TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications [Additional Experiments]

1 Advanced Queries Performance

In this section, we report the results of several advanced tasks. We will first describe the results of the SQL implementation and then the UDF one. We evaluate the query performance on the D-LONG dataset, running each 100 times and reporting the average execution time. Results are depicted in Figure 1

1.1 SQL implementation

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.

Q9: Dynamic Time Warping. This query computes Dynamic Time Warping (DTW) [1, 10, 13] 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.

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

Q11: Simple 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.

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.

1.2 User Defined Functions.

We have also implemented additional time series tasks using User Defined Functions (UDFs). We consider the following tasks: data normalization [7], matrix decomposition [5], anomaly detection [9], anomaly repair [15], clustering [3], and classification [2].

We evaluated 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.

2 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 [12] and complemented with common time series features from the literature.

In Figure 3, 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.

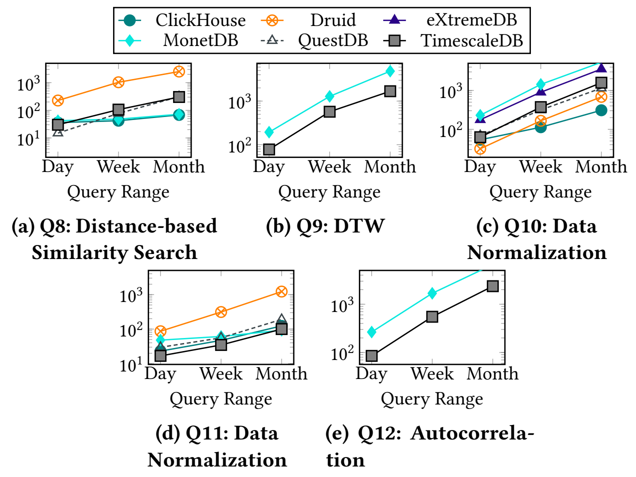


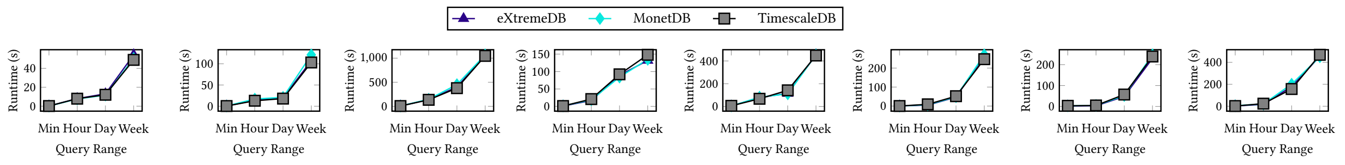
Figure 1: Advanced Queries Results.

3 Discarded Systems

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 1 and 2 illustrate that the performance of these systems is significantly inferior to the selected systems.

Table 1: Loading and compression of discarded systems.

System	Avg. Through-put (data-points/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

