TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications

Abdelouahab Khelifati
University of Fribourg
Switzerland
abdelouahab.khelifati@unifr.ch

Mourad Khayati
University of Fribourg
Switzerland
mourad.khayati@unifr.ch

Anton Dignös
Free University of Bozen-Bolzano
Italy
dignoes@inf.unibz.it

Djellel Difallah
NYU Abu Dhabi
United Arab Emirates
djellel@nyu.edu

Philippe Cudré-Mauroux
University of Fribourg
Switzerland
pcm@unifr.ch

ABSTRACT

Time series databases are essential for the large-scale deployment of many critical industrial applications. In infrastructure monitoring, for instance, a database system should be able to process large amounts of sensor data in real-time, execute continuous queries, and handle complex analytical queries such as anomaly detection or forecasting. Several benchmarks have been proposed to evaluate and understand how existing systems and design choices handle specific use cases and workloads. Unfortunately, none of them fully covers the peculiar requirements of monitoring applications. Furthermore, they fall short of providing an automated way to generate representative real-world data and workloads for testing and evaluating these systems.

We present TSM-Bench, a benchmark tailored for time series database systems used in monitoring applications. Our key contributions consist of (1) representative queries that meet the requirements that we collected from a water monitoring use case, and (2) a new scalable data generator method based on Generative Adversarial Networks (GAN) and Locality Sensitive Hashing (LSH). We demonstrate, through an extensive set of experiments, how TSM-Bench provides a comprehensive evaluation of the performance of seven leading time series database systems while offering a detailed characterization of their capabilities and trade-offs.

PVLDB Reference Format:
Abdelouahab Khelifati, Mourad Khayati, Anton Dignös, Djellel Difallah, and Philippe Cudré-Mauroux. TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications. PVLDB, 16(11): 3363 - 3376, 2023.
doi:10.14778/3611479.3611532

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eXascaleInfolab/TSM-Bench.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611532

1 INTRODUCTION

Time series database systems (TSDBs) are specialized systems designed to store, manage, and query large volumes of time-series data fast and efficiently [29]. The growing need for real-time analytics in various applications such as financial analysis and the Internet of Things (IoT) has made TSDBs essential for stream processing and monitoring. The development of TSDBs has been driven by significant research efforts in devising novel methods to process, compress, and store time series data. Due to the diversity and sophistication of these systems, it is difficult for end users to pick the most suitable system for their particular use case.

Benchmarking provides a systematic way to understand the capabilities of TSDBs by comparing their performance using a dataset, a set of queries, and predefined workloads. Several TSDB benchmarks have been proposed in this context, but unfortunately, little is known about systems' trade-offs, as existing benchmarks focus on static queries and utilize a narrow set of real-world datasets and workloads [25, 28, 60, 66, 68]. Moreover, ingestion and querying are often evaluated in isolation from each other, which mischaracterizes the streaming requirements that accompany time series monitoring. Those issues make the results of existing evaluations difficult to generalize and motivate the need for a new comprehensive benchmark.

We argue that a TSDB benchmark should provide insights that validate the choice for a system deployment by offering at least three desiderata. First, the benchmark should implement a variety of performance metrics, such as query latency and throughput, data ingestion rate, and scalability. Second, it should be able to efficiently generate a large stream of data representative of real-world datasets. Third, it should help debug the root performance results and causes of bottlenecks in the system. We describe below a real-world application for such functionalities.

Motivating Example. Our work is motivated by the challenges we faced when evaluating and selecting systems for the monitoring of data from hydrometric stations in Swiss watercourses [13, 22, 31, 38, 41, 50]. After consultations with hydrologists and data analysts at the Federal Office for the Environment in Switzerland (FOEN)[1], we identified a set of requirements that must be covered when developing a benchmark for real-time monitoring systems to help practitioners analyze sensor data. These requirements include data

[1] https://www.bafu.admin.ch/bafu/en/home/topics/water.html

exploration [49, 72], anomaly detection [39, 52, 59], predictive modeling [18, 64], trend analysis [18, 42], missing value recovery [35–37], and data metrics comparison [16, 17, 51]. The outcomes of our study are applicable beyond our use case as they reflect the data management needs of a wider variety of monitoring applications such as the IoT [25, 32, 57], healthcare analytics [12, 20, 70], or power grid management [1, 56, 68].

In this paper, we introduce a new evaluation benchmark, which we call TSM-Bench, specifically designed for time series data used in monitoring applications. TSM-Bench packs several salient features into a comprehensive framework that includes: (1) A suite of fundamental queries found in common workloads, serving as building blocks for more complex time series analytical operations; (2) Various realistic workloads that support dynamic query variability; (3) Generation mechanism of large and multiple realistic time series based on seed data collected from real-world applications.

We apply TSM-Bench to perform a wide range of tests to evaluate the capabilities and trade-offs of seven leading time series database systems. Our experiments analyze the performance of these systems and provide extensive insights into their characteristics. Moreover, to promote reproducibility and future extensions, we follow the general guidelines of benchmark design, including portability, scalability, and simplicity [14, 24]. In summary, we make the following contributions:

- We introduce a comprehensive time series benchmark for monitoring applications that features temporal workloads, realistic data, and configurable query variability.
- We propose a novel and efficient data generation method that augments seed real-world time series datasets enabling realistic and scalable benchmarking.
- We perform an extensive analysis to evaluate the impact of synthetic data quality on time series benchmarking.
- We provide a detailed examination of seven TSDB systems, accompanied by practical recommendations for understanding and navigating their architectural designs.

The rest of this paper is organized as follows. Section 2 gives an overview of the related work and discusses the key differences. Section 3 introduces our benchmark application, its architecture, the new data generation method, and the benchmark queries. The experimental setup is detailed in Section 4 and the results are presented in Section 5. Section 6 provides a general discussion of the findings and guidelines for system selection and architecture considerations. Finally, we conclude in Section 7.

2 RELATED WORK

In this section, we present a review of existing time series database benchmarks and their associated data generators. Table 1 provides a comparative summary of existing benchmarks, highlighting how TSM-Bench complements the time series benchmarking landscape. The columns "use-case", "workload tier", and "query variation" characterize each benchmark by its application domain, the type of its loading process (bulk or batch), the evaluation of querying and ingestion (with 'offline' denoting separate query execution from insertion, and 'online' indicating concurrent operations), and the support of variation in the number of sensors and dynamic changes in predicate ranges. We describe each benchmark briefly below.

2.1 Time series benchmarks

SciTS [28] is a recent benchmark for TSDBs that focuses on monitoring time series recorded in scientific instrumentation. It implements queries to retrieve and sample data from multiple experimental runs and compares their query latency and data ingestion. The list of supported systems includes three TSDBs—ClickHouse, InfluxDB, and TimescaleDB—and one RDBMS—PostgresDB. SciTS evaluates the queries on randomly generated time series using a static number of sensors and time range. The queries are run in an offline manner where reads and writes are executed asynchronously.

In [68], Hao et al. present TS-Benchmark, a benchmark designed for monitoring electricity data collected from wind turbines. They evaluate fetching and aggregating from multiple sensors that produce anomalous time series data. The benchmark compares bulk loading, data ingestion, and query latency of four TSDBs—Druid, InfluxDB, TimescaleDB, and OpenTSDB. It generates synthetic time series that have akin properties to real-world electricity data. Similarly to SciTS, TS-Benchmark evaluates only the offline setup with static query parameters.

SmartBench [25] is a benchmark that focuses on smart buildings. The implemented queries retrieve various types of dependencies (e.g., correlation, co-evolution, etc.) across different sensors. It evaluates seven systems; only two are TSDBs—InfluxDB and GridDB—on seed time series contaminated with noise and duplicates. SmartBench evaluates insertion and query times and studies the impact of time ranges and hard disk types on the performance of the evaluated systems. Queries are executed both in an offline and online manner, computing query performance for the former and insertion throughput for the latter.

IoTDB-Benchmark [44] is a benchmark for heterogeneous IoT devices (different frequencies, out-of-order, etc.). It evaluates data retrieval queries, ingestion throughput, and resource usage. It compares four TSDB systems—InfluxDB, OpenTSDB, KairosDB, and TimescaleDB. The benchmark evaluates the queries in an offline setup by varying the number of sensors. The authors provide a tool that generates cyclic time series of different data distributions.

YCSB [11] is a collection of micro-benchmarks with a workload that contains various combinations of read/write operations (both random and sequential) and access distributions. Those workloads are run on distributed key-value storage systems. YCSB-TS [66] is a test suite that builds on top of YCSB by adding time functions to evaluate TSDBs. The implemented workloads consist only of simple aggregation queries.

Some commercial TSDB systems have conducted benchmarks for their products. InfluxDB-comparison [27] and its fork Time Series Benchmark Suite (TSBS) [60] compare several systems by computing storage size, loading performance, and aggregation runtime. ClickBench [10] is another benchmark tool proposed by ClickHouse that evaluates several systems optimized for time series, including TimescaleDB, InfluxDB, Druid, and MonetDB. The queries implemented by those benchmarks resort to simple selections and aggregations, which are applied on randomly generated data.

To our knowledge, none of the existing benchmarks evaluates TSDB systems using offline and online workloads with variable query parameters. Including those features in a new benchmark leads to more nuanced results and a realistic system evaluation.

Table 1: Comparison of existing benchmarks. ✓: Supported. (✓): Requires non-trivial extension. ✗: Not supported.

Benchmark	Use-case	Workload Tier			Query Variation		Data Generation	
		Bulk	Offline	Online	# Sensors	Time Range	Realistic	Scalable
Sci-TS [28]	instrumentation	(✓)	✓	✗	✗	✗	✗	✓
TS-Benchmark [68]	wind turbines	✓	✓	✗	✗	✗	✓	✗
SmartBench [25]	smart building	(✓)	✓	(✓)	✗	✓	✗	✓
IoTDB-Benchmark [44]	IoT	✗	✓	✗	✓	✗	✗	✓
TSM-Bench	hydrology	✓	✓	✓	✓	✓	✓	✓

2.2 Data Generation Methods

Some of the benchmarks we described above provide tools to generate large datasets. The "Data Generation" column in Table 1 summarizes the main features of data generation techniques associated with existing systems' benchmarks. It describes their ability to generate real-like time series and to efficiently process seed data.

