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

This report serves as a supplement to our original paper. Due to space constraints, we were unable to include certain additional materials, experiments, and results that we believe are important for a more comprehensive understanding of the benchmark. In this document, we present the omitted material including 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.

Experiment 1 (Systems Configuration):

It is important to ensure that the TSDBs are optimally configured before evaluating their performance. We studied the impact of the different parameters on systems performance. Most of the TSDBs have a myriad of parameters to tune. Our experiments showed that using the recommended values by each TSDB ensures in most cases a stable behavior of the system. We have interacted with systems providers in some cases where the performance of the default configuration was suspicious. For example, TimescaleDB had poor ingestion performance in our online workload when compressing the entire dataset. We have updated the compression settings based on the feedback of the developers to not compress the last chunk, which has led to a consistent ingestion performance. Another example was eXtremeDB and QuestDB, whose default memory allocation was insufficient for our datasets, which led to slower query execution and poor performance. We have increased the default memory allocation size of these systems based on the developer's feedback, which has enhanced their loading and query performance.

There was still one important parameter for which there does not exist any recommendation, which is the size of partitions. We ran several experiments to evaluate the impact of partitioning size on systems' behavior. We depict the results in Figures 1d and 1b. 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.

The last aspect we evaluated is how to represent and store our data for each system. 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 2. We adopted this layout for all systems except eXtremeDB. The latter uses a hybrid row-columnar format where sensor values are grouped into a single array.

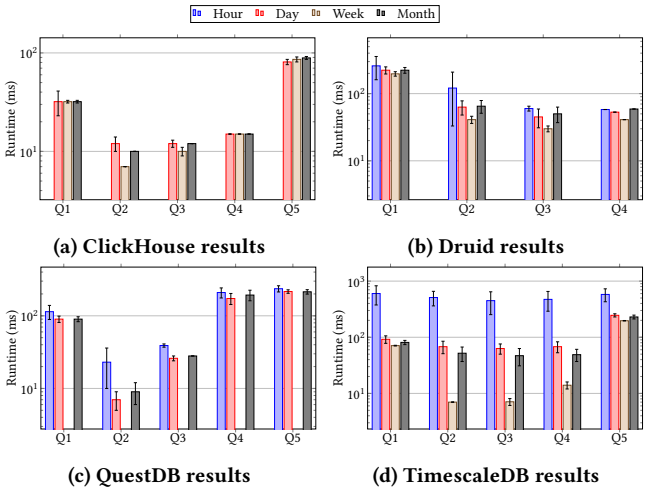


Figure 1: System partitioning. ClickHouse and QuestDB do not support partitioning by hour and week, respectively. eXtremeDB and MonetDB only support manual time partitioning, and InfluxDB has no support for partitioning.

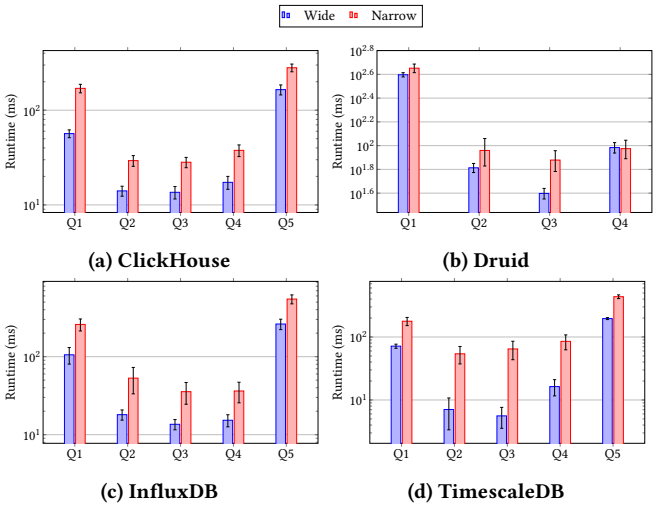


Figure 2: Impact of Storage Layout on System Performance.

Experiment 2 (Discarded Systems):

In order to select the appropriate TSDBs to evaluate, we used the best contender systems from other benchmarks including InfluxDB, Druid, and ClickHouse, we made certain that systems are properly configured such as TimescaleDB, and we included a cottage industry of popular systems such as eXtremeDB, MonetDB, or QuestDB is missing in those benchmarks. We decided to include them in our

benchmark. To further ensure that we did not dismiss any potential candidate, we also checked less-performing systems from other benchmarks including OpenTSDB, KairosDB, Graphite, and Apache IoTDB. As Table 1 illustrates, the performance of these systems is significantly inferior to the selected systems.

