TS-Benchmark: A Benchmark for Time Series Databases

Yuanzhe Hao, Xiongpai Qin, Yueguo Chen, Yaru Li, Xiaoguang Sun, Yu Tao, Xiao Zhang and Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China, Beijing, China
Email: {haoyuanzhe, qxp1990, chenyueguo, liyaru, xg.sun, taoyu2019, zhangxiao, duyong}@ruc.edu.cn

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00057

***Abstract*—Time series data is widely used in scenarios such as supply chain, stock data analysis, and smart manufacturing. A number of time series database systems have been invented to manage and query large volumes of time series data. We observe that the existing benchmarks of time series databases are focused on workloads of complex analysis such as pattern matching and trend prediction whose performance may be highly affected by the data analysis algorithms, instead of the back-end databases. However, in many real applications of time series databases, people are more interested in the performance metrics such as data injection throughput and query processing time. A benchmark is still required to extensively compare the performance of time series databases in such metrics. We introduce such a benchmark called TS-Benchmark which majorly applies a scenario of device monitoring for wind turbines. A DCGAN-based data generation model is proposed to generate large volumes of time series data from some real time series data. The workloads are categorized into three folds: data loading (in batch), streaming data injection, and historical data access (for typical queries). We implement the benchmark and compare four representative time series databases: InfluxDB, TimescaleDB, Druid and OpenTSDB. The results are reported and analyzed.**

***Index Terms*—benchmark, time series database, Internet of Things, generative adversarial network**

I. Introduction

In the age of Internet of Things, large volumes of data can be generated from many kinds of devices, which are further collected and stored in data center as big time series data. Taking sensors as an example, we can continuously collect sensor data from remote systems, to monitor the status of the systems, to predict some failures in advance, and to trace the root cause of some failures occurred recently.

Sensor-generated data comes in (almost) temporal order. They are therefore usually attached with timestamps recording the time when the data items are sensed. A data item may also have additional features such as *device_id* and *sensor_id*. Although general relational database management systems (RDBMS) can be used to store such temporal data, it is more natural to store such continuous sensor-generated data in a time series database, where sequences of data are individually clustered and indexed on a temporal dimension. A number of time series databases such as InfluxDB [26] and TimescaleDB [38] have been invented to manage time series data and provide data services more efficiently. They are widely used in many IoT applications where big time series data are collected.

Yueguo Chen is the corresponding author.

Time series databases (TSDBs) are typically applied for storing historical data so that they can be further used for querying and analyzing the status of remote systems where data are collected. Consequently, real-world applications have high requirements for the read/write performance of TSDBs. Depending on whether an external DBMS is relied, existing mainstream open source TSDBs can be divided into two categories [5], [27]. Examples of TSDBs that depend on external DBMS are TimescaleDB (runs on top of PostgreSQL) and OpenTSDB (based on HBase). Examples of TSDBs that do not depend on external DBMS are InfluxDB and Druid. We therefore choose InfluxDB, TimescaleDB, Druid, and OpenTSDB for comparison in this work.

The TPC-C [10] workload simulates a simple OLTP environment including order handling and good delivery. It is usually used to test RDBMSs. Our benchmark is also different from analytic benchmarks such as TPC-H [11] and TPC-DS [34]. TPC-H and TPC-DS are running against databases (usually RDBMS) of star schema or snowflake data model, and most queries involve joining multiple tables and doing aggregations. Our benchmark is running on TSDBs which primarily include jobs of range queries, aggregation queries, rolling window queries, etc. There are some benchmarks designed specifically for TSDBs, such as IoTAbench [4] and Linear Road [3]. IoTAbench is based on a use case of smart metering. Linear Road simulates a traffic monitoring system. They typically focus on complex analysis tasks where high level workloads such as pattern matching and trend prediction are applied. The performance of such workloads however are highly affected by the analysis algorithms, and therefore the read performance of TSDBs can hardly be measured accurately. Moreover, the write performance (data injection) of TSDBs is ignored by these benchmarks. Analytics in Motion (AIM) [6] evaluates the ingestion performance of streaming events and SQL analytics on materialized state, which is based on billing data. However, AIM is designed to process mixed workloads with high volume of events, not exclusively for the performance of TSDBs. We argue that the emerging of new networking techniques (e.g., 5G) will allow large-scale online monitoring applications where massive time series data are collected and sent to data center timely for failure monitoring and diagnosis. As a result, the write performance will be critical for back-end TSDBs. We therefore need a new benchmark of TSDBs that is able to evaluate the write performance, as well as the pure read performance for typical

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00057

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

analysis tasks on time series data.

To fill the gap of existing benchmarks for TSDBs, we propose a new benchmark called TS-Benchmark[1], which is based on the requirements of managing massive time series data. A DCGAN-based data generation model produces high-quality synthesized data after being trained with real data. Specially, we simulate a scenario where a large number of sensors (of different types) from wind turbines frequently report the sensed data to the data center. The number of wind farms, devices and sensors used for data generation are treated as scale factors of the benchmark. For write workloads, each wind turbine is assumed to periodically (7 seconds by default) report the collected data to TSDB. For read workloads, we identify some typical data access patterns from high level analysis tasks. We also test the performance of data loading when a certain size of time series data are loaded to TSDB in batch. We apply the TS-Benchmark to test the performance of four TSDBs: InfluxDB, TimescaleDB, Druid and OpenTSDB. The contributions of our work are as follows:

- TS-Benchmark is able to systematically test the performance of TSDBs under various workloads of data loading, data injection, and data fetching.
- TS-Benchmark is majorly based on a scenario of IoT applications. The proposed data generator is able to generate high-quality synthesized time series in high throughput.
- We test the performance of some typical TSDBs using TS-Benchmark, and obtain some insights of the underlying techniques by analyzing the results.

The paper is organized as follows. Section 2 presents related work. Section 3 give details of the benchmark, including application scenarios, data generation model, workload, test procedure, performance metrics and benchmark implementation. We use the results on testing InfluxDB, TimescaleDB, Druid and OpenTSDB to show the functionality of TS-Benchmark in Section 4. Finally, Section 5 concludes the paper.

II. RELATED WORK

A. State-of-the-art Benchmarks

Stream bench [31] is a benchmark for distributed stream processing systems such as Storm [15]. In Stream bench, operations such as *sampling* and *projection* are relevant to the speed of data serving/fetching in a database system. However, the performance of other operations such as *grep* and *word-count* will be highly dependent on the efficiency of upper layer software. RIoTBench [36] is also a benchmark for distributed stream processing systems, which includes 27 common IoT tasks spanning data pre-processing, statistical analysis and predictive analysis. Like Stream bench, it is not suitable for benchmarking a TSDB where read/write performance need to be extensively tested. Other benchmarks on big data such as HiBench [22], BigBench [16] and [9], designed for key-value data, multi-model data and graph data respectively, are not suitable for benchmarking TSDBs either.

IoTAbench [4] is a benchmark for TSDBs and RDBMS, which is based on a scenario of smart metering. It's workloads include data loading, and 6 distinct analytical queries: 1) total readings: counts the total number of readings (i.e., rows) within a given time period; 2) total consumption: sums up the resource consumption within a given time period; 3) peak consumption: create a sorted list of the aggregate consumption in each ten-minute interval within a given time period; 4) top consumers: create a list of the distinct consumers, sorted by their total (monthly) consumption; 5) sequences of consumption data: compute the sequences of aggregated consumption per ten-minute interval within a given time period; 6) time of usage billing: calculate the monthly bill for each consumer based on the time of usage. We can see that the queries of IoTAbench are mainly aggregation queries. It still lacks of some basic query types such as snapshot query, sub-sequence query, and so on. In the meanwhile, as stated, IoTAbench cannot tests the performance of data injection.