The authors of TS-Benchmark propose the most similar data generator to ours. They introduce a graph-based model, which we refer to as TS-Graph, that uses Generative Adversarial Network (GAN) [19, 26, 53, 54] to generate long time series. The proposed method takes as input time series segments generated by GAN and constructs a graph, where the segments represent nodes and the edges indicate the transition probability between different nodes. A random walk is then performed on the constructed graph to generate new time series. The graph construction time is quadratic with the size of the segments making the processing of large seed time series time-consuming. We show in Section 5 that our generation outperforms TS-Graph not only in efficiency but also in quality.

IoTAbench [5] introduces a data generation model that uses a Hidden Markov Model (HMM) to generate synthetic time series data that mimics device power consumption. The method assumes that the data follows the Markov property, which is not representative of real-world time series complexity. SmartBench introduces another data generation tool that augments the data duration and frequency by adding noise to the original data. The produced data has a low variability and resembles the input data, which leads to the same limitations as using a random data generator.

In addition to those generators, various categories of standalone time series generation techniques exist. Decomposition methods [6, 23, 34, 62] extract underlying patterns from the dataset, such as trend or independent components, and adapt them to generate new patterns. Those methods can only augment the number of time series. Model-based augmentation methods [2, 7, 33, 69] build a statistical model of the real data and then use it to generate new time series. Methods from this category are typically used for forecasting and can only augment the length of the series. Time-domain methods [21, 63] transform the original time series using simple techniques such as adding noise or advanced ones such as computing a weighted average of time series [46]. Such techniques require highly correlated time series, which is not representative of many real-world datasets.

We seek a new generation technique able to efficiently augment both the length and the number of time series without mischaracterizing the properties of the data.

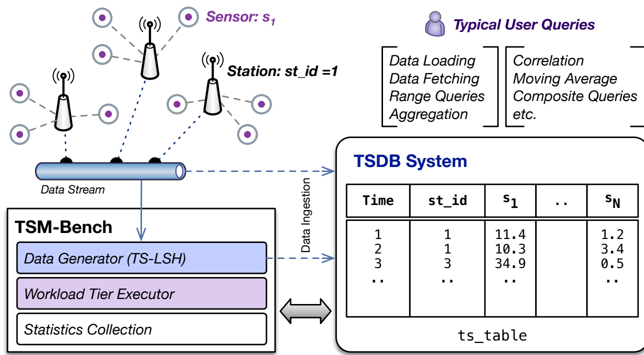
3 THE TSM-BENCH BENCHMARK

As mentioned earlier, our work is motivated by the challenges we encountered when evaluating and selecting systems for monitoring data from hydrometric stations in Swiss watercourses. Stations for water surfaces, such as rivers and lakes, are equipped with telemetric sensors that measure a wide range of water metrics. Each station collects its sensor records and transmits them to a database system where data is ingested and made available for analytical queries. Stations can incorporate up to hundreds of sensors that portray different water metrics and locations within the station. After consultations with hydrologists and data scientists, we identified a number of recurrent requirements related to hydrometric time series data monitoring:

- **(R1) Data exploration.** Basic data analysis involves fetching data within a certain time range from certain sensors and stations and exploring the result, for example, by visualizing it on a dashboard.
- **(R2) Anomaly detection.** Identifying sensor anomalies by fetching readings that exceed a specified threshold is a routine operation done during data exploration.
- **(R3) Prediction models.** Hydrologists are interested in forecasting data behaviors by computing statistical metrics and using them in building predictive models.
- **(R4) Data trends analysis.** Hydrologists also need to analyze data trends over extended time periods, which requires downsampling the data while preserving the most salient properties of the time series.
- **(R5) Recovery of missing values.** Failures in power, communication, or interference in sensors can cause missing values in the collected data. For many tasks, such as the recovery of missing values, it is necessary to first fill the missing values with linear interpolation.
- **(R6) Metrics comparison.** Hydrologists often need to compare data from various sensors. A frequently used method is to compute similarity metrics between time series.

3.1 Architecture

The architecture of the TSM-Bench is illustrated in Figure 1. Our data generation module, TS-LSH, uses sample data to generate a large realistic data stream that is fed into the target TSDB before or during the evaluation. We discuss in detail our generator in the next section. The benchmark Executor launches configurable workload tiers with queries such as data loading, data fetching, and complex analytical queries. The statistics collection module records the performance of the TSDB and a variety of system metrics.



Time	st_id	s_1	..	s_N
1	1	11.4		1.2
2	1	10.3		3.4
3	3	34.9		0.5
..	..	..		..

Figure 1: The Architecture of TSM-Bench.

Our framework adopts a conventional time series format where each station has a single entry for each timestamp including values for multiple sensors from the station. Figure 1 shows an example of such a wide-format data model, where each column of `ts_table` represents a sensor from a given station.

3.2 TS-LSH Generation Technique

A comprehensive benchmark requires access to a large amount of data. Unfortunately, existing real-world time series are often limited in size and/or number. Synthetic data can be a viable option but can lead to biased benchmark results when the properties of the original and synthetic data are dissimilar. Moreover, synthetic data should be generated efficiently because the benchmark needs high throughput data to simulate realistic scenarios.

We introduce TS-LSH, a new scalable data generator that closely emulates the properties of real-world time series. One of the benefits of this tool is to facilitate data sharing for benchmarking tasks, particularly when datasets are non-public due to privacy issues. Our method relies on Generative Adversarial Network (GAN) to create large volumes of time series data. GAN takes an input time series, partitioned into segments of the same length, and generates new segments that look real. It does so by playing an adversarial game: a generator creates new segments to fool a discriminator and the latter tries to distinguish between the real segments and fake ones [61]. At the end of the GAN training phase, the generator produces segments akin to the real ones, which cannot be distinguished by the discriminator.

Given a time series partitioned into segments $X = \{s_1, s_2, \ldots, s_n\}$ and a random noise $z \in \mathbb{R}^z$, the generator produces fake segments $G(z)$, while the discriminator $D(x)$ determines whether $G(z)$ is real or not. The adversarial game G and D play is expressed as a min max function: $\min_G \max_D V(G,D) = \mathbb{E}_{s \in p_{seg}(s)}[\log D(s)] + \mathbb{E}_{z \in p_z(z)}[log(1 - D(G(z))]$, where p_{seg} is the distribution of the real segments transformed from our original time series and p_z is the distribution of the input noise z. In other words, G tries to minimize the distance between the generated segments and the real ones while D tries to maximize it.

One possible way to create larger time series is to increase the length of the GAN's produced segments. This is, however, prohibitive in time as GAN consists of two deep neural networks each with an exponential complexity with the size of the segments [45]. Instead, we propose to concatenate the generated segments using the original data as a reference. This allows us to capture the *global* properties of the data such as trends or order without the need to change GAN's internals. We adopt Locality Sensitive Hashing (LSH) to identify the synthetic segments that are similar to the original ones and concatenate them to produce new series. Figure 2 depicts the generation pipeline, where the input of the process consists of real time series collected from multiple stations.

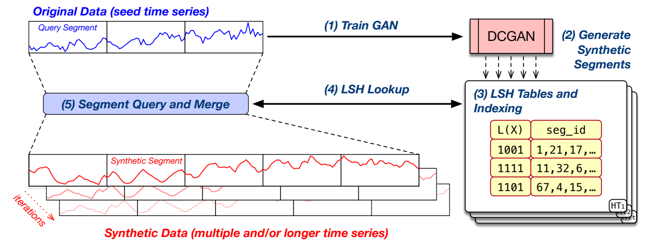


Figure 2: Time Series Generation using TS-LSH.

(Step 1 and 2) Train the GAN and Generate Segments. In TS-LSH we use DCGAN [55], a deep convolutional generative adversarial model frequently used for time series augmentation [30, 71]. We train a new GAN model for a given seed dataset by splitting the original time series into short and overlapping segments used by the discriminator. The length of seed segments and the window shift values are commonly set to 3k and 10, respectively [55]. Once the adversarial training is finished, the generator is able to produce new synthetic time series segments with the properties of real data. The obtained model is then used to generate a high number of synthetic seed segments that capture the local data properties.

(Step 3) LSH Tables and Indexing. We build LSH hash tables from the generated segments to locate synthetic candidates similar to the original data. We aim to achieve a high similarity between the generated segments and the original ones while making room for novelty in the generation. The number of hash tables can impact this goal. We empirically find that using 10 hash tables provides an order of magnitude speedup with a marginal loss in accuracy compared to using a larger number of tables [58]. Another aspect that impacts the indexing result is the similarity measure. There are a number of distance metrics for time series [15] including Euclidean Distance (ED) and its variants [8, 47], Dynamic Time Warping (DTW) [9, 67], Longest Common Subsequence (LCSS) [67]. Using the Euclidean Distance as a similarity measure between segment codes is sufficient for our application, as it can easily handle our highly correlated and generally time-aligned hydrological time series, where similar segments are in phase.

In our example, synthetic segment 21 is mapped to code 1001 in the first hash table. Thanks to the hashing function, similar segments will have a high probability of sharing the same hash bucket. For example, segments with ids 17 and 1 will be stored in the same bucket as segment 21 since they share the same code.

(Step 4) LSH Lookup. The generation process starts by selecting synthetic segments that (when combined) have the same properties as the original data. This helps preserve the global trends in the time series by sequentially extracting the codes of the original segments for each hash table. Next, these codes are used to look up similar synthetic segments in the corresponding hash tables. For each hash table, only the bucket corresponding to the segment's code is accessed, which significantly reduces the lookup time.