Table 1: 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

In addition to TSDBs, our pre-evaluation also included monitoring tools such as Prometheus to evaluate its efficiency to store and process time series used in monitoring applications. One of the main drawbacks of this system is that it does not support loading historical data. Instead, it 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 be able 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. We report the results only for D-LONG as we could not convert D-MULTI into Prometheus's format in a reasonable time. 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.

Table 2: Prometheus loading and compression.

System	Avg. Throughput (datapoints/s)	Loading Time (s)	Storage (GB)
ClickHouse	18'386'867	28	1.97
eXtremeDB	2'368'378	218	4.10
TimescaleDB	2'627'138	197	4.31
Prometheus	297'530	1'742	4.70

The query performance of Prometheus was also subpar. Table 3 shows that Prometheus does not appear among the top 3 systems in any of the queries and configurations. The only exception is Q5, where eXtremeDB performs poorly compared to the other systems.

Note that Prometheus's querying language PromQL does not support complex sub-queries. This limits its ability to perform advanced filtering. Because of this limitation, we could not implement Q6 and Q7 for Prometheus.

Experiment 3 (Complex Queries Performance):

During the development of the benchmark, we considered additional queries with diverse complexity as well as UDFs. We describe the selected queries and report their results to support our selection process.

Table 3: Prometheus query runtime.

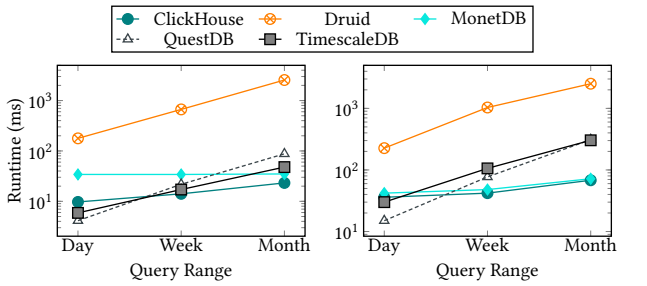
Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7
ClickHouse	30±1	10±1	13±2	13±1	**80±5**	53±2	9.72±5
eXtremeDB	34±1	**3±1**	**2±1**	**2±1**	296±1	**31±4**	14.06±2
TimescaleDB	**6±1**	6±1	12±2	12±2	95±1	62±3	**5.9±1**
Prometheus	160±49	18±1	25±3	20±3	248±7	-	-

Similarity Search in Time Series.

We implemented two complex queries that cover similarity search in time series [1, 4, 5, 14]. The two queries use different types of aggregations. We evaluate the query 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.

Q8
```
WITH distance_table AS (
  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;
```



(a) Q7: Pearson Correlation **(b) Q8: Distance-based Similarity Search**

Figure 3: Advanced Queries Results for queries with small output (Q7 and Q8). Note that eXtremeDB does not support the square root function and is thus not evaluated for Q8.

Figure 3b compares the results of Q8 to Q7 (our benchmark's *Correlation* query) by varying the time range interval. Both queries

involve multiple aggregations and have comparable output sizes. The results show that, albeit more complex, Q8 shows similar trends as the correlation query Q7 with a few marginal differences. More specifically, ClickHouse and MonetDB remain the fastest for high-range queries, followed by QuestDB. This result is explained by the fact that the behavior of the systems is impacted by the small output of both queries more than the complexity of the query itself.

Q9: Dynamic Time Warping. This query computes Dynamic Time Warping (DTW) [2, 13, 16] 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.

Q9

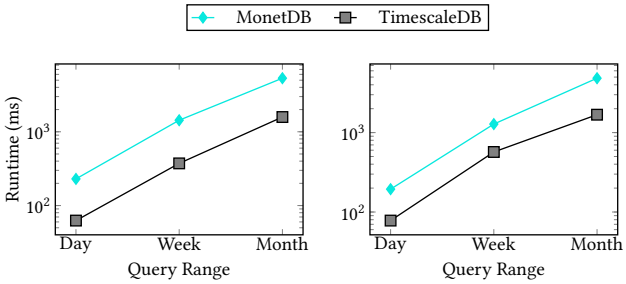
```
WITH distances AS (
  SELECT time, s, POWER(s - LAG(s, 1)
         OVER (ORDER BY time), 2) AS distance
  FROM ts_table
  WHERE st_id=?station)
  AND time < ?timestamp
  AND time > ?timestamp - ?range
),
segments AS (
  SELECT time, s,
         SUM(distance) OVER (ORDER BY time
         ROWS BETWEEN 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;
```

In this experiment, we contrast Q9 with our benchmark query Q6 (our benchmark's Cross Average query) based on the output size which exceeds 10K data points. Both queries perform cross-sensor analytics. Figure 4b illustrates the experiment results, revealing similar runtime trends for both DTW and Cross Average queries due to most of the query time being consumed by data output, given the large size of their results. TSDBs perform similarly for queries that share the same operations, even when the queries are more complex.

