*, 89(1):237–247, 2020.

[66] Markus F.X.J. Oberhumer. Lzo home page. http://www.oberhumer.com/opensource/lzo/, 2008.

[67] Tobi Oetiker. Rrdtool: round robin database tool. http://oss.oetiker.ch/rrdtool/, 2021.

[68] Themis Palpanas, Michail Vlachos, Eamonn Keogh, and Dimitrios Gunopulos. Streaming time series summarization using user-defined amnesic functions. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):992–1006, 2008.

[69] Thanasis G Papaioannou, Mehdi Riahi, and Karl Aberer. Towards online multi-model approximation of time series. In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 1, pages 33–38. IEEE, 2011.

[70] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. Vergedb: a database for iot analytics on edge devices. In *CIDR*, 2021.

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

[72] PostgreSQL. Postgresql home page. https://www.postgresql.org/, 2020.

[73] AH Robinson and Colin Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

[74] Kexin Rong and Peter Bailis. Asap: prioritizing attention via time series smoothing. *Proceedings of the VLDB Endowment*, 10(11), 2017.

[75] Ariel Rosenfeld, Joseph Keshet, Claudia V Goldman, and Sarit Kraus. Online prediction of exponential decay time series with human-agent application. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 595–603, 2016.

[76] Rohan Basu Roy, Tirthak Patel, Raj Kettimuthu, William Allcock, Paul Rich, Adam Scovel, and Devesh Tiwari. Operating liquid-cooled large-scale systems: long-term monitoring, reliability analysis, and efficiency measures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 881–893. IEEE, 2021.

[77] SAS. The connected vehicle: big data, big opportunities. https://www.sas.com/content/dam/SAS/en_-us/doc/whitepaper1/connected-vehicle-107832.pdf, 2021.

[78] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[79] Theo Schlossnagle, Justin Sheehy, and Chris McCubbin. Always-on time-series database: keeping up where there's no way to catch up. *Communications of the ACM*, 64(7):50–56, 2021.

[80] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. Financial time series forecasting with deep learning: a systematic literature review: 2005–2019. *Applied Soft Computing*, 90:106181, 2020.

[81] Zakhele Phumlani Shabalala, Mokhele Edmond Moeletsi, Mphethe Isaac Tongwane, and Sabelo Marvin Mazibuko. Evaluation of infilling methods for time series of daily temperature data: case study of limpopo province, south africa. *Climate*, 7(7):86, 2019.

[82] Syed Ahsin Ali Shah, Wajid Aziz, Majid Almaraashi, Malik Sajjad Ahmed Nadeem, Nazneen Habib, and Seong-O Shim. A hybrid model for forecasting of particulate matter concentrations based on multiscale characterization and machine learning techniques. *Mathematical Biosciences and Engineering*, 18(3):1992–2009, 2021.

[83] SIBROS. Smart data logging for connected vehicle value creation. https://www.sibros.tech/post/smart-data-blogging-for-connected-vehicle-value-creation, 2019.

[84] Apache Solr. Open source enterprise search platform. http://lucene.apache.org/solr, 2021.

[85] Sveinn Steinarsson. Downsampling time series for visual representation. Master's thesis, 2013.

[86] Joachim Stolze, Angela Nöppert, and Gerhard Müller. Gaussian, exponential, and power-law decay of time-dependent correlation functions in quantum spin chains. *Physical Review B*, 52(6):4319, 1995.

[87] Marco Storace and Oscar De Feo. Piecewise-linear approximation of nonlinear dynamical systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(4):830–842, 2004.

[88] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. P-store: an elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219, 2018.

[89] Yinyan Tan, Zhe Fan, Guilin Li, Fangshan Wang, Zhengbing Li, Shikai Liu, Qiuling Pan, Eric P Xing, and Qirong Ho. Scalable time-decaying adaptive prediction algorithm. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 617–626, 2016.

[90] TimescaleDB. Timescaledb home page. https://www.timescale.com/, 2020.

[91] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[92] Robert Underwood, Sheng Di, Jon C Calhoun, and Franck Cappello. Fraz: a generic high-fidelity fixed-ratio lossy compression framework for scientific floating-point data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–577. IEEE, 2020.

[93] Sergio Verdu. Fifty years of shannon theory. *IEEE Transactions on information theory*, 44(6):2057–2078, 1998.

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

[95] Maheswari Visvalingam and James D Whyatt. Line generalisation by repeated elimination of points. *The cartographic journal*, 30(1):46–51, 1993.

[96] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. Apache iotdb: time-series database for internet of things. *Proceedings of the VLDB Endowment*, 13(12):2901–2904, 2020.

[97] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[98] Steven R Wilkinson, Cyrus F Bharucha, Martin C Fischer, Kirk W Madison, Patrick R Morrow, Qian Niu, Bala Sundaram, and Mark G Raizen. Experimental evidence for non-exponential decay in quantum tunnelling. *Nature*, 387(6633):575–577, 1997.

[99] George Edward Williams. Critical aperture convergence filtering and systems and methods thereof, July 2006. US Patent 7,076,402.

[100] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: a real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, 2014.