TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications [Technical Report]

This report serves as a supplement to our original paper. It describes additional experiments and results that provide further insights into the performance and capabilities of the time series database systems under evaluation. These experiments were conducted using the same methodology as described in the original paper, ensuring comparability to the results reported in the paper.

In what follows we describe in turn the query performance for more advanced queries in section 1, the workloads setup in section 2, the parameterization of our data generation in section 3 and the systems' selection in section 4.

1 Performance of Additional Queries

Our benchmark focuses on fundamental queries which represent key operations that form most of the complex time series analytics workloads. 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. This approach offers a systematic understanding of TSDBs and helps us reason about their performance and predict their behavior when subjected to queries with more complex access paths.

We have implemented several queries with various levels of complexity inside the benchmark; any additional query can seamlessly be integrated. Our results show that the combination of simple operations and the size of the output are the bottleneck of TSDBs making their behavior predictable for any new query. Our extensive evaluations show that TSDBs are suitable for data storage, simple processing, and data access more than complex analytical tasks, which are commonly implemented outside of the database.

A challenge we encountered while considering complex queries was the lack of support for common operators on certain systems to effectively implement complex logic using SQL or the corresponding system's native language. During the development of the benchmark, we considered additional queries with increased complexity as well as UDFs. In this section, we illustrate systems query performance for more complex queries and for UDFs.

1.1 SQL Queries

We implemented more complex queries that use different types of aggregations. We evaluate the queries performance on the D-LONG dataset, running each 100 times and reporting the average execution time.

Q8: Distance-based Similarity Search. This query employs a distance-based similarity search to identify stations with two sensor readings (si, sj) similar to an input pair $(?x, ?y)$ within a specified time window. The query fetches data from the sensors, performs distance computations, and orders the results. Figure 1 depicts the results for this query.

```
WITH distance_table AS (                                    Q8
  SELECT st_id,
    SQRT(
      POWER(si - ?x, 2) + POWER(sj - ?y, 2)
    ) AS distance
    FROM ts_table
    WHERE time < ?timestamp
    AND time > ?timestamp - ?range
)
SELECT st_id, distance
FROM distance_table
ORDER BY distance ASC
LIMIT 5;
```

Q9: Dynamic Time Warping. This query computes Dynamic Time Warping (DTW) [1, 11, 15] using a self-join to calculate the Euclidean distance between each pair of adjacent values. It then uses a rolling window to sum up the distances between adjacent values of a given sensor s and defines a new segment whenever the distance exceeds a threshold (in this case, 0.1). Finally, it groups the segments by the segment ID and calculates each segment's start time, end time, and average value. Figure 1 depicts the results for this query.

```
WITH distances AS (                                         Q9
  SELECT time, s, POWER(s - LAG(s, 1)
         OVER (ORDER BY time), 2) AS distance
  FROM ts_table
  WHERE st_id = ?station)
  AND time < ?timestamp
  AND time > ?timestamp - ?range
),
segments AS (
  SELECT time, s,
         SUM(distance) OVER (ORDER BY time
         ROWS BETWEEN UNBOUNDED PRECEDING
         AND CURRENT ROW) AS segment_id
  FROM distances
  WHERE distance > 0.1
)
SELECT segment_id, MIN(time) AS start_time,
MAX(time) AS end_time, AVG(s) AS avg_value
FROM segments
GROUP BY segment_id;
```

Q10: Data Normalization. This query computes the *mean* and the *standard deviation* of a given sensor then uses it to compute the Z-normalization of the given sensor within a time range. Figure 1 depicts the results for this query.

```
WITH series AS (                                        Q10
    SELECT time, s
    FROM  ts_table
    WHERE st_id in (?station)
    AND time > ?timestamp
    AND time < ?timestamp - ?range;
),
stats AS (
    SELECT MEAN(s) as series_mean,
    stddev(s) as series_stddev
    FROM series
)
SELECT value,
    (s - series_mean)/ series_stddev AS zscore
FROM
    series, stats;
```