(Step 5) Segment Query and Merge. Each lookup returns multiple synthetic segments as candidates to represent the original segment. In our example, segments 1, 21, and 17 are candidates to represent the original segment with the code 1001. One of these segments is randomly selected and appended to the generated time series. Selected segments are removed from the LSH tables to avoid using identical segments repeatedly in the generation. Following TS-Graph [68], once the next segment is selected, a fitting function is used to smoothly append it to the generated data. The fitting function is applied at the end of the last segment and the start of the selected segment, resulting in a smooth transition.

The generation process continues by iteratively performing the querying process. When half of the synthetic segments are used or a particular entry of the hash tables is depleted, we execute an update that reconstructs the hash tables using new GAN-generated segments. We show that this iterative generation is efficient. Specifically, we prove that it is sub-linear with the size of the input series.

LEMMA 3.1. *Let D be a dataset of m datapoints. The complexity of TS-LSH to augment D to a larger size n is $O(m^\rho \cdot n)$ where $\rho < 1$.*

PROOF. The runtime complexity of TS-LSH is dominated by the construction and lookup times of the hash tables. LSH construction is performed on h hash tables using $n_s = \frac{m}{s}$ synthetic segments, where s is the length of the shifting window used to create segments. It follows that LSH construction has the cost of $O(h \cdot n_s)$.

TS-LSH uses the similarity between series to determine which segments are stored in the same bucket. LSH query time depends on the probability of two segments being hashed to the same bucket, i.e., $\rho = \log p1 / \log p2$, where p_1 and p_2 refer to the lower bound and the upper bound probabilities of the hashing tables, respectively [3]. The cost for querying or deleting one segment in h hash tables containing n_s segments each is $O(h \cdot n_s^\rho)$ [4]. The iterative process of hash tables construction and querying is performed $\frac{n}{n_s}$ times. The total complexity is therefore $O((h \cdot n_s + 2 \cdot h \cdot n_s^\rho \cdot n_s) \cdot \frac{n}{n_s}) = O(h \cdot n \cdot n_s^\rho)$. By substituting n_s with $\frac{m}{s}$ and using the fact that h and s are two small constants, we obtain a total time complexity of $O(m^\rho \cdot n)$. □

3.3 TSM-Bench Queries

Our benchmark focuses on fundamental queries, typically found in common workloads, which underpin most of the complex time series analytics operations. Those queries allow us to isolate single performance dimensions, such as the impact of input and output size, data access, and the number of operations.

Since most systems support a pseudo-SQL-like language, we use SQL to describe our queries. However, the query implementation may vary from one system to another. Our queries are executed on a list of stations `<st_list>`, a list of sensors `<s_list>`, and in a time range `[?timestamp - ?range, ?timestamp]`.

Q1: Data Fetching. This query selects intervals of time series data from given sensors and stations. It is typically used to compare multiple series through data exploration (**R1**) and aims to evaluate the performance of data access and output.

```
SELECT time, st_id, <s_list>                              Q1
FROM ts_table
WHERE st_id in <st_list>
AND time < ?timestamp
AND time >= ?timestamp - ?range;
```

Q2: Data Fetching with Filter. This query selects from a sensor the values that exceed a threshold within a time interval. In our use case, hydrologists commonly use it to detect anomalous values (**R2**). The filter condition is applied to one sensor, and the condition value is sampled from the data distribution to output 5% from the sensor. This number represents the percentage of out-of-bound values for the data. Q2 evaluates the systems' efficiency for filtering and outputting the results.

```
SELECT time, st_id, <s_list>                              Q2
FROM ts_table
WHERE st_id in <st_list>
AND time < ?timestamp
AND time >= ?timestamp - ?range
AND s_k > ?value;   /* s_k in <s_list> */
```

Q3: Data Aggregation. This query calculates the average value recorded by some sensors from multiple stations and groups them by station. The aggregation operation is often applied for computing statistical values used to build hydrological prediction models (**R3**). Q3 returns very small-sized results and aims to evaluate data access and aggregation computation.

```
SELECT st_id, AVG(s_i)...AVG(s_j)                         Q3
FROM ts_table
WHERE st_id in <st_list>
AND time < ?timestamp
AND time >= ?timestamp - ?range
GROUP BY st_id;
```

Q4: Downsampling. The downsampling operation reduces the granularity of time series by executing multiple aggregations along the time dimension and using the results to create a new time series with a lower frequency. This operation is frequently used to reduce the size of the data while preserving its main trends (**R4**). For example, Q4 can downsample the values of sensors and stations from 10 seconds to a 1-hour granularity. It evaluates the systems' performance to handle window operations.

```
SELECT time, st_id, AVG(s_i)...AVG(s_j)                   Q4
FROM ts_table
WHERE st_id in (<st_list>)
AND time < ?timestamp
AND time >= ?timestamp - ?range
GROUP BY st_id, time
SAMPLE BY 1H;
```

Q5: Upsampling. This operation increases the frequency of time series. This is done by generating new timestamps and using a replacement strategy (e.g., interpolation, previous value, etc.) to construct time-aligned data with a higher frequency. Hydrologists apply such an operation to recover missing values in their series (**R5**). Note that this query is not supported by some systems, such as Druid and MonetDB, and that eXtremeDB does not support interpolation, instead, it fills the surrogate values with zeros.

```
SELECT time, st_id, <s_list>                    Q5
FROM ts_table
WHERE st_id in <st_list>
AND time < ?timestamp
AND time >= ?timestamp - ?range
SAMPLE BY 5s
FILL(LINEAR);
```

In addition to the five simple temporal queries, we identify two advanced analytical queries that combine basic operations from the previous workload. The two queries operate on multiple sensors at a time. The first one returns large-sized data, while the second returns small-sized results.

Q6: Cross Average. This query computes the average values of sensors within a station (**R6**). The output of this query has the same size as the original time series.

```
SELECT time, s_i, s_j, AVG(s_i, s_j)            Q6
FROM ts_table
WHERE st_id = st_k
AND time < ?timestamp
AND time >= ?timestamp - ?range;
```

Q7: Correlation. This query computes the Pearson correlation [43] between two time series (**R6**). We use a built-in primitive for ClickHouse, eXtremeDB, and TimescaleDB and combine the `SUM()` and `COUNT()` operators for the remaining systems. The output of this query is a scalar.

```
SELECT CORR(s_i, s_j)                           Q7
FROM ts_table
WHERE st_id = st_k
AND time < ?timestamp
AND time >= ?timestamp - ?range;
```

We also considered additional queries with a higher complexity as well as User Defined Functions (UDFs). Our results [40] show that the behavior of the best-performing systems on those queries showed similar trends to the ones we include in the paper.

3.4 TSM-Bench Workload Tiers

We propose three benchmark tiers to effectively evaluate the ability of a database system to fulfill the requirements of our water-sensing use case. These tiers have been designed with the intention of assessing query complexity, data volume, and scalability. Additional tiers can be easily added to TSM-Bench to assess alternative scenarios such as real-time stream processing.

Bulk-Loading Workload is intended to evaluate the efficiency of data loading into the system, which is done separately from query processing. This involves taking a large amount of historical data and inserting the data into the system in batch mode.

Offline Workload consists of queries that run on historical data without concurrent insertions. It evaluates two types of queries, basic queries [Q1-Q5] that use built-in operators, and complex queries [Q6-Q7] that combine these operators to perform a time series downstream task.

Online Workload consists of concurrent workloads for both insertion and querying. This involves continuously streaming data into the system, and running simple queries [Q1-Q5] on both historical and new data.

4 EXPERIMENTAL SETUP

4.1 Setup Notes

The machines we used in the following have an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (32cores), 128GB of memory, 4TB TOSHIBA Hard disk operating under Ubuntu 20.04 LTS and connected over a 10 Gbit/sec Ethernet switch. Each system is deployed on one dedicated machine. We use a separate client machine to launch the workloads and collect the results from each system. We use the following TSDB versions: ClickHouse v22.6.1, Druid v0.24.1, eXtremeDB v8.2, InfluxDB v1.7.10-1, MonetDB v11.43.9, QuestDB v6.2.1 and TimescaleDB v2.6.0. The default configuration was used for each system. We consulted with systems support teams in cases where a system's behavior was unexpected such as poor resource utilization or inefficient data ingestion/query performance. Our data generator is built using Python 3.10 and PyTorch 2.0.

4.2 Datasets

To evaluate the systems on large datasets, we apply TS-LSH to generate two new datasets, D-LONG and D-MULTI, that include a small number of long time series and a large number of short time series, respectively. We use as seed a real-world dataset of water temperature and level time series recorded using multiple hydrometric sensors. Temperature series contain duplicates and similar consecutive values while level series are erratic and contain abrupt changes. We augment the seed dataset by scaling i) the number of time series—we increase the number of stations and/or sensors—and ii) the length of time series—we increase their duration. Table 2 summarizes the main properties of the two datasets.

Table 2: Datasets used in TSM-Bench. The sensors have a 10-second frequency.

Dataset	# Stations	# Sensors	Range	# Datapoints
Original	1	10	5 days	430K
D-LONG	10	100	60 days	518M
D-MULTI	2000	100	10 days	17.2B

4.3 Evaluated TSDB Systems

The selection of the evaluated systems is based on the following criteria: (a) their popularity, (b) their performance in other benchmarks, (c) the support of the necessary operators to implement the queries, and (d) the results of pre-evaluation experiments.

The first step of the selection process consists in picking the top ten TSDBs based on their popularity according to DB-Engines and OSS Insight and their performance in previous benchmarks. We conduct preliminary experiments on the pre-selected systems using a subset of queries with fixed parameters. Our results led to the selection of four systems as top competitors—InfluxDB, TimescaleDB, Druid, and QuestDB. In addition to the popularity criterion, our goal was to strike a good balance between the systems evaluated in other benchmarks and the differences in the underlying architecture. For this reason, we complemented the initial list with three additional systems—eXtremeDB, ClickHouse, and MonetDB.

ClickHouse is an open-source column-store DBMS that, while not specialized for time series, supports temporal analytics. It uses a MergeTree as its engine, which stores series separately in parts indexed and sorted by a sparse primary key based on time. When data is queried, the relevant ranges are located from the primary key and the offset files. During insertions, new parts are created from the incoming records after being sorted by the primary key. The new parts are periodically selected and merged by the MergeTree. ClickHouse supports vectorized query execution by processing data as columns, which allows SIMD CPU instructions usage.