Another benchmark of TSDBs, Linear Road [3] is based on a toll system for the motor vehicle expressways of a large metropolitan area. It tests how well a Stream Data Management Systems (SDMS) serves different analytical queries such as accident detection and alerts, traffic congestion measurements, toll calculations, as well as historical queries. InfluxDB-comparison [24] is a tool to benchmark InfluxDB database against other TSDBs. It focuses on the DevOps [14] use cases in a data center scenario. Data and queries are created to mimic what a system administrator would see when operating a fleet of thousands of virtual machines. The queries are focused on the aggregation of some metrics such as CPU load, RAM usage, number of active, sleeping, or stalled processes, disk used etc. The testing procedure is as follows, some data is generated and loaded to the database. Then the queries are executed against the loaded data. Throughput and response time of the queries are reported. InfluxDB-comparison does not include the test of data injection either. Time Series Benchmark Suite [37] is extended from InfluxDB-comparison by considering a variety of use cases, query types, and databases to be tested. However, it still does not test continuous data injection. It randomly generate datasets for DevOps and IoT cases, without guaranteeing the real-world data distribution.

B. Some Time Series Databases and Their Designs

- **InfluxDB** is a NoSQL database. The storage engine of InfluxDB is based on time structured merge tree (TSM), which is adapted from log structured merge tree [25]. TSM brings excellent performance improvement to InfluxDB both on data writing and reading [24].
- **TimescaleDB** is a TSDB built on top of PostgreSQL [21], which is a relational database. It inherits the full features of PostgreSQL and creates adaptive time chunks to minimize the usage of system resources. TimescaleDB has a complete SQL support.
- **Druid** [39] is an open-source columnar data store, designed for OLAP queries. It includes different kinds of nodes to support a specific set of functionalities. Interactions between different nodes have been minimized.

[1]open sourced at https://github.com/dbiir/TS-Benchmark

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

- **OpenTSDB** is a TSDB built on top of HBase, which is also a NoSQL database. HBase is a key-value database modeled after Google's Bigtable [8], which is built upon Google File System [17]. It is based on columnar storage.

C. Data generation

The work [41] proposes a hybrid time series forecasting model based on autoregressive integrated moving average and neural network. The periodic autoregressive moving average model [1] takes into account the periodicity of time series data, which is also proposed for time series forecasting. Both of these time series forecasting methods can be used for data generation. Scalable smart meter data generator [23] is a data generation tool based on smart meters scenario. Scalable smart meter data generator trains data models by flattening of time series fluctuations and deseasonalization firstly, then it generates data using the model and reseasonalization. It applies a supervised learning method to create the models, and generates synthetic data by these models. Feature-based generation (FBG) [29] proposes feature-based similarity and take it into account to generate highly similar time series. FBG does not exploit the correlation between components which could improve the recombination.

III. The Benchmark

A. Application Scenario: Monitoring Wind Turbines

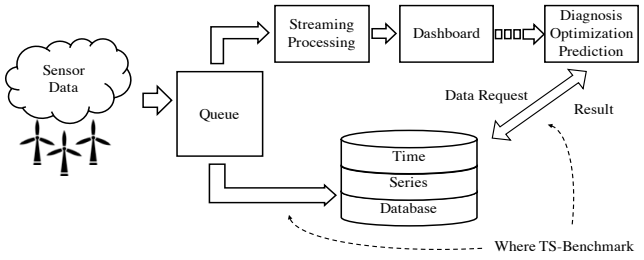


Fig. 1: A typical data processing pipeline of time series data collected from wind turbines

The TS-Benchmark is majorly based on an application of failure monitoring and diagnosis in wind power generation, although it also supports other types of time series data. A typical data processing pipeline in this application is shown in Figure 1. Basically, all sensor readings of wind turbines within a wind farm are collected and sent back to the data center periodically. The number of sensors on each device (a turbine-generator) is up to hundreds. We assume that there are 50 devices in each wind farm, and wind farms send their data to data center independently. A large wind power company may have thousands of wind farms, and therefore need to process data from hundreds of thousands of devices. The streaming data are typically routed to stream processing systems for monitoring applications and TSDBs for persistent storage.

The stream processing system typically supports visual analysis of time series data. Users can select particular wind farm, device, or sensor for monitoring. The most recent data within a small time window are accessible by the monitoring application. Alerts for some failures will be reported on dashboard. Further access to some historical data stored in TSDB will be required if users want to do some exploratory or advanced data analysis. This usually happens when some abnormalities/failures are observed or predicted.

A TSDB is responsible for persisting the streaming data so that they can be efficiently retrieved for querying and further analysis in some scenarios:

- **Problem identification**. When sensor readings exceed a threshold, people need to fetch more data for identifying the problem. In some cases, people want to know the frequency of abnormalities. This requires to fetch data within a certain period covering the failure point.
- **Operational optimization**. Users may be interested in which turbine is performing the best recently. This requires to fetch major readings of sensors related to power generation for deep analysis, so that some ideal parameters can be found to optimize the turbines.
- **Problem prediction**. When users are interested in predicting what will be happening in the near future, more data is typically required for predictive analysis.
- **Correlation analysis**. People need to perform correlation analysis across different devices, or readings of different sensors of the same device. This requires to read historical data of different devices and sensors in the same period to conduct correlation analysis.

B. Data Generation Model

Many IoT monitoring applications require a large volume of sensor data to be generated and analyzed continuously. The benchmark need to be able to generate massive high-quality sensor readings in high throughput. "High-quality" means that the generated time series are very similar (in terms of shapes) to the real time series data. "High throughput" means massive sensor readings can be generated in parallel. To mimic time series more effectively and generate them more efficiently, we propose a framework of massive time series data generation which has the following steps:

- Create seed fragments (sequences) from real time series data[2], which usually has limitation on data size.
- Generate synthetic fragments from real seeds using some generative adversarial network (GAN) [18] model.
- Build a directed graph of the synthetic fragments.
- Continuously generate time series using a random walk on the directed graph of synthetic fragments.

Note that the last two steps are for generating streaming time series, and they are therefore optional.

1) Generative Adversarial Network for synthetic data: A GAN model has two adversarial parts, a generator G and a discriminator D. The generator G learns a distribution P_g over data x via a mapping $G(z)$ of samples z, where a sample z

[2]A real dataset (wind turbines) and some other real datasets from the UCR Time Series Classification Archive [12] are available with TS-Benchmark.

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

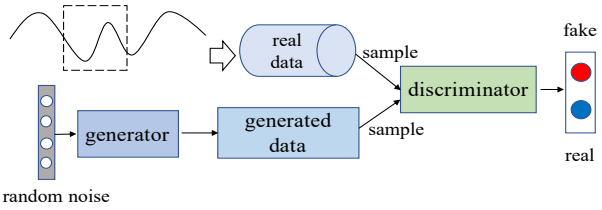


Fig. 2: The architecture of the GAN for time series generation

is an 1D uniformly distributed input noise. The input of the discriminator D is synthetic fragments generated by G, and it outputs $D(\cdot)$, indicating the quality of the synthetic data. Figure 2 illustrates the architecture of GAN. The generator G and discriminator D are simultaneously optimized in a two-player minimax game defined by the value function $V(G, D)$:

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{1}$$

In practice, the generator is trained to maximise $log(D(G(z)))$ instead of minimizing $log(1 - D(G(z)))$, because it provides stronger gradients while training [18].

GAN has been successfully applied in many areas for data generation. Deep convolutional generative adversarial networks (DCGAN) [35] is proposed because it shows excellent capacity of capturing semantic image content. The GAN model used in TS-Benchmark is based on DCGAN. The generator is the deconvolutional layer, and the discriminator is the convolutional layer. DCGAN makes some changes to the structure of convolutional neural network (CNN) to improve the quality of samples and the speed of convergence [35]. CNN in GAN ensures that a feature is learned independently of its position in the time series [13]. In this architecture, the generator G is equivalent to a convolutional decoder. By applying multiple filters, the network can learn multiple features of the data. The output dimensions are calculated as:

$$\frac{W - K + 2P}{S} + 1 \tag{2}$$

where W is the number of input dimensions, K is the kernel or filter size, S is the stride and P is the amount of zero-padding applied to the border of the data.