Q11: Anomaly Detection. This query retrieves the *mean* and the *stddev* from a given column, then labels as an anomaly each value that's not between *mean - stddev* and *mean + stddev* and as a normal datapoints all points that are between these two bounds. Figure 1 depicts the results for this query.

```
WITH series AS (                                        Q11
    SELECT value
    FROM  ts_table
    WHERE st_id in (?station)
    AND time > ?time_start
    AND time < ?time_start - ?range;
),
bounds AS (
    SELECT mean(value) - stddev(value)
        as lower_bound,
    SELECT mean(value) + stddev(value)
        as upper_bound
    FROM series
)
SELECT value,
    value NOT BETWEEN lower_bound AND upper_bound
    AS is_anomaly
FROM
    series,
    bounds;
```

Q12: Autocorrelation This query calculates the autocorrelation of a given sensor reading for each station with a lag of a time period. Autocorrelation can be used to identify potential seasonality or repetitive patterns in time series data.

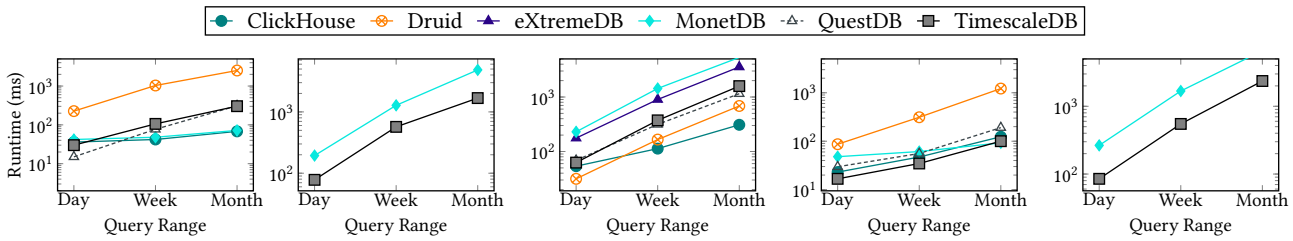
```
WITH lagged_data AS (                                   Q12
    SELECT
        id_station,
        time,
        si,
        LAG(si,10) OVER (PARTITION BY id_station
        ORDER BY time) AS si_lag
    FROM d1 where id_station='st<stid>'
    AND time > ?time_start
    AND time < ?time_start - ?range
)
SELECT
    id_station,
    time,
    (si - AVG(si)
    OVER (PARTITION BY id_station)) *
    (si_lag - AVG(si_lag)
    OVER (PARTITION BY id_station)) AS autocorr
FROM lagged_data
ORDER BY id_station, time;
```

Figure 1 illustrates the experimental results, revealing that TSDBs perform similarly for queries that share the same operations, even when the queries are more complex.

1.2 User Defined Functions

We have also implemented additional time series tasks using User Defined Functions (UDFs). We consider the following tasks:

- Anomaly detection [5]: We implement Time Series anomaly detection with HOT-SAX. The algorithm uses SUM and COUNT for the normalization part of the algorithm.
- Similarity search [13]: We implement similarity search using DSTree because it is a tree-based algorithm that answers queries fast at the cost of slow indexing. This algorithm uses only mathematical operators for single values and memory allocation operators for constructing the tree.
- Data normalization [8]: We evaluated the Z-score normalization, one of the most commonly used normalization methods, which centers the data around zero. Z-score relies mostly on statistical operators AVG and STDDEV.
- Matrix decomposition [6]: We implement the Centroid Decomposition (CD) technique, which includes a mix of matrix operations including addition, subtraction, multiplication, and transpose.
- Anomaly repair [17]: We implement SCREEN, a popular technique for anomaly repair based on speed constraints. The algorithm is incremental and relies mostly on mathematical operators such as addition, subtractions, and division.
- Pattern discovery [10]: We implement Time Series Discord discovery with Sax-Representation. The algorithm uses SUM and COUNT for the normalization part of the algorithm.



(a) Q8: Distance-based Similarity Search **(b) Q9: DTW** **(c) Q10: Data Normalization** **(d) Q11: Data Normalization** **(e) Q12: Autocorrelation**

Figure 1: Advanced Queries Results.

- Clustering [3]: We implement the classical K-Means function with a fixed number of iterations and clusters. The algorithm uses MEAN over a filtered set of datapoints.
- Classification [2]: We implement the classical KNN algorithm. This algorithm relies on mathematical operations.

,

We evaluate the tasks on the D-LONG and report the average execution time of 10 runs. Figure 2 depicts the runtime results only for the systems that support UDFs.