User Defined Functions.

We have also implemented additional time series tasks using User Defined Functions (UDFs). Those queries involve matrix operations and iterative computations, which are very hard to implement in the systems' native language. UDFs execution in TSDBs is comprised of three steps: (i) accessing and fetching the data, (ii) converting the data from and back to the TSDB format to the programming language format, and finally, (iii) executing the UDF.

We consider several tasks such as data normalization [10], matrix decomposition [8], anomaly detection [12], anomaly repair [18], clustering [6], and classification [3]. Furthermore, we have implemented various missing values imputation techniques from a recent benchmark [9].



(a) Q6: Cross-Sensor Average **(b) Q9: DTW**

Figure 4: Advanced Queries Results for queries with a large output (Q6 and Q9). We only evaluate MonetDB and TimescaleDB, as they were the only ones to support the necessary operators `LAG` and `OVER`.

We evaluated the tasks on the D-LONG and report the average execution time of 10 runs. As suggested by Reviewer #2, we present the results of the HotSax [12] technique, which is one of the most popular algorithms for anomaly detection. We omit the results of the remaining tasks, as they exhibit similar trends. Figure **??** depicts the runtime breakdown 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.

Experiment 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 [7, 17]. 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.

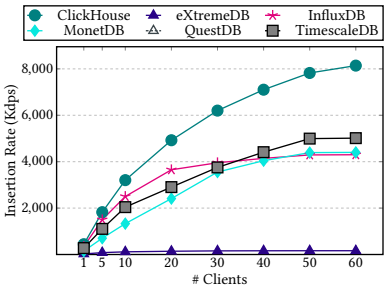
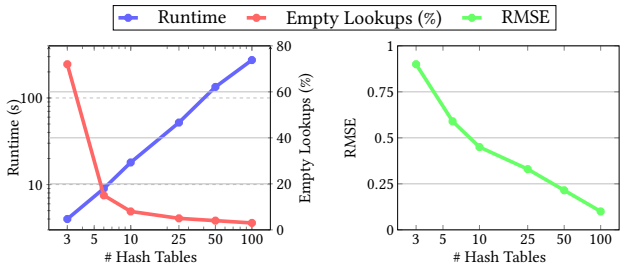


Figure 5: Insertion Performance.

Experiment 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 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 [11, 15], we found that determining the optimal number of hash tables for LSH is a trade-off between quality and computational efficiency.



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

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

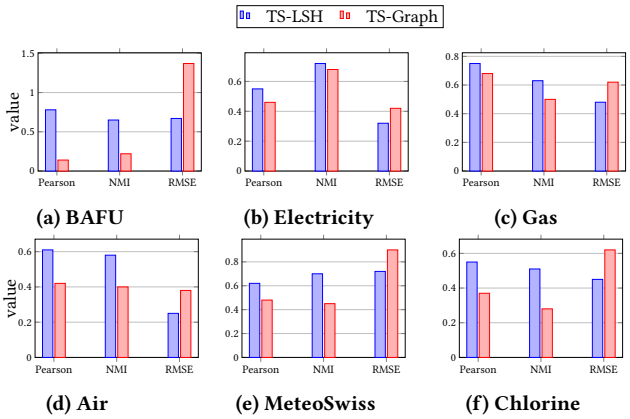
Experiment 6 (Generation Performance on different seed sources):

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 [9]. Table 4 describes the main properties of those datasets; Complex time series datasets exhibit irregular patterns, high volatility, multiple seasonality, and the presence of outliers.

Table 4: 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 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 technique. In simpler datasets, such as Electricity, the difference is less pronounced but is still visible.



(a) BAFU **(b) Electricity** **(c) Gas**

(d) Air **(e) MeteoSwiss** **(f) Chlorine**

Figure 7: Data Quality of TS-LSH.

References

[1] Ira Assent, Marc Wichterich, Ralph Krieger, Hardy Kremer, and Thomas Seidl. 2009. Anticipatory DTW for efficient similarity search in time series databases. *Proceedings of the VLDB Endowment* 2, 1 (2009), 826–837.

[2] 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.

[3] 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.

[4] Karima Echihabi. 2020. High-dimensional vector similarity search: from time series to deep network embeddings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2829–2832.

[5] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. 2019. Progressive similarity search on time series data. In *BigVis 2019-2nd International Workshop on Big Data Visual Exploration and Analytics*.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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).

[18] 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.