During adversarial training process, the generator continuously improves the quality of the synthetic fragments, and the discriminator continuously improves the ability to discriminate between real data and fake data. When the adversarial training process is completed, the parameters of the generator and discriminator are fixed. Generator has learned mapping $G(\mathbf{z}) = \mathbf{z} \mapsto \mathbf{x}$ from latent space representations $\mathbf{z}$ to realistic fragment $\mathbf{x}$. However, GAN can not yield the inverse mapping $\mu(\mathbf{x}) = \mathbf{x} \mapsto \mathbf{z}$ automatically. Given a real time series fragment $\mathbf{x}$, the goal is to find the point z corresponding to the sequence $G(\mathbf{z})$ in the latent space, which is most similar to the real fragment $\mathbf{x}$. The generated fragment $G(\mathbf{z})$ is the closest to the real sequence $\mathbf{x}$, and the generated fragment is satisfied with the distribution of real data. In this way, synthetic time series fragments are successfully generated.

The quality of synthetic data is determined by two aspects. On one hand, it is related to the output of the discriminator, which represents the similarity between the generated sequence and the real data. On the other hand, it is also related to the similarity of the targeted real fragments $\mathbf{x}$. The loss of the discriminator is defined as:

$$\mathcal{L}_D(\mathbf{z}_\gamma) = \sum |\mathbf{f}(\mathbf{x}) - \mathbf{f}(G(\mathbf{z}_\gamma))| \tag{3}$$

where $f(\cdot)$ denotes the statistics of an input fragment and $G(\mathbf{z}_\gamma)$ is the generated fragment. The second loss function for the similarity between targeted data and synthetic data is:

$$\mathcal{L}_R(\mathbf{z}_\gamma) = \sum |\mathbf{x} - G(\mathbf{z}_\gamma)| \tag{4}$$

where $\mathbf{x}$ is the targeted real data and $G(\mathbf{z}_\gamma)$ is the generated sequence. The final loss function is defined as the weighted sum of the above two loss functions:

$$\mathcal{L}(\mathbf{z}_\gamma) = (1 - \lambda) \cdot \mathcal{L}_R(\mathbf{z}_\gamma) + \lambda \cdot \mathcal{L}_D(\mathbf{z}_\gamma) \tag{5}$$

where λ is the weight parameter. In order to balance the two parts of the loss function, we set the $\lambda = 0.5$, and our results prove that it brings good synthetic time series.

2) Directed Graph of Synthetic Fragments: With the above data generation model, a large number of synthetic time series fragments are generated and used as seeds for continuously generating time series. To achieve this, we build a directed graph of the synthetic sequences so that those connectable fragments are linked with some edges. We define the connectability of a sequence a to another sequence b as:

$$s(a,b) = \text{sim}_\text{l}(a_t, b_h) = \frac{1}{\sqrt{\sum_{i=1}^{l}(a_{ti} - b_{hi})^2}} \tag{6}$$

where a_t denotes the tail of sequence a whose length is l, and b_h denotes the head of sequence b whose length is l too. The similarity $\text{sim}_\text{l}(a_t, b_h)$ of the two sequences is defined by the Euclidean distance between fragments a_t and b_h. Then, given a sequence a, its spliceable fragment set is defined as $N_a = \{b | s(a,b) > \bar{s}\}$, where $\bar{s}$ is a threshold to filter those fragments should not be spliceable to a. Note that in case of $N_a = \emptyset$, we simply set $N_a = \{argmax_b\ s(a,b)\}$.

A directed graph can then be built by creating edges from any segment a to those in N_a. Given a pair of spliceable fragments a and b, a probabilitic weight $w(a,b)$ is assigned to the edge of (a,b):

$$w(a,b) = \frac{s(a,b) - \bar{s}}{\sum_{b' \in N_a} s(a,b') - \bar{s}} \tag{7}$$

3) Generating Synthetic Time Series via Random Walk: Given an initial seed sequence a, a subsequent sequence b is generated by a random walk process on the directed graph, meaning that b has a probability of $w(a,b)$ to be the picked as the subsequent sequence of a. A synthetic time series can then be continuously generated via a random walk process. Given two segments a and b, to connected them smoothly, a_t

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

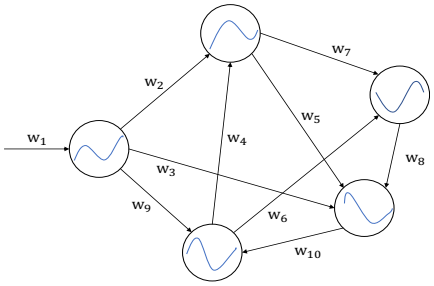


Fig. 3: The directed graph constructed by sequences

and b_h of the two adjacent sequences are spliced using a fitting function defined as:

$$c_i = (1 - \sigma_i)\, a_i + \sigma_i b_i \tag{8}$$

where $i = [-\frac{l}{2}, ..., \frac{l}{2})$ is the index of overlaps between sequences, and σ is the sigmoid function $\sigma_i = \frac{1}{1+e^{-i}}$. During the transition from sequence a to b, the weight of a and b gradually changes, leading to a smooth transition.

Note that some outlier shapes seldom appear in the real time series, which will be hard to mimic even using the proposed DCGAN-based model. To address this challenge, we allow users to manually create some segments of abnormal shapes (with two smooth ends). They are also applied to build the directed graph for random walks, so that abnormal shapes are also randomly generated as part of the random walk process.

C. Metrics

Throughput of batch import. When a local dataset is loaded into a target database, the throughput of the entire data loading process is measured, which is defined as the average number of data points loaded per time unit.

Compression ratio of batch import. It is defined as the ratio of data size stored in database to the raw data size. A lower compression ratio means that the data is compressed in a higher rate, which saves the storage space.

Appending throughput. When streaming data is continuously injected into a target database with multi-thread connections, the overall throughput of the target database is measured. We define the appending throughput as the number of data points successfully injected into the database per time unit.

Query latency. It is the response time that the database serves a query. For each kind of queries, 10 queries of different parameters will be executed. The average response time is computed as the performance metric of query latency.

D. Workloads

TS-Benchmark has 3 types of workloads: data loading, data injection, and data fetching.

1) Data Loading: Two basic datasets are generated in text format with different data sizes. The first dataset includes data from 2 wind farms. There are 50 devices in each wind farm. Each device has 50 sensors. The first dataset covers time series data of one week, including 4.32×10^8 data points. The total size of the first dataset is 3.9 GB. The second dataset includes data from 50 wind farms. There are also 50 devices in each wind farm. Each device has 150 sensors. The second dataset also cover data of one week, including 3.24×10^{10} data points. The total size of the dataset is 284.5 GB. The size of dataset can be dynamically set according to the three scale factors: the number of wind farms, the number of devices per wind farm, and the number of sensors per device, although the number of wind farms is recommended as the major scale factor for resizing. During data loading test, the batch file is imported to the target database using either its loading functionality (if available), or a data loading program specially written for it.

2) Continuous Data Injection: After a batch of time series data is loaded into the target TSDB, a data injection workload will be applied to test the write performance of database. The data injection task is implemented in a multi-threading manner. Each thread (as a connection) is responsible for injecting data of one farm. The number of threads equals to the number of farms (scale factor). For each thread, a snapshot of all sensor data of devices is packed and sent to the target database periodically. We gradually increase the number of threads from 1 to 512, to simulate the growing workload on the target system. It is expected that at some point the target system will get saturated, and then it will not be able to persist more data points timely. We measured the throughput in terms of data points successfully written per second.

3) Data Fetching for Problem Identification: There are two query cases for this workload, Query 1.1 and Query 1.2.