The results show that the runtimes of the systems are almost identical. The UDF execution step has similar runtime across systems and is the most time-consuming part of the total UDF runtime. Data fetching and format conversion times represent less than 10% of the overall UDF runtime. Format conversion is faster for some systems such as MonetDB, whose data format is similar to that of Python. These differences remain negligible compared to the total UDF execution time.

2 Workloads Setup

2.1 Systems Configuration

We ran several experiments to evaluate the impact of partitioning size on systems' behavior. We depict the results in Figures 3d and 3b. We found that partitioning by week is preferred for all the systems in our setup as it provides the optimal trade-off between inter- and intra-partition look-ups.

Most of the systems we include in this benchmark allow representing data using two storage layouts. In the wide format, each station has a single entry for each timestamp including values for multiple sensors from the station while the narrow format stores each sensor in a separate timestamp entry, resulting in multiple entries with the same timestamp. We evaluated both formats and found out that the wide layout provides faster query runtimes and better loading and compression performance as shown in Figure 4.

2.2 Online Workloads Setup

For our online workloads, we have examined the impact of other parameters in addition to the insertion rate. We have investigated how varying the number of clients influences performance [4, 16]. We concurrently inserted 1k datapoint batches into the TSDBs while increasing the number of concurrent clients and measured the insertion rate reached by each system. Our results, depicted in Figure 5, indicate that the performance of the systems –measured in data points per second (dps)– generally increases with the number of concurrent clients. ClickHouse achieves the highest ingestion rate of 8Mdps using 60 clients. InfluxDB, MonetDB, and TimescaleDB each hit their maximum insertion capacity of 4Mdps using 30 clients. However, eXtremeDB and QuestDB are notable under-performers: The former cannot reach more than 30Kdps, while the latter cannot scale beyond single-client online insertions.

We notice that ClickHouse achieves the best ingestion rate where it reaches 8Mdps under 60 clients. InfluxDB, MonetDB and TimescaleDB show similar trends where they reach their maximum insertion capacity of around 4Mdps under 30 clients. We also notice that eXtremeDB's storage model is challenged reaching rates higher than 30Kdps and that QuestDB does not support multi-client insertion.

3 Data Generation

3.1 TS-LSH Performance

In this experiment, we elucidate the process of choosing the optimal number of hash tables. The experiment consists of varying the number of hash tables and examining both the CPU time required to generate new segments and the percentage of empty lookups in the LSH. The results in Figure 6a show that using 5 hash tables results in over twice the number of empty outputs compared to using 10. This difference has a substantial impact on the quality of the generation as indicated by the contrasting RMSE values shown in Figure 6b – 0.9 versus 0.45. Using a larger number of hash tables (e.g., 25) incurs an important increase in the runtime with a marginal reduction in the number of lookups, and thus in the generation quality. Similarly to the elbow rule in clustering [9, 12], we found that determining the optimal number of hash tables for LSH is a trade-off between quality and computational efficiency.

We also chose the commonly used Euclidean distance as a similarity measure because it fits the requirement needs of our use case: i) it is fast and ii) it applies to time series with data inconsistencies such as missing values or anomalies. Other distance measures such as Dynamic-Time Warping or the Longest Common Subsequence (LCSS) would be more effective in time series with time shifts, which is not the case of our data.

3.2 Generation Performance on different seed sources

In this experiment, we examine the generalizability of our generator using various datasets, each with different features and sizes. We

