MOST: Model-Based Compression with Outlier Storage for Time Series Data

ZEHAI YANG and SHIMIN CHEN*, SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China

Time series data are used in a wide variety of applications. The explosive growth of the amount of time series data poses a significant challenge in efficient data storage and query processing. Unfortunately, existing compression techniques either show only low to medium compression ratio on time series data, or incur significant decompression overhead during query processing.

We propose a novel compression technique, MOST (**M**odel-based compression with **O**utlier **ST**orage) for time series data. As measurement values often change smoothly in a period of time, we divide a time series into segments of smooth changes, then compute a linear model for each segment. Since tiny errors are often acceptable in analysis tasks, we omit data points whose computed values are within a pre-specified error threshold from the actual values, thereby effectively reducing the data size. Outliers are rare but important for many applications, and therefore we store outliers explicitly. Moreover, for processing MOST compressed data, we propose a segment-outlier dual-mode query engine that computes segments as a whole as much as possible, and build a prototype MostDB. Experimental results on real-world data sets show that MOST achieves 9.45–15.04x compression ratios. Compared to existing time series databases, MostDB achieves up to 11.68x speedups for common queries from the IoTDB Benchmark.

CCS Concepts: • **Information systems → Data compression**; **Stream management**; **Database query processing**.

Additional Key Words and Phrases: Model-Based Compression, Outlier, Dual-Mode Query Engine

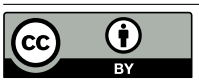
ACM Reference Format:
Zehai Yang and Shimin Chen. 2023. MOST: Model-Based Compression with Outlier Storage for Time Series Data. *Proc. ACM Manag. Data* 1, N4 (SIGMOD), Article 250 (December 2023), 29 pages. https://doi.org/10.1145/3626737

1 INTRODUCTION

Time series data are used in a wide variety of applications, ranging from Internet of Things (IoT) [70] and cloud server monitoring [31] to stock market prices [64], supporting many important tasks, including forecasting [30] , trend analysis [81], real-time monitoring [80], anomaly detection [79, 87]. To support these applications, time series databases (TSDB), such as InfluxDB [34] and IoTDB [92], provide a unified system for time series data storage and relational query processing. However, recent years have seen an explosive growth of the amount of time series data. According to an IDC report [14], the amount of new data created is growing at a compound annual growth rate of about 26% between 2015 and 2025. About 65% of data are created at endpoints, such as IoT devices. The rapidly growing data size poses a significant challenge in the efficient storage and processing of time series data.

*Shimin Chen is the corresponding author.

Authors' address: Zehai Yang, yangzehai20z@ict.ac.cn; Shimin Chen, chensm@ict.ac.cn, SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, No. 6 Ke Xue Yuan South Rd, Haidian District, Beijing, China and University of Chinese Academy of Sciences, No.1 Yanqihu East Rd, Huairou District, Beijing, China.



2836-6573/2023/12-ART250
https://doi.org/10.1145/3626737

We study compression techniques and query processing on compressed data in TSDBs for addressing this challenge. An ideal compression technique can (i) greatly reduce the size of time series data (e.g., by 10x), thereby saving storage space and lowering I/O costs for accessing the data, and (ii) support efficient relational query processing without decompressing the data. Unfortunately, existing compression techniques fail to meet such goals. We find that most compression techniques obtain only low to medium compression ratios for time series data, including general-purpose compression techniques (e.g., LZ77 [95], Snappy [28], ZStd [65], LZ4 [12]), record-oriented compression techniques (e.g., dictionary encoding [56], delta encoding [49], run-length encoding, Gorilla [75], SplitDouble [74], BUFF [55]), and model-based compression techniques (e.g., Sprintz [8]). A recent model-based technique, SZ3 [94], can achieve good compression ratios for a large amount of data. However, SZ3 performs multi-layer spline interpolation on the data, and requires the decompression of all the data before query processing, incurring significant overhead for query processing.

We propose a novel compression technique, MOST (**M**odel-based compression with **O**utlier **ST**orage) for time series data. Our design is inspired by the following observations. First, measurement values often change smoothly or stay stable in a period of time. Hence, prediction models can effectively capture the characteristics of time series and accurately describe most data points. Second, existing model-based compression either discards or stores all prediction errors. The former has high compression ratio but poor data accuracy, while the latter preserves good data accuracy but sees lower compression ratio. Neither approach is ideal. Third, given a pre-defined error threshold, we can avoid storing data points whose prediction errors are within the error threshold. This idea can effectively reduce the data size because models are expected to be accurate for most points. Finally, we can explicitly store outliers, whose prediction errors are beyond the error threshold, to preserve good data accuracy. The storage cost for outliers is expected to be reasonably low. As a result, MOST can achieve both good compression ratio and good data accuracy.

The MOST compression algorithm consists of three steps: 1) outlier detection, 2) segmentation, and 3) model and outlier encoding. We find that Step 1) before Step 2) is important because outliers may divide an otherwise smooth segment into multiple small segments, which can adversely impact the compression ratio. The segmentation step divides the points into segments and computes linear models for each segment. We use linear models since they are well-studied [20, 21, 47, 48] and amenable to fast computation and query processing without decompression. We propose a Combined Shrinking Cone (CSC) algorithm for outlier detection and segmentation, after experimentally studying the combinations of existing outlier detection and segmentation methods. In Step 3), we exploit quantization and matissa truncation to reduce the data size of encoded model parameters and outlier values.

Then, we design MostDB, a prototype TSDB for storing and processing MOST compressed times series data. MostDB stores segment model parameters and outliers in an underlying database, specifically InfluxDB [34] in our implementation. We propose a *dual-mode query engine* for query processing. In the engine, measurement values are passed between query operators as a segment with its associated outliers. Then the query operators have two modes: the segment mode and the outlier mode. For the outlier mode, the operator processes each outlier record similar to the vanilla relational operator. For the segment mode, the computation is performed on the linear model of a segment as a whole. For example, the segment mode of a filter can accept or reject an entire segment, or accept a partial segment. The segment mode of a sum aggregate can compute the sum for the entire line segment. We support dual-mode scan, filter, aggregate, and output operators. In this way, MostDB performs per-segment processing and defers the reconstruction of values as much as possible, thereby significantly reducing query costs.

Contribution. The contributions of this paper are threefold. First, we propose MOST, a novel compression technique for time series data. While previous model-based techniques store either no errors or all errors, MOST stores errors only for outliers, thereby achieving both high compression ratio and good data accuracy. MOST detects outliers before segmentation to reduce the number of segments for better compression results. Second, we propose a dual-mode query engine for processing MOST compressed data. We present a MostDB prototype, which stores data in the underlying database and processes relational queries in the dual-model query engine. Finally, we perform an extensive experimental study to evaluate the benefits of MOST and MostDB. Our results show that MOST achieves 9.45–15.04x compression ratios on real-world data sets. Compared to InfluxDB, IoTDB and ModelarDB, MOST achieves up to 11.68x, 9.75x and 4.99x speedups for common IoT queries from the IoTDB Benchmark [57], and 3.73x, 3.82x, 1.82x speedups for TSBS [89] queries. We have made the code of MostDB publicly available: https://github.com/schencoding/mostdb.

Outline. The remainder of the paper is organized as follows. Section 2 introduces relevant background and discusses related work. Section 3 presents the MOST compression technique. Section 4 describes MostDB, a TSDB based on MOST, with an emphasis on its query processing engine. Then, Section 5 reports evaluation results. Finally, Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, we begin by reviewing background on time series data. Then, we discuss related work on compression techniques, TSDBs, and query processing in TSDBs.

2.1 Background on Time Series Data

Time Series. A time series is a sequence of data points ordered in time order. Typically, it contains the measurement values of a certain data source (e.g., an IoT device) collected at regular intervals. Formally, a time series is defined as follows:

$$S = \{(t_1, v_1), (t_2, v_2), ..., (t_i, v_i), ...\} \tag{1}$$

where v_i is the measurement value and usually a floating point number, and t_i is the timestamp when v_i is collected.

Error and Error Bound. Compression techniques can be categorized into lossless and lossy techniques. Lossless compression reconstructs the original values from the compressed data. However, since floating point numbers are difficult to compress, lossy compression has been used to attain better compression ratio for time series data. We would like to constrain the lossiness of a lossy compression technique by providing data accuracy guarantees. Suppose v_i' is the decompressed value of the original measurement v_i. Given an error bound $\epsilon > 0$, we define the following constraint:

$$\begin{cases} \text{absolute error-bound}: & \forall i, \text{absolute\_error}_i = |v_i' - v_i| \leq \epsilon \\ \text{relative error-bound}: & \forall i, \text{relative\_error}_i = |\frac{v_i' - v_i}{v_i}| \leq \epsilon \end{cases} \tag{2}$$

Our solution supports both relative and absolute error bounds.

2.2 Compression for Time Series Data

Existing compression techniques can be divided into four categories: (1) general-purpose, which treats the input data as a byte stream; (2) record-oriented, which understands the record structures in the input data; (3) model-only, which stores only computed models to represent time series data and discards all data points; and (4) model with errors, which stores models as well as prediction errors at each data point. Note that since time series data are mainly floating point numbers, compression techniques targeting specific data types (e.g., audio [67], EEG recordings [60]) are ill-suited and excluded from our discussion. In addition, neural network based compression techniques [93]

Table 1. Compression techniques for time series data.

Compression Techniques	Compression Ratio	Data Accuracy	Query Types	Queries on Compressed Data
General-Purpose	low	lossless	general	no
Record-Oriented	low/medium	lossless/good	general	yes
Model-Only	highest	poor	special	yes
Model w/ Errors	medium/high	ok/good	general	no
MOST (this paper)	high	good	general	yes

may incur significant computation overhead, slowing down data ingestion. Hence, they are not generally suitable to time series applications. Table 1 summarizes and compares the features of the compression techniques.

General-purpose compression. General-purpose byte-oriented compression techniques encode a byte stream of input data. LZ77 [95] and its variants, such as GZip [61], Snappy [28], ZStd [65], LZ4 [12], and Brotli [4], are widely used. The core idea is to find and encode repeated substrings in the sliding window of the input byte stream. However, general-purpose compression is not optimized for the characteristics of time series data. Consequently, the compression ratio is low. Moreover, it is impossible to distinguish individual records in the compressed data. Hence, queries and analysis operations cannot be supported directly on the compressed data, causing data decompression overhead in query processing.

Record-oriented compression. Record-oriented compression transforms each record into a compact representation. Run length encoding (RLE) represents consecutive records having the same value as the value and the number of occurrences. Dictionary encoding creates a mapping between values and compact integer codes, then replaces the original data entries with the corresponding codes. Delta encoding [49] stores the difference between two adjacent records, and Delta-of-Delta uses Delta encoding twice in succession. Xor encoding is similar to delta encoding except that it computes the Xor of adjacent records. Gorilla [75], an in-memory TSDB developed by Facebook, employs Delta-of-Delta encoding for timestamps and XOR-based encoding for floating point numbers. SplitDouble in VergeDB [74] encodes the integer and decimal parts of floating-point numbers separately. The former is bit compressed and the latter is encoded by Gorilla. Recently, BUFF [55] exploits bounded range and precision to reduce the number of bits for storing the integer and the decimal parts of floating point numbers. The resulting encodings are stored in byte-oriented columnar layout to improve query processing.

As shown in Table 1, record-oriented compression techniques retain the record information and thus are capable of supporting query processing on compressed data. Except BUFF, the techniques are lossless and their compression ratio are low. In comparison, BUFF is a lossy technique with good accuracy. It achieves medium compression ratio (cf. Section 5.2) by bounding the range and precision of the floating point numbers.