Query 1.1. When there is an alert on dashboard, e.g., the readings of some sensor exceed a threshold, people need to fetch more data to identify what problem it is. Query 1.1 is expressed in SQL as follows:

```
SELECT * FROM ts_table
WHERE f_id = ?1
AND d_id = ?2
AND s_id = ?3
AND time >?time_start
AND time <(?time_start + 1 hour);
```

where *f_id*, *d_id*, *s_d* and *time* denote farm ID, device ID, sensor ID, and timestamp respectively. Parameters ?1, ?2, ?3, and $?time\_start$ can be set randomly from dictionary, or manually by users. The width of the time window is fixed as an hour to guarantee that a certain size of data is fetched.

Query 1.2. There may also be a requirement to compute the frequency that the same problem occurs. Let the timestamp at which the problem is currently identified (e.g, the current reading exceeds some threshold) be a problem point. Data analysts need to fetch the data around the problem points for further analysis. The query identifies the timestamps when some sensor readings exceed the threshold. Query 1.2 is expressed in SQL as follows:

```
SELECT s_id, d_id, f_id, time FROM ts_table
WHERE f_id = ?1
AND s_id = ?2
AND time >?time_start
AND time <(?time_start + 1 week)
AND value >?3;
```


Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

where ?3 denotes the threshold of some sensor readings. Note that the width of the time window is fixed as one week, which also can be set by users. In this case, the device is not restricted to one device *d_id*. Readings of the same sensor from all devices of a wind farm are required.

4) Data Fetching for Optimization: In some cases, people are interested in which turbine of a wind farm has performed the best recently, hour by hour. Based on the aggregation data, we can fetch readings of major sensors related to power generation to do some deep analysis, with the expectation that other turbines can be optimized to improve the efficiency of overall power generation. We propose two queries for this purpose, Query 2.1 and Query 2.2.

Query 2.1 computes the average of the power sensor (measuring the amount of power generated) of each device hour by hour within one week.

```
SELECT f_id, d_id, avg(value) FROM ts_table
WHERE f_id = ?1
AND time >?time_start
AND time <?time_start + 1 week
AND s_id = "power"
GROUP BY f_id, d_id, hour (time);
```

Query 2.2 returns readings of major sensors which are related to power generation for deep analysis conducted in the upper layer software. The time parameter is set according to the results of Query 2.1 in real applications. After executing the Query 2.1, people find out that within some hours, a device *d_id* generates the most power. Based on that, people want to fetch the readings of major sensors of all devices (including the turbine performs the best) at the same hour for deep analysis.

```
SELECT * FROM ts_table
WHERE f_id = ?1
AND s_id in (S1, S2, S3, S4, S5)
AND time >?time_start
AND time <(?time_start + 1 hour)
ORDER BY d_id;
```

5) Data Fetching for Problem Prediction: In many cases, people concern not only the immediate problems, but also the potential problems that will happen in the near future. Recent data of a wind farm is required for predictive analysis (e.g., linear regression). **Query 3** serves this purpose. It fetches the sensor readings of all devices of the same wind farm within a time window.

```
SELECT * FROM ts_table
WHERE f_id = ?1
AND time >?time_start
AND time <(?time_start + 15 min);
```

6) Data Fetching for Correlation Analysis: Sometimes people need to perform correlation analysis between readings of different sensors, within a sliding window. **Query 4** is designed for this demand. It is used to continuously acquire all sensor values of fixed window size.

TABLE I: Selectivity of the different queries

Query	Dataset 1	Dataset 2
Query 1.1	1.18×10^{-6}	1.58×10^{-8}
Query 1.2	1.00×10^{-2}	1.33×10^{-3}
Query 2.1	1.00×10^{-2}	1.33×10^{-3}
Query 2.2	2.98×10^{-4}	1.19×10^{-5}
Query 3	7.44×10^{-4}	2.98×10^{-5}
Query 4	2.98×10^{-3}	1.19×10^{-4}

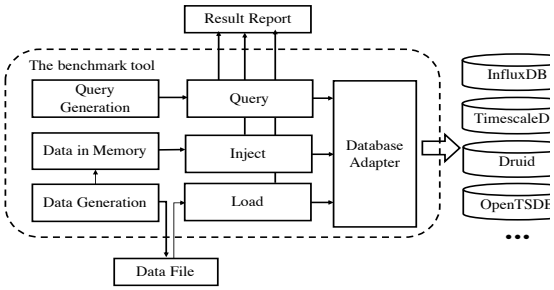


Fig. 4: The Architecture of the TS-Benchmark

```
for ?time_start in range(0, 30):
SELECT * FROM ts_table
WHERE f_id = ?1
AND time >?time_start
AND time <(?time_start + 30 min);
end for
```

Note that some TSDBs (e.g., OpenTSDB) do not support the standard SQL syntax. In this situation, we convert the 6 queries into the equivalent query language to suit the database under test. Selectivity of the queries is listed in Table I.

E. Implementation of the Benchmark

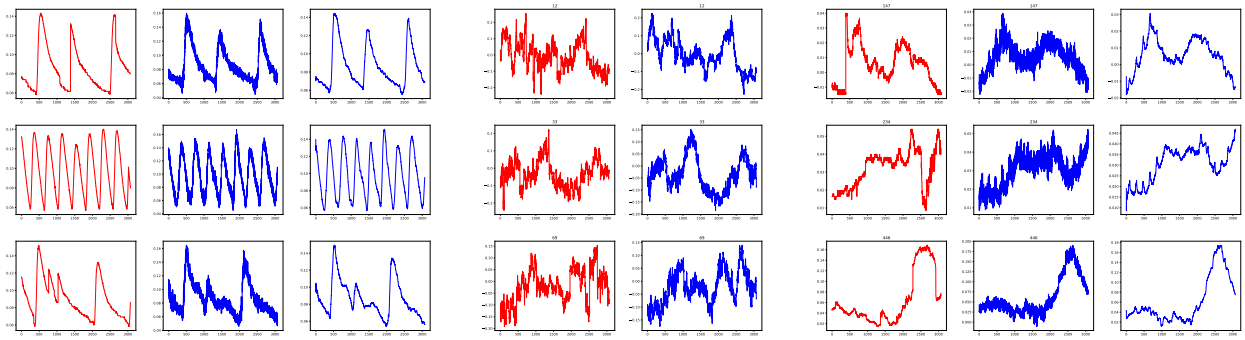
TS-Benchmark is implemented in Java. The whole architecture of the tool is depicted in Figure 4. Each module is described in brief as follows. The data generation module generates the basic dataset (for data loading). The load module tests the loading capability of the target systems, which imports the local batch file to target databases. The query generation module generates queries and submit them to the query module. The query module then executes the queries against the target databases. The inject module directs the data stream to the target systems. The "database adapter" module hides the difference of various target systems by providing a unified interface. We can test a new database by implementing a driver which inherits the interface.

IV. Experiments and Results

A. Systems under Test

To verify the TS-Benchmark, we test and compare 4 TSDBs: InfluxDB V1.7, TimescaleDB V1.2.1, Druid V0.13 and OpenTSDB V2.3. The cluster version of InfluxDB is not free, and TimescaleDB has no cluster version. To fairly compare them, we test single-node versions. The configuration of these TSDBs are given on the TS-Benchmark website.

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.



(a) shape 1 real (b) shape 1 basic (c) shape 1 synthetic (d) shape 2 real (e) shape 2 synthetic (f) shape 3 real (g) shape 3 basic (h) shape 3 synthetic

Fig. 5: Examples of both real (red) and synthetic (blue) fragments generated by the data generation model. (a), (d) and (f) are the real sequences of shape 1, shape 2 and shape 3. (b), (e) and (g) are the corresponding generated sequences. (c) and (h) are results of (b) and (g) when Kalman filter is applied.

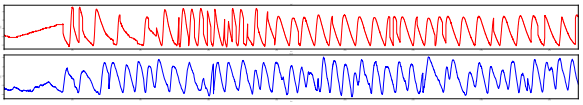


Fig. 6: Real (red) and synthetic (blue) time series of shape 1.