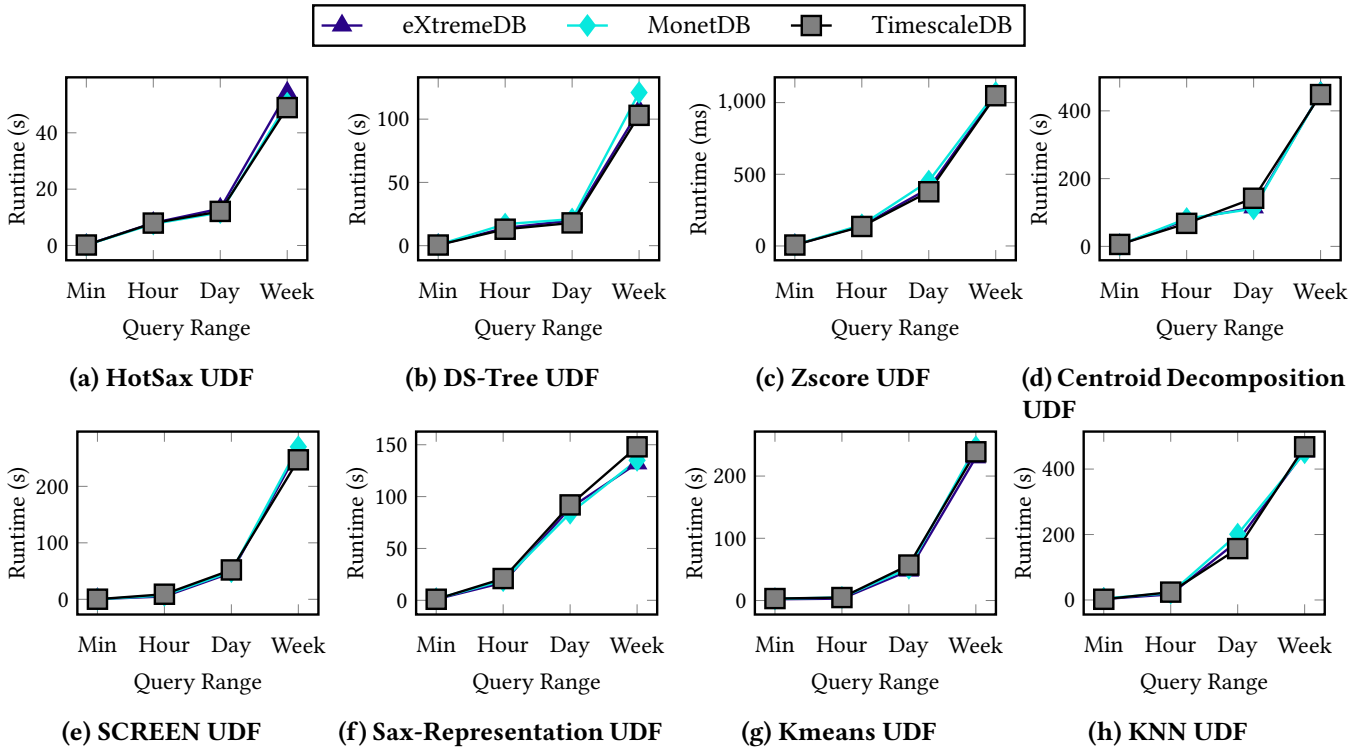


Figure 2: User-Defined Functions Results. Note that only eXtremeDB, MonetDB and TimescaleDB supported UDFs.

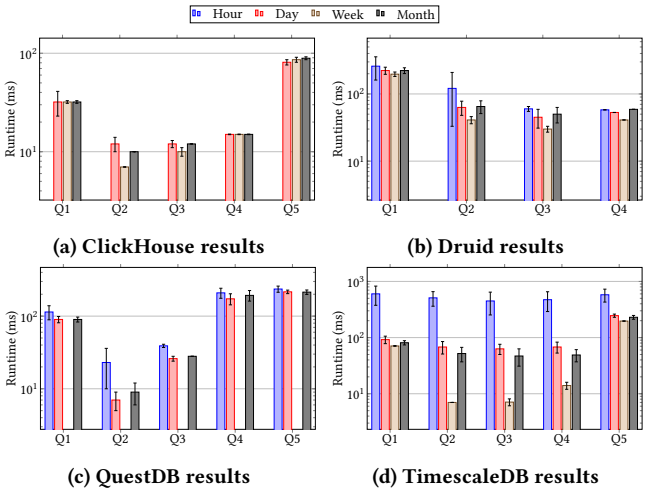


Figure 3: System partitioning. ClickHouse and QuestDB do not support partitioning by hour and week, respectively. eXtremeDB and MonetDB only support manual time partitioning, and InfluxDB has no support for partitioning.

