Benchmarking and Evaluating Time-Series Databases for Appliance-Level Energy Data

by

Simin Shehbaz

Bachelor of Technology in Computer Science and Engineering
MAHE Dubai, 2021

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Kenneth B. Kent, Ph.D., Computer Science

Examining Board: Suprio Ray, Ph.D., Computer Science, Chair
Aaron Tabor, Ph.D., Computer Science,
Julian Cardenas-Barrera, Ph.D., Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

February, 2026

© Simin Shehbaz, 2026

Abstract

This thesis benchmarks five Time-series databases (TSDBs)—TimescaleDB, ClickHouse, QuestDB, InfluxDB v1.8, and Apache IoTDB, using a controlled dataset simulating 26 appliances across 100 households at minute-level resolution. Extending the TSM-Bench methodology, ingestion throughput, query latency, storage efficiency, and compression characteristics across five workloads are evaluated. Wide-format schemas are compared against narrow-format schemas to understand schema effects on performance.

Results show dramatic performance disparities. ClickHouse achieves faster ingestion than competitors and completes billion-row queries in seconds versus hours for other TSDBs. QuestDB fails catastrophically beyond 684 million rows due to memory exhaustion. TimescaleDB incurs a performance penalty when enabling time-series optimizations for narrow-format data. Apache IoTDB achieves best-in-class storage compression at the cost of slower ingestion. Wide-format schemas universally outperform narrow formats across all TSDBs except ClickHouse, which demonstrates format-agnostic performance.

These findings provide evidence-based TSDB and schema selection guidance for smart-home energy monitoring deployments requiring billion-row scalability.

Dedication

This thesis is dedicated to my parents, Shehbaz and Almas for giving me a better life—thank you for so many things, especially the things I only understand because of you. To Sara and Omar—thanks for sharing Mom and Dad, our secrets, and our childhood. I will always remember us fondly. To Haya, for making me *masho*. To Sahar, my *jaan*, for being, and growing together. And to Roush, for everything and for the future.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Kenneth B. Kent, for giving me the opportunity to work at CAS—Atlantic and for his steady support, guidance, and encouragement. Under his mentorship, I have gained not only technical expertise but also valuable business insight and broader professional skills.

I am sincerely grateful to my teammates, Mohammad and Bhavani, for their constant help and encouragement throughout the course of this thesis. I would also like to thank Mr. Stephen MacKay for reading my paper and thesis and offering thoughtful feedback and suggestions.

My appreciation extends to Mr. DeVerne Jones for setting up my system, ensuring access to all necessary resources, answering every question—no matter how small—and always being willing to help. I would also like to thank Nidhi Bangera for proofreading an earlier version of this work. Lastly, I extend my sincere thanks to thank Rameela Ramesh for all the support throughout the course of this study.

This research was conducted at the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick.

Table of Contents

Dedication **iii**

Acknowledgements **iv**

Table of Contents **v**

List of Tables **xi**

List of Figures **xiii**

Abbreviations **xv**

1 Introduction **1**

2 Background **6**

2.1 Overview of Time Series Data 6

2.1.1 Schema Models for Time Series Data 7

2.1.2 Core Operations on Time Series Data 8

2.2 Limitations of Traditional DBs 9

2.2.1 Ingestion Bottlenecks with High-Frequency Writes 9

2.2.2 Inefficient Time-Based Indexing and Querying 10

2.2.3 Lack of Temporal Compression 10

2.2.4 Summary 10

2.3 TSDBs: Design and Architecture 11

2.3.1 Architectural Differences . . . 11
2.3.1.1 Time-centric Storage Engine . . . 11
2.3.1.2 Temporal Indexing and Metadata Tagging . . . 12
2.3.1.3 Common Features of TSDBs . . . 12
2.3.2 Common Compression Techniques in TSDBs . . . 13
2.4 Working of Selected TSDBs . . . 14
2.4.1 TimescaleDB . . . 14
2.4.2 QuestDB . . . 15
2.4.3 ClickHouse . . . 15
2.4.4 InfluxDB . . . 16
2.4.5 Apache IoTDB . . . 17
2.5 Importance of Benchmarking TSDBs . . . 18

3 Related Work **19**

3.1 TSM-Bench (2023) . . . 19
3.2 TSBS (2016) . . . 21
3.3 IoTDB-Benchmark (2019) . . . 21
3.4 IoTAbench (2015) . . . 21
3.5 SmartBench (2020) . . . 22
3.6 TS-Benchmark (2021) . . . 22
3.7 SciTS (2022) . . . 22
3.8 ClickBench (2022) . . . 23

4 Methodology **24**

4.1 Purpose and Motivation . . . 24
4.2 Overview . . . 25
4.3 Dataset Generation and Characteristics . . . 26
4.3.1 Data Generation . . . 26

4.3.2 Data Characterization and Profiling . . . 27
4.3.2.1 Wide-Format . . . 28
4.3.2.2 Narrow-Format . . . 29
4.3.3 Data Preprocessing . . . 29
4.3.4 File System . . . 32
4.4 TSDB Selection . . . 32
4.4.1 TSDB System Requirements (Inclusion Criteria) . . . 33
4.4.2 TSDB Exclusion Criteria . . . 33
4.4.3 Selected TSDBs for the Benchmarking Study . . . 34
4.5 Benchmark Design . . . 36
4.5.1 Pre-Defined Benchmark WLs . . . 37
4.5.2 Ingestion Workload Design . . . 38
4.5.2.1 Ingestion Approach . . . 39
4.5.2.2 Performance Metrics . . . 40
4.5.3 Query Workload Design . . . 41
4.5.3.1 Point Queries . . . 42
4.5.3.2 Analytical Queries . . . 42
4.6 Benchmarking Environment Setup . . . 44
4.7 Benchmark Execution . . . 45
4.7.1 TimescaleDB . . . 46
4.7.2 QuestDB . . . 49
4.7.3 ClickHouse . . . 50
4.7.4 InfluxDB . . . 51
4.7.5 Apache IoTDB . . . 53
4.8 Summary . . . 55

5 Results and Analysis **57**

5.1 Ingestion Performance Analysis . . . 57

5.1.1 TimescaleDB Ingestion Performance 58
5.1.1.1 Wide-Format Performance 58
5.1.1.2 Narrow-Format Performance 59
5.1.1.3 HT Overhead 60
5.1.1.4 WL5-Narrow Performance Collapse 61
5.1.1.5 Key Findings 62
5.1.2 QuestDB Ingestion Performance 63
5.1.2.1 Wide-Format Performance 63
5.1.2.2 Narrow-Format Performance 65
5.1.2.3 Key Findings 66
5.1.3 ClickHouse Ingestion Performance 67
5.1.3.1 TP Performance 68
5.1.3.2 Ingestion Latency 69
5.1.3.3 Data-Point Processing Rate 69
5.1.3.4 Scaling Behavior 70
5.1.3.5 Key Findings 71
5.1.4 InfluxDB Ingestion Performance 72
5.1.4.1 Implementation Context 73
5.1.4.2 Wide-Format Performance: 74
5.1.4.3 Narrow-Format Performance 75
5.1.4.4 Data-point Processing Rate 76
5.1.4.5 Key Findings 76
5.1.5 Apache IoTDB Ingestion Performance 77
5.1.5.1 Wide-Format Performance 78
5.1.5.2 Narrow-Format Performance 79
5.1.5.3 Datapoint Processing Rate 80
5.1.5.4 Key Findings 81