Druid is an open-source OLAP DBMS that stores the data by time natively. Each record in Druid consists of three column types: timestamp, metrics, and additional attributes to the data records called dimensions. Druid partitions the data by time and supports additional partitioning based on other dimensions. It uses bitmap indexes to perform filtering and searching across multiple columns. During query execution, Druid first identifies which rows to select, decompresses them, and pulls out the relevant rows.

eXtremeDB is a commercial TSDB that supports hybrid tables where each table can have vertical and horizontal fields. It implements a new data type for time series stored vertically as *sequences*. The remaining data types (e.g., float, string, etc.) are kept in a traditional horizontal (n-ary) fashion. The sequences type allows eXtremeDB to access the data using the indices of the start and end of the queried series. eXtremeDB includes additional operations such as user-defined indexes and collations that can further optimize access to vertical data.

InfluxDB is an open-source column-store TSDB where each datapoint consists of a timestamp, a value, and one or multiple tags. The latter are key-value pairs used to add data to the record. InfluxDB does not require building a schema before ingesting data. The schema is instead inferred from the tags and fields following the data ingestion. InfluxDB uses a Time-Structured Merge Tree (TSM Tree) [48] as a storage format. It indexes the data using timestamps and tags and it organizes time series into shards when storing data on disk. A shard contains encoded and compressed time series data for a given time range.

MonetDB is an open-source column-store relational DBMS that, albeit not specialized for time series, is optimized for temporal analytics. It stores each time series as a separate table called BAT (Binary Association Table). During query processing, MonetDB generates a logical plan, expressed in the MonetDB Assembly Language (MAL), then optimizes it through various modules before execution. MonetDB supports manual partitioning of time series data horizontally by time using a merge table.

QuestDB is an open-source column-store TSDB. It stores data by time natively and partitions the data by intervals of time, where data for each interval is stored in separate sets of files. Columns in QuestDB are append-only, allowing insertions in the most recent partition only. QuestDB supports SIMD instructions to perform multiple filtering and aggregation operations simultaneously.

TimescaleDB is an open-source row-store TSDB built on top of PostgreSQL. It stores time series data using hypertables that partition the data by time into several chunks while making it similarly accessible as a PostgreSQL table. TimescaleDB supports a compact storage format that groups multiple records into a single array. During query execution, it selects chunks containing targeted data based on the query's time range. During insertion, TimescaleDB compresses each inserted row before storing it within its respective chunk. Periodically, it recompresses the new chunks, combining the newly inserted rows with the previously compressed ones in a more compact format.

5 EXPERIMENTAL EVALUATION

Before reporting the results of our experiments, we first describe how we calibrate the systems. We then delve into our three workloads, and lastly, evaluate the performance of our data generation and show its impact on the systems.

To reduce disk IOs, computations, and seek times, TSDBs use partitioning by time, which physically separates data files. The partition time range is a critical parameter that has to be manually set. Its optimal value depends on many factors, including data cardinality, data granularity, and typical queries. We evaluate the impact of partitioning by varying the partition time range (Hour, Day, Week, and Month) for each system and comparing the query runtime using our two datasets.

Our results (described as a technical report in [40]) show that partitioning by week represents the optimal choice. This is the smallest partition size that would invoke, in our setting, single partition access for these systems minimizing both inter-partition and intra-partition costs. We also found that ClickHouse is the only system that is not noticeably impacted by the variation of the partition size. Thanks to its sparse indexing, ClickHouse can directly access the data within a partition, even for large partitions.

5.1 Data Bulk Loading

We evaluate systems performance using their bulk loading utility to import datasets files. Table 3 compares the performance in terms of throughput and storage size. The reported loading times consist of the time to ingest the data and to make it available to queries. Note that we do not report the performance of InfluxDB on D-MULTI as its loading takes a significant amount of time (i.e., over a couple of days). Unlike the other systems, InfluxDB does not have a bulk-loading utility.

The results in Table 3 show that ClickHouse achieves the best throughput on both datasets reaching 18M data points per second in the first dataset and 11.9M in the second one. ClickHouse copies data in bulk upon insertion and then performs merging and sorting using its MergeTree. Moreover, it uses the maximum number of available CPU cores, which further improves the loading rates. MonetDB achieves the second-best throughput reaching 7.5M and

Table 3: Loading and compression performance. Throughput the higher the better and storage the lower the better.

Dataset	System	Avg. Throughput (datapoints/s)	Loading Time (s)	Storage (GB)
D-LONG	ClickHouse	**18'386'867**	28	1.97
	Druid	903'135	574	3.52
	eXtremeDB	2'368'378	218	4.10
	InfluxDB	692'122	749	3.50
	MonetDB	7'526'460	68	4.00
	QuestDB	2'938'892	176	4.00
	TimescaleDB	2'627'138	197	4.31
D-MULTI	ClickHouse	**11'933'701**	1'448	65.50
	Druid	616'702	28'020	118.81
	eXtremeDB	2'415'877	7'152	138.74
	InfluxDB	-	-	-
	MonetDB	5'760'000	3'000	137.00
	QuestDB	1'920'000	9'000	132.00
	TimescaleDB	1'749'976	9'874	134.00

5.7M data points for the two datasets. The low loading times of both systems confirm their superiority.

The results also show that QuestDB, TimescaleDB and eXtremeDB achieve comparable throughputs of over 2M data points per second, despite having different loading mechanisms. For instance, QuestDB maps column files into a memory page and performs a column append as a memory write. Once the memory page is exhausted, it is unmapped and a new page is used. TimescaleDB, on the other hand, uses its loading tool (timescaledb-parallel-copy) to map the dataset file directly into a table. This loading tool uses more resources by importing data in parallel. We notice that eXtremeDB achieves similar throughputs for the two datasets.

Finally, the results in Table 3 show that ClickHouse outperforms all the systems in terms of compression. It implements both time series and general-purpose compression schemes and uses sparse indexing, storing one index entry for each part (a partition in MergeTree engine). Druid and InfluxDB implement similar compression schemes but their indexes use one entry per row, which yields additional disk storage consumption. Row-based systems, such as TimescaleDB, achieve lower compression ratios because they cannot leverage the similarity between values within each column.

5.2 Offline Workload

In this section, we evaluate the systems' performance on our two datasets and report the average query execution time (in milliseconds). For each query, we create 100 query instances with randomly generated parameters. We run several random queries before evaluation to guarantee that they are not executed on a cold cache.

5.2.1 Evaluation with Long Series. We begin evaluating the systems for queries on D-LONG by varying the number of stations/sensors and the time interval for each query. When we vary each dimension, the remaining ones are set to their default values, i.e., one station, three sensors, and one day.

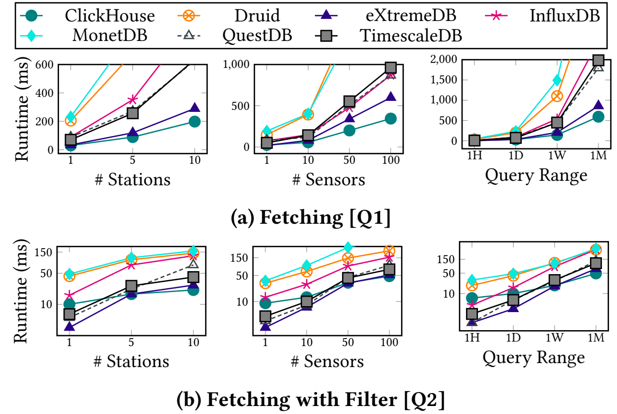


Figure 3: Runtime of Queries Q1 and Q2 on D-LONG.

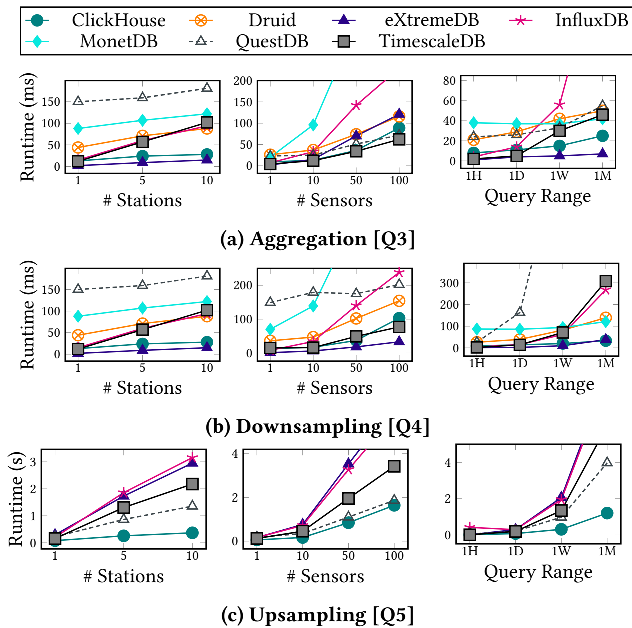
Q1: Data Fetching. The results in Figure 3a show that ClickHouse is the best contender in most cases. It achieves high efficiency by grouping column values into granules (a set of rows) that are indexed and read jointly. This means that instead of reading individual rows, ClickHouse outputs granules, making it particularly efficient for high selectivity range queries. The results also show that eXtremeDB handles very well queries with low selectivity that involve only one sensor or cover one hour range. Thanks to its vertical sequences format, fetching queries in eXtremeDB returns the first and the last elements' positions (index) of the selected subsequence, which is then materialized into memory, making it highly efficient for short subsequences.

TimescaleDB and QuestDB achieve comparable runtimes in most cases. The former uses hypertables that partition the data by the time dimension into several chunks where each chunk is a PostgreSQL table. Indexing the table by chunks that can fit in memory reduces query runtimes. QuestDB, on the other hand, partitions the data by intervals of time, where data from each interval is stored in separate sets of files, which reduces disk reads for range queries.