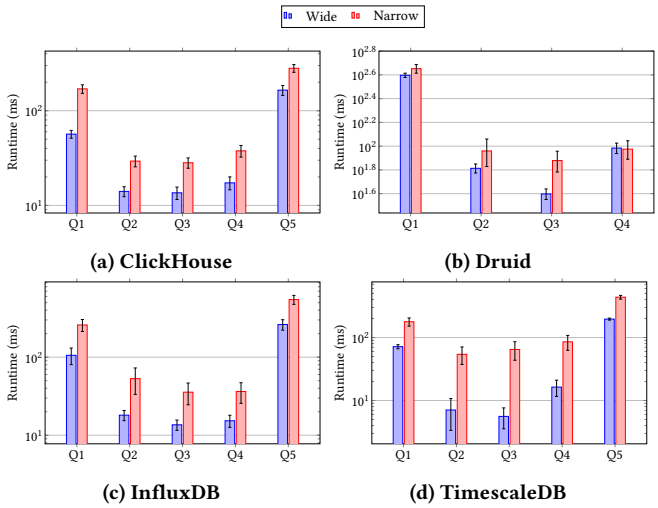


Figure 4: Impact of Storage Layout on System Performance.

show the results on six datasets extracted from one of our time series benchmarks [7]. Table 1 describes the main properties of those datasets; Complex time series datasets exhibit irregular patterns, high volatility, multiple seasonality, and the presence of outliers.

The results in Figure 7 show that our technique outperforms the state-of-the-art baseline TS-Graph in all datasets. The performance difference between our technique and the baselines depends on the properties of the datasets. For instance, in complex datasets such as BAFU, the difference goes up to 5.6x and 3x higher correlation and NMI, respectively compared to the second-best generation

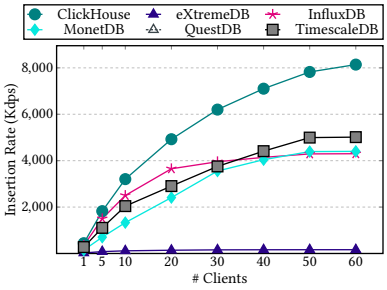
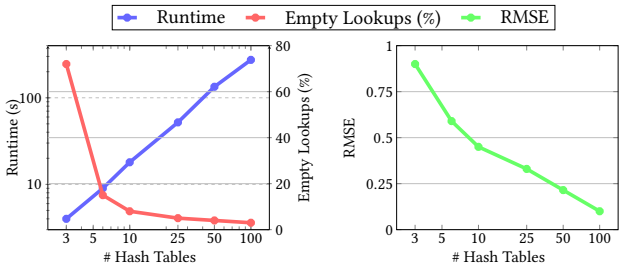


Figure 5: Insertion Performance.



(a) Runtime v. Empty Lookups (%) **(b) RMSE**

Figure 6: Impact of the number of hash tables on generation performance.

Table 1: Description of time series datasets.

Name	TS length	# of TS	Main features
BAFU	43'000	10	irregular trends
Electricity	5'000	20	time-shifted series
Gas	1'000	100	high variations
Air	1'000	10	repeating trends, sudden changes
Meteo	10'000	10	repeating trends, sporadic anomalies
Chlorine	1'000	50	cyclic

technique. In simpler datasets, such as Electricity, the difference is less pronounced but is still visible.

4 Systems Selection

The selection of systems for our benchmark considered a diverse set of factors. First, we used DB-Engines[1] as an initial selection criterion to determine the popularity of the TSDBs. DB-Engines ranks databases based on search engine popularity, social media mentions, job postings, and technical discussion volume. We selected the top-10 TSDB systems recommended by DB-Engines. We verified that most of those systems are also recommended by OSS Insight[2], which ranks the systems based on repository stars, pull requests, pull request creators, and issues.

Second, 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 other systems

[1]https://db-engines.com/en/ranking/time+series+dbms
[2]https://ossinsight.io/collections/time-series-database/

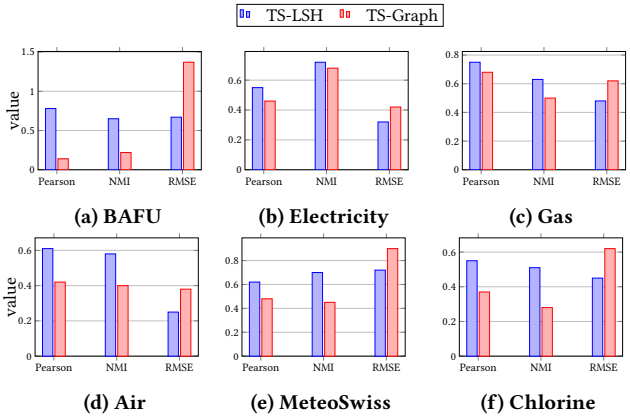


Figure 7: Data Quality of TS-LSH.

such as eXtremeDB, ClickHouse, and MonetDB, which are known in the community for their efficiency on time series data. We have amended the description of the selection criteria in section 4.3.