5.1.6 Cross-Database Ingestion Performance Comparison 83
5.1.6.1 Wide-Format Ingestion Throughput Comparison . . 83
5.1.6.2 Narrow-Format Ingestion ThroughputComparison . . 84
5.1.6.3 WL5 Ingestion Time Comparison 85
5.1.6.4 Schema Design Impact 87
5.1.6.5 Summary Rankings 87
5.2 Storage Performance 89
5.2.1 TimescaleDB 89
5.2.2 QuestDB 92
5.2.3 ClickHouse 94
5.2.4 InfluxDB 96
5.2.5 Apache IoTDB 98
5.2.6 Cross-Database Storage Comparison 99
5.2.6.1 Compression Effectiveness Comparison 99
5.2.6.2 Storage Performance Summary 101
5.3 Query Performance 102
5.3.1 Query Execution Success Rates 104
5.3.2 Cross-Query Performance Comparison 107
5.3.2.1 Per-Query Analysis 107
5.3.2.2 Per-Query Speedup Analysis 109
5.3.2.3 TimescaleDB Speedup Patterns 110
5.3.2.4 QuestDB Speedup Patterns 111
5.3.2.5 ClickHouse Speedup Patterns 111
5.3.3 Key Findings from Query Performance Analysis 112

6 Conclusion and Future Work **114**

6.1 Conclusion 115
6.2 Limitations 116

6.3 Future Work 117

Bibliography **124**

A InfluxDB Queries **125**

Vita

List of Tables

2.1 Narrow-Format Energy Data 7

2.2 Example of Wide-Format Energy Data 8

3.1 Comparison of TSDB Benchmarks 20

4.1 Characteristics of the Appliance-Level Energy Consumption Dataset . 28

4.2 Wide-format Dataset Sample 29

4.3 Narrow-Format Dataset Sample. Note: In the actual dataset, metric values are stored as integers (0–26); labels shown here for readability. 30

4.4 Preprocessing steps per data format 31

4.5 Excluded TSDBs (25 of 31 Initial Candidates) 35

4.6 Summary of Benchmark Workloads 38

4.7 Ingestion Implementation Approaches by TSDB 39

4.8 Benchmarking Environment Setup 45

5.1 Ingestion metrics per WL in TimescaleDB 58

5.2 Ingestion metrics per WL in QuestDB. *WL4-narrow and WL5-narrow ingestion failed due to memory exhaustion and write back-pressure. . 63

5.3 Ingestion metrics per WL in Clickhouse 67

5.4 Ingestion metrics per WL in InfluxDB 72

5.5 Ingestion metrics per WL in Apache IoTDB 78

5.6 TSDB Ingestion Performance Rankings 88

5.7 TimescaleDB Table Storage and Compression Metrics Across WLs . . 90

5.8 QuestDB Storage Metrics Across Workloads 93
5.9 ClickHouse Table Compression Metrics Across Workloads 95
5.10 InfluxDB Table Size Comparison Across Workloads 98
5.11 Apache IoTDB Table Size Comparison Across Workloads 100
5.12 WL5 Storage Efficiency Rankings 100

List of Figures

4.1 Apache IoTDB hierarchical storage structure showing root-level organization of workload storage groups 54
4.2 Apache IoTDB wide-format structure 55
4.3 Apache IoTDB narrow-format structure 56

5.1 TimescaleDB Throughput Comparison 59
5.2 TimescaleDB Data-points Ingested 60
5.3 TimescaleDB Latency Comparison: With and Without HTs 61
5.4 QuestDB Throughput Comparison 64
5.5 QuestDB Latency Comparison 65
5.6 ClickHouse Throughput Comparison 68
5.7 ClickHouse Latency Comparison 70
5.8 ClickHouse Data-points Ingested: Wide vs. Narrow Comparison . . . 71
5.9 InfluxDB Throughput Comparison 73
5.10 InfluxDB Latency Comparison 74
5.11 InfluxDB Datapoint Comparison 75
5.12 Apache IoTDB Throughput Comparison 79
5.13 Apache IoTDB Latency Comparison 80
5.14 Apache IoTDB Datapoint Comparison 81
5.15 Wide-Format Ingestion Throughput Comparison 83
5.16 Narrow Format Ingestion Throughput Comparison 84
5.17 WL5 Ingestion Time Comparison (Log Scale) 86

5.18 TimescaleDB - Compression Ratio Comparison 91
5.19 TimescaleDB–Storage Performance Comparison 92
5.20 QuestDB - Storage Performance Comparison 94
5.21 ClickHouse–Storage Performance Comparison 96
5.22 ClickHouse - Compression Ratio Comparison 97
5.23 InfluxDB - Storage Performance Comparison 99
5.24 Apache IoTDB - Storage Performance Comparison 101
5.25 Apache IoTDB Query Success/Fail Matrix (Q1–Q6) 104
5.26 Query Success/Fail Matrix (Q1–Q5) 105
5.27 Query 6 Success/Fail Matrix 106
5.28 Query 1 Performance: Wide (Solid) vs. Narrow (Striped) 107
5.29 Query 2 Performance 108
5.30 Query 3 Performance 109
5.31 Query 4 Performance 110
5.32 Query 5 Performance 111
5.33 Query 6 Performance 112
5.34 Per-Query Speedup Ratios for Q1–Q6. Ratios above 1.0 indicate that the narrow format is slower than the wide format. 113

List of Abbreviations

DB	Database
TSDB	Time Series Database
IoT	Internet of Things
NILM	Non-Intrusive Load Monitoring
RDBMS	Relational Database Management Systems
ACID	Atomicity, Consistency, Isolation, Durability
WL	Workloads
HT	Hypertable
LSM	Log-Structured Merge
TSM	Time-Structured Merge
RLE	Run Length Encoding
WAL	Write-Ahead Log
CR	Compression Ratio

Chapter 1

Introduction

The rise of smart homes, Internet of Things (IoT)-enabled devices, and renewable energy systems have made appliance-level energy consumption data a critical resource for building intelligent infrastructure. This data supports crucial real-world use-cases like energy usage feedback, non-intrusive load monitoring (NILM), and demand response systems. These require timely, granular, and device-specific feedback. Capturing this data at minute-level frequency from dozens of appliances produces large-scale, multi-dimensional time series data—demanding efficient storage, querying, and analysis [1].

Time series databases (TSDBs) are essential for managing large volumes of time-stamped data with low-latency read and write performance [23]. They are preferred over general-purpose databases in real-time monitoring applications, due to their scalability, compression efficiency, and native support for streaming analytics [23]. TSDBs use time-based indexing, columnar storage, down-sampling strategies, and offer built-in support for temporal queries such as sliding windows, aggregations, and time filters [23]. Unlike traditional relational databases, TSDBs are optimized for high ingestion throughput and storage efficiency [23, 22].