Model-only compression. Model-only techniques include singular value decomposition [40], discrete wavelet transformation [11], discrete Fourier transformation [3], PAA [42], PLA [82]. For example, PAA divides the time series into segments and uses the average value to represent each segment, while PLA uses a linear function to model each segment. Symbolic Aggregate Approximation (SAX) [54] employs PAA then discretizes the PAA coefficients using equal-area buckets of the Gaussian distribution. Model-only techniques are typically optimized for specific analysis tasks, e.g., monitoring pre-defined aggregates, classifying time series, comparing time series. Since the models themselves suffice the targeted tasks, the techniques store only the model parameters and discard all data records. As a result, they achieve the highest compression ratio, as

shown in Table 1. However, it is not possible to accurately reconstruct the original measurement values. Hence, model-only techniques cannot support general-purpose queries.

Model-based compression with errors. A number of model-based compression techniques store both model parameters and errors for all data points so as to achieve better data accuracy and support general-purpose analysis tasks. Sprintz [8] was initially targeted at compressing integer values. It proposes a Fire (Fast Integer REgression) model to predict the next value based on a linear combination of a fixed number of previous values. Then, it saves the (integer) errors using zigzag encoding. To support floating point time series, Sprintz quantizes floating point values before applying the Fire model. For each segment of data, it divides the value range into a pre-defined number of buckets (e.g., 256 or 65536 buckets), then maps floating point values to the bucket IDs. The quantization can result in very large error bounds because the value range must contain all outliers. SZ3 performs multi-layer spline interpolation on the values [94]. It stores the first and last points. Then, each layer predicts the intermediate points by interpolation. All the intermediate points can be reconstructed hierarchically through the multiple interpolation calculations. SZ3 quantizes the errors and provides error bound guarantees.

As shown in Table 1, since prediction errors can be represented more compactly than the original values, model-based techniques usually obtain better compression ratio than lossless techniques. Sprintz and SZ3 attain medium and high compression ratio, respectively (cf. Section 5). While the data accuracy of Sprintz suffers from its quantization, SZ3 provides good data accuracy. Finally, data compressed by these techniques must be decompressed before query processing, incurring significant overhead.

Our proposed solution: MOST. MOST (model-based compression with outlier storage) selectively stores prediction errors, neither discarding them completely as in model-only compression, nor retaining all errors as in model-based compression with errors. To improve compression ratio, MOST exploits models to represent most values whose prediction errors are small. For outliers, whose values are very different from predicted values, MOST stores them explicitly to ensure data accuracy. As shown in Table 1, MOST is capable of supporting high compression ratio, good data accuracy, general-purpose queries, and queries on compressed data.

2.3 Time Series Databases and Query Processing

TSDB. Time series database (TSDB) aims to support high-throughput data insertion and general-purpose relational queries on time series data. InfluxDB [34] is a popular open source TSDB written in Go. Its storage engine is based on TSM-Tree [69], a variant of LSM-Tree [72]. InfluxDB compresses integers with Zigzag and Simple8b [7], and floating point numbers with Gorilla. IoTDB [92][91] is an open source TSDB written in Java. It designs an optimized columnar file format for efficient time-series data storage. It supports Gorilla, RLE, Delta-of-Delta, Sprintz, as well as general-purpose compression, such as LZ4. Other TSDBs include TimescaleDB [83], OpenTSDB [2], KairosDB [86], TDengine [85], Druid [24], Gorilla [75], and VergeDB [74]. They mainly use general-purpose and/or record-oriented compression techniques.

We compare MostDB with InfluxDB, IoTDB, and ModelarDB using real-world data sets in our experiments (cf. Section 5). Please note that MostDB and ModelarDB perform lossy compression, while InfluxDB and IoTDB use lossless compression by default. MostDB allows users to choose the error bound based on application requirements. We evaluate the impact of varying error bounds on compression performance (cf. Figure 9).

Approximate query processing. When querying data compressed with lossy compression techniques, TSDBs compute query results from reconstructed values. The baseline approach employed by TSDBs is to simply ignore the errors and treat the reconstructed values as accurate ones. In

this paper, we would like also to provide guarantees on result accuracy. Our scheme is built on the techniques of approximate query processing and probabilistic databases.

Approximate query processing can greatly reduce the computation time of aggregates for massive data sets [51]. Online aggregation methods, such as sampling [71] and online aggregation [29, 32], use data samples to compute approximate query answers and the confidence intervals. Offline synopses, such as offline samples [1], histograms [13, 76], and sketches [13, 84], generate synopses offline and use them in query processing. Probabilistic databases [15, 17] assign a probability to each record in the data table to represent uncertain values or the results from uncertain query predicates. A recent study [53] performs model-only compression for time series data, and computes deterministic error bounds for a number of analytic tasks by combining per-segment models and pre-computed error measures (e.g., L2-norm of predication errors). We borrow the ideas of these studies to compute expectation, standard deviation, and/or deterministic bounds for aggregations in MostDB. The main difference is that MostDB combines both models and outliers in the computation (cf. Section 4.5) .

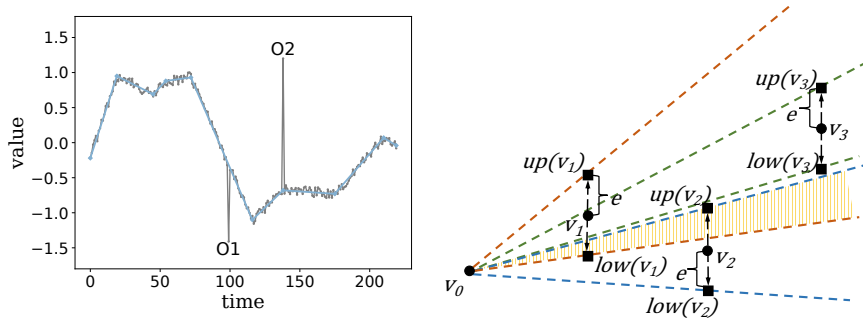
Model-based database systems. Previous works store models and support query processing on models in DBMSs. MauveDB [18] proposes model-based views to hide the irregularities of the underlying sensor data. It supports a declarative language for defining and querying over model-based views. Queries use ScanView or IndexView operators to retrieve tuples computed from the models. FunctionDB [88] stores models in function tables and supports algebraic operations (e.g., variable substitution, equation solving/approximation, function inference) based on hypercubes in query processing. Plato [41] regards models as the ground truth of spatio-temporal sensor data and as first-class citizens in DBMS. In query processing, models are viewed either as functions or as tables with an infinite number of tuples. A recent study, ModelarDB [38], builds a TSDB that performs model-based compression for time series with regular timestamps and infrequent gaps. Query processing is based mostly on data points reconstructed from the models. In case that query results can be computed directly from the models (e.g., aggregation with time predicates), ModelarDB performs pure-model computation.

In the above studies, query processing uses either reconstructed values or pure-model computation. In comparison, we propose a dual-mode query engine in MostDB that combines the processing of both segment models and outliers in query operators. Compared to pure-model computation, our approach (i) includes outliers naturally and (ii) supports a wide range of query types. We compare MostDB with ModelarDB in our experiments (cf. Section 5).

3 MOST COMPRESSION

Model-based compression captures the characteristics of time series as values in a time series often change smoothly in a period of time. At one extreme, model-only compression does not store any prediction errors. This attains high compression ratio but sacrifices data accuracy. At the other extreme, model-based compression with errors preserves good data accuracy by storing all prediction errors. However, this leads to less ideal compression ratio. We propose MOST (**M**odel-based compression with **O**utlier **ST**orage) to strike a balance of compression ratio and data accuracy by storing only rare outliers, i.e., data points whose prediction errors are beyond a pre-specified error threshold. In this way, MOST achieves both high compression ratio and good data accuracy.

The MOST compression algorithm consists of three steps: 1) *outlier detection*, 2) *segmentation*, and 3) *model and outlier encoding*. Note that it is important to detect outliers before segmentation. Figure 1a depicts the segments and outliers of a time series. If Step 2) were before Step 1), $O1$ would cut the segment (75,117) into two parts. If we can detect and skip $O1$, then segmentation produces



(a) (b)

Fig. 1. (a) Segments and outliers of a time series; (b) Segmentation process of *Shrinking Cone*

a single segment for (75,117). In other words, Step 1) can prevent smooth segments from being split by scattered outliers in Step 2).

In the following, Section 3.1– 3.2 focus on Step 1) and 2), then Section 3.3 describes Step 3).

3.1 Outlier Detection and Segmentation Methods for Time Series Data

In this subsection, we review existing methods for outlier detection and segmentation for time series data. Then, in the next subsection, we experimentally choose the best strategy and propose a combined shrinking cone (CSC) algorithm for the chosen strategy.

Outliers. In this work, we construct (linear) models for the purpose of compressing time series data. In this context, we define outliers as the data points whose prediction errors based on the models are beyond the pre-defined error bound.

Please note that the outliers as defined are different from outliers in previous outlier/anomaly detection work [9, 10]. First, the outliers in the context of data compression may or may not correspond to actual anomalies meaningful to upper level applications. The definition suits our goal of designing a general-purpose compression solution. Second, while outliers in previous work can be individual points, a sequence of points, or even an entire time series [9, 10], we mainly focus on point outliers in a single time series. It would be interesting but beyond the scope of this paper to exploit application knowledge in the outlier detection for data compression.

Outlier detection methods. Existing methods can be divided into the following three categories:

- *Density-based outlier detection*: Density-based methods [5, 6] examine each point in sliding windows and count the number of neighbors (i.e., points with close values) in the windows. A point is regarded as an outlier if it does not have sufficient number of neighbors in any window. The time complexity of the methods is $O(n)$, where n is the number of data points.
- *Histogram-based outlier detection*: A number of methods [37, 68] are based on the optimal histogram vopt [36] given a fixed number of buckets. A point is considered as an outlier if its removal reduces the estimation error of the vopt histogram. The methods look for the optimal set of points whose removal minimizes the estimation error of vopt. Previous work [37] proposes a dynamic programming algorithm that computes the optimal set in $O(n^2k^2B)$ time for n data points, k outliers, and B histogram buckets. Improved algorithms [68] reduce the power of n in the time complexity, but the power of k is raised to at least 3. These methods incur high computation overhead, and therefore are not used in our study.
- *Model-based outlier detection*: Methods in this category employ models to detect outliers. ML-based models, such as LSTM [33, 63] and recurrent autoencoder ensembles [45], have high accuracy, but suffer from high computation cost. Hence, they cannot be employed for rapid

data ingestion in IoT scenarios. Shrinking cone [26] is an algorithm derived from feasible space window [59]. As shown in Figure 1b, given an error-bound e, the cone-shaped feasible slope range shrinks as more points are added. V_0 is the starting point of the segment. After V_1 is added, the feasible slope range is $[low(V_1), up(V_1)]$. After V_2 is added, the range shrinks to $[low(V_1), up(V_2)]$. If V_3 were added,then $\bigcap_{i=1}^{3}[low(V_i), up(V_i)] = \emptyset$, the range becomes empty. Thus, the segment stops at V_2.

Originally, Shrinking Cone is a segmentation method. We modify Shrinking Cone to detect outliers as follows. Since an outlier (e.g, $O1$) often disrupts a segment or forms a very short segment, we mark a point as an outlier in two cases: a) it stops a segment but the removal of the point continues the growth of the segment; b) it starts a short segment that contains less than *min_seg_len* (e.g., 5) points.

Segmentation methods. We mainly consider linear segment models as they are well-studied [20, 21, 47, 48] and amenable to fast computation and query processing without decompression.

There are a number of linear segmentation methods in the literature [21]. The top-down method [50] recursively partitions a time series until certain stopping criterion is met, while the bottom-up method [44] iteratively merges adjacent short segments to longer segments. The extreme/trend point method cuts series into different segments at extreme/trend points. In sliding window methods [46], a segment grows until its total error exceeds a user-defined threshold. Sliding window and bottom-up (SWAB) [43] combines the ideas of sliding window and bottom-up methods. In SWAB, bottom-up is used to segment points in the sliding window till there remains 5 to 6 segments. The first segment is popped from the sliding window, and new points are added. Then, SWAB performs the bottom-up computation again. In addition, the Shrinking cone method [26] as described above naturally supports segmentation. When the feasible slope range becomes empty, the growth of the current segment stops and the next point starts a new segment.