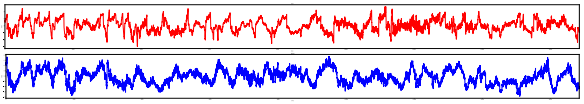


Fig. 7: Real (red) and synthetic (blue) time series of shape 2.

B. Experiment Environment

The databases are installed on a server with two Intel(R) Xeon(R) CPUs E5-2620 @ 2.00GHz (12cores), 32GB of memory, 2TB of 7200rpm SAS Hard disk. An operating system of CentOS 7.6 is installed on the server.

We have another server with the same hardware configuration to run the benchmark tool. Two machines are on the same LAN, connected via a 10 Gbit/s Ethernet. They work as a C/S model. The network bandwidth is sufficient. We choose Prometheus [2] and Grafana [30] as the monitoring tools for the tested systems.

The whole test follows such a procedure. Firstly, a base generated dataset is loaded into the target database, the throughput and compression ratio of data loading are measured. Secondly, we continuously generate streaming time series and run the data injection test against the target database with various number of farms. We increase the number of farms until the target database gets saturated, from which the throughput is measured. Thirdly, we run six different queries of the benchmark to evaluate the query performance based on query latency. The results of the above-mentioned tests are described in details in the following subsections.

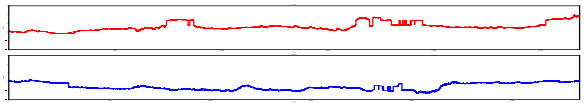


Fig. 8: Real (red) and synthetic (blue) time series of shape 3.

C. Results of Data Generation

First, we use sliding windows of size 500 to chop seed time series fragments from real data. Then, the data generation model is trained based on the extracted fragments to generate a large number of synthetic fragments. Finally, Kalman filter will be applied to those synthetic sequences with high frequency jitter. The synthetic sequences of some typical time series is shown in the Fig. 5.

In other general literature of time series generation, most of the GAN models used are based on LSTM and CNN [13] [40], i.e., the generator is LSTM, and the discriminator is CNN, or both the generator and the discriminator are LSTM. However, when applying this kind of GAN model, we find that they do not perform well because the generated fragments are almost noisy, especially for those non-periodical time series. For this reason, we only show the results of DCGAN-based model.

Synthetic streaming time series are generated based on the directed graph of synthetic fragments. Figure 6-8 show some real time series (in red) and synthetic ones (in blue) respectively. Basically, the synthetic time series are very similar to those real ones in terms of general shapes. Note that the throughput of data generation is very high because the directed graph of synthetic fragments are in memory and the generation process is a simple random walk process. We therefore do not conduct experiments on the throughput of data generation.

D. Results of Data Loading

The local data files are imported via the built-in import tool of the respective database. The databases are restarted before data import to guarantee no cache is used. The performance of data loading using the first dataset is shown in Table II. It can be seen that InfluxDB has the best compression ratio which is 68%. This is because InfluxDB uses different compression algorithms for different data types, such as Facebook Gorilla [33] for float, simple8b [28] encoding for timestamp, snappy [20] for string and so on. However, InfluxDB's batch file import throughput ranks third. Druid also applies a similar

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Loading performance under dataset 1 (3.9 GB)

Database	Throughput (points/s)	Throughput (MB/s)	Compression Ratio
InfluxDB	69,722	0.660	68%
TimescaleDB	124,138	1.175	118%
Druid	142,339	1.346	135%
OpenTSDB	38,652	0.366	366%

TABLE III: Loading performance under dataset 2 (284.5 GB)

Database	Throughput (points/s)	Throughput (MB/s)	Compression Ratio
InfluxDB	66,215	0.610	69%
TimescaleDB	1,093,302	10.072	102%
Druid	133,016	1.225	111%
OpenTSDB	NULL	NULL	NULL

strategy for different storage methods of different data types. However, Druid need a permanent backup storage called "deep storage" for distributed file system. Since each segment is replicated in deep storage, the storage space consumption is enlarged [39]. Druid has slightly better throughput and compression ratio than TimescaleDB for the first dataset. The throughput of TimescaleDB is 123,138 points/s, and the throughput of Druid is 142,339 points/s. OpenTSDB has the lowest throughput, and its data compression ratio is a bit poor. OpenTSDB's data storage space is expanded to 3.6 times of the original data size. The storage of OpenTSDB is based on HBase [19], which applies a key-value storage model. We find that HBase does not compress well when storing the name of "metric". Each data cell is stored as a key-value pair, and the same key is stored for many times, so they consume a lot of extra storage space.

When turning to the second dataset (Table III), InfluxDB still holds the best compression ratio, which is 69%. However, the throughput of TimescaleDB is far ahead of that of Druid under the second dataset. It is almost an order of magnitude higher than that of Druid. The loading tool of TimescaleDB is specially designed, called "timescaledb-parallel-copy". For both data sets, we use the single-thread import tool. The "timescaledb-parallel-copy" maps the data file to the table in database directly, which close the transaction processing functionality and pauses the index synchronization functionality. It automatically uses more resources and threads to improve the import speed. Therefore, the throughput of TimescaleDB is much better than that of the others under large dataset.

OpenTSDB fails to import the dataset 2. HBase always crashes when the data has not been imported successfully. Our experiments all run on a single-node version of the database, which may not be fair to OpenTSDB. The underlying storage system of OpenTSDB, HBase, is originally designed for distributed architecture. Since the built-in import tools of the databases are different in implementation, the time taken to read the data file is different. So it cannot reflect the true throughput performance of the databases. However, the results provide a reference for offline data loading tasks of these TSDBs.

E. Results of Data Injection

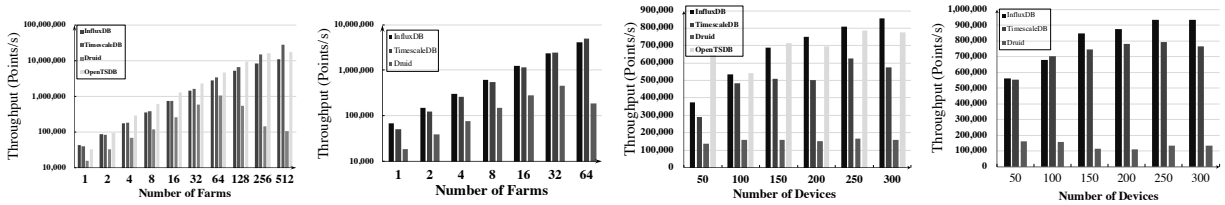
The results of data injection are summarized in Fig. 9a and Fig. 9b. We gradually enlarge the number of threads/connections (i.e. scale factor) to increase the injection pressure on target systems. On each threads level, we run the injection for 35 seconds to measured an average injecting throughput in terms of points per second. The threads are created before the test, and put in a thread pool. When more threads are needed, they are fetched from the pool and resume running immediately. The delay is minimized. We also monitored the resource usage during data injection, including CPU utilization, memory usage, as well as the I/O bandwidth usage. The results are shown in Fig. 10.

When the base data size is small (Fig. 9a), OpenTSDB maintains the best throughput from 2 to 256 threads (farms), which varies from 93,931 points/s to 16,199,638 points/s (almost linearly to the number of threads in most cases). However, OpenTSDB goes into saturation when the number of threads reaches 256. When the number of threads increases to 512, its throughput will hardly increase. When the number of threads is no more than 32, InfluxDB's throughput closely follows TimescaleDB. As the number of threads increases, the throughput of InfluxDB is about 80% of the throughput of TimescaleDB. However, when the number of threads increases from 128 to 256, InfluxDB is nearly saturated and reach saturation in 512 threads. The throughput of TimescaleDB has been increasing with the enlargement of the number of threads. TimescaleDB is not saturated when the number of threads reaches 512. It achieved the highest throughput of 27,765,420. Druid achieves the lowest throughput and less performance than the other three databases. It saturates when the number of threads is 64, and the throughput it achieves is only 1,032,805 points/s.