A crucial design choice when storing time series data is the schema format—wide vs. narrow. In the wide format, each row contains multiple sensor readings for a single timestamp (e.g., one row for a household may include readings from all 26 appliances). In contrast, the narrow format stores one measurement per row, requiring additional columns for the sensor identifier and its value. Wide-format schemas are typically better suited for high-dimensional sensor data where all sensors report at the same interval, enabling faster cross-sensor operations and improved compression due to columnar locality [9]. However, they may be less flexible for variable-frequency streams or semi-structured data, where narrow formats are more common [22].

In the context of the appliance-level energy data that are addressed in this study, typical analyses on this data would include: trends over time (e.g., how the heater consumption varies seasonally), anomaly detection that identifies surges in energy readings (e.g., devices left on), device-level usage comparison, and checking for peak demands, showing the critical role of TSDBs in energy analytics.

Existing benchmarks have not evaluated systems under appliance-level energy workloads (WLs). They have focused on specific domains like financial transactions, smart buildings, and industrial instruments [40, 13, 21]. TSM-Bench does come close in scope, however, it caters specifically to hydrological data [23]. Hence, this framework is extended and tailored to the fine-grained, appliance-level energy dataset that exhibits: high dimensionality, frequent timestamps, realistic WL patterns, and is format-aware.

In addition, this study explicitly investigates how schema design—wide vs. narrow—impacts performance across ingestion, compression, and querying. Both formats are implemented using the same dataset and WLs generated by DGSim, allowing a controlled comparison [1]. For each database, identical WLs in both formats are

executed, while modifying the TSM-Bench query templates to cater to our domain and querying energy data use cases. These queries (Q1–Q6) represent core analytical operations: raw data retrieval, filtered fetching, cross-appliance averages, house-type aggregations, and hourly downsampling. By isolating the schema as an independent variable, the benchmark reveals how system internals (indexing, compression, query planners) interact with the schema design under different WL tiers, for both wide and narrow data formats.

The extended benchmark therefore captures not only system-level differences in the databases, but also the influence of schema layout on practical performance for household-scale energy analytics. This dual-format evaluation is critical since storage design choices directly impact query flexibility, latency, and scalability for smart home deployments. This tailored benchmarking framework caters to a fine-grained, appliance-level, energy dataset generated using DGSim that exhibits:

- **High dimensionality** (e.g., 26 appliances $\times$ 5 home types),
- **Both schema types** (wide and narrow) for comparative evaluation,
- **Frequent data points** (minute-level),
- **Scalable WLs** (100 to 1000 homes across variable time ranges), and
- **Realistic WLs** (values representing household trends)

Benchmarking allows researchers to evaluate ingestion speed under bulk and streaming scenarios, query latency for diverse temporal WLs, compression efficiency across different data distributions, and system resource utilization under realistic energy analytics use cases [23, 22, 40, 13, 21]. Without tailored benchmarks, selecting a system is difficult and becomes more ad-hoc. This could lead to poor storage deployment selection, sub-par user experience, and delayed querying time [40].

To address the above mentioned challenges, this work presents a comprehensive benchmarking framework for appliance-level energy data that extends the TSM-Bench. In this thesis, the following contributions are made:

- Developing a benchmarking framework tailored to appliance-level energy consumption, evaluating both ingestion and query performance.
- Generating data WLs using 'DGSim', a custom simulator that models realistic energy consumption patterns for 26 appliances across five household types, producing both wide and narrow format, minute-level time series data [1].
- Defining a varying set of WL strains including write-heavy streams, range queries, time aggregations (daily/monthly/yearly), and appliance-level comparisons.
- Benchmarking five leading TSDBs—TimescaleDB, QuestDB, ClickHouse, InfluxDB and Apache IoTDB—analyzing ingestion rates, query latencies, compression ratios, and resource usage.
- Providing insights and recommendations for selecting the most suitable TSDB system for smart home energy analytics, while highlighting both, system-level, and schema-level trade-offs.

The primary goal of this thesis is to provide a systematic evaluation of TSDB performance within the specific context of appliance-level energy data. Given the lack of existing benchmarks that account for both high-dimensional energy workloads and varying schema designs, this work seeks to answer fundamental questions about database performance in this context. Such as, how do different TSDBs perform in terms of ingestion throughput, latency, and resource utilization when handling appliance-level energy data? What is the impact of data schema design on the performance of TSDBs for appliance-level energy data? What TSDB is best suited

for storing and querying appliance-level energy data, considering both ingestion and query performance?

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary background and theoretical context, introducing the key concepts discussed throughout this thesis. Chapter 3 surveys related work on TSDB benchmarking and appliance-level datasets. Chapter 4 details the study setup, benchmarking framework, and experimental methodology. Chapter 5 reports and analyzes the experimental results. Chapter 6 concludes the thesis, and outlines future research directions.

Chapter 2

Background

This chapter provides the necessary background on time-series data, the limitations of traditional DBs, and the design principles of TSDBs. It also discusses the importance of benchmarking and the working of the selected TSDBs in this study.

2.1 Overview of Time Series Data

Time-series data consists of observations collected over time. Each of these are associated with a specific timestamp and one or more measurement values [22]. Unlike transactional data that captures isolated events, time-series data emphasizes the changing of values over time and their temporal dependencies. It is inherently ordered, append-only, and exhibits a strong correlation between consecutive points [11], thus making it fundamental to domains like financial analyses, environmental monitoring and smart-grid analytics [14].

The exponential growth of sensor networks and monitoring applications has caused a surge in the volume and velocity of time-series data. Modern IoT environments produce billions of records per day across thousands of devices [34]. Managing data like

this requires systems that are capable of handling high-frequency ingestion, querying in real-time, and have long-term storage solutions with efficient compression. Traditional relational or document-oriented databases are typically not optimized for these patterns, motivating the development of specialized databases [14, 22].

2.1.1 Schema Models for Time Series Data

Two main schema representations are commonly adopted for storing time-series data: narrow and wide formats [35]. The choice between these representations influence ingestion throughput, compression ratio, and query latency, making it a central design consideration in benchmarking studies.

- **Narrow Schema:** Each record typically contains three key fields: a timestamp, a metric, and its value. This schema model offers high ingestion efficiency and flexibility especially when the number of sensors or metrics change dynamically [35]. An example of narrow-format schema is shown in Table 2.1.
- **Wide Schema:** In contrast, a wide schema stores multiple metrics within a single record as separate columns indexed by timestamp. This format can improve query performance for multi-metric retrievals but may also lead to sparsity if all the metrics are not recorded at every timestamp (missing values) [22]. An example of wide-format schema is shown in Table 2.2.

Table 2.1: Narrow-Format Energy Data

timestamp	**house_id**	**appliance**	**value (kWh)**
2024-01-01 00:00:00	H1	fridge	0.45
2024-01-01 00:01:00	H2	tv	0.32
2024-01-01 00:02:00	H3	oven	0.60
...			

Table 2.2: Example of Wide-Format Energy Data

timestamp	**house_id**	**fridge**	**tv**	**oven**	**washer**	**dryer**	...
2024-01-01 00:00:00	H1	0.45	0.32	0.50	0.10	0.08	...
2024-01-01 00:00:00	H2	0.42	0.40	0.60	0.11	0.09	...
2024-01-01 00:01:00	H1	0.44	0.35	0.48	0.09	0.08	...
2024-01-01 00:01:00	H2	0.43	0.38	0.59	0.12	0.10	...
...							