3.2 Optimal Strategy Selection for Outlier Detection and Segmentation Steps

In this subsection, we evaluate different combinations of outlier detection and segmentation methods.

Metrics. We consider the following four metrics in the comparison:

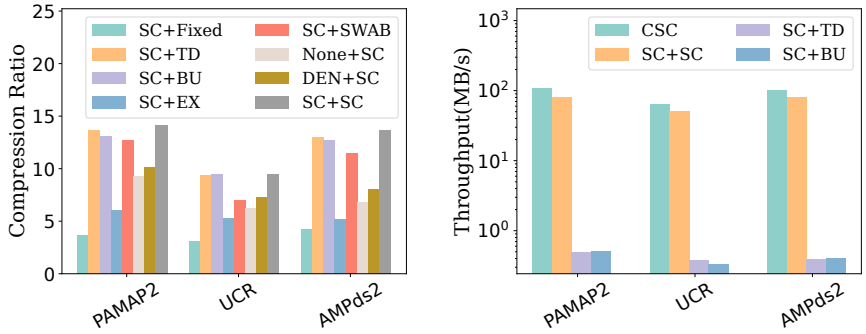
- *SPTP* (number of Segments Per Thousand Points): For each segment, MOST stores the start position, length, and model parameters. Hence, SPTP should be lowered as much as possible.
- *OR* (Outlier Rate): OR is the fraction of points that are outliers. Since MOST stores outliers, the lower the OR, the better.
- *Compression Ratio*: It is computed as the original data size divided by the compressed data size. Thus, the higher the better.
- *Compression Throughput*: We measure the throughput of a strategy for compressing data in memory.

Data sets. The experimental evaluation in Section 5 uses five real-world data sets. In the strategy selection in this subsection, we use three of the five data sets (i.e., PAMAP2, UCR, AMPds2) with diverse characteristics as the "training" data sets. The resulting MOST compression algorithm will be tested against the two "test" data sets in Section 5.

Strategy combinations. We implement three outlier detection methods: none, density-based method (DEN), and Shrinking Cone (SC). We implement six segmentation methods: fixed50 (50 points per segment), top-down (TD), bottom-up (BU), extreme points (EX), SWAB, and Shrinking Cone (SC). We set the relative error-bound $\epsilon = 0.01$. For TD, BU, and SWAB, ϵ is used in the

Table 2. Comparing SPTP and OR of different combinations of outlier detection and segmentation strategies.

SPTP / OR	None	Density (DEN)	Shrinking Cone (SC)
Fixed50	20.00 / 0.2725	20.00 / 0.2304	20.00 / 0.1696
Top-Down (TD)	39.05 / 0.0014	12.35 / 0.0491	5.49 / 0.0784
Bottom-Up (BU)	38.29 / 0.0018	12.40 / 0.0491	5.52 / 0.0776
Extreme (EX)	55.11 / 0.0521	18.45 / 0.0682	12.29 / 0.0773
SWAB	42.23 / 0.0074	15.80 / 0.0466	7.66 / 0.0777
Shrinking Cone (SC)	11.20 / 0.0777	8.81 / 0.0830	**4.89 / 0.0764**



(a) Compression ratio (b) Insertion throughput

Fig. 2. Comparing different strategy combinations.

stopping criterion. For EX, the difference between an extreme point and its neighbors should be at least ϵ. For SC, ϵ is used in the slope range computation.

Result analysis. Table 2 compares the SPTP and OR of different combinations of outlier detection and segmentation strategies. Each column corresponds to an outlier detection method. Each row corresponds to a segmentation method. Every cell reports the average SPTP / average OR for the three data sets. First, comparing SPTP across the columns, we see that SC outlier detection produces the smallest number of segments. Specifically, compared to none, detecting and skipping outliers with SC effectively reduces the number of dynamically generated segments by 2.3–7.1x. Second, we focus on the SC column, and see that SC+SC is the best combination. The SPTP/OR of SC+TD and SC+BU are close to those of SC+SC.

Figure 2a compares the compression ratio of different strategy combinations. Due to space limitation, we show only the more competitive strategy combinations. From the figure, we see that SC+SC is the best, while SC+TD and SC+BU are also quite competitive. This result is consistent with the SPTP and OR in Table 2. The lower the SPTP and OR, the better the compression ratio.

Figure 2b compares the compression throughput of SC+SC, SC+TD, and SC+BU. We see that SC+SC achieves 132–208x higher throughput than SC+TD and SC+BU. This is because TD and BU pay the cost of iterative processing of the segments.

In summary, from the experiments, it is clear that SC+SC is the best strategy of outlier detection and segmentation.

Combined Shrinking Cone (CSC) algorithm.

We observe that SC+SC performs Shrinking Cone twice on the time series data. Therefore, we propose an optimized algorithm in MOST that performs one-pass Shrinking Cone computation to improve efficiency. The combined Shrinking Cone algorithm is listed in Algorithm 1.

The algorithm computes and updates the slope range of the current segment for each point (Line 4-5, 8-9). When the slope range becomes empty, we denote the current point as a *splitter* because it tends to split the current segment. At this moment, the algorithm does not immediately start a new

Algorithm 1: Combined Shrinking Cone algorithm for outlier detection and segmentation in MOST.

```
   Input: points, ϵ /* error threshold */, min_seg_len
   Output: segments, outliers
1  segments = ∅; outliers = ∅;
2  start = 0; next = start + 1; splitter = -1; slope_lo = −∞; slope_hi = +∞;
3  while next < points.size() do
4      (cur_lo, cur_hi) = SlopeRange(points[start], points[next], ϵ);
5      if (cur_lo, cur_hi) ∩ (slope_lo, slope_hi) != ∅ then
6          if splitter >= 0 /* segment grows so skip the splitter */ then
7              outliers = outliers ∪ { splitter }; splitter = -1;
8          (slope_lo, slope_hi) = (cur_lo, cur_hi) ∩ (slope_lo, slope_hi);
9          next = next + 1;
10     else if splitter < 0 /* try skipping the point and continue */ then
11         splitter = next; next = next + 1;
12     else
13         if splitter - start >= min_seg_len /* new segment */ then
14             segments = segments ∪ { current segment };
15             start = splitter;
16         else
17             /* segment is too short! start is an outlier */
18             outliers = outliers ∪ { start }; start = start + 1;
19         next = start + 1; splitter = -1; slope_lo = −∞; slope_hi = +∞;
20 return (segments, outliers);
```

segment. Instead, it records the splitter and moves on to check the next point (Line 10-11). If the next point successfully extends the current segment, then the algorithm records the splitter as an outlier (Line 6-7), and continues growing the segment (Line 8-9). In case that the next point after the splitter splits the segment again, the algorithm tries to begin a new segment (Line 13–15). It ensures that the current segment's length is at least *min_seg_len*. If not, then we regard the segment start point as an outlier, and re-examine the other points in the segment (Line 17–18). Note that the two outlier detection cases (Line 7 and Line 18) correspond to case a) and b) as described in Section 3.1, respectively.

The time complexity of the CSC algorithm is $O(n)$ for n points. As shown in Figure 2b, the CSC algorithm improves the compression throughput of SC+SC by a factor of 1.27–1.40x.

3.3 Model Parameter and Outlier Encoding

We further reduce the space for storing outliers and model parameters with quantization and matissa truncation as follows.

Quantization for outlier encoding. Inspired by recent works on error-bounded lossy compression for scientific data [39, 94], we quantize outliers to reduce the space cost. We associate each outlier v to the segment that it resides in or the segment before the outlier if the outlier is between two segments. Then, we compute the prediction value v_p for the outlier using the segment's model. The quantization result q and the reconstructed value v' after decompression are computed as follows:

- *Absolute error-bound*: $q = round(\frac{v-v_p}{2\epsilon})$, and $v' = v_p + 2q\epsilon$.

- *Relative error-bound*: $q = round(\frac{\ln v - \ln v_p}{2\epsilon'})$, and $v' = v_p e^{2q\epsilon'}$, where $\epsilon' = ln(1+\epsilon)$.

It is easy to see that the error-bound constraints in Eqn 2 are satisfied. After quantization, we store q with Zigzag [27] and variable-length integer encoding.

Mantissa truncation for encoding slopes. Since long mantissa prevents floating point compression algorithm (e.g., Gorilla [75]) from achieving high compression ratio, MOST truncates the mantissa of slopes. We employ BFLOAT16 [35], a floating point format widely used in deep learning. BFLOAT16 has 1 sign bit, 8 exponent bits, and 7 mantissa bits (rather than 23 matissa bits in the IEEE 754 32-bit floating point format). We obtain matissa-truncated slopes by bit operations on the matissa bits. The matissa-truncated slopes are used in the main computation. Specifically, in Algorithm 1, we make sure that truncated cur_{lo} >= cur_{lo}, and truncated cur_{hi} <= cur_{hi}. We find that using BFLOAT16 to represent slopes would not considerably decrease the diversity and accuracy for linear functions. However, this is not the case for intercepts. As intercepts can be much larger than slopes, truncating matissa of intercepts has a much larger impact on prediction accuracy. Therefore, intercepts are stored as 32-bit single precision floating point numbers.

Error-bound guarantee. For normal points, Algorithm 1 guarantees that normal points fall in the Shrinking Cone slope range of the associated segment. Since the slope range computation observes the error bound ϵ, it follows that normal points satisfy the error bound. For outliers, as described in the quantization process, the error bounds hold for the reconstructed values. In summary, MOST satisfies the error-bound constraint of Eqn 2 for all points.

4 MOSTDB

We design MostDB, a TSDB prototype that stores and processes MOST compressed data, as shown in Figure 3. In the following, Section 4.1 describes the MOST storage. Then, Section 4.2 proposes a dual-mode query engine to support efficient relational queries. After that, Section 4.3– 4.6 describe the dual-mode scan, filter, aggregation, and output operators, respectively.

4.1 Segment and Outlier Storage

We store MOST compressed data in an underlying database. A time series table can be represented as:

$$data(tags, time, value_1, ..., value_k)$$

tags specify the descriptive information of the data source[1]. Every $value_j$ (j=1, ..., k) column stores a time series. The k time series are correlated and collected together. Usually, tags are strings, while measurement values are (floating point or integer) numbers.

For MOST compressed data, we create the following two tables:

- Segment table: $segment(tags, time_{start}, intercept, slope)$.
- Outlier table: $outlier(tags, time, q)$.

We require that the underlying database supports efficient data retrieval given tags and time ranges.

Given the above generic description, MostDB can be supported by any TSDB or relational database. Specifically, we employ InfluxDB [34, 69] as the underlying database. Since InfluxDB does not have native support for BFLOAT16, we declare the slope column as 32-bit floating point type and clear the lower 16 matissa bits. We leverage the default compression methods in InfluxDB (i.e., Gorilla for $intercept$ and $slope$, Simple8b for q) to compress the columns.

Regular and irregular time series. In regular time series [38], the measurement values are captured at regular intervals. Most real-world scenarios satisfy this condition. In this case, the timestamps can be computed and do not need to be stored for normal points. For irregular time

[1] *tags* may consist of multiple columns, providing different levels of description, e.g., to support a hierarchy of devices.