However, when the data size changes from dataset 1 to dataset 2, Druid's write performance drops significantly. It is unable to handle large-scale write requests when the number of threads is up to 64. InfluxDB performs slightly better than TimescaleDB when the number of threads is less than 32. When the number of threads increases to 32 and more, TimescaleDB's write performance surpasses InfluxDB.

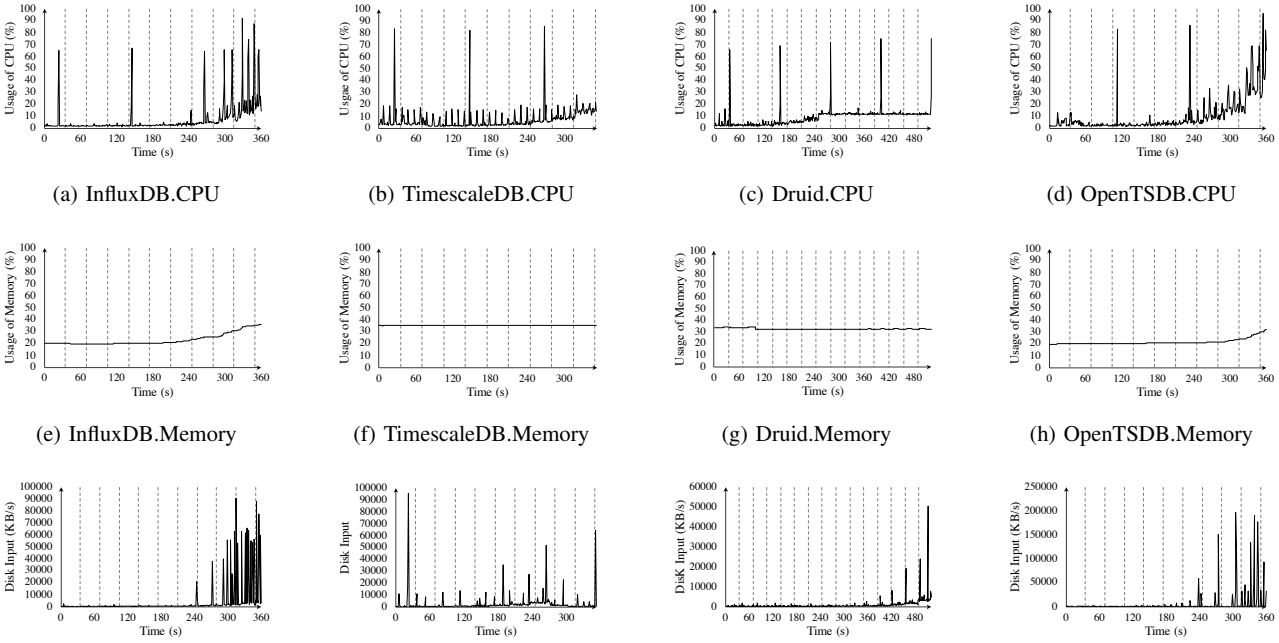
When the amount of data sent by each thread increases, in order to observe the impact to the throughput of each database, we perform a test of increasing the amount of data transmitted while fixing the number of threads. Results can be seen from Fig. 9c and Fig. 9d. When the number of writing threads is fixed as 8, we increase the size of the written data from 50 devices per thread (farm) to 300 devices per thread. The step size added for each level is set to 50. The test time for each level also lasts for 35 seconds. As the data size of each thread increases linearly, the throughput of the four databases will increase as well. However, the write capacity of the database will gradually approach the saturation. InfluxDB's performance is the best when tested under both datasets. For dataset 1, InfluxDB is not saturated when the number of devices is up to 300. OpenTSDB ranks in the second place. In saturation situation, the throughput of OpenTSDB is about 10% worse than that of InfluxDB. TimescaleDB is in third place for the small dataset. The saturated throughput of TimescaleDB is about 80% of that of InfluxDB. Druid is in the

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.



(a) Appending throughput under dataset 1 (3.9 GB)
(b) Appending throughput under dataset 2 (284.5 GB)
(c) Appending under dataset 1 (3.9 GB, 8 threads)
(d) Appending under dataset 2 (284.5 GB, 8 threads)

Fig. 9: Results of data injection



(a) InfluxDB.CPU
(b) TimescaleDB.CPU
(c) Druid.CPU
(d) OpenTSDB.CPU
(e) InfluxDB.Memory
(f) TimescaleDB.Memory
(g) Druid.Memory
(h) OpenTSDB.Memory
(i) InfluxDB.Input
(j) Timescale.Input
(k) Druid.Input
(l) OpenTSDB.Input

Fig. 10: CPU usage, memory usage and disk input bandwidth of four databases under dataset 1 (3.9 GB) during data injection
Note: Data injection test increases the number of threads. Each update of the number of threads lasts for 35 seconds during data are injected into the database. Each thread manages data of one farm.

last place. When the amount of data is small, its throughput increases slightly by 23% with the enlargement of the number of devices. When the test is conducted under the large dataset, the final throughput of Druid drops by 19%.

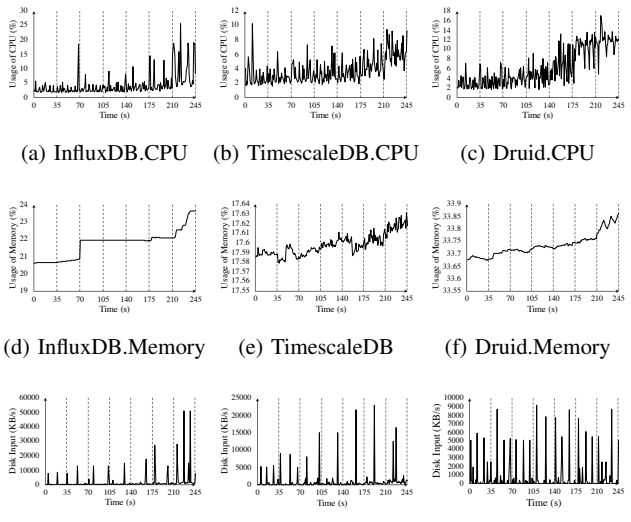
We modify the batch size to perform injection tests under different batch sizes. Instead of fixing the update interval as 7 seconds, we adjust the time interval of injecting data, so that it is set from 0.1 seconds to 14 seconds. Accordingly, the batch size increases from 35 points per batch to 5000 points per batch. This allows the rate (rows per second) of data points injected into database to be roughly equal. The throughput of each TSDB under different batch sizes is reported in Table IV. In general, as the amount of data written in each batch increases, the throughput of the four TSDBs increases too. When the data injection interval is short and the batch size is small, InfluxDB has the highest throughput. The throughput of TimescaleDB and OpenTSDB is less than that of InfluxDB, followed by Druid. As the amount of data in each batch increases, the throughput of TimescaleDB gradually surpasses that of InfluxDB. One interesting point is that the throughput is not in proportion to the amount of data written per batch, simply because an appending operation has some other overheads such as network protocols.

TABLE IV: Appending throughput under different batch sizes

interval (sec.)	# of rows	Throughput (points/s)			
		InfluxDB	TimescaleDB	Druid	OpenTSDB
0.1	35	6,392	5,973	2,083	5,482
1	350	10,470	8,362	7,692	8,978
7	2,500	43,427	41,136	30,981	31,972
14	5,000	81,036	83,003	31,271	78,367

As a relational database, TimescaleDB inherits the excellent high-concurrency write capability of PostgreSQL. TimescaleDB creates adaptive time “chunks” to maintain stable write performance. The size of chunks is based on the data volume. Different chunks have different time intervals or metrics, which enable better parallelization. This also helps

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.