Note: These tables are a simplified illustration of the narrow-format and wide-format dataset used in this study. Each house record in the actual dataset contains 26 appliance-level energy measurements per timestamp for the wide-format data. Both datasets also include additional metadata fields such as house type and aggregated power consumption, which are omitted here for clarity. Both data formats are discussed in detail in Chapter 4.

2.1.2 Core Operations on Time Series Data

Time-series WLs typically involve operations that prioritize temporal ordering [22, 11]. Common tasks include data ingestion (continuous streaming of measurements at high frequency), range queries (fetching data for specific time intervals), aggregation (computing statistical summaries), anomaly detection (identifying outliers or threshold violations) [41], and forecasting (predicting future values based on historical trends).

These create demanding requirements for DBs. Traditional relational and document-oriented DBs were not designed for continuous, time-ordered data streams. They struggle with high write throughput, efficient time-based indexing, and compression of temporally correlated values. As the scale and frequency of this data increases, these limitations become amplified, leading to high storage costs and degraded query performance [22]. TSDBs have emerged to address these challenges through modified

architectures and optimizations tailored specifically for time-centric WLs [23]. The following section examines why conventional DBs are inadequate for time-series WLs and how TSDBs overcome these constraints.

2.2 Limitations of Traditional DBs

Relational Database Management Systems (RDBMS) and general-purpose data store systems were designed for transactional WLs. Consistency, integrity, and flexible relational querying was prioritized over sustained write throughput and time-based analytics[22]. Time-series data introduces different access patterns: append-heavy, time-ordered, and often requiring aggregation or filtering over continuous ranges over discrete lookups [35]. These characteristics expose limitations of traditional DBs when applied to time-series WLs.

2.2.1 Ingestion Bottlenecks with High-Frequency Writes

Traditional DB engines maintain strict atomicity, consistency, isolation, and durability (ACID) properties and commonly use B-tree–based index structures to support random access and consistency [15, 36]. These design choices can incur write amplification with continuous high-frequency inserts, when compared with write-optimized access methods like Log-Structured Merge(LSM) trees [16, 29, 27]. Each incoming record requires index maintenance, log writes, and page flushing, which limits ingestion throughput to thousands of rows per second, which is several orders of magnitude lower than what modern monitoring systems demand [23, 22]. Batching or buffering only partially mitigates this issue and also adds latency that is not ideal for real-time analytics.

2.2.2 Inefficient Time-Based Indexing and Querying

Most relational DBs index data using a primary key or by attribute combinations, but they lack time-optimized access paths. Time-based queries for this study, such as "retrieve all appliance readings between 18:00 and 20:00 for the past month" would require sequential scans or expensive index traversals if using traditional DBs. As the dataset grows, these operations become increasingly I/O-bound. While partitioning can be configured manually, temporal locality is not accessed automatically, unlike TSDBs that partition and index natively by time [35, 26]. As a result, latency increases non-linearly with increased WLs, and query performance degrades severely.

2.2.3 Lack of Temporal Compression

General DBs store each record with full schema metadata and per-row overhead, leading to substantial redundancy in time-series data, where successive timestamps often differ by a fixed interval and values change gradually [42]. They typically use block-level compression (e.g., LZ4, Snappy) optimized for general data rather than delta-based or run-length encoding techniques that take advantage of temporal correlation. This causes increased storage footprints and I/O steps during query execution. TSDBs integrate specialized encoders like Gorilla, delta, or XOR encoding, that achieve order-of-magnitude reductions in storage while enabling direct decompression during query execution [23, 42].

2.2.4 Summary

In summary, traditional DBs face limitations when managing time-series WLs creating a clear motivation for TSDBs. The next section explores how TSDBs address these challenges through architectural innovations and system-level design choices.

2.3 TSDBs: Design and Architecture

TSDBs are specialized DB engines optimized for high-frequency, temporally ordered data. They address the ingestion, storage, and analytical bottlenecks that general-purpose DBs face when managing large-scale time-dependent WLs [23, 22].

2.3.1 Architectural Differences

Unlike conventional DBs, TSDBs integrate architectural innovations in data partitioning, compression, and query execution to support real-time monitoring and analytics at scale.

2.3.1.1 Time-centric Storage Engine

TSDBs adopt append-optimized storage engines rather than update-intensive record stores. Incoming data has sequential writes in timestamp order and is stored in time-partitioned chunks (or shards), each covering a fixed time window (e.g., one hour or one day) [23, 26]. This can be observed in the DBs used in this study:

- TimescaleDB extends PostgreSQL through hyptertables (HTs) that automatically partition by time and space dimensions. Each chunk acts as an independent table, improving parallelism and cache locality [39].
- ClickHouse employs a column-oriented storage engine called MergeTree. This sorts data based on a user-defined primary key, commonly including a timestamp for time-series WLs. These parts are periodically merged in the background to reduce fragmentation, preserve sort order, and improve query performance and compression efficiency. [8].

- InfluxDB uses the Time-Structured Merge Tree (TSM), an evolution of LSM-Trees optimized for time-based key ordering and background compaction [26].
- Both QuestDB and Apache IoTDB natively partition time-series data by fixed time intervals (e.g., daily or monthly). Each partition is stored as a separate physical directory or file group on disk, allowing the systems to perform parallel ingestion and query execution across partitions without blocks [30, 3].

2.3.1.2 Temporal Indexing and Metadata Tagging

Many TSDBs employ specialized time-aware indexes like skip-lists, segment trees, or sparse indexes to locate relevant time ranges efficiently [22]. They commonly complement these with tag-based secondary indexes to filter data by device, location, or measurement type. For example, InfluxDB maintains series keys combining tag–field pairs hashed into its index, whereas TimescaleDB uses PostgreSQL's B-tree on HT dimensions. ClickHouse employs a sparse primary index that stores key ranges per data block, enabling partition and segment elimination even on terabyte-scale datasets, which significantly reduces query latency for analytical WLs [23].

2.3.1.3 Common Features of TSDBs

TSDBs are designed to handle time-series data, some of the features of TSDBs include:

- High Ingestion Throughput: Append-only writes and sequential I/O allowing ingestion of millions of records per second [23].
- High Compression Ratios: Columnar storage combined with time-aware encoders (delta, XOR, RLE) drastically reduce disk usage [42]. Compression is

applied transparently at chunk boundaries, preserving query ability without decompression overhead.

- Native Support for Temporal Queries: TSDBs natively implement operations such as SAMPLE BY, GROUP BY time, and continuous aggregates directly within the engine [22].

2.3.2 Common Compression Techniques in TSDBs

Compression is a an important part of TSDBs; it enables long-term retention of massive datasets without sacrificing performance. TSDBs employ time-domain encoding methods that exploit correlations among consecutive timestamps and values [42, 23].

- Delta Encoding: These techniques store only the difference between successive timestamps or values. When intervals are uniform, the deltas are small and easily compressed with variable-length encoding. The delta-of-delta variant further compresses the differences of differences [42].

- Run Length Encoding (RLE): RLE condenses sequences of identical or slowly varying values into a single (value, count) pair [23].

- XOR and Gorilla Encoding: Originating from the Gorilla TSDB of Facebook, using XOR encoding to compress floating-point values by storing XORs between successive binary representations and omitting leading zeros [28, 26].