Next, we conducted preliminary experiments on the pre-selected systems on a subset of queries with fixed parameters. Our results led to the selection of four systems: InfluxDB, TimescaleDB, Druid, and QuestDB as the best competitors.

Lastly, to ensure that we did not dismiss any potential candidate, we checked less-performing systems from other benchmarks including OpenTSDB, KairosDB, Graphite, Apache IoTDB and Prometheus. Note that Prometheus does not support the bulk-loading features so we wrote scripts that convert the data format into its format. The runtimes for these scripts are considered as the loading time. As Tables 2 and 3 illustrate that the performance of these systems is significantly inferior to the selected systems.

Table 2: Loading and compression of discarded systems.

System	Avg. Throughput (datapoints/s)	Loading Time (s)	Storage (GB)
OpenTSDB	72'412	7'159	12.5
KairosDB	462'512	1'120	18.6
Graphite	34'454	15'046	11.8
Apache IoTDB	42'141	12'301	5.2
Prometheus	297'530	1'742	4.70

Prometheus is an open-source monitoring and alerting system that also serves as a TSDB. Prometheus support ingesting, storing, and querying time series data. It groups the time series data into blocks, each block corresponding to a directory containing a chunks subdirectory containing all the time series samples for that window of time, a metadata file, and an index file. Prometheus further supports multiple monitoring and modern alerting features.

Prometheus does not support loading historical data and does not support complex sub-queries. For data insertion, Prometheus scrapes metrics data from targets at regular intervals. We worked out a solution using data conversion scripts provided by Prometheus'

developers. Those scripts allow us to convert our historical data to the Prometheus storage format and then launch Prometheus to visualize and query the data. We consider the script's completion time as the bulk-loading runtime.

Table 2 compares the data loading performance of Prometheus with the top 3 most reliable systems of our benchmark. The results show that Prometheus is 9x slower than the average loading time while producing a higher storage size than the selected set of systems.

We also evaluated Promtheus's query performance. Table 3 shows that while Prometheus does not appear among the top 3 systems in the queries and configurations, it still shows promising results by performing better than eXtremeDB for Q5.

Table 3: Prometheus query runtime.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7
ClickHouse	30±1	10±1	13±2	13±1	**80±5**	53±2	9.72±5
eXtremeDB	34±1	**3±1**	**2±1**	**2±1**	296±1	**31±4**	14.06±2
TimescaleDB	**6±1**	6±1	12±2	12±2	95±1	62±3	**5.9±1**
Prometheus	160±49	18±1	25±3	20±3	248±7	-	-

5 Data Characteristics Impact

To evaluate the impact of different characteristics on systems performance, we generated synthetic datasets by varying a list of ten characteristics: mean of values, variance of values, mean of deltas, variance of deltas, number of repeats, count of increases, number of outliers, number of missing values, data correlation, and seasonality. This list was primarily selected from a research benchmark of data encoding techniques for time series [14] and complemented with common time series features from the literature.

In Figure 8, 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.

References

[1] 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

[2] Michele Dallachiesa, Themis Palpanas, and Ihab F. Ilyas. 2014. Top-k Nearest Neighbor Search In Uncertain Data Series. *Proc. VLDB Endow.* 8, 1 (2014), 13–24. https://doi.org/10.14778/2735461.2735463

[3] Shalmoli Gupta, Ravi Kumar, Kefu Lu, Benjamin Moseley, and Sergei Vassilvitskii. 2017. Local Search Methods for k-Means with Outliers. *Proc. VLDB Endow.* 10, 7 (2017), 757–768. https://doi.org/10.14778/3067421.3067425

[4] 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

[5] Eamonn J. Keogh, Jessica Lin, and Ada Wai-Chee Fu. 2005. HOT SAX: Efficiently Finding the Most Unusual Time Series Subsequence. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*. IEEE Computer Society, 226–233. https://doi.org/10.1109/ICDM.2005.79