Q2: Data Fetching with Filter. The results in Figure 3b show that eXtremeDB achieves the fastest execution time for most configurations. Filtering queries are executed in eXtremeDB by traversing a B-tree without materializing the full sequence into memory, significantly improving the query execution time. In queries with high selectivity, ClickHouse achieves the fastest query execution as it processes data within partitions in parallel.

We can observe that QuestDB is the second-fastest system in multiple configurations. QuestDB supports SIMD instructions that allow executing a filter query for multiple rows simultaneously. This allows QuestDB to reach a high level of parallelization. Also, InfluxDB is noticeably fast for small-range queries. InfluxDB's TSM index uses binary search to find data blocks, making InfluxDB fast for short-range queries where a few blocks are retrieved.

Q3: Data Aggregation. The results in Figure 4a show that the fastest runtime for Q3 is achieved by eXtremeDB and TimescaleDB, respectively. Both systems store data values by grouping them into a single array, making aggregations possible in a single read. TimescaleDB is the fastest for queries with a high number of sensors



(a) Aggregation [Q3]

(b) Downsampling [Q4]

(c) Upsampling [Q5]

Figure 4: Runtime of Queries Q3, Q4 and Q5 on D-LONG.

while being less efficient in high-range queries because it has to access multiple chunks.

We notice that, unlike the two previous queries, TimescaleDB and QuestDB show different results. This is because QuestDB scans data rows to perform the aggregation, whereas TimescaleDB performs one read per sensor to compute the aggregation due to its array-like format. We notice also that MonetDB shows similar runtime for short and long queries. MonetDB's BAT representation makes it better suited for aggregate operations on a few columns, even for long time series.

Q4: Downsampling. Figure 4b shows the query runtimes for Q4. Similarly to Q3, eXtremeDB achieves the fastest runtime for most query parameters. eXtremeDB performs downsampling by manipulating the timestamps at an hourly granularity and aggregating the values in each window. Because of its array format, eXtremeDB is able to efficiently perform multiple aggregations for each sensor.

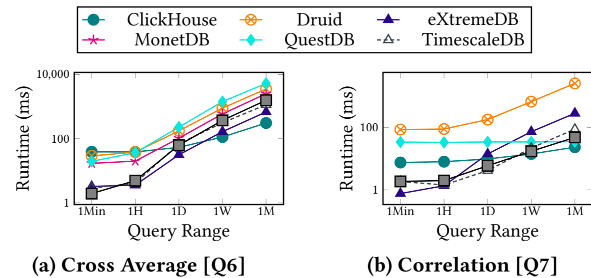
ClickHouse provides the second-best performance for most configurations. It creates a new time column with the reduced frequency by manipulating the timestamp and performing the aggregation for each window. Because aggregations are performed on neighboring rows, ClickHouse can efficiently downsample large data ranges. MonetDB and Druid show a reasonable performance for all configurations as the output of this query is relatively small.

Q5: Upsampling. Figure 4c shows the query runtimes for Q5. ClickHouse and QuestDB achieve the best runtimes for most parameters. Both systems perform upsampling by creating new timestamps matching the higher granularity series and then interpolating in parallel sensor values. TimescaleDB achieves the best runtime for short-range queries but is less efficient for higher ranges because more data is sequentially processed.

eXtremeDB and InfluxDB achieve a decent performance in highly selective queries, but their efficiency significantly decreases for lowly selective queries. eXtremeDB performs upsampling by stretching the timestamps into a new array matching the higher granularity. It copies elements from the original series into the new ones by matching their timestamps before filling the intermediary values. InfluxDB shows a reduced performance for queries with high selectivity because more blocks need to be processed.

Q6: Cross Average. Figure 5a shows that eXtremeDB and TimescaleDB provide the best runtime results for short window ranges while eXtremeDB and ClickHouse provide the best results for query ranges over 1 day. As expected, this query shows similar performance trends to data fetching in Q1. This shows that data output impacts the runtime more than executing the operations.

Q7: Correlation. Note that InfluxDB is not evaluated for this query as it does not support advanced combinations of aggregate functions. Figure 5b shows that eXtremeDB is the fastest for short ranges up to 1 hour starting from which QuestDB and TimescaleDB are the fastest. ClickHouse and MonetDB are the fastest for high-range queries thanks to their parallel computation ability. eXtremeDB shows a slowdown for higher ranges because of the increasing query cost for long sequences. This query shows that the performance of the systems heavily depends on the query's computational cost, even for small outputs.



(a) Cross Average [Q6]

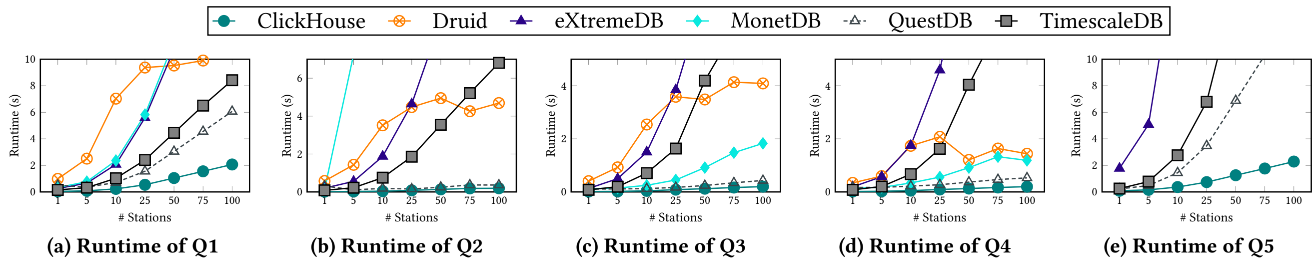
(b) Correlation [Q7]

Figure 5: Runtime of Queries Q6 and Q7 on D-LONG.

5.2.2 Evaluation with a High Number of Time Series. In this experiment, we evaluate the systems' performance using the second dataset D-MULTI. To assess the impact of the number of time series, we vary, for each query, the number of stations and keep the number of sensors and range to their default values. Figure 6 depicts the results for all queries but Q6 and Q7, which are performed on one station only. We do not report the numbers for InfluxDB in D-MULTI for the same reasons as above.

We observe that ClickHouse and QuestDB become the fastest for this dataset. Both systems process multiple time series simultaneously, with each series represented as a vector. This results in fast query runtimes, even for high-selectivity queries (i.e., 100 stations). In queries with large output, such as Q1 or Q5, the two systems behave in different ways. Similarly to the other dataset, ClickHouse is efficient for large output queries. QuestDB, on the other hand, shows a slowdown due to the additional data outputting overhead.

The results also show unexpected trends. For example, the runtime of Druid plateaus when the number of stations increases. This



(a) Runtime of Q1 (b) Runtime of Q2 (c) Runtime of Q3 (d) Runtime of Q4 (e) Runtime of Q5

Figure 6: Runtime of Queries Q1-Q5 on D-MULTI.

is visible for a number of stations above 10. Druid processes the queried data segments in parallel, generates partial results for each segment, and then merges the partial results into the final result set. Such parallelization explains the good performance of Druid when querying multiple stations.

MonetDB shows another interesting trend. It performs well for highly selective queries involving fewer than ten stations and as the output increases, we notice a substantial decrease in its performance. During query execution, MonetDB requires that the inputs and outputs of each single MAL operation fit in memory. Subsequently, queries involving a high number of time series are processed sequentially, leading to a slowdown.

Unlike the other systems, eXtremeDB and TimescaleDB show a slowdown both in data output and data access. eXtremeDB performs a join on its hybrid table and then sequentially loads the time series into memory causing high runtimes for a large number of series, even for highly selective queries. In TimescaleDB, the time-partitioned chunks become significantly large for this dataset as each chunk contains data from all the stations. This slows down its loading and query execution times.

5.3 Online Workloads

In the previous set of experiments, we assumed an offline setup where queries are executed separately from data ingestion. In monitoring applications, however, data is often received in an online manner, making queries and insertions happen simultaneously. We ran an experiment to compare systems performance on online queries. Before reporting the online results, we describe how we set the window size during continuous insertion. We do so by evaluating the impact of the window size on systems insertion latency. Figure 7 depicts the results where each box plot describes the insertion time distribution (i.e., median, quartiles, and outliers).

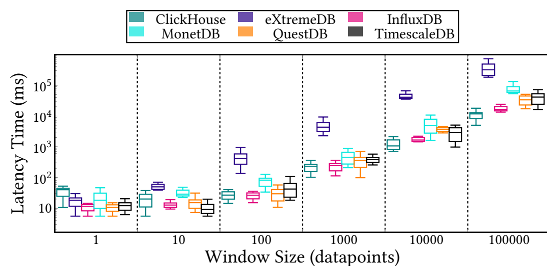


Figure 7: Insertion Latency by Varying Window Size.

The online results reveal two key trends. For the *point-by-point* insertion, all systems can properly handle streaming. eXtremeDB shows the best latency, but the performance gap between all the systems is marginal. We also notice that as the window gets bigger, ClickHouse becomes the fastest because it uses MergeTree to write directly to storage. We set the maximum window size to 10K rows to ensure a reasonable insertion latency ratio for all systems.

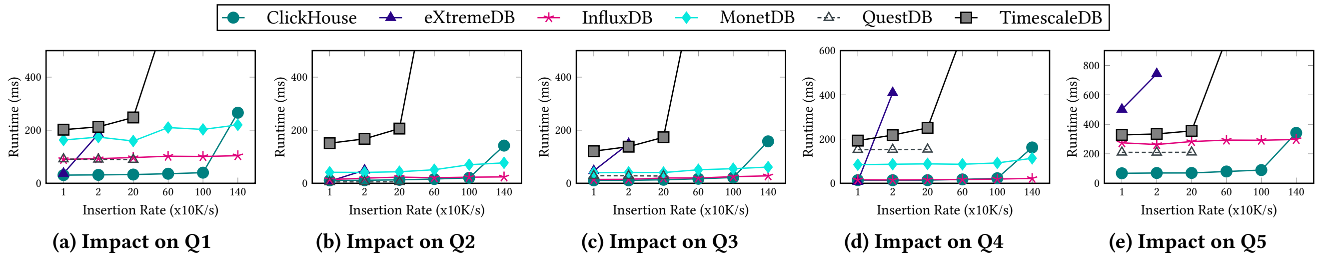
We now move to describe the online experiment. We first load two months of historical data and then vary the insertion rate between 10K and 1.4M data points per second. The queries are run in parallel on recent data after ensuring the insertion rates are properly reached. We report the results only for the D-LONG dataset, as it takes considerable time to produce the results for D-MULTI. Note that Druid does not support short frequent insertions. Also, since QuestDB does not support multi-connection insertions, we insert the whole rate using one connection. Figure 8 reports the runtime by varying the insertion rates.