- Dictionary and Bit-Packing Techniques: Categorical fields (e.g., sensor IDs, locations) are often encoded using dictionary mappings combined with bit-packing for compact storage. ClickHouse integrates these methods with SIMD decoding to enable column scans at memory speed [7, 5].

2.4 Working of Selected TSDBs

The TSDB selection for this study is discussed in Chapter 4.4. This section details the internal workings of the TSDBs benchmarked in this study—TimescaleDB, QuestDB, ClickHouse, InfluxDB, and Apache IoTDB. For each system, the storage engine and partitioning, indexing and query execution model, and compression/retention features relevant to time-series WLs are summarized.

2.4.1 TimescaleDB

Storage Engine and Partitioning: TimescaleDB extends PostgreSQL with HTs that automatically partition data into time-based chunks. Chunks are normal PostgreSQL tables managed by the extension, enabling append-optimized writes and pruning by the time windows during scans [23]. Timescale provides continuous aggregates, materialized over HTs with a background refresh and optional compression policies [37, 38].

Indexing and Querying: HTs employ standard PostgreSQL indexing on time and secondary (space) dimensions. During query planning, TimescaleDB automatically performs constraint exclusion on chunk boundaries, allowing range queries to scan only relevant time intervals. Continuous aggregates support real-time aggregation by combining materialized and recent data, while maintaining incremental refresh semantics for efficient updates [39].

Compression and Retention: It compresses data at a chunk level using columnar-style encodings, applying algorithms such as delta-of-delta, delta, and dictionary compression. The system provides automated compression and retention policies that periodically compress older chunks and drop data exceeding a specified time window, reducing the storage footprint while preserving recent data [37].

2.4.2 QuestDB

Storage Engine and Partitioning: QuestDB stores tables in a column-oriented layout, partitioned by configurable time intervals. Each partition is stored as a separate directory, with every column written into its own data and index files. The query planner applies partition pruning, scanning only the partitions that overlap the query time range, which accelerates time-bounded analytical queries. [32].

Indexing and Querying: It has column-oriented engines optimized for sequential reads and vectorized execution. It employs pre-compiled, SIMD-optimized operator implementations that leverage CPU vector instructions to process data in batches, achieving low-latency scans over wide temporal ranges [30].

Compression and Retention: It applies lightweight column encodings, storing each column as a contiguous binary file and using dictionary compression for symbol columns. Its vectorized, column-oriented engine performs CPU-efficient sequential scans for analytical workloads [31].

2.4.3 ClickHouse

Storage Engine and Partitioning: ClickHouse is a column-oriented OLAP engine. Its core MergeTree family stores data as immutable parts on disk and periodically merges them in the background. Each part is sorted by the primary key and divided into granules, with a sparse primary index that stores marks for efficient granule skipping. Custom partitioning keys (e.g., by day or month) constrain queries further through directory-level pruning, minimizing scan overhead [8].

Indexing and Querying: It combines sparse primary indexes with data-skipping indexes to accelerate range filtering on time and tag or key columns. The execution engine is fully vectorized, processing columnar blocks of data using SIMD-optimized

operators. Aggregation and downsampling operations are optimized for block-level execution, achieving high throughput on analytical and time-series workloads [6].

Compression and Retention: It applies column-level encodings such as delta, run-length, and bit-packing, configured via per-column codecs. These compress both, the column data and their associated index mark files, minimizing storage while preserving efficient access. Data retention is typically enforced by dropping old partitions, enabled by the engine's partition-per-directory layout [5].

2.4.4 InfluxDB

Storage Engine and Partitioning: InfluxDB's storage engine is built on the TSM tree. Incoming writes are first batched through a Write-Ahead Log (WAL) and periodically compacted into sorted, compressed TSM files on disk. Data is organized into shards that each cover a fixed time range and are grouped into shard groups for metadata management [19].

Indexing and Querying: In InfluxDB, a series is uniquely identified by the combination of a measurement name and its tag set. The TSM engine, together with the Time Series Index (TSI), organizes data by series key and timestamp, enabling efficient scans for queries with tag filters and time-range predicates. InfluxDB's query language (InfluxQL) includes native temporal operators and functions for aggregation and downsampling, supporting expressive time-based analytics [18].

Compression and Retention: The TSM engine applies Gorilla-style XOR compression for floating-point fields and delta or delta-of-delta encoding for timestamps and integer values, reducing the on-disk footprint and CPU overhead during sequential scans. Combined with shard-level retention policies that automatically remove expired time ranges, these optimizations enable high-throughput ingestion with pre-

dictable, bounded storage growth [28, 20].

2.4.5 Apache IoTDB

Storage Engine and Partitioning: Apache IoTDB implements a hybrid storage design that organizes data by devices and measurements. It supports time and space partitioning, maintaining separate partitions for distinct time windows and storage groups. Incoming data is flushed into sequence and un-sequenced TsFiles, optimized for append operations and background compaction [3].

Indexing and Querying: It employs time-oriented indexing within its TsFile format and a hierarchical metadata tree (MTree) to manage devices and measurements. Its SQL-like query language and APIs support time-window scans, aggregations, and downsampling over defined intervals. IoTDB also supports aligned measurements, where multiple sensors of a device share a common timestamp column, enabling efficient wide-style row alignment for multi-sensor telemetry data [2].

Compression and Retention: Apache IoTDB supports multiple column encodings, including PLAIN, RLE, TS-2DIFF (delta-of-delta), and GORILLA, which are applied before general compression (LZ4, Snappy, or GZIP) [2]. Studies show that TS-2DIFF alone achieves high compression ratios for typical sensor signals, leaving little room for improvement from secondary compression [43]. Data retention is managed via TTL policies and file-level deletion of expired time partitions, ensuring bounded on-disk storage.

2.5 Importance of Benchmarking TSDBs

Benchmarking is central to understanding performance, scalability, and design trade-offs among TSDBs. Given the diversity of TSDB architectures, empirical evaluation is the only reliable way to determine which system best suits a given WL [22, 23]. Different engines are optimized with different priorities—ingestion throughput, query latency, compression, scalability, or flexibility. Without standardized benchmarks, users must rely on claims or ad-hoc testing, which is an unreliable approach that often leads to sub-optimal deployments.

Building on this motivation, the next chapter surveys the existing literature on TSDB benchmarking, identifies methodological gaps in current suites and studies, and articulates the relevance to appliance-level energy TSDB benchmarking.

Chapter 3

Related Work

This chapter surveys representative benchmarks, analyzing their methodologies, WL designs, and applicability to appliance-level residential energy monitoring.

3.1 TSM-Bench (2023)

TSM-Bench represents a state-of-the-art comprehensive TSDB evaluation, providing the foundation for this thesis [23]. TSM-Bench introduces several innovations. It separates offline (batch ingestion) and online (streaming writes) scenarios, enabling evaluation of both historical data loading and real-time monitoring patterns. It also defines 14 fundamental query types spanning point queries, range scans, aggregations, downsampling, and joins, covering the core operations required for time-series analytics. It employs TS-LSH, a GAN-based generator trained on real hydrological monitoring data, producing synthetic time series with authentic temporal autocorrelation, seasonal patterns, and noise characteristics.