(a) InfluxDB.CPU (b) TimescaleDB.CPU (c) Druid.CPU

(d) InfluxDB.Memory (e) TimescaleDB (f) Druid.Memory

(g) InfluxDB.Input (h) TimescaleDB (i) Druid.Input

Fig. 11: CPU usage, memory usage and disk input bandwidth of four databases under dataset 2 (284.5 GB) during data injection

to save CPU usage, and lead to less memory swapping. For TimescaleDB, the CPU and memory usage have been relatively stable when increasing the number of threads, with a slight rise, except for occasional spikes in CPU usage, as shown in Fig. 10b. Based on the design of BigTable [8], OpenTSDB also takes advantage of the excellent write capabilities of the LSM tree of HBase, and therefore it demonstrates excellent write performance on small datasets. Besides, data is not compressed immediately when stored in HBase. Occasional compaction operations cause huge disk input bandwidth cost, as shown in Fig. 10l. However, for OpenTSDB, when the number of threads is 256, the system's CPU resources are almost exhausted, as shown in Fig. 10d. InfluxDB's excellent write performance lies in its unique design of time-structured merge tree and the backpressure mechanism. For each thread, InfluxDB has the highest write performance. The TSM tree of InfluxDB preserves the excellent write capability of the LSM tree [32] while building indexes for data files, specifically for high concurrent writes. However, when InfluxDB's data writes are near the saturation case, the system's CPU resource usage often arrives peaks, even close to 100%, as shown in Fig. 10a. Under dataset 2, InfluxDB also uses more CPU resource, as shown in Fig. 11a. From Fig. 10d or h, we can see that Druid's system resource usage under high concurrency has some peaks, but not so much. Druid's stream data processing relies on tranquility, and we find that tranquility does not perform well in high concurrency situations. This may cause a bottleneck in Druid's write performance.

F. Results of Queries

For the four groups of queries, we run each query for 10 times with different query parameters, and measured the average latency of each query. The results of queries over two datasets are shown in Table V and VI respectively. OpenTSDB does not support queries with thresholding clause, so the response time for Query 1.2 is set to a NULL value. We analyze the result of each query individually as follows.

For Query 1.1, as we can see, InfluxDB has the fastest response time under the two datasets. When the data size is small, TimescaleDB and Druid have similar query latency. However, Druid's response time is obviously worse than TimescaleDB under the dataset 2. In contrast, OpenTSDB's latency ranks the last. Query 1.1 specifies all data dimensions, and InfluxDB performs the best on such queries.

For Query 1.2, we can observe that Druid has the fastest response time under the dataset 1, while InfluxDB instead performs the best under the dataset 2. The third place in speed is achieved by TimescaleDB, whose response time increases significantly under the dataset 2. Query 1.2 has a threshold specified in the where clause. In general, InfluxDB performs the best on this query, followed by Druid.

For Query 2.1, InfluxDB has the fastest response time under the two datasets. OpenTSDB's response time ranks the third place under the dataset 1. Druid is fast under the dataset 1, while the response time of Druid increases significantly under the dataset 2. We know that Query 2.1 fetches data of certain sensors. Some databases do not indexes data based on their sources (e.g., sensor IDs). Comparatively, InfluxDB performs the best on such queries.

For Query 2.2, Druid has the fastest response time under two datasets. TimescaleDB is faster than InfluxDB under the dataset 1, while InfluxDB is obviously faster under the dataset 2. OpenTSDB's response time is in third place under the dataset 1. Query 2.2 specifies the set of sensors to be fetched, and Druid performs best on such queries.

For Query 3, Druid has the fastest response time under the two datasets, followed by TimescaleDB and OpenTSDB. The performance of InfluxDB is poor when serving the Query 3. Query 3 is a full data query without filtering conditions, which tests the performance of the databases to scan the data table.

For Query 4, Druid also has the fastest response time under the two databases, followed by TimescaleDB and OpenTSDB. InfluxDB spends the most time on Query 4. Query 4 fetches sensor data multiple times, with time range conditions of increasing start-timestamps.

Overall, InfluxDB's query response time has an absolute advantage in Query 1.1, Query 1.2, and Query 2.1. Druid has clear advantages in Query 2.2, Query 3 and Query 4.

InfluxDB and Druid are outstanding in query performance because of the unique data storage format and the way the indexes are created for time series data, especially the secondary index. For InfluxDB, data in different time periods is divided into different data blocks and stored in columnar form. At the same time, InfluxDB also benefits from its unique TSM indexing mechanism. When a query is issued, the files covered by the query time period are first retrieved. The TSM index is then used for binary search to find the data block. Druid separates the data in different time periods, and the storage unit is called "segment". At the same time, Druid also builds a bitmap index for the columns [7], which also plays a

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

TABLE V: Query latency under dataset 1 (3.9 GB)

Query	Response Time (millisecond)			
	InfluxDB	TimescaleDB	Druid	OpenTSDB
Query 1.1	22	63	65	117
Query 1.2	146	422	69	NULL
Query 2.1	387	1,861	1,102	1,666
Query 2.2	367	56	17	204
Query 3	2,946	297	118	376
Query 4	103,110	9,358	4,248	14,061

TABLE VI: Query latency under dataset 2 (284.5 GB)

Query	Response Time (millisecond)			
	InfluxDB	TimescaleDB	Druid	OpenTSDB
Query 1.1	242	5,977	2,833	NULL
Query 1.2	357	19,683	3,741	NULL
Query 2.1	4,020	5,681	34,331	NULL
Query 2.2	590	38,977	62	NULL
Query 3	11,209	1,039	298	NULL
Query 4	381,106	37,565	11,026	NULL

significant role in improving the query performance, especially in queries with filters.

Although TimescaleDB has blocks of data for different time periods called "chunk", when handling queries with filtering conditions under large amounts of data, the disadvantages of row storage is obvious. At the same time, TimescaleDB can not automatically create an index for the columns. The data in OpenTSDB is stored on HBase. HBase is an excellent NoSQL database. It also takes the advantage of columnar storage. However, when OpenTSDB receives the query request, it will process it and send it to HBase for query results. After HBase processes the request, it returns results to OpenTSDB, and OpenTSDB returns them to the user. During the process, a lot of time is wasted on the transmission (e.g., data serialization/de-serialization), resulting in poor performance of the query of OpenTSDB.

V. Conclusion

In this paper, we propose a new benchmark for TSDBs called TS-Benchmark. It is a tool to test the performance of TSDBs majorly in IoT analytic application scenarios. We define metrics which are more suitable for evaluating the performance of TSDBs. We build the benchmark tool majorly based on a real-world use case of time series data management for wind farms. We apply the benchmark to compare databases' performance including batch file loading, highly concurrent streaming data appending, and query performance. We conduct experiments on four typical TSDBs with the proposed TS-Benchmark, and analyze the results.

TS-Benchmark's contributions include the following:

1) TS-Benchmark simulates workloads on TSDBs in real world IoT applications, i.e. wind farm data management and operation analysis. After careful analysis of real-world applications, we clearly separate two kinds of jobs, i.e. jobs of upper-layer software such as pattern matching, which has nothing to do with database performance, and jobs should be done by TSDB, i.e. how data is going into and coming out of the database. Testing of such jobs should tell us how good each database is persisting and serving the data. TS-Benchmark is a comprehensive benchmark in terms of workloads, the workloads include data loading, data injection, and data fetching in analytic applications. Loading is different from injection, some existing benchmark tools such as "InfluxDB comparison" test data loading, however it does not test continuous data injection. TS-Benchmark is designed to test target TSDB under different concurrency so that weak points and strong points of the system under test can be fully exposed. It is worth to mention that, TS-Benchmark is not designed for streaming processing systems, which process data in a streaming manner. It is used to test data operations on TSDBs.

2) Based on DCGAN, we design a data generation model to produce different scale of high-quality synthesized dataset in high throughput, after trained with real world seed data.