The results show that queries do not block writes for all systems. For slow insertion rates, B-tree-based systems such as QuestDB provide the best runtimes. They also show that slow insertion rates have a significant impact on array-based systems. In eXtremeDB, for instance, the insertion is done by appending new elements to the end of sequences, which incurs a query slowdown and instability in the runtime of queries. In TimescaleDB, the slowdown is due to the newly inserted data remaining uncompressed. As such, the system cannot benefit from IO optimization of its compressed format as it needs to load the uncompressed data from the disk. We observe similar trends for the medium insertion rates for most systems, with the exception of eXtremeDB, which cannot process more than 20K data points per second.

The results of the fast insertion rates—over 1M data points per second—show that only InfluxDB, MonetDB, and ClickHouse are able to reach high rates. In this case, ClickHouse becomes slower than InfluxDB and MonetDB. This is attributed to the fact that a significant portion of the new data has to be merged, surpassing the system's parallelization ability. InfluxDB and MonetDB append new data into their storage directories, which allows them to query the new data efficiently without any additional merging overhead.

5.4 Generation Performance and Impact

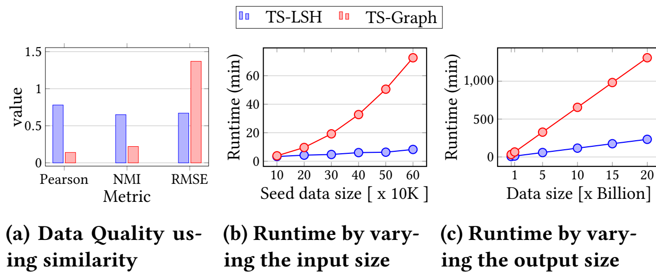
In this section, we evaluate the performance of TS-LSH in terms of efficiency and data quality and compare it to the state-of-the-art baseline TS-Graph (see Section 2.2). We also examine the impact of data quality on storage using different generation methods.



(a) Impact on Q1 (b) Impact on Q2 (c) Impact on Q3 (d) Impact on Q4 (e) Impact on Q5

Figure 8: Online Workload: Runtime Results on D-LONG by Varying Insertion Rate (x10K/s).

5.4.1 *Generation Performance.* We begin by evaluating the quality of the generated data by computing the similarity between an original time series and a synthetic one of the same length. We use standard similarity metrics: (1) *the Pearson correlation coefficient—Pearson* (for shape), where higher values are better, (2) *Normalized Mutual Information—NMI* (for the amount of shared information), where higher values are better, and (3) *Root Mean Squared Error—RMSE* (for distance), where lower values are better.



(a) Data Quality using similarity **(b) Runtime by varying the input size** **(c) Runtime by varying the output size**

Figure 9: Data Generation Performance.

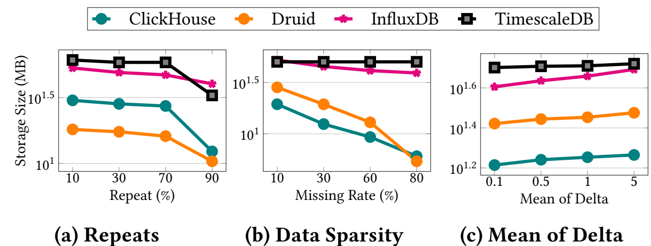
The results in Figure 9a show that TS-LSH outperforms the baseline on all metrics. It achieves 0.8 in Pearson correlation being 6x higher than TS-Graph, indicating that it better preserves the global trends of the time series. TS-Graph relies on smooth transitions between segments, hindering the learning of both the local and global properties of the data. The synthetic time series produced by this technique are shifted in time, which explains the low correlation and NMI values. Comparatively, TS-LSH achieves better results using both metrics.

Next, we compare the generation efficiency by varying the size of the seed data used to generate our D-LONG dataset. The results in Figure 9b show that the size of the seed data has a marginal impact on the runtime of TS-LSH. This is because the hash tables are constructed less frequently for higher inputs. Unlike our solution, TS-Graph generates a graph, which is quadratic in the length of the original series. The runtime difference between the two techniques grows with the size of the dataset.

Lastly, we report the runtimes to generate the two datasets used for our experiments, D-LONG (518M data points) and D-MULTI (17.2B data points). Figure 9c shows that both techniques scale linearly with the length of the generated series. Our technique is, on average, 5.7x faster than TS-Graph. This difference is due to the faster construction of LSH tables compared to graph generation.

5.4.2 *Compression Performance* . In this section, we evaluate how effectively the systems utilize time series properties. We generate time series with different features and measure their impact on the systems' compression performance. We consider three different features from a recent benchmark of data encoding techniques [65]. The features include i) repeated values, ii) missing values, and (iii) mean of delta between consecutive values.

In Figure 10, we incrementally increase the intensity of each feature and compute the resulting storage size. We report the results only for the systems where we could observe some impact. The results show three different trends. First, all reported systems can benefit from the existence of repeats to further compress the time series. ClickHouse and Druid show the best performance. The former system applies delta encoding and Run-length encoding (RLE) to store the difference between consecutive values. The latter uses bitmap encoding where a bit represents the presence or absence of each unique value.



(a) Repeats **(b) Data Sparsity** **(c) Mean of Delta**

Figure 10: Impact of Data Characteristics on Systems Storage.

Second, we observe that only ClickHouse and Druid can take advantage of the existence of missing values. Their storage size decreases for datasets with a high missing rate. Both systems support nullable data types to represent missing values allowing them to not allocate additional space for those values.

Finally, the results in Figure 10c show a different trend from the two previous experiments. Increasing the delta between consecutive values yields a higher storage size. When the delta between consecutive values is small, compression algorithms such as delta encoding or run-length encoding can use the similarity between adjacent data points to reduce storage. When the delta is large, this similarity is weaker and the compression algorithms are less able to reduce the storage size effectively.

6 RESULTS DISCUSSION

6.1 Performance Summary

To simplify the interpretation of the results and enable a more actionable comparison, we identify seven discriminative dimensions for comparing the performance of TSDBs (see Figure 11). Using these dimensions, we articulate a performance summary through a Kiviat diagram. Each system's performance for different query types is ranked along these dimensions on a 0-5 scale, with 5 representing the best performance. To assign the scores, the runtimes are normalized to [0, 1] and then log scaled.

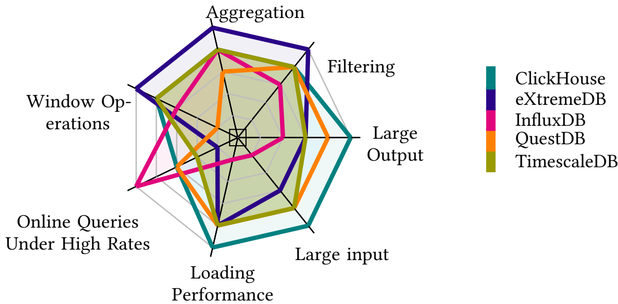


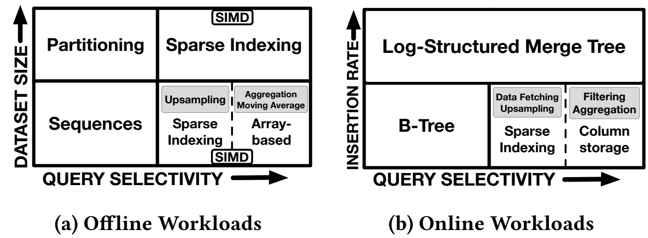
Figure 11: Characterization of the Performance of the Five Best Systems Under Various Workload Dimensions.

From Figure 11, we observe several performance patterns across the various systems. Most notably, some systems excel in specific categories, while others present a balanced trade-off across several dimensions. For example, ClickHouse stands out for queries with large output, data loading, and on datasets with many time series thanks to its parallelization ability. eXtremeDB leverages its sequence storage format to be best-in-class in filtering, aggregations, and window operations. InfluxDB is best suited for queries under fast insertion rates as they support data appending without any additional merging step. TimescaleDB can produce competitive results in all query types, while QuestDB is a reliable system in all queries except those which involve window operations.

6.2 Architecture Impact

Our empirical findings indicate that no single architecture dominates all the workload tiers. Some design choices, however, excel for specific workloads. Those results show also that the performance of those architectures depends on whether the workload is offline or online. Figure 12 showcases two decision matrices that illustrate the best design to use for a given workload query and mode.

The results of the offline workloads show that the query selectivity and the size of the data determine the choice of the appropriate architecture. For queries with low selectivity that involve small datasets, sequence-based systems, such as eXtremeDB, are likely to perform the best. When increasing the dataset size, using a partitioning mechanism is highly recommended. Other systems' designs can better handle queries with high selectivity. In smaller datasets, the array format is best suited for aggregations and window operations, whereas sparse indexing is recommended for upsampling. Systems that adopt SIMD, such as ClickHouse, can easily handle large datasets when executing queries with high selectivity.



(a) Offline Workloads **(b) Online Workloads**

Figure 12: Design Choices.

For online workloads, where queries and data ingestion are executed concurrently, the insertion rate and the query selectivity are the determining factors. Our results show that systems implementing B-Tree, such as QuestDB, achieve the best results in queries with low selectivity and insertion rate. Similarly to the offline mode, we notice that the query type has an impact only on queries with high selectivity. As such, sparse indexing is particularly suited for data fetching and upsampling queries while systems that implement column-store representation such as InfluxDB or MonetDB are best suited for filtering and aggregation queries. InfluxDB's Log-Structured Merge (LSM) tree yields the best overall query performance under high insertion rates.