However, TSM-Bench's hydrological focus introduces limitations for appliance-level energy monitoring applications:

Benchmark	**Domain**	**Schema Formats**	**Data Generator**	**DBs Tested**	**Scale (Rows)**
TSM-Bench [23]	Hydrology	Both	GAN (TS-LSH)	7	1B
TSBS [40]	Finance	Narrow	Synthetic	5+	100M
SciTS [21]	Scientific	Narrow	Synthetic	4	10M
SmartBench [13]	Buildings	Wide	Real (noisy)	3	1M
IoTDB-Bench [24]	IoT Sensors	Narrow	Synthetic	1	100M
IoTAbench [4]	IoT Analytics	Event	Real sensors	2	N/A
TS-Benchmark [14]	Wind Turbines	Wide	Real turbine	5	50M
ClickBench [9]	Web Analytics	Wide	Real (GitHub)	10+	100M
This Study	**Energy Monitoring**	**Both**	**DGSim**	**5**	**1.4B**

Table 3.1: Comparison of TSDB Benchmarks

- Low dimensionality: Hydrological stations report 3-5 metrics per location versus 26 appliances per household in energy monitoring. This affects schema design trade-offs—wide format with 5 columns is trivial; wide format with 30 columns introduces materialized view and index challenges.

- Narrow value range: River gauge measurements vary within 1-2 orders of magnitude, whereas appliance power consumption spans 3 orders (LED: 10W, EV charger: 10kW), impacting compression effectiveness and query selectivity.

- Hydrological events are weather-driven and stochastic, whereas household energy exhibits strong daily/weekly periodicity (heating cycles, appliance usage patterns), enabling different query optimization strategies.

This thesis extends TSM-Bench's WL and query template while adapting the data generation (using DGSim instead of TS-LSH), adjusting the queries for energy analytics use cases, and implementing both schema formats across all TSDBs to understand schema effects.

3.2 TSBS (2016)

TSBS, developed by InfluxData, focuses on DevOps monitoring workloads simulating CPU utilization, memory consumption, and disk I/O metrics across server fleets [40]. TSBS provides pluggable data generators, write-focused evaluation, and multi-TSDB loaders. However, TSBS differs from this work in several ways. It targets homogeneous metric types (all CPU measurements have similar value ranges) rather than heterogeneous appliances (10W LEDs vs. 10kW chargers). It does not systematically evaluate schema design impact (uses narrow format). Lastly, query evaluation is limited to simple aggregations rather than the dimensional grouping and cross-entity analytics required for energy monitoring.

3.3 IoTDB-Benchmark (2019)

IoTDB-Benchmark [24] is the official benchmarking tool for Apache IoTDB, supporting heterogeneous sensor types (temperature, humidity, pressure) with varying reporting frequencies. It evaluates ingestion throughput, out-of-order write handling, and basic query latency. However, it assumes narrow-format schemas (device-metric-value tuples) and does not explore schema design trade-offs or high-dimensional per-entity data (26 metrics per device).

3.4 IoTAbench (2015)

IoTAbench [4] benchmarks IoT analytics platforms using real sensor data streams, focusing on system-level TP and event processing latency. While it includes power consumption scenarios, it uses simplified event models rather than continuous power measurements, and evaluates platform-level APIs, over raw TSDB performance.

3.5 SmartBench (2020)

SmartBench [13] evaluates data systems for commercial smart buildings, focusing on HVAC control, lighting automation, and occupancy sensing. It uses real-world sensor data from campus buildings with intentionally noisy measurements to test noise tolerance. SmartBench's query suite emphasizes correlation analysis (temperature vs occupancy) and basic filtering but lacks the dimensional grouping and aggregation patterns required for residential energy analytics.

3.6 TS-Benchmark (2021)

TS-Benchmark [14] benchmarks electricity generation data from wind turbines, incorporating realistic workloads including anomaly detection (turbine faults) and multi-sensor fetching (wind speed, blade rotation, power output). While energy-focused, TS-Benchmark assumes relatively simple schema layouts (5-10 metrics per turbine) and does not evaluate schema format impact. Furthermore, industrial turbine monitoring differs from residential energy monitoring in data characteristics. Turbines exhibit continuous operation with gradual state changes, whereas household appliances show discrete on/off cycles with rapid power transitions.

3.7 SciTS (2022)

SciTS [21] targets scientific instrumentation workloads such as particle accelerators, telescopes, and laboratory equipment generating high-frequency sensor streams. SciTS evaluates 4 TSDBs (InfluxDB, TimescaleDB, QuestDB, Cassandra) across ingestion throughput and query latency metrics using synthetic data. While scientific instrumentation shares high-frequency characteristics with energy monitoring (both

generate minute-level or finer measurements), SciTS focuses on extremely high write rates (millions of points per second) from specialized hardware versus the more moderate write rates of residential IoT devices. Additionally, scientific queries emphasize precise time-range extraction and raw data retrieval, whereas energy analytics require in-database aggregation and dimensional grouping.

3.8 ClickBench (2022)

ClickBench [9] offers an open benchmarking framework for analytical workloads across columnar databases. Using real-world datasets, ClickBench evaluates SQL aggregation performance, join efficiency, and compression ratios. However, ClickBench focuses on static batch analytics (historical analysis of append-only logs) rather than the continuous ingestion and real-time query patterns characteristic of monitoring systems. Additionally, ClickBench does not evaluate time-series-specific features or schema design trade-offs.

Chapter 4

Methodology

This chapter describes the methodology used to design and implement the benchmark for evaluating TSDBs for appliance-level energy data. It first discusses the motivation and purpose behind the benchmarking. Following this, an overview of the benchmark design process is provided, outlining key steps taken. Finally, details of the benchmark implementation are presented, including data generation, WL definition, TSDB selection, benchmark design, setup and execution.

4.1 Purpose and Motivation

Generating large, appliance-level energy datasets from DGSim posed challenges for storage and querying due to the highly disaggregated nature of the data. Dumping this data into CSV files was not an ideal solution. Early experiments using CSV exports exposed bottlenecks—slow I/O cycles, significant storage overhead, issues with append and poor support for appliance queries. Consequently, multiple TSDBs were evaluated to identify a system offering efficient storage and querying for this data. Given the scope of available systems, existing benchmarks were reviewed to

compare TSDB performance for appliance-level energy data. However, as discussed in Chapter 3, none of the existing benchmarks specifically addressed appliance-level, granular WLs or systematically compared wide vs. narrow schemas in this context. As a result, this methodology extends TSM-Bench to the appliance-energy context, and additionally examines the effect of the data schema design on ingestion and query performance. This thesis aims to answer the following research questions (RQs):

RQ1: How do different TSDBs perform in terms of ingestion throughput, latency, and resource utilization when handling appliance-level energy data?

RQ2: What is the impact of data schema design (wide vs. narrow) on the performance of TSDBs for appliance-level energy data?

RQ3: What TSDB is best suited for storing and querying appliance-level energy data, considering both ingestion and query performance?

4.2 Overview

To implement this benchmark, a series of steps were followed. First, the appliance-level energy data was generated by DGSim. An important part of the data generation is the formatting of this data. Therefore, both wide and narrow schemas were produced for testing. Following this, its characteristics were analyzed to understand its schema, size, and features. Based on this, the data was preprocessed to ensure consistency and correctness before being used in the benchmark tests.