3) TS-benchmark can be easily adapted to test representative TSDBs such as InfluxDB, TimescaleDB, Druid and OpenTSDB. InfluxDB is a native TSDB using Time-Structured Merge Tree (a variant log-structured merge-tree) in its storage engine. TimescaleDB, which is based on PostgreSQL, represents databases using row-wise storage. Druid stores data in columnar format, and builds indexes during data injection. OpenTSDB is based on HBase, which is a popular column family NoSQL database using key-value data model inside, in essence it represents key-value databases.

4) We analyze the experiment results and obtain some preliminary yet lightening conclusions as follows: I) InfluxDB enjoys high injection throughput and high query throughput, owing to the special storage technique it uses, i.e. time-structured merge tree supporting high speed writing. Since an index is built during writing, it also enjoys fast reading. II) in Druid, a component called tranquility splits data into segments and builds bitmap index during pre-processing, it does not support high speed data injection, Druid stream processing (tranquility) also does not perform well under high concurrency. However columnar storage format and indexes pay off later. Druid achieves high query throughput, outperforming InfluxDB in some cases. III) TimescaleDB achieves poor query performance, since it uses a row-wise storage format inherited from PostgreSQL. The poor performance is also due to the reason that TimescaleDB does not create any indexes automatically. As for continuous data injection, TimescaleDB creates adaptive time "chunks" to achieve very high write performance. IV) Although HBase also takes the advantage of columnar storage, overall it does not achieve comparable query performance with InfluxDB and Druid. The poor performance is partly due to the reason that frequent serialization and de-serialization between underlying HBase and upper layer of OpenTSDB during query processing. The poor performance is also partly due to the fact that redundant storage of the key (column family, column qualifier) of the cells. HBase is designed to run on large-scale clusters, not on a single node. We also monitored resource usage during continuous data injection, InfluxDB consumes much resource when data writes are near the saturation level, CPU usage is close to 100%. The CPU and memory usages of TimescaleDB have been relatively stable with a slight rise when increasing the number of threads,

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.

except for occasional spikes in CPU usage. Data compaction in HBase causes huge disk input bandwidth cost occasionally. Resource usage of Druid under high concurrency has some peaks, but not too much.

For the workload of batch file import, InfluxDB has the best compression performance, and TimescaleDB has the highest import efficiency. For the workload of data injection, OpenTSDB has the best write performance, while TimescaleDB performs better at high concurrency. InfluxDB has the highest performance for each thread. For the query workloads, InfluxDB and Druid handle queries with filters and time intervals more quickly than the other two.

We also find that some unique designs of the TSDBs bring performance improvements. Time series data usually includes multiple types. Different compression strategies for different types of data lead to better data compression ratios. A good compression strategy can effectively save space on the hard disk and reduces the I/O cost of the disk, although it may slightly increase the serialization/de-serialization cost. Considering the write performance of the database, the LSM tree has a significant improvement in the concurrent write performance of the database. However, at the same time it also loses some of the read performance and consumes more system resources. For non-relational databases, the improvement of LSM tree may be a good way to improve write performance. For relational databases such as TimescaleDB, changing the size of data blocks adaptively is also a good way to improve the write performance and resource usage. For the queries, data chunking according to the time interval can save a lot of time for the range queries on the time dimension. Time series data usually includes many dimensions, designing a columnar storage for the data, and creating a secondary index can result in significant improvements of the query performance.

VI. Acknowledgment

This work is supported by National Key Research and Development Program (No. 2016YFB1000702) and the National Science Foundation of China under the grant U1711261.

References

[1] P. L. Anderson, M. M. Meerschaert, and K. Zhang. Forecasting with prediction intervals for periodic autoregressive moving average models. *Journal of Time*, 34(2):187–193, 2013.

[2] C. Anglano, M. Canonico, and M. Guazzone. Prometheus: A flexible toolkit for the experimentation with virtualized infrastructures. *Concurrency and Computation: Practice and Experience*, 30(11), 2018.

[3] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, and et al. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[4] M. F. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. Iotabench: an internet of things analytics benchmark. In *ACM/SPEC*, pages 133–144, 2015.

[5] A. Bader, O. Kopp, and M. Falkenthal. Survey and comparison of open source time series databases. In *BTW*, pages 249–268. GI, 2017.

[6] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, and et al. Analytics in motion: High performance event-processing AND real-time analytics in the same database. In *ACM SIGMOD*, pages 251–264, 2015.

[7] S. Chambi, D. Lemire, R. Godin, K. Boukhalfa, C. R. Allen, and F. Yang. Optimizing druid with roaring bitmaps. In *IDEAS*.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, and et al. Bigtable: A distributed storage system for structured data. In *OSDI'06*.

[9] Y. Cheng, P. Ding, T. Wang, W. Lu, and X. Du. Which category is better: Benchmarking relational and graph database management systems. *Data Sci. Eng.*, 4(4):309–322, 2019.

[10] T. P. P. Council. Tpc benchmark c standard specification, 1990.

[11] T. P. P. Council. Tpc benchmark h (decision support). *Standard Specification, Revision*, 1(0), 1999.

[12] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, and et al. The ucr time series classification archive, October 2018. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.

[13] A. M. Delaney, E. Brophy, and T. E. Ward. Synthesis of realistic ECG using generative adversarial networks. *CoRR*, abs/1909.09150, 2019.

[14] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[15] A. S. Foundation. Apache storm. https://storm.apache.org/, 2020.

[16] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and et al. Bigbench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208, 2013.

[17] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *SOSP, 2003*.

[18] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, and et al. Generative adversarial networks. *CoRR*, abs/1406.2661, 2014.

[19] Google. Hbase. https://hbase.apache.org/, 2020.

[20] Google. snappy. http://google.github.io/snappy/, 2020.

[21] P. G. D. Group. Postgresql. https://www.postgresql.org/, 2020.

[22] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE*, pages 41–51, 2010.

[23] N. Iftikhar, X. Liu, S. Danalachi, F. E. Nordbjerg, and J. H. Vollesen. A scalable smart meter data generator using spark. In *OTM, CoopIS, C&TC, and ODBASE*, pages 21–36. Springer, 2017.

[24] Influxdata. Influxdb-comparisons. https://github.com/influxdata/influxdb-comparisons. Accessed 2020.

[25] InfluxData. Influx documentation. https://docs.influxdata.com/influxdb/v1.7/concepts/storage_engine/, 2020.

[26] Influxdata. Influxdb. https://www.influxdata.com/, 2020.

[27] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time series management systems: A survey. *IEEE TKDE*, 29(11):2581–2600, 2017.

[28] Jwilder. simple8b. https://github.com/jwilder/encoding/tree/master/simple8b, 2019.

[29] L. Kegel, M. Hahmann, and W. Lehner. Feature-based comparison and generation of time series. In D. Sacharidis, J. Gamper, and M. H. Böhlen, editors, *SSDBM*, pages 20:1–20:12. ACM, 2018.

[30] G. Lab. Grafana. https://grafana.com/, 2020.

[31] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *UCC*, pages 69–78, 2014.

[32] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[33] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and et al. Gorilla: A fast, scalable, in-memory time series database. *PVLDB*, 8(12):1816–1827, 2015.

[34] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[35] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

[36] A. Shukla, S. Chaturvedi, and Y. Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21), 2017.

[37] timescale. Time series benchmark suite. https://github.com/timescale/tsbs, 2020.

[38] Timescale. Timescaledb. https://www.timescale.com/, 2020.

[39] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: a real-time analytical data store. In *SIGMOD*, pages 157–168, 2014.

[40] F. Ye, F. Zhu, Y. Fu, and B. Shen. ECG generation with sequence generative adversarial nets optimized by policy gradient. *IEEE Access*, 7:159369–159378, 2019.

[41] G. P. Zhang. Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing*, 50:159–175, 2003.

Authorized licensed use limited to: JILIN UNIVERSITY. Downloaded on September 23,2021 at 07:25:57 UTC from IEEE Xplore. Restrictions apply.