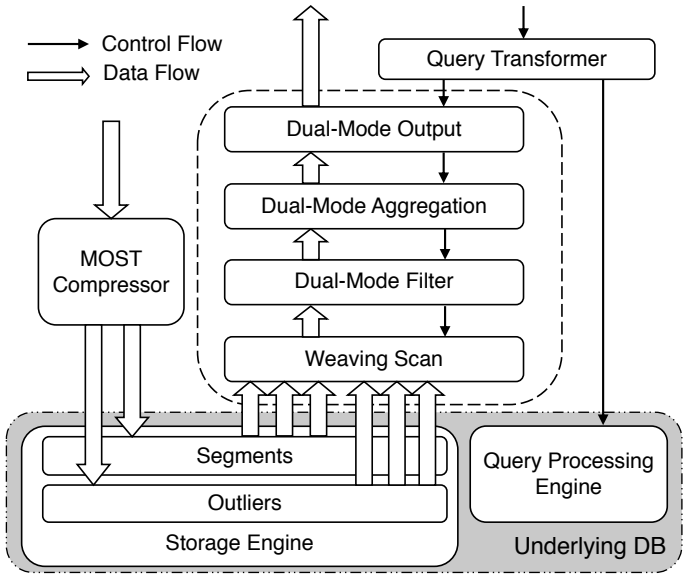


Fig. 3. Architecture of MostDB.

series, we store the timestamps as a measurement value column, and use an auto-increment id as the regular timestamp. The actual timestamp column is compressed using Delta-of-Delta encoding [75]. To support queries, time ranges are converted to the regular id ranges by checking the timestamp column.

4.2 Dual-Mode Query Engine

Query operation analysis. According to the IoTDB Benchmark [58], there are ten common query types in IoT applications, as listed in Table 3. The query operations can be divided into four categories:

- *Operations on time*: Many queries are interested in the data either at a time point (Q1, Q9) or in a time range (Q2, Q4-6, Q8, Q10). Time range can also be used as the group by key (Q10). For these operations, MostDB specifies appropriate time conditions when retrieving data from the underlying DB. Suppose the time range in a query is $[T_b, T_e]$. It is straightforward to retrieve outlier data in $[T_b, T_e]$. However, segments are different because a segment can span the boundaries of the time range. To address this problem, we take a simple approach. We maintain the maximum segment length, *max_seg_len*, and use $[T_b\text{-}max\_seg\_len, T_e+max\_seg\_len]$ to retrieve segment data. Note that *max_seg_len* is often small compared to the total data size. Hence, while this approach may load a few extra segments outside the original time range, they can be easily filtered with little overhead.
- *Operations on tags*: Since data sources are identified by tags, almost all queries contain either tag conditions (Q1,Q2,Q4,Q5,Q9, Q10) or tags as group by keys (Q6-8). To support these operations, MostDB specifies the tag conditions when retrieving data from the underlying DB.
- *Operations on values*: There are three kinds of operations on measurement values: a) Value in select clause: Q1-5 and Q9 return values as query results; b) Filter on values: Q4, Q5, Q7 and Q8 filter data with value predicates; and c) Aggregate of values: Q6-8 and Q10 compute aggregates of values.
- *Limit clause*: Q3 and Q5 use limit clause to retrieve a given number of records. This can be supported by specifying appropriate limit clauses when retrieving data from the underlying

Table 3. Common relational queries in IoT applications [58].

	Description	SQL
Q1	exact point query	`select [values] from data where time=? and [tag conditions]`
Q2	time range query	`select [values] from data where time between ? and [tag conditions]`
Q3	limit query	`select [values] from data limit ?`
Q4	time range + value filter	`select [values] from data where time between ? and [value conditions] and [tag conditions]`
Q5	Q4 with limit clause	`select [values] from data where time between ? and [value conditions] and [tag conditions] limit ?`
Q6	aggregation w/ time range	`select [aggr(value)] from data where time between ? group by [tags]`
Q7	aggregation w/ value filter	`select [aggr(value)] from data where [value conditions] group by [tags]`
Q8	Q6 + Q7	`select [aggr(value)] from data where time between ? and [value conditions] group by [tags]`
Q9	latest point query	`select time, [values] from data where [tag conditions] and time=max(time)`
Q10	group by time range query	`select [aggr(value)] from data where time between ? and [tag conditions] group by [time range]`

DB. For outliers, we use the same limit clause. For segments, we set the limit to $\frac{original\_limit}{min\_seg\_len}$. In case that query processing does not produce sufficient result records (e.g., in Q5), more data are retrieved from the underlying DB.

Since values are MOST compressed, the support for value operations is the main focus of the rest of this section.

Dual-mode query engine. As discussed in Section 2.3, query processing in existing model-based database systems uses either reconstructed values or pure-model computation. We find neither approach appealing for the following reasons. First, pure-model computation cannot support outliers, and is applicable only to limited query types. Second, it is expensive to decompress the data and evaluate queries on reconstructed values.

We propose a dual-mode query engine for processing relational queries on MOST compressed data. By dual-mode, we mean that the main query operators have two modes: the segment mode and the outlier mode. The outlier mode processes each outlier record, while the segment mode computes results for each segment as a whole. The engine performs a variant of volcano-like query execution. The main difference is that *next()* on a dual-mode operator returns a segment and a list of outliers associated with the segment.

The dual-mode query engine has three benefits. First, it naturally combines outliers with models. Second, it supports a wide range of query types. We will describe dual-mode scan, filter, aggregation, and output operators. Third, the engine performs per-segment processing and defers the reconstruction of values as much as possible, thereby substantially reducing query costs.

Result styles. MostDB provides several styles for generating query results. In the default style, the query engine computes results based on the values predicted by the segment models. This style attains the best query performance. However, predicted values are not exactly the original measurement values. To model this uncertainty, we assume that a predicted value is uniformly distributed in $[value_{lo}, value_{hi}]$, where the value range is computed using the error bound constraint. Then, MostDB provides three detailed styles that take the uncertainty into consideration. They mainly differ in the evaluation of aggregation operations. Style-E provides the expectation of the result. Style-EB generates both the expectation and the deterministic bounds of the aggregation result. Style-EBS computes the standard deviation in addition to the output of Style-EB.

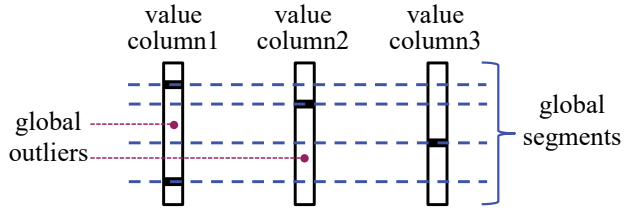
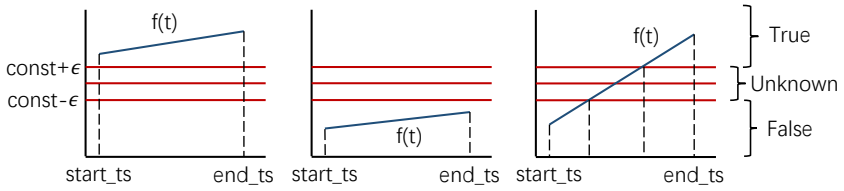


Fig. 4. Weaving scan obtains global segments and outliers.



(a) fully satisfy (b) fully dissatisfy (c) partially satisfy

Fig. 5. Evaluating a filter ($value > const$) on a segment.

4.3 Weaving Scan

As shown in Figure 3, the query engine transforms an incoming query to retrieve data from the underlying DB. Time, tags, and limit operations are handled appropriately. The scan operator collects the segments and outliers of all measurement value columns involved in the query. There can be multiple value columns (e.g., in predicates, aggregates, and/or select results). Interestingly, their segment start points and their outliers can be different because the outlier detection and segmentation of MOST is performed on each column of values alone. Therefore, the scan operator must align the segments and outliers of multiple value columns. We call this alignment action *weaving* the columns.

As shown in Figure 4, we treat all the start points from the columns as the global segment split points[2]. Next, an outlier in any value column becomes a global outlier. If it is a normal point in some column val_j, then the relevant outlier entry is computed from the associated segment of val_j. After weaving, *next()* returns a global segment and a list of global outliers.

4.4 Dual-Mode Filter Operator

Filter on segments in detailed styles. For every segment, we consider three cases when evaluating the segment against a filter predicate. Figure 5 depicts an example, where the filter is $value > const$. In case (a), the segment fully satisfies the filter. We set the result to true for the segment. In case (b), the segment fully dissatisfies the filter. We set the result to false. In case (c), the segment partially satisfies the filter. We divide the segment into (up to) three smaller segments, as shown in Figure 5c. We set the results for the true and false parts accordingly. For the unknown part, we need to consider each point. Here, we simply insert all the points of the unknown part into the outlier list, and handle them with other outliers. The unknown segment becomes empty and we set its result to false.

Filter on outliers in detailed styles. For every outlier, we generate a probability p for the filter evaluation. We first compute $[value_{lo}, value_{hi}]$ based on the predicted value and the error bound constraint. Then we compare the value range with the filter predicate. There are also three cases.

[2]In our prototype, we weave all columns in the query at once in the weaving scan operator. However, since weaving reduces segment sizes and lower the benefit of per-segment processing, a more sophisticated solution can build a weaving operator, and judiciously place the operator to delay weaving until necessary in the query plan.

First, the entire range satisfies the filter. We set p=1. Second, the entire range does not satisfy the filter. We set p=0. Third, the range intersects with the filter. We compute p according to the uniform distribution. For an example filter $value > const$, if $const \in [value_{lo}, value_{hi}]$, then $p=\frac{value_{hi}-const}{value_{hi}-value_{lo}}$.

Multiple Filters. Logical operations (i.e., AND, OR, NOT) of filter predicates can be easily supported. Please recall that segments are aligned across all value columns by the weaving scan. We may further divide a segment into smaller segments if case (c) occurs in any filter evaluation, as shown in Figure 5c. We have a true/false result after evaluating an individual filter on a given segment. Hence, it is easy to compute and/or/not on filter predicates for a segment.

For outliers, we assume that the filter results on different columns are independent. Suppose the probabilities of two filter predicates are $p(pred_1)$=p_1 and $p(pred_2)$=p_2. Then $p(pred_1\ and\ pred_2) = p_1p_2$, $p(pred_1\ or\ pred_2) = 1-(1-p_1)(1-p_2)$, $p(not\ pred_1)$=$1-p_1$.

Filter in default style. The default style is a special case of the detailed styles. For segments, case (c) has only the true and false parts. There is no unknown part. For outliers, only fully satisfy and fully dissatisfy cases exist.

Filter operator output. In general, the filter operator outputs all input segments and outliers, enhancing each segment with a true/false result, and enhancing each outlier with a probability. A segment can be safely discarded only if its result is false, and either it does not have any outliers or all its associated outliers have probability=0. An outlier can be discarded only if its probability=0 and its associated segment's result is false.

4.5 Dual-Mode Aggregation Operator

The aggregation operator obtains a segment and a list of outliers in every *child.next()* call. The returned segment has a true/false result, and every outlier has a probability. If the child operator is the weaving scan (i.e., there is no value filters), then the segment result is default to true, and the outlier probability to 1. We describe how to compute the aggregates progressively after each *child.next()*.

SUM and COUNT. For space constraint, we consider SUM with absolute error bound below. SUM with relative error bound and COUNT can be supported similarly. Suppose there are m segments $s_1, s_2, ..., s_m$ that are independent of each other. As $sum = \Sigma_{i=1}^{m} sum(s_i)$, we have $E(sum) = \Sigma_{i=1}^{m} E(sum(s_i))$, $Var(sum) = \Sigma_{i=1}^{m} Var(sum(s_i))$, $UB(sum) = \Sigma_{i=1}^{m} UB(sum(s_i))$, and $LB(sum) = \Sigma_{i=1}^{m} LB(sum(s_i))$. UB and LB stand for upper and lower bounds, respectively. Therefore, we can progressively accumulate sum, E, Var, UB, and/or LB for each segment depending on the result style. In the following, we focus on a single segment and omit the segment subscript.