Next, a selection of TSDBs relevant to appliance-level energy data storage and querying were identified for evaluation, based on certain inclusion and exclusion criteria.

Following this, the benchmark was designed. Based on prior research, existing benchmarks, and energy-domain use cases, a set of representative WLs were defined. These WLs increase in size and complexity to provide a comprehensive view of performance

across different scenarios. The benchmark is broadly divided into two parts, ingestion and querying. For ingestion, metrics like throughput, latency, and resource utilization were defined. For querying, a set of representative queries were designed to measure execution time, resource usage, and scalability. After finalizing the benchmark design, the benchmark environment was set up, including hardware and software configurations to ensure consistency across tests.

Finally, the benchmark was executed across the selected TSDBs, and results were collected for analysis. The subsequent sections detail each of these steps.

4.3 Dataset Generation and Characteristics

This section describes the data generation process and the characteristics of the appliance-level energy data used in the benchmark, for both, wide and narrow formatting.

4.3.1 Data Generation

The dataset for this study is generated using a simulator, 'DGSim', developed under the same study that generates data based on the energy consumption patterns of various household appliances [1]. DGSim is designed to model realistic energy consumption behaviors by appliances including those of heating, ventilation, and air conditioning (HVAC) systems, refrigerators, and washing machines. DGSim adopts a Markov, state-based modeling technique, where each appliance is modeled as a probabilistic state machine. The simulator determines the transitions between 'on' and 'off' states based on pre-defined probabilities, thus producing realistic usage patterns [25]. This simulator provides a scalable, reproducible environment for generating data as required by the experiment. The data generation process simulates

energy consumption of these appliances over time by considering factors such as appliance types, usage patterns, and operational conditions, including the time of day and seasonal variation. This approach addresses a key challenge in energy data research—the lack of publicly available datasets that offer an appliance-level, granular resolution. Due to storage limitations, DGSim originally generated aggregated energy consumption profiles for a large number of households. For the purpose of this benchmarking study, the simulator code was modified to output disaggregated, appliance-level energy consumption data for individual households. This generated more granular data, hence allowing benchmarking of database capabilities such as time-based grouping, filtering on metadata, and handling high-ingestion WLs for a variety of household appliances.

The simulator produced its data in a default wide format. However, to evaluate the impact of data schema design on TSDB performance, a narrow-format version of the same dataset was also created through a transformation process. This involved restructuring the wide-format data into a long, narrow format where each row represents a single measurement for an appliance at a specific timestamp. The transformation process ensured that both formats retained the same underlying information, allowing for a fair comparison of their performance in the benchmark tests. The characteristics of the data, in both formats are discussed in the next subsection.

4.3.2 Data Characterization and Profiling

The data generated by DGSim has a user-defined time range and a minute level time-stamp granularity. The number of households is configurable, DGSim assigns a `house_id` for each house. The houses simulated by DGSim include five distinct types: single-family home (attached), single-family home (detached), small apart-

Characteristic	Description
Time Granularity	1-minute intervals
Time Range	User-defined
Number of Houses	User-defined (each house is assigned an ID)
Appliances Modeled	26 appliances
Number of Columns	30 columns (wide); 5 columns (narrow)
House Types	5 distinct types

Table 4.1: Characteristics of the Appliance-Level Energy Consumption Dataset

ment, large apartment, and mobile home. Every house is assigned a house type using a distribution mapping technique during data generation. Each house type includes a unique combination of appliances and usage profiles, designed to reflect the diversity of real-world households. It captures energy consumption data for twenty-six different appliances commonly found in residential settings [1]. The dataset includes metadata such as `house_id` and `house_type`. Along with each appliance's power consumption values, the dataset also holds the total power consumption for the house at that given timestamp. The generated dataset is exported to a CSV file. A summary of the dataset characteristics is provided in Table 4.1

4.3.2.1 Wide-Format

The data generated by DGSim by default is structured in a wide format, one row per timestamp—each row has individual columns for each appliance, having a total of thirty columns per row, and capturing minute-level granularity of energy consumption data for twenty-six different appliances, within a user-defined time range, allowing scalability across different dataset sizes. It includes metadata such as `house_id` and `house_type`, along with individual columns for each appliance's power consumption values, and a column (`PC`) for the total power consumption for the house at that given timestamp. The first 5 rows of the wide-format data are shown in Table 4.2.

Appliance columns are named generically as `App 1`, `App 2`, `···`, `App 26` to represent

time	House type	PC	House id	App 1	App 2	...	App 26
2024-01-01 00:00	apt	78.00	House_1	1.97	0.45	···	11.14
2024-01-01 00:01	apt	81.48	House_1	8.39	0.40	···	2.13
2024-01-01 00:02	apt	198.58	House_1	6.54	22.54	···	28.27
2024-01-01 00:03	apt	232.12	House_1	2.65	1.24	···	5.00
2024-01-01 00:04	apt	191.68	House_1	7.94	4.23	···	19.84

Table 4.2: Wide-format Dataset Sample

the 26 different appliances. The PC column represents the total power consumption for the house at that timestamp.

4.3.2.2 Narrow-Format

The narrow-format version restructures this data into five columns: timestamp, house_id, house_type, metric, and value. The metric field contains integer values (0–26) representing the total power consumption (metric 0) and the twenty-six individual appliances (metrics 1–26). This transformation results in twenty-seven rows per timestamp (one for each metric), creating a longer dataset with more rows but fewer columns.

For this study, a Python script was developed to transform the wide-format data into narrow-format by restructuring each row into multiple rows—one per appliance metric. This transformation ensured that both formats retained the same underlying information, allowing for a fair comparison of their performance in the benchmark tests. The first five rows of the narrow-format data are shown in Table 4.3.

4.3.3 Data Preprocessing

The base dataset file generated by DGSim was in CSV format. A small dataset was generated to learn the structure and characteristics of the data, and to see if there

time	house_type	house_id	metric	value
2024-01-01 00:00	single_family_detached	1	pc	78.00
2024-01-01 00:01	single_family_detached	1	pc	81.48
2024-01-01 00:02	single_family_detached	1	pc	198.58
2024-01-01 00:03	single_family_detached	1	pc	232.12
2024-01-01 00:04	single_family_detached	1	pc	191.68

Table 4.3: Narrow-Format Dataset Sample. Note: In the actual dataset, metric values are stored as integers (0–26); labels shown here for readability.

were any issues with the data being generated. This data had values of appliances used by a household for one year. The following findings were made from this initial dataset:

- There were no missing values in the dataset values generated for both formats.
- The time stamps were generating data for every minute, but TSDBs generally require timestamps with a second level precision; minute level is not acceptable, therefore this needed to be fixed.
- The file size was relatively larger than needed because of string type meta data like `house_type` and `house_id` that had large values.
- The generated numbers had a high level of precision (many decimal places), which was not necessary for energy consumption data.
- For the narrow-format, the file size was significantly larger due to the increased number of rows. This was made worse because of the metrics column having string values.

Due to the above issues, the following preprocessing steps were taken before using the data for benchmarking, as described below.

- The timestamp values were updated to have second level precision instead of minute level.