7 CONCLUSION

This work aims to fill an important gap in the time series databases (TSDB) system benchmarking space. It presents a new benchmark, called TSM-Bench, designed to evaluate the performance of TSDBs used in monitoring applications. TSM-Bench implements a new time series generator that can efficiently augment the size of seed real-world time series using GANs and LSH. We conducted a comprehensive study on seven leading TSDBs to evaluate their performance and capabilities using the generated data. Our results provide valuable insights into how these systems work and offer useful guidance in selecting the best system.

Future development plans for TSM-Bench include evaluating TSDB systems with mixed-queries workloads and multitenancy scenarios using a load-generating framework. Also, new performance metrics such as energy consumption or cloud costs of the TSDBs can be added as a recommendation axis.

ACKNOWLEDGMENTS

This work was funded by the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (grant agreement No. 101092908). Additional funding was provided by the NYUAD Center for Interacting Urban Networks (CITIES) funded by Tamkeen under the NYUAD Research Institute Award CG001 and by the DIADEM project funded by the Free University of Bozen-Bolzano.

We would like to thank the anonymous reviewers for their insightful comments and suggestions. We would like also to thank Gabriela Dinica for her contribution in the early stage of this work. Lastly, we would like to give special thanks to Ganesh Vernekar and Jennie Zhang as well as the support teams from eXtremeDB, MonetDB, TimescaleDB, and Druid for their invaluable help in setting up and configuring the systems.

REFERENCES

[1] Michael Opoku Agyeman, Zainab Al-Waisi, and Igla Hoxha. 2019. Design and Implementation of an IoT-Based Energy Monitoring System for Managing Smart Homes. In *Fourth International Conference on Fog and Mobile Edge Computing, FMEC 2019, Rome, Italy, June 10-13, 2019*. IEEE, 253–258. https://doi.org/10.1109/FMEC.2019.8795363

[2] Paul L Anderson, Mark M Meerschaert, and Kai Zhang. 2013. Forecasting with prediction intervals for periodic autoregressive moving average models. *Journal of Time Series Analysis* 34, 2 (2013), 187–193.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. (2015), 1225–1233. https://proceedings.neurips.cc/paper/2015/hash/2823f4797102ce1a1aec05359cc16dd9-Abstract.html

[4] Alexandr Andoni, Piotr Indyk, Huy L. Nguyen, and Ilya P. Razenshteyn. 2014. Beyond Locality-Sensitive Hashing. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, Chandra Chekuri (Ed.). SIAM, 1018–1028. https://doi.org/10.1137/1.9781611973402.76

[5] Martin F. Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. 2015. IoTAbench: an Internet of Things Analytics Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, Lizy K. John, Connie U. Smith, Kai Sachs, and Catalina M. Lladó (Eds.). ACM, 133–144. https://doi.org/10.1145/2668930.2688055

[6] Christoph Bergmeir, Rob J Hyndman, and José M Benítez. 2016. Bagging exponential smoothing methods using STL decomposition and Box–Cox transformation. *International journal of forecasting* 32, 2 (2016), 303–312.

[7] Hong Cao, Vincent Y. F. Tan, and John Z. F. Pang. 2014. A Parsimonious Mixture of Gaussian Trees Model for Oversampling in Imbalanced and Multimodal Time-Series Classification. *IEEE Trans. Neural Networks Learn. Syst.* 25, 12 (2014), 2226–2239. https://doi.org/10.1109/TNNLS.2014.2308321

[8] Matteo Ceccarello and Johann Gamper. 2022. Fast and Scalable Mining of Time Series Motifs with Probabilistic Guarantees. *Proc. VLDB Endow.* 15, 13 (2022), 3841–3853. https://www.vldb.org/pvldb/vol15/p3841-ceccarello.pdf

[9] Zemin Chao, Hong Gao, Yinan An, and Jianzhong Li. 2022. The Inherent Time Complexity and An Efficient Algorithm for Subsequence Matching Problem. *Proc. VLDB Endow.* 15, 7 (2022), 1453–1465. https://www.vldb.org/pvldb/vol15/p1453-chao.pdf

[10] ClickHouse. 2022. ClickBench. https://github.com/ClickHouse/ClickBench/.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[12] Benjamin F Crabtree, Subhash C Ray, Priscilla M Schmidt, Patrick T O'Connor, and David D Schmidt. 1990. The individual over time: time series applications in health care research. *Journal of clinical epidemiology* 43, 3 (1990), 241–260.

[13] Lars Dietrich and Ansgar Kahmen. 2019. Water relations of drought-stressed temperate trees benefit from short drought-intermitting rainfall events. *Agricultural and forest meteorology* 265 (2019), 70–77.

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[15] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.* 1, 2 (2008), 1542–1552. https://doi.org/10.14778/1454159.1454226

[16] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. 2022. ProS: Data Series Progressive k-NN Similarity Search and Classification with Probabilistic Quality Guarantees. *CoRR* abs/2212.13310 (2022). https://doi.org/10.48550/arXiv.2212.13310 arXiv:2212.13310

[17] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New trends in high-d vector similarity search: al-driven, progressive, and distributed. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3198–3201.

[18] Christos Faloutsos, Jan Gasthaus, Tim Januschowski, and Yuyang Wang. 2018. Forecasting Big Time Series: Old and New. *Proc. VLDB Endow.* 11, 12 (2018), 2102–2105. https://doi.org/10.14778/3229863.3229878

[19] Ju Fan, Tongyu Liu, Guoliang Li, Junyou Chen, Yuwei Shen, and Xiaoyong Du. 2020. Relational Data Synthesis using Generative Adversarial Networks: A Design Space Exploration. *CoRR* abs/2008.12763 (2020). arXiv:2008.12763 https://arxiv.org/abs/2008.12763

[20] Spilios D Fassois and John S Sakellariou. 2009. Statistical time series methods for structural health monitoring. *Encyclopedia of structural health monitoring* (2009), 443–472.

[21] Germain Forestier, François Petitjean, Hoang Anh Dau, Geoffrey I. Webb, and Eamonn J. Keogh. 2017. Generating Synthetic Time Series to Augment Sparse Datasets. In *2017 IEEE International Conference on Data Mining, ICDM 2017, New Orleans, LA, USA, November 18-21, 2017*, Vijay Raghavan, Srinivas Aluru, George Karypis, Lucio Miele, and Xindong Wu (Eds.). IEEE Computer Society, 865–870. https://doi.org/10.1109/ICDM.2017.106

[22] Frédéric Frappart, Fabien Blarel, Ibrahim Fayad, Muriel Bergé-Nguyen, Jean-François Crétaux, Song Shu, Joël Schregenberger, and Nicolas Baghdadi. 2021. Evaluation of the performances of radar and lidar altimetry missions for water level retrievals in mountainous environment: The case of the Swiss lakes. *Remote Sensing* 13, 11 (2021), 2196.

[23] Jingkun Gao, Xiaomin Song, Qingsong Wen, Pichao Wang, Liang Sun, and Huan Xu. 2020. RobustTAD: Robust Time Series Anomaly Detection via Decomposition and Convolutional Neural Networks. *CoRR* abs/2002.09545 (2020). arXiv:2002.09545 https://arxiv.org/abs/2002.09545

[24] Jim Gray. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publishers Inc.

[25] Peeyush Gupta, Michael J. Carey, Sharad Mehrotra, and Roberto Yus. 2020. SmartBench: A Benchmark For Data Management In Smart Spaces. *Proc. VLDB Endow.* 13, 11 (2020), 1807–1820. http://www.vldb.org/pvldb/vol13/p1807-gupta.pdf

[26] Geon Heo, Yuji Roh, Seonghyeon Hwang, Dayun Lee, and Steven Whang. 2020. Inspector Gadget: A Data Programming-based Labeling System for Industrial Images. *Proc. VLDB Endow.* 14, 1 (2020), 28–36. https://doi.org/10.14778/3421424.3421429

[27] InfluxData. 2016. InfluxDB comparison. https://github.com/influxdata/influxdb-comparisons.

[28] Mostafa Jalal, Sara Wehbi, Suren Chilingaryan, and Andreas Kopmann. 2022. SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things. (2022), 12:1–12:11. https://doi.org/10.1145/3538712.3538723

[29] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2581–2600. https://doi.org/10.1109/TKDE.2017.2740932

[30] Haneul Jeon and Donghun Lee. 2021. A New Data Augmentation Method for Time Series Wearable Sensor Data Using a Learning Mode Switching-Based DCGAN. *IEEE Robotics Autom. Lett.* 6, 4 (2021), 8671–8677. https://doi.org/10.1109/LRA.2021.3103648

[31] Katharina Kaelin and Florian Altermatt. 2016. Landscape-level predictions of diversity in river networks reveal opposing patterns for different groups of macroinvertebrates. *Aquatic Ecology* 50, 2 (2016), 283–295.