Suppose there are n points in the segment. The predicted value of point j is $f(j)$. We use r_s=1 or 0 to denote if the segment result is true or false. Suppose the set of outliers is out. For an outlier $u \in out$, denote its quantized error as q_u. Then, the reconstructed $v'_u = f(u) + 2q_u\epsilon$, and the measurement v_u follows the uniform distribution with $E(v_u) = v'_u$ and $Var(v_u) = \epsilon^2/3$. Suppose the filter columns and the aggregate column are independent. Let the outlier probability be p_u. Then, the event that the filter conditions are satisfied has mean p_u and variance $p_u(1-p_u)$. We have:

$$
\begin{aligned}
sum(s) &= r_s\Sigma_{j=1}^{n} f(j) + \Sigma_{u\in out}(p_u v_u - r_s f(u)) \\
E(sum(s)) &= r_s\Sigma_{j=1}^{n} f(j) + \Sigma_{u\in out}(p_u - r_s) f(u) + \Sigma_{u\in out} 2p_u q_u \epsilon \\
Var(sum(s)) &= \Sigma_{u\in out} p_u(1-p_u)(f(u) + 2q_u\epsilon)^2 \\
&\quad + \tfrac{\epsilon^2}{3}(r_s n + \Sigma_{u\in out}(p_u - r_s)) \\
UB(sum(s)) &= r_s\Sigma_{j=1}^{n} f(j) + r_s n\epsilon + \Sigma_{u\in out}(UB_u - r_s f(u)) \\
LB(sum(s)) &= r_s\Sigma_{j=1}^{n} f(j) - r_s n\epsilon + \Sigma_{u\in out}(LB_u - r_s f(u))
\end{aligned}
$$

$$UB_u = max\{f(u) + (2q_u + 1)\epsilon, 0\}$$
$$LB_u = min\{f(u) + (2q_u - 1)\epsilon, 0\}$$

Note that $f()$ is a linear function. Hence, $\Sigma_{j=1}^{n} f(j)$ is the sum of an arithmetic series. It is easy to see that the time complexity of the above computation is $O(|out|)$. As the number of outliers $|out|$ is expected to be small, we can efficiently compute and accumulate the result of each segment.

AVG. In the default style, we accumulate sum and count, then compute $avg = \frac{sum}{count}$. In detailed styles, for $E(avg)$ and $Var(avg)$, we use the Monte Carlo method. For each uncertain outlier u, we simulate whether it exists with probability p_u, and generate the value v_u to be uniformly distributed in $[f(u) + (2q_u - 1)\epsilon, f(u) + (2q_u - 1)\epsilon]$. To calculate $UB(avg)$, we first compute the sum and count of the certain points. Then, we sort all the uncertain points in descending order of UB_u. After that, we examine each UB_u from the largest to the smallest. If $UB_u >$ current average, then we include it and update the average. We stop when the check fails. The lower bound is computed in a similar fashion.

4.6 Dual-Mode Output Operator

The output operator supports measurement values in the select clause of queries. It obtains a segment and its associated outliers in each *child.next()* call. For every point j, it computes the reconstructed value $f(j)$ from the segment model unless it is an outlier. For an outlier u, it computes $v'_u = f(u) + 2q_u\epsilon$. The output operator returns a set of tuples as the query result. In this way, we hide the details of models and outliers from users.

5 EVALUATION

We perform extensive experiments to evaluate MOST and MostDB using real-world data sets in this section.

5.1 Experimental Setup

Machine configuration. We conduct all experiments on a server equipped with an Intel i7-4790 CPU (3.60GHz, 4 cores, 32 MB L3 cache), 32GB 1600MHz DDR3 memory, and a 1TB SSD using SATA 3.0. The server runs Ubuntu Linux 22.04 LTS. C/C++ code is compiled with GCC 11.3.0. and Java code with OpenJDK 11.0.17.

Implementation. We implement MostDB in C++ (1320 lines) and Java (1525 lines). The interaction with the underlying DB, the query transformer, and the user interface functions (e.g., insert and query operations) are coded in Java. The MOST compressor and the query engine are written as C++ sub-routines. We invoke the C++ code with JNI, and use the Java Unsafe off-heap memory to store intermediate data. For data insertion, MostDB buffers the time series data in off-heap memory. Each time series occupies a buffer. As soon as a buffer is full, MostDB gets an idle work thread from the thread pool to process the buffer. The thread uses JNI to invoke the C++ MOST compressor to compress the data, then calls InfluxDB's Java API to insert the segments and outliers with InfluxDB's default compression setting. For a user query, MostDB transforms the query, and calls InfluxDB's Java API with the transformed query. It retrieves MOST compressed data into the off-heap memory. Then, MostDB uses JNI to invoke the C++ dual-mode query engine to compute the query results. Besides InfluxDB, we port MostDB to run on top of IoTDB to show the generality of the design.

Solutions to compare. We compare MOST with state-of-the-art general-purpose compression methods (i.e., Snappy, LZ4, GZip), record-oriented methods (i.e., RLE, SplitDouble [74], Gorilla [75],

Table 4. Data sets used in our experiments.

Data Set	Size[1]	Measurement Columns	Length of Longest Series	Tags
MSRC-12	516MB	80	24K	30
PAMAP2	1.7GB	52	440K	9
UCI GAS	1.2GB	18	4.2M	2
UCR	855MB	40–24K	2709 (27K)[2]	0 (1)[3]
AMPds2	229M	2–24	1M	0 (1)[3]

1. This reports the size of the binary uncompressed data (not the text data size). Each measurement value and each timestamp take 8B.
2. Compression tests use the original UCR data sets. In query experiments, we interpolate 9 values between every 2 points to obtain 10x longer series.
3. UCR and AMPds2 do not contain tags. We add a single tag to the time series to support tag conditions in query experiments.

TS-2DIFF, BUFF [55]), model-based methods (i.e., Sprintz [8] and SZ3 [52]), and the default compression methods in InfluxDB [34], IoTDB [92], and ModelarDB [38]. The source code of BUFF is from github [73]. We set Buff's precision to achieve similar error bound as MOST for each column. For Sprintz, we quantize floating point data into 16-bit integers, then employ its FIRE model and Huffman encoding to compress the data. We denote the resulting method sprintzFIRE+Huff. We obtain SZ3 code from github [25]. We compare MostDB with three state-of-the-art TSDBs. For InfluxDB 2.1.0 [34] and IoTDB 0.11.2 [92], we use their Linux release packages and follow the default configurations. For ModelarDB [38], we obtain its Rust code from github [66].

Unless otherwise noted, we set the relative error bound $\epsilon = 0.01$ in the experiments.

Data sets. Our experiments use five real-world data sets, as summarized in Table 4.

- **MSRC-12** [23]: The Microsoft Research Cambridge-12 Kinect gesture dataset consists of sequences of human movements and gestures when the subjects are interacting with a video game. Body part locations are captured at a sample rate of 30Hz.
- **PAMAP2** [77, 78]: This data set describes physical activities performed by 9 subjects wearing 3 inertial measurement units and a heart rate monitor with a sample rate of 100Hz and ~9Hz, respectively. We replace missing values (i.e., NaN) in the data with linear interpolation based on the neighbor points.
- **UCI GAS** [22]: This data set contains gas concentration readings during chemical experiments at a sample rate of 100Hz.
- **UCR** [16]: The UCR Time Series Archive (2018) consists of 128 sub-datasets without timestamps from various domains. Our compression experiments use the original UCR data sets. In query experiments, since most sub-datasets are short, we interpolate 9 values between every 2 points to obtain 10x longer time series. Random fluctuations within the error bound are added to the interpolated values.
- **AMPds2** [62]: The data set collects electricity, water, and natural gas measurements at one minute intervals in a house. The number of measurement columns varies from 2 to 24 in the 38 sub-datasets. There is no tag column.

Three data sets (i.e., PAMAP2, UCR, AMPds2) are used as training data for selecting the optimal strategy in Section 3.2. We report the experimental results for all five data sets in this section.

The timestamps in the data sets are regular. In the experiments of varying irregular rate of timestamps, we randomly choose data points to add random timestamp fluctuations.

Queries. We formulate four representative test queries based on the ten common queries in IoT applications as discussed in Section 4.2. We make sure to have all three kinds of operations on

Table 5. Test query templates in our experiments.

	Description	SQL
TQ1	exact point/time range query	`select [value] from data where time between ? and tag=?`
TQ2	value filter query	`select [value1] from data where time between ? and [value2 condition] and tag=?`
TQ3	aggregation query w/ value filter	`select sum/avg(value1) from data where time between ? and [value2 condition] group by [tag]`
TQ4	aggregation query w/ two value filters	`select sum/avg(value1) from data where time between ? and [value2 condition] and [value3 condition] group by [tag]`

measurement values: a) value in select clause (i.e., retrieving decompressed data), b) filter on values, and c) aggregate of values. Specifically, TQ1 contains a) and represents Q1 and Q2. TQ2 contains a) and b), and represents Q4. TQ3 and TQ4 both contain b) and c), and represent Q8. Since Q8 combines the functionality of Q6 and Q7, TQ3 and TQ4 essentially cover Q6–Q8[3]. Table 5 lists the four query types[4].

Each data set contains multiple time series or sub-datasets with different lengths. In the query experiments, the largest range size is at least 10^4 points. So, we exclude time series shorter than this size. Then, we randomly generate queries according to the query templates using the remaining time series. Note that the compression and insertion experiments still use all data in a data set.

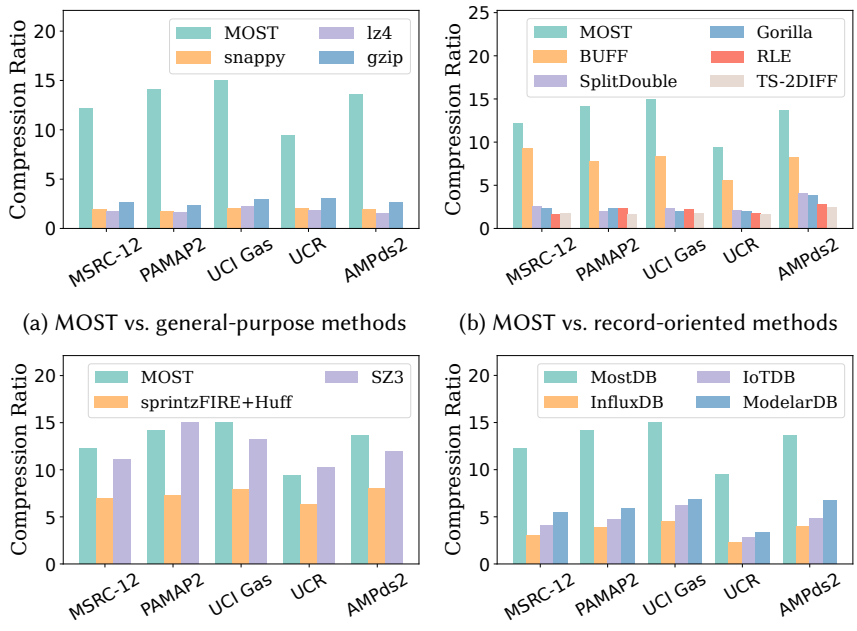
TSBS Benchmark. The Time Series Benchmark Suite (TSBS) [89] consists of a set of data generation tools and queries for evaluating TSDBs. It supports two use cases: dev-ops and IoT. A dev-ops use case simulates the monitoring of data center machines. We employ the CPU-only dev-ops data set, which contains 10 CPU metrics. The IoT use case simulates data collected from the trucks in a fictitious trucking company, which consists of 7 reading metrics and 3 diagnostic columns. For both use cases, we follow the popular settings in existing TSBS tests [90] to generate time series data for 100 and 4000 devices at a sample interval of 10 seconds. In all cases, we generate 300 million records for each time series. We use double for metric values. We set $\epsilon = 0.02$ for TSBS.