[6] Mourad Khayati, Michael H. Böhlen, and Johann Gamper. 2014. Memory-efficient centroid decomposition for long time series. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 100–111. https://doi.org/10.1109/ICDE.2014.6816643

[7] 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, 768–782. https://doi.org/10.14778/3377369.3377383

[8] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *Proc. VLDB Endow.* 11, 13 (2018), 2236–2248. https://doi.org/10.14778/3275366.3275372

[9] Patrick Schäfer and Ulf Leser. 2022. Motiflets - Simple and Accurate Detection of Motifs in Time Series. *Proc. VLDB Endow.* 16, 4 (2022), 725–737. https://www.vldb.org/pvldb/vol16/p725-schafer.pdf

[10] Jin Shieh and Eamonn J. Keogh. 2008. *i*SAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, Ying Li, Bing Liu, and Sunita Sarawagi (Eds.). ACM, 623–631. https://doi.org/10.1145/1401890.1401966

[11] Machiko Toyoda, Yasushi Sakurai, and Yoshiharu Ishikawa. 2013. Pattern discovery in data streams under the time warping distance. *VLDB J.* 22, 3 (2013), 295–318. https://doi.org/10.1007/s00778-012-0289-3

[12] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Timos Sellis, and Xiaolin Qin. 2019. Fast Large-Scale Trajectory Clustering. *Proc. VLDB Endow.* 13, 1 (2019), 29–42. https://doi.org/10.14778/3357377.3357380

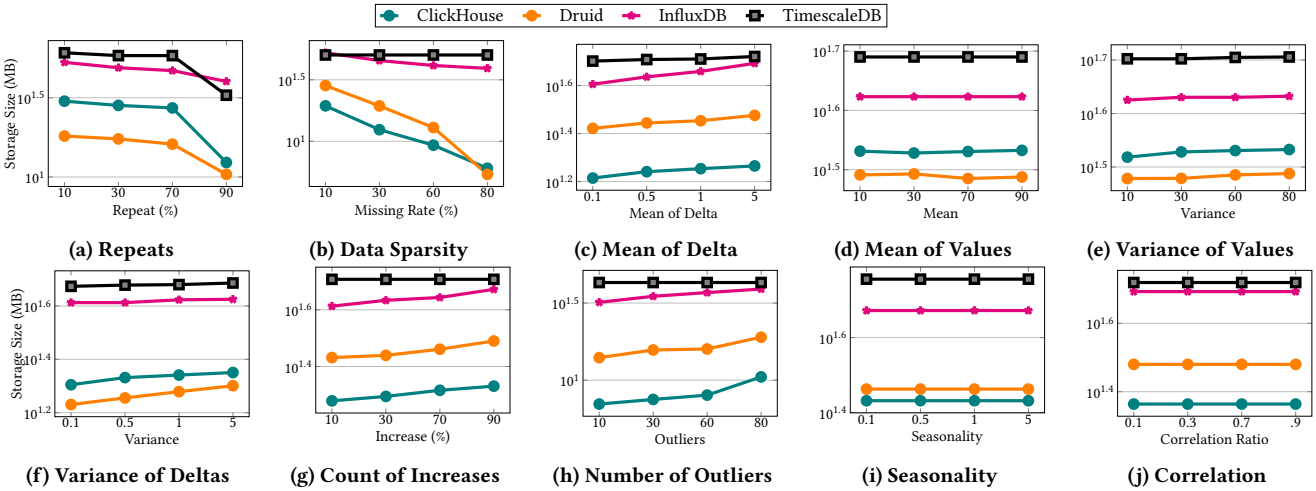
[13] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *Proc. VLDB Endow.* 6, 10 (2013), 793–804. https://doi.org/10.14778/2536206.2536208

[14] 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

[15] 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

[16] 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).

[17] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S. Yu. 2017. Time Series Data Cleaning: From Anomaly Detection to Anomaly Repairing. *Proc. VLDB Endow.* 10, 10 (2017), 1046–1057. https://doi.org/10.14778/3115404.3115410



(a) Repeats **(b) Data Sparsity** **(c) Mean of Delta** **(d) Mean of Values** **(e) Variance of Values**

(f) Variance of Deltas **(g) Count of Increases** **(h) Number of Outliers** **(i) Seasonality** **(j) Correlation**

Figure 8: Impact of Data Characteristics on Systems Storage.