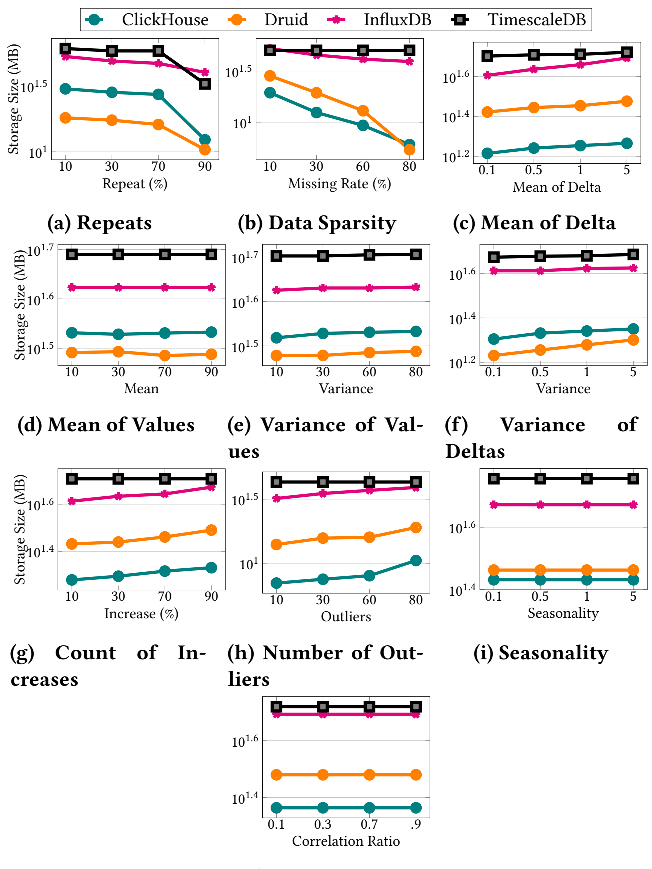


(a) HotSax UDF (b) DS-Tree UDF (c) DS-Tree UDF (d) Sax-Representation UDF (e) Centroid Decomposition UDF (f) RecovDB UDF (g) Kmeans UDF (h) KNN UDF

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

Table 2: 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	-	-



(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 3: Impact of Data Characteristics on Systems Storage.

4 Online Workloads Configuration

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, 14]. 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 4, 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.

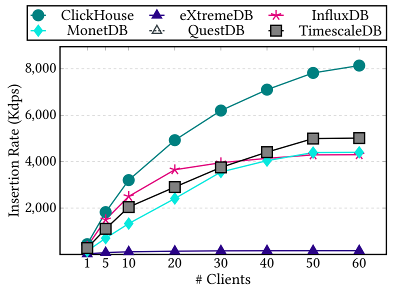


Figure 4: Insertion Performance.

5 Impact of the number of hash tables on 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 5a 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 5b – 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 [8, 11], we found that determining the optimal number of hash tables for LSH is a trade-off between quality and computational efficiency.

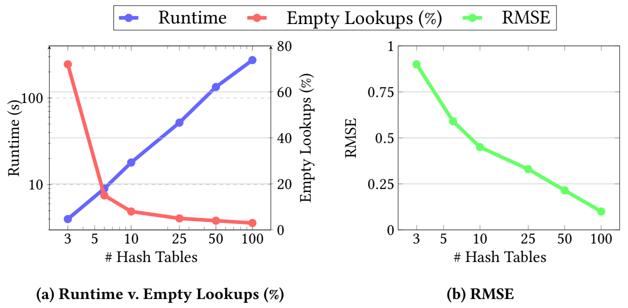


Figure 5: Impact of the number of hash tables on generation performance.

6 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 show the results on six datasets extracted from one of our time series benchmarks [6]. Table 3 describes the main properties of those datasets; Complex time series datasets exhibit irregular patterns, high volatility, multiple seasonality, and the presence of outliers.

Table 3: 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

The results in Figure 6 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 technique. In simpler datasets, such as Electricity, the difference is less pronounced but is still visible.

7 Systems Configuration

We ran several experiments to evaluate the impact of partitioning size on systems' behavior. We depict the results in Figures 7d and 7b. 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

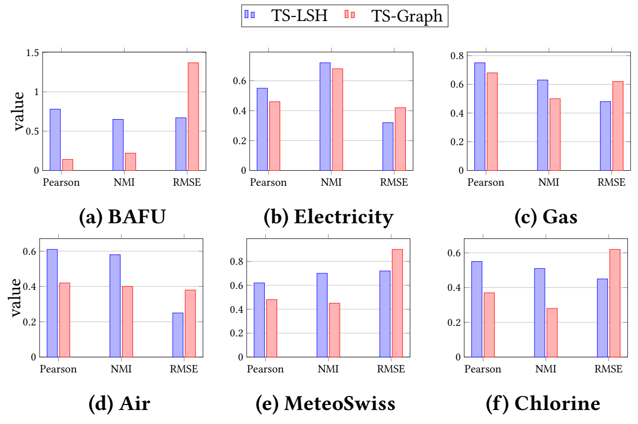


Figure 6: Data Quality of TS-LSH.