We choose six representative query types, three for each use case: 1) cpu single-groupby-1-8-1: compute max(value) for 8 hosts for every 5 minutes in a one-hour period; 2) cpu double-groupby-1: compute avg(value) per host per hour for a 24-hour period; 3) cpu high-cpu-1: for a chosen host, find all readings where value > threshold; 4) iot low-fuel: find all trucks with less than 10% fuel; 5) iot stationary-trucks: find all trucks that are at low average velocity for the last 10 minutes; and 6) iot avg-daily-driving-duration: calculate the average daily driving duration per truck. For each query type, we generate 1000 random queries. The insertion experiments use both 100-device and 4000-device data sets, while the query experiments focus on the 100-device data sets.

5.2 Compression Performance

Comparison with state-of-the-art compression techniques. Figure 6 reports the compression ratio of MOST and state-of-the-art compression techniques. Compression ratio is computed as before-compression size divided by after-compression size. So the higher the better. For Figure 6a–c, we measure the input and output sizes of the compression techniques. For a data set, we compress each measurement column separately. Then, we consider the total size of all measurement columns before and after compression.

[3]Q6 is less challenging than Q8. Q6 can be efficiently supported by storing pre-computed aggregates for time ranges.
[4]We omit four queries with (i) limit clauses (Q3, Q5), (ii) latest time point (Q9), and group by on time ranges (Q10). These operations are tangential to our focus of queries on MOST compressed data. They are not yet supported by the current prototype.



(a) MOST vs. general-purpose methods

(b) MOST vs. record-oriented methods

(c) MOST vs. model-based methods

(d) MostDB vs. state-of-the-art TSDBs

Fig. 6. Compression ratio on real-world data sets.

As shown in Figure 6a and Figure 6b, MOST achieves 3.13–9.03x improvements compared to general-purpose and record-oriented lossless compression methods. The compression ratio of lossless compression is relatively low because small differences in floating point values can cause significant changes in the binary representation. Compared with BUFF, MOST attains 1.32–1.83x improvements. BUFF truncates matissa based on the precision setting. This reduces the impact of matissa changes on compression ratio. In comparison, MOST exploits linear models with bounded errors and outlier storage to effectively deal with floating point fluctuations.

Figure 6c compares MOST with state-of-the-art model-based compression techniques with errors. We see that compared to sprintzFIRE+Huff, MOST is 1.48x–1.94x better. On the other hand, MOST and SZ3 have comparable compression ratio. SZ3 requires the data to be fully decompressed before query processing. This incurs poor query performance, as will be shown in Section 5.3.

Comparison with state-of-the-art TSDBs. Figure 6d compares MostDB with InfluxDB, IoTDB, and ModelarDB. Note that MostDB uses InfluxDB as the underlying database. We employ the default compression settings for InfluxDB and IoTDB, and set the relative error $\epsilon = 0.01$ for ModelarDB. The data size after compression is computed as the disk space consumed by all files in the data directory, including metadata, and indices on timestamps and tags.

From Figure 6d, MostDB achieves 2.36–4.24x, 2.42–3.37x, 2.02–2.84x higher compression ratio than InfluxDB, IoTDB, and ModelarDB, respectively. This result shows that MOST can effectively improve the compression ratio in TSDBs. While also a model-based TSDB, ModelarDB has lower compression ratio because it starts new segments when encountering outliers, producing a large number of short segments, adversely impacting the compression ratio.

Ablation study for MOST compression. To understand the benefit of the components in MOST, we perform an ablation study in Figure 7. We compare the following variants of MOST with different component removed or replaced: 1) *MOST W/O OD* removes outlier detection in MOST; 2) *MOST(FIXED50)* replaces our SC-based dynamic segmentation with FIXED50 (50 points per

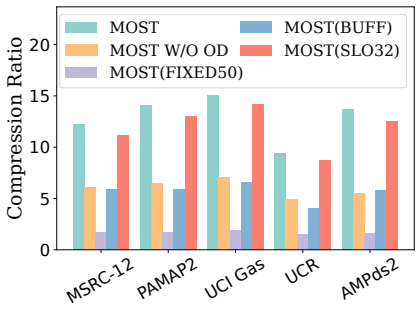


Fig. 7. Ablation study for MOST.

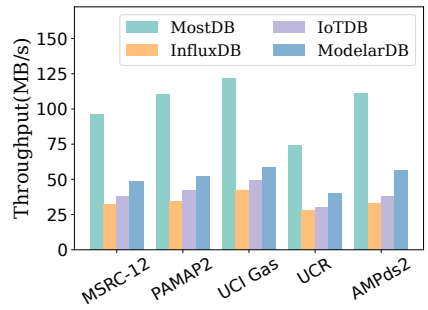
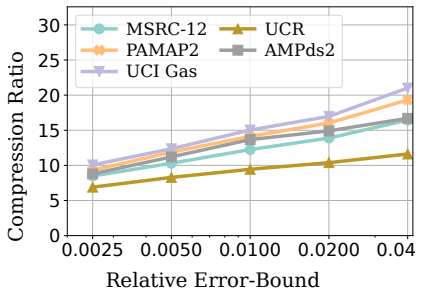
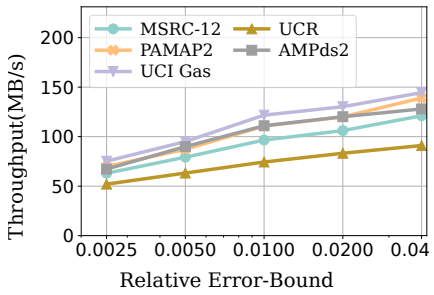


Fig. 8. Insertion for real-world data sets.

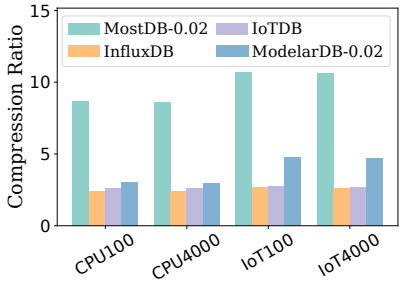


(a) Compression ratio of MOST

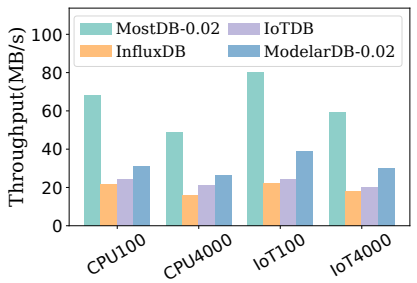


(b) Insertion speed of MostDB

Fig. 9. Compression performance varying error bound ϵ.



(a) Compression ratio



(b) Insertion throughput

Fig. 10. Compression performance for TSBS benchmark.

segment); 3) *MOST(BUFF)* applies BUFF rather than quantized error to encode outliers; and 4) *MOST(SLO32)* replaces BFLOAT16 with 32-bit floating point numbers to encode the slopes of linear models. From Figure 7, we see that all MOST variants show worse compression ratio, indicating the contribution of the individual techniques to MOST. Specifically, the compression ratio improves by a factor of 1.92-2.49x with outlier detection, 6.29-8.57x with SC-based dynamic segmentation, 2.07-2.42x with quantized error encoding for outliers, and 1.06-1.10x with BFLOAT16 encoding for slopes.

Insertion throughput. Figure 8 reports the insertion throughput for the five real-world data sets. InfluxDB, IoTDB, and ModelarDB use their default compression settings to compress the data. The throughput is calculated as the original data volume divided by the insertion execution time. From the figure, we see that compared to InfluxDB, IoTDB, and ModelarDB, MostDB attains 2.65–3.36x, 2.44–2.93x, and 1.86–2.12x higher insertion throughput, respectively. While data is

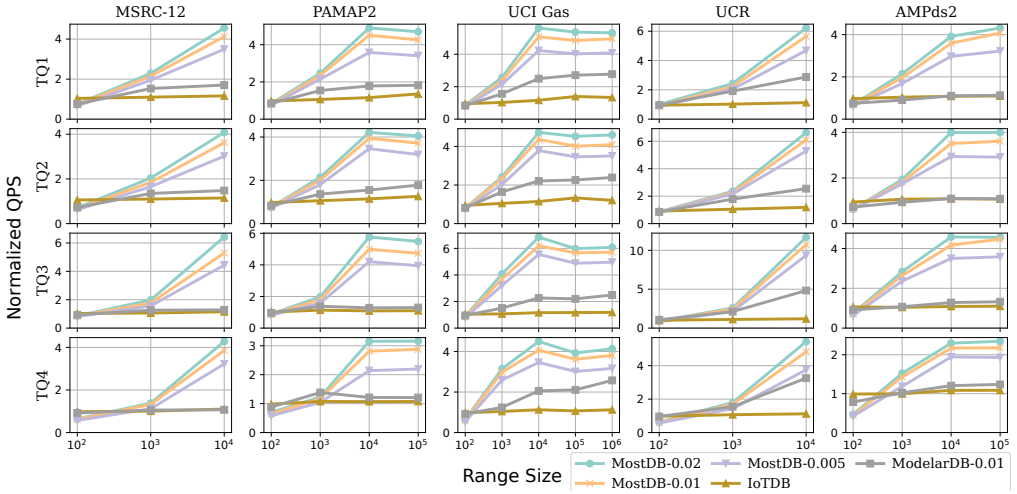


Fig. 11. Query processing performance of TQ1-TQ4 (query throughput normalized to that of InfluxDB).

stored in InfluxDB, MostDB sees better insertion performance than InfluxDB because MostDB has higher compression ratio, and therefore saves a substantial amount of I/O.

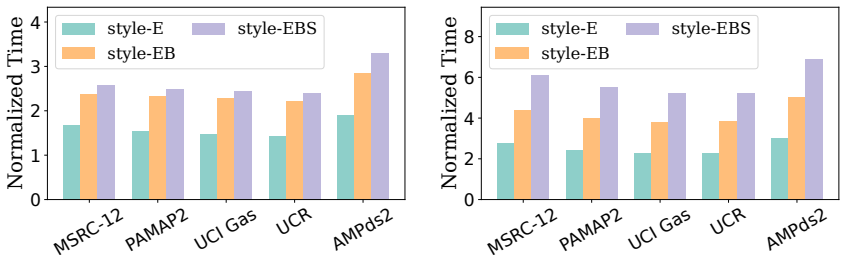
Varying error bound. Figure 9 shows compression ratio and insertion throughput while varying the error bound ϵ from 0.0025 to 0.04. The default ϵ is 0.01.

From Figure 9a and 9b, we see that both compression ratio and insertion throughput increase as the error bound increases. This is because there are fewer outliers when the error bound is larger, which leads to better compression ratio. As the compression ratio increases, the amount of data to store decreases, and therefore insertion throughput improves. Overall, when ϵ increases from 0.0025 to 0.04, the compression ratio increases from 6.90–10.07 to 11.62–21.01. In summary, MOST compression is capable of achieving high compression ratio.

Compression performance for TSBS benchmark. Figure 10 shows the compression ratio and the insertion throughput of MostDB and the TSDBs for the dev-ops cpu-only and the IoT use cases. From the figure, we see that compared to the lossless compression in InfluxDB and IoTDB, MostDB achieves a factor of 3.28-4.08x improvement for compression ratio. Compared to the lossy compression in ModelarDB, MostDB improves the compression ratio by a factor of 2.27-2.89x. Moreover, MostDB achieves 1.84–3.61x better insertion throughput compared to InfluxDB, IoTDB, and ModelarDB. As the number of devices increases, the insertion throughput decreases. This is because the underlying DB incurs many random I/O accesses to deal with the various device tags.

5.3 Query Performance

Query performance in default style. Figure 11 reports the performance of TQ1–TQ4 in default result style. For each query template, we randomly generate 10,000 queries in a data set to form a query workload. The X-axis varies range size from 10^2 to 10^4 points. Since the data in all five data sets are collected at regular intervals, we can easily convert the range size to the time range. To generate a query, we first randomly choose the measurement value column(s) according to the query templates. Specifically, TQ1 requires one value column, TQ2 and TQ3 require two value columns, and TQ4 requires three value columns. For a value condition, we generate a random constant value in the column's value range and create a predicate like $value\ \theta\ const$, where θ is randomly chosen from >, =, and <. We run a generated query workload on a TSDB using a single thread to issue the queries one at a time, and measure the query throughput in QPS. In the figure,



(a) SUM aggregation query (b) AVG aggregation query

Fig. 12. Query execution time in different styles (normalized to time in default style).

we report QPS normalized to that of InfluxDB. We compare InfluxDB, IoTDB, ModelarDB, and three variants of MostDB with error bound ϵ from 0.005 to 0.02.