[32] Vignesh Kamath, Jeff Morgan, and Muhammad Intizar Ali. 2020. Industrial IoT and Digital Twins for a Smart Factory : An open source toolkit for application design and benchmarking. In *2020 Global Internet of Things Summit, GIoTS 2020, Dublin, Ireland, June 3, 2020*. IEEE, 1–6. https://doi.org/10.1109/GIOTS49054.2020.9119497

[33] Yanfei Kang, Rob J. Hyndman, and Feng Li. 2020. GRATIS: GeneRAting TIme Series with diverse and controllable characteristics. *Stat. Anal. Data Min.* 13, 4 (2020), 354–376. https://doi.org/10.1002/sam.11461

[34] Lars Kegel, Martin Hahmann, and Wolfgang Lehner. 2018. Feature-based comparison and generation of time series. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018*, Dimitris Sacharidis, Johann Gamper, and Michael H. Böhlen (Eds.). ACM, 20:1–20:12. https://doi.org/10.1145/3221269.3221293

[35] Mourad Khayati, Ines Arous, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. ORBITS: Online Recovery of Missing Values in Multiple Time Series Streams. *Proc. VLDB Endow.* 14, 3 (2020), 294–306. https://doi.org/10.5555/3430915.3442429

[36] Mourad Khayati, Philippe Cudré-Mauroux, and Michael H. Böhlen. 2020. Scalable recovery of missing blocks in time series with high and low cross-correlations. *Knowl. Inf. Syst.* 62, 6 (2020), 2257–2280. https://doi.org/10.1007/s10115-019-01421-7

[37] Mourad Khayati, Alberto Lerner, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. Mind the Gap: An Experimental Evaluation of Imputation of Missing Values Techniques in Time Series. *Proc. VLDB Endow.* 13, 5 (2020), 768–782. https://doi.org/10.14778/3377369.3377383

[38] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudré-Mauroux. 2019. CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding. In *2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019*, Chaitanya K. Baru, Jun Huan, Latifur Khan, Xiaohua Hu, Ronay Ak, Yuanyuan Tian, Roger S. Barga, Carlo Zaniolo, Kisung Lee, and Yanfang (Fanny) Ye (Eds.). IEEE, 2289–2298. https://doi.org/10.1109/BigData47090.2019.9005580

[39] Abdelouahab Khelifati, Mourad Khayati, Philippe Cudré-Mauroux, Adrian Hänni, Qian Liu, and Manfred Hauswirth. 2021. VADETIS: An Explainable Evaluator for Anomaly Detection Techniques. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2661–2664. https://doi.org/10.1109/ICDE51399.2021.00298

[40] Abdelouahab Khelifati, Mourad Khayati, Anton Dignös, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2023. TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications [Technical Report]. https://github.com/eXascaleInfolab/TSM-Bench

[41] Anne-Marie Kurth and Mario Schirmer. 2014. Thirty years of river restoration in Switzerland: implemented measures and lessons learned. *Environmental earth sciences* 72, 6 (2014), 2065–2079.

[42] Li Li, Xiaonan Su, Yi Zhang, Yuetong Lin, and Zhiheng Li. 2015. Trend Modeling for Traffic Time Series Analysis: An Integrated Study. *IEEE Trans. Intell. Transp. Syst.* 16, 6 (2015), 3430–3439. https://doi.org/10.1109/TITS.2015.2457240

[43] Yuhong Li, Leong Hou U, Man Lung Yiu, and Zhiguo Gong. 2013. Discovering Longest-lasting Correlation in Sequence Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1666–1677. https://doi.org/10.14778/2556549.2556552

[44] Rui Liu and Jun Yuan. 2019. Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios. (2019). arXiv:1901.08304 [cs.DB]

[45] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. The Expressive Power of Neural Networks: A View from the Width. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6231–6239. https://proceedings.neurips.cc/paper/2017/hash/32cbf687880eb1674a07bf717761dd3a-Abstract.html

[46] Victor Maus, Gilberto Câmara, Ricardo Cartaxo, Alber H. Sánchez, Fernando M. Ramos, and Gilberto Ribeiro de Queiroz. 2016. A Time-Weighted Dynamic Time Warping Method for Land-Use and Land-Cover Mapping. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* 9, 8 (2016), 3729–3739. https://doi.org/10.1109/JSTARS.2016.2517118

[47] Abdullah Mueen, Krishnamurthy Viswanathan, CK Gupta, and Eamonn Keogh. 2015. The fastest similarity search algorithm for time series subsequences under Euclidean distance. url: www cs unm edu/ mueen. *FastestSimilaritySearch html (Accessed 24 May 2016)* (2015).

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

[49] Rodica Neamtu, Ramoza Ahsan, Elke A. Rundensteiner, and Gábor N. Sárközy. 2016. Interactive Time Series Exploration Powered by the Marriage of Similarity Distances. *Proc. VLDB Endow.* 10, 3 (2016), 169–180. https://doi.org/10.14778/3021924.3021933

[50] Vincent Nouchi, Tiit Kutser, Alfred Wüest, Beat Müller, Daniel Odermatt, Theo Baracchini, and Damien Bouffard. 2019. Resolving biogeochemical processes in lakes using remote sensing. *Aquatic Sciences* 81, 2 (2019), 1–13.

[51] John Paparrizos and Michael J. Franklin. 2019. GRAIL: Efficient Time-Series Representation Learning. *Proc. VLDB Endow.* 12, 11 (2019), 1762–1777. https://doi.org/10.14778/3342263.3342648

[52] John Paparrizos, Yuhao Kang, Paul Boniol, Ruey S. Tsay, Themis Palpanas, and Michael J. Franklin. 2022. TSB-UAD: An End-to-End Benchmark Suite for Univariate Time-Series Anomaly Detection. *Proc. VLDB Endow.* 15, 8 (2022), 1697–1711. https://www.vldb.org/pvldb/vol15/p1697-paparrizos.pdf

[53] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. 2018. Data Synthesis based on Generative Adversarial Networks. *Proc. VLDB Endow.* 11, 10 (2018), 1071–1083. https://doi.org/10.14778/3231751.3231757

[54] Jinfeng Peng, Derong Shen, Nan Tang, Tieying Liu, Yue Kou, Tiezheng Nie, Hang Cui, and Ge Yu. 2022. Self-supervised and Interpretable Data Cleaning with Sequence Generative Adversarial Networks. *Proc. VLDB Endow.* 16, 3 (2022), 433–446. https://www.vldb.org/pvldb/vol16/p433-peng.pdf

[55] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. (2016). http://arxiv.org/abs/1511.06434

[56] Lida Rashidi, Sutharshan Rajasegarar, Christopher Leckie, Michele Nati, Alexander Gluhak, Muhammad Ali Imran, and Marimuthu Palaniswami. 2014. Profiling spatial and temporal behaviour in sensor networks: A case study in energy monitoring. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*. IEEE, 1–7. https://doi.org/10.1109/ISSNIP.2014.6827606

[57] Stefano Rinaldi, Federico Bonafini, Paolo Ferrari, Alessandra Flammini, Emiliano Sisinni, and Devis Bianchini. 2019. Impact of Data Model on Performance of Time Series Database for Internet of Things Applications. In *IEEE International Instrumentation and Measurement Technology Conference, I2MTC 2019, Auckland, New Zealand, May 20-23, 2019*. IEEE, 1–6. https://doi.org/10.1109/I2MTC.2019.8827164

[58] Kexin Rong, Clara E. Yoon, Karianne J. Bergen, Hashem Elezabi, Peter Bailis, Philip Alexander Levis, and Gregory C. Beroza. 2018. Locality-Sensitive Hashing for Earthquake Detection: A Case Study Scaling Data-Driven Science. *Proc. VLDB Endow.* 11, 11 (2018), 1674–1687. https://doi.org/10.14778/3236187.3236214

[59] Sebastian Schmidl, Phillip Wenig, and Thorsten Papenbrock. 2022. Anomaly Detection in Time Series: A Comprehensive Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1779–1797. https://www.vldb.org/pvldb/vol15/p1779-wenig.pdf

[60] Timescale. 2020. *Time Series Benchmark Suite.* https://github.com/timescale/tsbs

[61] Zhengwei Wang, Qi She, and Tomás E. Ward. 2022. Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy. *ACM Comput. Surv.* 54, 2 (2022), 37:1–37:38. https://doi.org/10.1145/3439723

[62] Qingsong Wen, Liang Sun, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. 2020. Time Series Data Augmentation for Deep Learning: A Survey. *CoRR* abs/2002.12478 (2020). arXiv:2002.12478 https://arxiv.org/abs/2002.12478

[63] Tailai Wen and Roy Keyes. 2019. Time Series Anomaly Detection Using Convolutional Neural Networks and Transfer Learning. *CoRR* abs/1905.13628 (2019). arXiv:1905.13628 http://arxiv.org/abs/1905.13628

[64] Xinle Wu, Dalin Zhang, Chenjuan Guo, Chaoyang He, Bin Yang, and Christian S. Jensen. 2021. AutoCTS: Automated Correlated Time Series Forecasting. *Proc. VLDB Endow.* 15, 4 (2021), 971–983. https://doi.org/10.14778/3503585.3503604

[65] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB. *Proc. VLDB Endow.* 15, 10 (2022), 2148–2160. https://www.vldb.org/pvldb/vol15/p2148-song.pdf

[66] Yahoo. 2023. Yahoo Cloud Server Benchmark for Time Series. https://github.com/TSDBBench/YCSB-TS.

[67] Byoung-Kee Yi and Christos Faloutsos. 2000. Fast Time Sequence Indexing for Arbitrary Lp Norms. (2000), 385–394. http://www.vldb.org/conf/2000/P385.pdf

[68] Hao Yuanzhe, Qin Xiongpai, Chen Yueguo, Li Yaru, Sun Xiaoguang, Tao Yu, Zhang Xiao, and Du Xiaoyong. 2021. TS-Benchmark: A Benchmark for Time Series Databases. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021).

[69] Guoqiang Peter Zhang. 2003. Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing* 50 (2003), 159–175. https://doi.org/10.1016/S0925-2312(01)00702-0

[70] Yandong Zheng, Rongxing Lu, Yunguo Guan, Songnian Zhang, and Jun Shao. 2021. Towards Private Similarity Query based Healthcare Monitoring over Digital Twin Cloud Platform. In *29th IEEE/ACM International Symposium on Quality of Service, IWQOS 2021, Tokyo, Japan, June 25-28, 2021*. IEEE, 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521351

[71] Ming Zhu, Zongxi Zhang, Jie Mei, Kejian Zhou, Pengan Chen, Yongka Qi, and Qinqing Huang. 2021. Data augmentation using DCGAN for improved fault detection of high voltage shunt reactor, In Journal of Physics: Conference Series. *Journal of Physics: Conference Series* 1944, 1, 012012.

[72] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2015. RINSE: Interactive Data Series Exploration with ADS+. *Proc. VLDB Endow.* 8, 12 (2015), 1912–1915. https://doi.org/10.14778/2824032.2824099