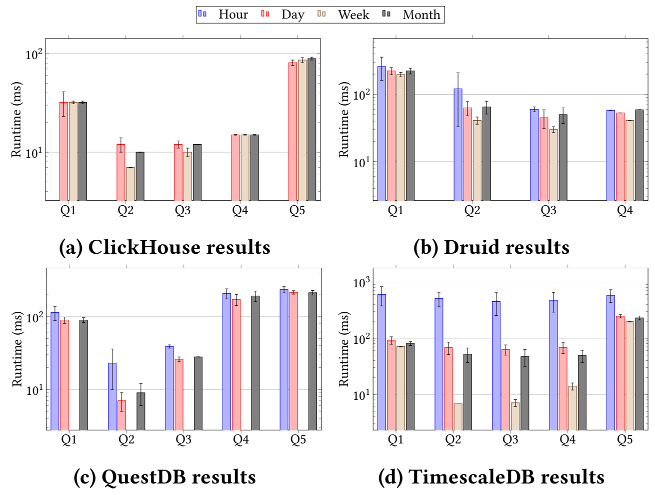
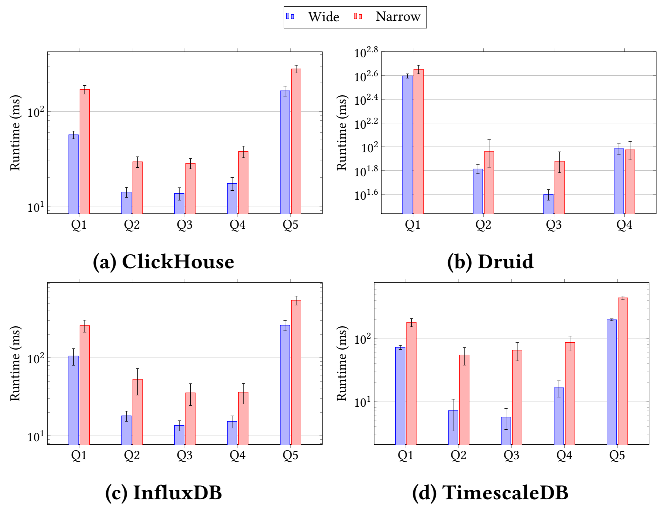


Figure 7: System partitioning. ClickHouse and QuestDB do not support partitioning by hour and week, respectively. eXtremeDB and MonetDB only support manual time partitioning, and InfluxDB has no support for partitioning.

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 8.



(a) ClickHouse (b) Druid (c) InfluxDB (d) TimescaleDB

Figure 8: Impact of Storage Layout on System Performance.

References

[1] Zemin Chao, Hong Gao, Yinan An, and Jianzhong Li. 2022. The inherent time complexity and an efficient algorithm for subsequence matching problem. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1453–1465.

[2] Michele Dallachiesa, Themis Palpanas, and Ihab F Ilyas. 2014. Top-k nearest neighbor search in uncertain data series. *Proceedings of the VLDB Endowment* 8, 1 (2014), 13–24.

[3] Shalmoli Gupta, Ravi Kumar, Kefu Lu, Benjamin Moseley, and Sergei Vassilvitskii. 2017. Local search methods for k-means with outliers. *Proceedings of the VLDB Endowment* 10, 7 (2017), 757–768.

[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.

[5] Mourad Khayati, Michael Böhlen, and Johann Gamper. 2014. Memory-efficient centroid decomposition for long time series. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 100–111.

[6] 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. In *Proceedings of the VLDB Endowment*, Vol. 13. 768–782.

[7] Michele Linardi and Themis Palpanas. 2018. Scalable, variable-length similarity search in data series: The ULISSE approach. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2236–2248.

[8] Patrick Schäfer and Ulf Leser. 2022. Motiflets: Simple and Accurate Detection of Motifs in Time Series. *Proceedings of the VLDB Endowment* 16, 4 (2022), 725–737.

[9] Sebastian Schmidl, Phillip Wenig, and Thorsten Papenbrock. 2022. Anomaly detection in time series: a comprehensive evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1779–1797.

[10] Machiko Toyoda, Yasushi Sakurai, and Yoshiharu Ishikawa. 2013. Pattern discovery in data streams under the time warping distance. *The VLDB Journal* 22 (2013), 295–318.

[11] Sheng Wang, Zhifeng Bao, J Shane Culpepper, Timos Sellis, and Xiaolin Qin. 2019. Fast large-scale trajectory clustering. *Proceedings of the VLDB Endowment* 13, 1 (2019), 29–42.

[12] 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. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2148–2160.

[13] Byoung-Kee Yi and Christos Faloutsos. 2000. Fast time sequence indexing for arbitrary Lp norms. (2000).

[14] 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).

[15] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S Yu. 2017. Time series data cleaning: From anomaly detection to anomaly repairing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1046–1057.

Appendices

A Advanced SQL Queries

Following are the SQL queries of the advanced queries discussed above.

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, 10, 13] 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: Simple Anomaly Detection. This query retrieve 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;
```