From Figure 11, we see the following trends. First, when the range size is small, MostDB performs worse than InfluxDB and IoTDB. This is because the queries retrieve only a small amount of data with random I/Os. There is little savings on I/O cost, and MostDB pays extra cost for processing data retrieved from InfluxDB. Interestingly, for the cases where MostDB's QPS is lower, the query latency ranges from 0.11ms to 27.47ms. Hence, MostDB can execute these queries sufficiently fast to support user interactions.

Second, as the range size increases, MostDB achieves significant performance benefits in all cases. As more data are retrieved, the I/O access becomes more and more sequential. The higher compression ratio of MostDB leads to smaller I/O costs. Dual-mode query processing further improves performance. Compared to InfluxDB and IoTDB, MostDB achieves up to 6.22x speedups for TQ1, 6.63x speedups for TQ2, 11.68x speedups for TQ3, and 5.44x speedups for TQ4. Compared to ModelarDB, MostDB performs up to 4.99x better. Note that the pure-model computation in ModelarDB cannot support value predicates because segments may partially satisfy the predicates. Hence, ModelarDB reconstructs the data points from models for evaluating the queries. In comparison, our dual-mode query engine is capable of combining the processing of models and outliers for a wide range of queries.

Third, as the error bound increases from 0.005 to 0.02, the query performance improves. This is because the compression ratio increases as the error bound (cf. Figure 9a), and MostDB retrieves smaller amount of data from the underlying database.

Finally, TQ4 processes one more value column than TQ3. With more value columns, weaving generates finer global segments, making segment mode computation less effective. Hence, we see that TQ4 has lower performance than TQ3.

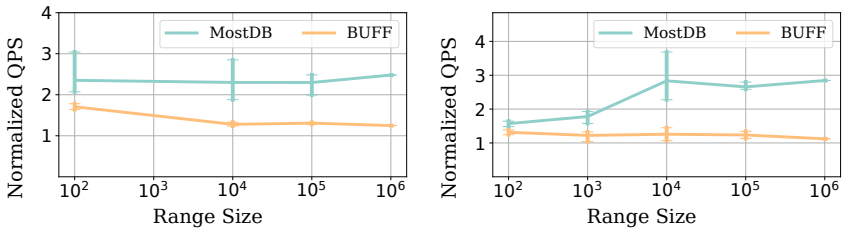
Overall, we conclude that MostDB supports general-purpose queries on compressed data. It can bring significant query performance benefits compared to InfluxDB, IoTDB and ModelarDB.

Query performance in detailed styles. We evaluate the three detailed result styles by running 10000 random TQ4 queries on UCI GAS. Figure 12a and 12b report the execution time of the detailed styles normalized to the default style. We see that the estimation of expectation, deterministic bounds, and standard deviation all incur additional overhead. Moreover, AVG is slower than SUM because of the cost of the Monte Carlo simulation for AVG.

As shown in Table 6, the expectation $E(aggr)$ is within 0.112%–0.146% of the accurate result $aggr$ computed with the original data. Both the standard deviation and the deterministic bound work well as expected. This verifies that our computation is reliable. Interestingly, the result $aggr'$ in the default style is already good, which is within 0.189%–0.294% of the accurate result. Moreover, we compute the distribution error [19] as the L^2 distance between normalized distributions of the approximate and the accurate group-by results. MostDB achieves low distribution errors (i.e.,

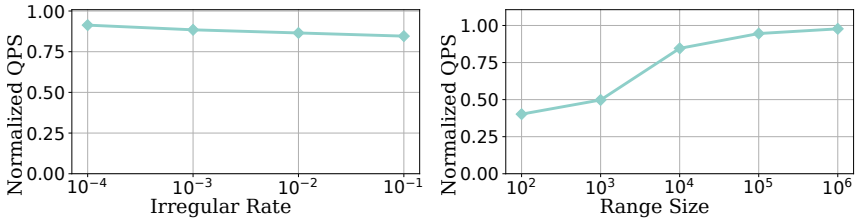
Table 6. Accuracy of the aggregation results.

	sum query	**avg query**
Accuracy of $aggr'$**:** $avg(\lvert\frac{aggr'-aggr}{aggr}\rvert)$	0.189%	0.294%
Accuracy of $E(aggr)$**:** $avg(\lvert\frac{E(aggr)-aggr}{aggr}\rvert)$	0.112%	0.146%
% within $E(aggr) \pm 3\sigma$	98.84%	99.20%
% within deterministic bound	100%	100%
Distribution error	0.278%	0.427%



(a) Filter performance (b) Aggregation performance

Fig. 13. Comparing query performance of MostDB, SZ3 and BUFF. (QPS is normalized to that of SZ3.)



(a) Varying ir-rate (range=10^4) (b) Varying range size (ir-rate=0.1)

Fig. 14. TQ3 with irregular timestamps. (QPS is normalized to that of regular timestamp case.)

0.278% for SUM and 0.427% for AVG) because it incorporates all relevant records in the group-by aggregate computation rather than using a small sample as in sample-based AQP.

MOST vs. SZ3 and BUFF. MOST and SZ3 have similar compression ratio. BUFF is the most competitive record-oriented compression method. We compare their query performance. Since no TSDB supports SZ3, we run the queries fully in memory. Figure 13a shows the performance of a filter query (TQ2), while Figure 13b reports the performance of an aggregate query (`select sum(value1) from data where time between ? and [value2 condition] and tag=?`) (i.e., TQ3 without group by). The line curves show both the mean and the range of normalized QPS over the five data sets.

From the figures, we see that MOST achieves much better query performance than SZ3. This is because SZ3 pays the cost to decompress a data block (containing 128 floating point numbers by default) before computing. BUFF supports progressive filter and aggregate computation on compressed data. For the filter query, BUFF exploits SIMD to achieve good performance. However, this is less effective for the aggregation of a scattered subset of records satisfying the value predicate. In comparison, the dual-mode processing in MostDB can process a segment of data with low cost, thereby achieving better performance.

Query performance with irregular timestamps. MostDB supports both regular and irregular time series, as described in Section 4.1. In this set of experiments, we vary both the irregular rate (or ir-rate) of timestamps and the range size. We randomly choose an ir-rate fraction of data points from UCI Gas to add random timestamp fluctuations while preserving the time order of the data points.

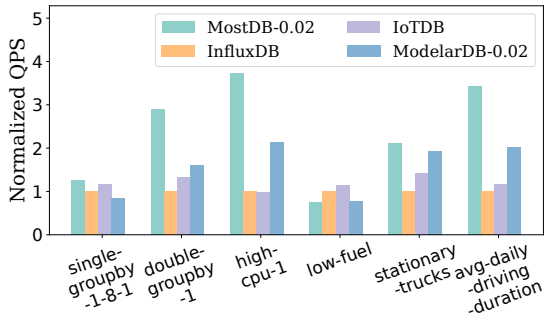


Fig. 15. Query performance for TSBS (normalized to InfluxDB).

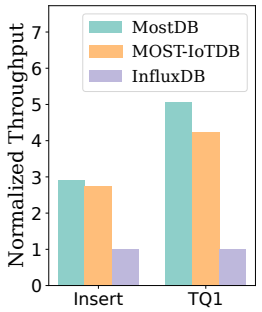


Fig. 16. MOST on IoTDB.

MostDB stores the timestamps as a separate column in InfluxDB. The extra space cost reduces MostDB's compression ratio slightly from 15.04 to 12.71 in the worst case (i.e., ir-rate=0.1). As shown in Figure 14a, we see that MostDB pays slight overhead for supporting irregular timestamps. This corresponds to the cost of calling InfluxDB to convert time ranges to regular id ranges for a query. As shown in Figure 14a, the conversion overhead stays flat when the ir-rate varies. As shown in Figure 14b, as the range size increases, the query time increases and the relative overhead of the conversion decreases drastically (to close zero for large ranges).

Queries in TSBS benchmark. Figure 15 shows the query performance of the TSBS benchmark. We see similar trends as in Figure 11: MostDB's performance benefit is higher for queries with larger time ranges. For double-groupby-1, high-cpu-1, and avg-daily-driving-duration, MostDB achieves significantly (up to 3.73x) better performance than the other TSDBs. These queries examine large time ranges, processing a large amount of time series data. For single-groupby-1-8-1 and stationary-trucks, the time ranges (i.e., 1 hour and 10 minutes) are smaller, and the performance advantage of MostDB is lower. For low-fuel, it is a point query, reading the latest data. While MostDB is slightly worse, its query latency is 108.3 ms, which is sufficiently fast to support user interactions.

MostDB with different underlying DB. Finally, we show the generality of the MostDB design by porting it to run on top of IoTDB. Figure 16 shows the insertion and query (TQ1) performance of MostDB-IoTDB. The experiments use the UCI Gas data set and the time range is 10^4 points. We see that MostDB-IoTDB achieves similar performance compared to MostDB-InfluxDB. Both MostDB implementations are faster than InfluxDB.

6 CONCLUSION

In conclusion, we propose and evaluate MOST (**M**odel-based compression with **O**utlier **ST**orage) for time series data in this paper. We build a prototype MostDB with a segment-outlier dual-mode query engine that computes segments as a whole as much as possible. Our experimental results on five real-world data sets confirm that MOST is capable of supporting high compression ratios, good data accuracy, general-purpose queries, and high performance queries on compressed data.

ACKNOWLEDGMENTS

This work is partially supported by Natural Science Foundation of China (62172390) and CCF-Huawei Database Innovation Research Funding. We thank the anonymous meta-reviewer and reviewers for their insightful comments and suggestions. Shimin Chen is the corresponding author.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European*

Conference on Computer Systems. 29–42.
[2] Bikash Agrawal. 2013. *Analysis of large time-series data in OpenTSDB*. Master's thesis. University of Stavanger, Norway.
[3] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. 1993. Efficient Similarity Search In Sequence Databases. In *Foundations of Data Organization and Algorithms, 4th International Conference, FODO'93, Chicago, Illinois, USA, October 13-15, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 730)*, David B. Lomet (Ed.). Springer, 69–84. https://doi.org/10.1007/3-540-57301-1_5
[4] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)* 37, 1 (2018), 1–30.
[5] Fabrizio Angiulli and Fabio Fassetti. 2007. Detecting distance-based outliers in streams of data. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão (Eds.). ACM, 811–820. https://doi.org/10.1145/1321440.1321552
[6] Fabrizio Angiulli and Fabio Fassetti. 2010. Distance-based outlier queries in data streams: the novel task and algorithms. *Data Min. Knowl. Discov.* 20, 2 (2010), 290–324. https://doi.org/10.1007/s10618-009-0159-9
[7] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Softw. Pract. Exp.* 40, 2 (2010), 131–147. https://doi.org/10.1002/spe.948
[8] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.
[9] Ane Blázquez-García, Angel Conde, Usue Mori, and José Antonio Lozano. 2021. A Review on Outlier/Anomaly Detection in Time Series Data. *ACM Comput. Surv.* 54, 3 (2021), 56:1–56:33. https://doi.org/10.1145/3444690
[10] Paul Boniol, John Paparrizos, Themis Palpanas, and Michael J. Franklin. 2021. SAND: Streaming Subsequence Anomaly Detection. *Proc. VLDB Endow.* 14, 10 (2021), 1717–1729.
[11] Kin-pong Chan and Ada Wai-Chee Fu. 1999. Efficient Time Series Matching by Wavelets. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia, March 23-26, 1999*, Masaru Kitsuregawa, Michael P. Papazoglou, and Calton Pu (Eds.). IEEE Computer Society, 126–133. https://doi.org/10.1109/ICDE.1999.754915
[12] Yann Collet. 2017. Lz4-extremely fast compression. Website. https://github.com/Cyan4973/lz4, Last accessed on 2022-10-05.
[13] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1-3 (2012), 1–294.
[14] International Data Corporation(IDC). 2020. Rethinking Data: Put More of Your Business Data to Work—From Edge to Cloud. Website. https://www.seagate.com/files/www-content/our-story/rethink-data/files/Rethink_Data_Report_2020.pdf, Last accessed on 2022-10-05.
[15] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16, 4 (2007), 523–544.
[16] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. 2018. The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.
[17] Guy Van den Broeck and Dan Suciu. 2017. Query Processing on Probabilistic Data: A Survey. *Found. Trends Databases* 7, 3-4 (2017), 197–341.
[18] Amol Deshpande and Samuel Madden. 2006. MauveDB: supporting model-based user views in database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 73–84. https://doi.org/10.1145/1142473.1142483
[19] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 679–694. https://doi.org/10.1145/2882903.2915249
[20] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984.
[21] Philippe Esling and Carlos Agón. 2012. Time-series data mining. *ACM Comput. Surv.* 45, 1 (2012), 12:1–12:34. https://doi.org/10.1145/2379776.2379788
[22] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and*

Actuators B: Chemical 215 (2015), 618–629.
[23] Simon Fothergill, Helena Mentis, Pushmeet Kohli, and Sebastian Nowozin. 2012. Instructing people for training gestural interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1737–1746.
[24] The Apache Software Foundation. 2020. Apache Druid Home Page. Website. https://druid.apache.org, Last accessed on 2022-11-03.
[25] SZ Lossy Compressor Framework. 2021. SZ3: A Modular Error-bounded Lossy Compression Framework for Scientific Datasets. Website. https://github.com/szcompressor/SZ3, Last accessed on 2022-10-05.
[26] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1189–1206. https://doi.org/10.1145/3299869.3319860
[27] Google. 2001. Protocol Buffers Encoding. Website. https://developers.google.com/protocol-buffers/docs/encoding#types, Last accessed on 2022-10-05.
[28] Google. 2011. Google Snappy home page. Website. https://www.gzip.org, Last accessed on 2022-10-05.
[29] Peter J Haas. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings. Ninth International Conference on Scientific and Statistical Database Management (Cat. No. 97TB100150)*. IEEE, 51–62.
[30] Zahra Hajirahimi and Mehdi Khashei. 2019. Hybrid structures in time series modeling and forecasting: A review. *Eng. Appl. Artif. Intell.* 86 (2019), 83–106. https://doi.org/10.1016/j.engappai.2019.08.018
[31] Christopher B. Hauser and Stefan Wesner. 2018. Reviewing Cloud Monitoring: Towards Cloud Resource Profiling. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE Computer Society, 678–685. https://doi.org/10.1109/CLOUD.2018.00093
[32] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Joan Peckham (Ed.). ACM Press, 171–182.
[33] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[34] InfluxData. 2015. InfluxData. Website. https://www.influxdata.com, Last accessed on 2022-10-05.
[35] Intel. 2018. BFLOAT16 - Hardware Numerics Definition. Website. https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf, Last accessed on 2022-10-05.
[36] Yannis E. Ioannidis and Viswanath Poosala. 1995. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 233–244.
[37] H. V. Jagadish, Nick Koudas, and S. Muthukrishnan. 1999. Mining Deviants in a Time Series Database. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 102–113. http://www.vldb.org/conf/1999/P9.pdf
[38] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. *Proc. VLDB Endow.* 11, 11 (2018), 1688–1701. https://doi.org/10.14778/3236187.3236215
[39] Sian Jin, Sheng Di, Jiannan Tian, Suren Byna, Dingwen Tao, and Franck Cappello. 2022. Improving Prediction-Based Lossy Compression Dramatically via Ratio-Quality Modeling. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2494–2507. https://doi.org/10.1109/ICDE53745.2022.00232
[40] Kothuri Venkata Ravi Kanth, Divyakant Agrawal, and Ambuj K. Singh. 1998. Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, Laura M. Haas and Ashutosh Tiwary (Eds.). ACM Press, 166–176. https://doi.org/10.1145/276304.276320
[41] Yannis Katsis, Yoav Freund, and Yannis Papakonstantinou. 2015. Combining Databases and Signal Processing in Plato. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper26.pdf
[42] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems* 3, 3 (2001), 263–286.
[43] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*. IEEE, 289–296.
[44] Eamonn J. Keogh. 1997. Fast Similarity Search in the Presence of Longitudinal Scaling in Time Series Databases. In *9th International Conference on Tools with Artificial Intelligence, ICTAI '97, Newport Beach, CA, USA, November 3-8, 1997*. IEEE Computer Society, 578–584.

[45] Tung Kieu, Bin Yang, Chenjuan Guo, and Christian S. Jensen. 2019. Outlier Detection for Time Series with Recurrent Autoencoder Ensembles. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 2725–2732. https://doi.org/10.24963/ijcai.2019/378

[46] Antti Koski, Martti Juhola, and Merik Meriste. 1995. Syntactic recognition of ECG signals by attributed finite automata. *Pattern Recognit.* 28, 12 (1995), 1927–1940. https://doi.org/10.1016/0031-3203(95)00052-6

[47] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504.

[48] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1001–1016.

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

[50] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. 1998. MALM: A Framework for Mining Sequence Database at Multiple Abstraction Levels. In *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, USA, November 3-7, 1998*, Georges Gardarin, James C. French, Niki Pissinou, Kia Makki, and Luc Bouganim (Eds.). ACM, 267–272.

[51] Kaiyu Li and Guoliang Li. 2018. Approximate query processing: What is new and where to go? *Data Science and Engineering* 3, 4 (2018), 379–397.

[52] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* (2022).

[53] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. 2020. Approximate Analytics System over Compressed Time Series with Tight Deterministic Error Guarantees. *Proc. VLDB Endow.* 13, 7 (2020), 1105–1118.

[54] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. 2–11.

[55] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed Bounded Floats for Fast Compression and Queries. *Proc. VLDB Endow.* 14, 11 (2021), 2586–2598. https://doi.org/10.14778/3476249.3476305

[56] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J Elmore. 2019. Mostly order preserving dictionaries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1214–1225.

[57] Rui Liu and Jun Yuan. 2019. Benchmark Time Series Database with IoTDB-Benchmark for IoT Scenarios. *CoRR* abs/1901.08304 (2019). arXiv:1901.08304 http://arxiv.org/abs/1901.08304

[58] Rui Liu and Jun Yuan. 2019. Benchmarking time series databases with IoTDB-benchmark for IoT scenarios. *arXiv preprint arXiv:1901.08304* (2019).

[59] Xiaoyan Liu, Zhenjiang Lin, and Huaiqing Wang. 2008. Novel Online Methods for Time Series Segmentation. *IEEE Trans. Knowl. Data Eng.* 20, 12 (2008), 1616–1626. https://doi.org/10.1109/TKDE.2008.29

[60] Yipeng Liu, Maarten De Vos, and Sabine Van Huffel. 2015. Compressed Sensing of Multichannel EEG Signals: The Simultaneous Cosparsity and Low-Rank Optimization. *IEEE Trans. Biomed. Eng.* 62, 8 (2015), 2055–2061. https://doi.org/10.1109/TBME.2015.2411672

[61] Jean loup Gailly and Mark Adler. 2003. The gzip home page. Website. https://www.gzip.org, Last accessed on 2022-10-05.

[62] Stephen Makonin, Bradley Ellert, Ivan V Bajić, and Fred Popowich. 2016. Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014. *Scientific data* 3, 1 (2016), 1–12.

[63] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. 2016. LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection. *CoRR* abs/1607.00148 (2016). arXiv:1607.00148 http://arxiv.org/abs/1607.00148

[64] Sidra Mehtab, Jaydip Sen, and Subhasis Dasgupta. 2020. Robust analysis of stock price time series using CNN and LSTM-based deep learning models. In *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE, 1481–1486.

[65] Inc. Meta Platforms. 2017. Zstandard - fast real-time compression algorithm. Website. https://facebook.github.io/zstd, Last accessed on 2022-10-05.

[66] ModelarData. 2023. ModelarDB: Model-Based Time Series Management from Edge to Cloud. Website. https://github.com/ModelarData/ModelarDB-RS, Last accessed on 2023-03-13.

[67] Jack Moffitt. 2001. Ogg Vorbis—open, free audio—set your media free. *Linux journal* 2001, 81es (2001), 9–es.

[68] S. Muthukrishnan, Rahul Shah, and Jeffrey Scott Vitter. 2004. Mining Deviants in Time Series Data Streams. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21-23 June 2004, Santorini Island, Greece*. IEEE Computer Society, 41–50. https://doi.org/10.1109/SSDBM.2004.51

[69] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[70] Irene CL Ng and Susan YL Wakenshaw. 2017. The Internet-of-Things: Review and research directions. *International Journal of Research in Marketing* 34, 1 (2017), 3–21.

[71] Frank Olken and Doron Rotem. 1986. Simple Random Sampling from Relational Databases. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 160–169.

[72] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[73] John Paparrizos. 2021. Rust implementation for BUFF: decomposed bounded floats for fast compression and queries. Website. https://github.com/johnpaparrizos/buff, Last accessed on 2023-03-11.

[74] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. 2021. VergeDB: A Database for IoT Analytics on Edge Devices.. In *CIDR*.

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

[76] Gregory Piatetsky-Shapiro and Charles Connell. 1984. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 256–276.

[77] Attila Reiss and Didier Stricker. 2012. Creating and benchmarking a new dataset for physical activity monitoring. In *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*. 1–8.

[78] Attila Reiss and Didier Stricker. 2012. Introducing a new benchmarked dataset for activity monitoring. In *2012 16th international symposium on wearable computers*. IEEE, 108–109.

[79] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. 2019. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3009–3017.

[80] Günter Schiepek and Guido Strunk. 2010. The identification of critical fluctuations and phase transitions in short term and coarse-grained time series—a method for the real-time monitoring of human change processes. *Biological cybernetics* 102, 3 (2010), 197–207.

[81] Zekâi Şen. 2012. Innovative trend analysis methodology. *Journal of Hydrologic Engineering* 17, 9 (2012), 1042–1046.

[82] Hagit Shatkay and Stanley B Zdonik. 1996. Approximate queries and representations for large data sequences. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 536–545.

[83] Elena Stefancova. 2018. *Evaluation of the timescaledb postgresql time series extension.* Technical Report.

[84] Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. 2020. *Delegation sketch*: a parallel design with support for fast and accurate concurrent operations. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 4:1–4:16.

[85] TDengine. 2022. TDengine Documentation. Website. https://docs.tdengine.com, Last accessed on 2022-10-05.

[86] KairosDB Team. 2022. KairosDB documentation v1.3.0. Website. https://kairosdb.github.io/docs/index.html, Last accessed on 2022-10-05.

[87] Mingyan Teng. 2010. Anomaly detection on time series. In *2010 IEEE International Conference on Progress in Informatics and Computing*, Vol. 1. IEEE, 603–608.

[88] Arvind Thiagarajan and Samuel Madden. 2008. Querying continuous functions in a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 791–804. https://doi.org/10.1145/1376616.1376696

[89] Inc Timescale. 2023. Time Series Benchmark Suite (TSBS). Website. https://github.com/timescale/tsbs, Last accessed on 2023-06-01.

[90] Inc Timescale. 2023. TimescaleDB vs. InfluxDB: Purpose Built Differently for Time-Series Data. Website. https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/, Last accessed on 2023-07-01.

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

[92] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[93] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-level data compression using machine learning in time series database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1333–1344.

[94] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[95] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

Received April 2023; revised July 2023; accepted August 2023