Data Format	Preprocessing Step	File Size (MB)
Wide Format	Initial	76.3
	Timestamp (Second level)	79.1
	House Type (Short Hand)	65.4
	House ID (int value)	58.3
	DP Precision Reduction	25.6
Narrow Format	Initial	384.6
	Timestamp Second level	392.3
	House Type Short Hand	268.7
	House ID int value	245.9
	DP Precision reduction	160.5
	Metric Updated from label to int	136.6

Table 4.4: Preprocessing steps per data format

- The `house_type` column values were reduced in size by applying lossless technique by using a shorthand label.
- The `house_id` column data type was changed from `STRING` to `INTEGER` to reduce the bit space taken by the values.
- The appliance power consumption values were rounded down to two decimal points (DP) to reduce unnecessary precision. This was done to reduce file size and minimize unnecessary precision that does not affect the integrity of the benchmarking results. This form of preprocessing is common in real-world energy monitoring pipelines, where data is often truncated or aggregated to reduce storage overhead. The reduced precision has a negligible effect on ingestion performance outcomes but significantly improves the consistency and speed of bulk insert operations across systems.
- Further, for the narrow-format, the `metric` column values were also reduced in size by applying lossless technique by index mapping each label to a number, twenty-seven in total, ranging from 0—26.

The data was preprocessed and values were updated to reduce the file size. Apart from this, the datasets are not altered in any way. The initial dataset covers one day

for 100 houses and served as the base formatting for all benchmark WLs, including both wide and narrow formats. The same preprocessing steps were implemented on all the WLs using a python script. The step-wise impact on file size for both formats for the base file are is summarized in Table 4.4.

4.3.4 File System

An investigation was conducted in deciding what file format to use as the base input for ingestion into the database. All this data was stored in CSV format since they are universally supported and portable, simple. The only different file format used apart from CSV was the line protocol format for InfluxDB, as it is the native format for ingestion into InfluxDB. The line protocol files were generated from the preprocessed CSV files using a python script.

4.4 TSDB Selection

This section first outlines the potential database WL requirements for this type of data, summarizing the inclusion criteria, followed by the criteria used to exclude the TSDBs for this study. Finally, the selected TSDBs for this benchmarking study are summarized. A 'good' database, in the context of this study, is defined as one capable of bulk loading data at 50,000–70,000 rows per second, executing aggregation queries on millions of rows in less than a minute, providing automatic storage compression, and offering simple setup and Python integration.

4.4.1 TSDB System Requirements (Inclusion Criteria)

As mentioned in 4.3.2, the simulated dataset consists of high-frequency (1-minute interval) energy consumption data for 26 different appliances. The database WL for this study is designed based on the potential operations that the database might be expected to handle. Based off these, the TSDB must include:

- Native support for time-series data ingestion, indexing, storage, and time-window grouping (e.g., hourly/daily).
- Support for high-throughput ingestion at 1-minute intervals across 26 appliances per household, with sustained performance at $\geq$10,000 datapoints/sec.
- Ability to filter metadata based on non-temporal fields such as `house_id`, and `house_type`.
- Availability of a SQL-like query language or API supporting selection, filtering, and time-based operations.
- Open-source or free for academic use.
- Should have an ability to ingest wide-format and narrow-format data.
- Should have native support for Docker-based deployment.

4.4.2 TSDB Exclusion Criteria

Based on the WL requirements, a set of exclusion criteria have been defined to exclude TSDBs from this benchmarking study.

- Unmaintained or inactive projects.
- TSDBs/projects with no updates in over a year.

- TSDBs requiring significant dataset transformation beyond the benchmark's native schema formats.
- Cloud-only managed services without self-hosted options.

4.4.3 Selected TSDBs for the Benchmarking Study

The TSDB selection process began with an initial survey of thirty-one time-series database systems identified through literature review and online resources, as detailed in Table 4.5. Each candidate was evaluated against the inclusion criteria stated in Section 4.4.1. Twenty-five TSDBs were excluded during initial screening, while six candidates—TimescaleDB, ClickHouse, QuestDB, InfluxDB, Apache IoTDB, and CrateDB—met all criteria.

ClickHouse is a columnar database, and natively supports wide schemas, while also supporting narrow-format. It also excels with high-dimensional time series by using its MergeTree engine. Each column is stored independently and is efficiently compressed by using delta encoding and SIMD-optimized scans. This enables fast read operations and aggregation operations [9].

TimescaleDB is built on PostgreSQL. It stores time series data in row-oriented hyper tables, that are partitioned by time and can also be done so by device. It handles rows effectively using array-based storage and chunk-level compression. Though, the performance may see a degradation with increased cardinality or wide schemas due to row-based access costs [22].

QuestDB also uses a column-store layout with time-partitioned tables. It supports vectorized execution and memory-mapped I/O, making it well-suited for appliance-level energy data, for example, where sensors report at fixed intervals [33].

InfluxDB uses the TSM storage engine with a shard-based organization, write-

No.	TSDB	Reason for Exclusion
1	VictoriaMetrics	Model requiring semantic dataset reinterpretation
2	Prometheus	Model requiring dataset restructuring
3	OpenTSDB	Rigid tag-based schema requiring non-trivial redesign
4	KairosDB	Unmaintained or inactive; no recent updates
5	TDengine	Supertable-based schema requiring explicit restructuring
6	Druid	Significant dataset transformation required
7	Graphite	Significant dataset transformation required
8	Blueflood	Unmaintained or inactive; no recent updates
9	Warp 10	Specialized data model requiring semantic transformation
10	DalmatinerDB	Unmaintained or inactive; no recent updates
11	KDB+	Licensing or academic access restrictions
12	Cassandra	Significant dataset transformation required
13	Beringei	Unmaintained or inactive; no recent updates
14	TDengine Cloud	Cloud-only managed service
15	Akumuli	Append-only numeric model requiring preprocessing
16	Riak TS	Unmaintained or inactive; no recent updates
17	Axibase ATSD	Unmaintained or inactive; no recent updates
18	Sidewinder	Unmaintained or inactive; no recent updates
19	Timescale Forge	Cloud-only managed service
20	AWS Timestream	Cloud-only managed service
21	Kusto	Cloud-only managed service
22	Chronix Server	Unmaintained or inactive; no recent updates
23	QuestDB Cloud	Cloud-only managed service
24	GreptimeDB	Significant dataset transformation required
25	SiriDB	Append-only model requiring dataset adaptation

Table 4.5: Excluded TSDBs (25 of 31 Initial Candidates)

optimized ingestion, and optional TSI indexing to mitigate high-cardinality costs. It uses native line protocol, and retention policies [18]. Apache IoTDB uses a columnar file format (TsFile) with per-column encoding and compression, partitioned by time and device [3].

CrateDB, built on Elasticsearch and Lucene, ingests wide-format data via JSON objects, storing each record in an inverted index structure [10]. While flexible, this

design is better suited for semi-structured or sparse data. It also introduces an overhead when managing large numbers of columns with regular updates. This limitation was observed during initial performance testing, where CrateDB exhibited significant inefficiencies under high-ingestion, wide-schema WLs [35]. Consequently, CrateDB was excluded from the final evaluation and is not included in the comparative analysis presented in this study.

The final TSDBs selected for this were: **TimescaleDB**, **ClickHouse**, **QuestDB**, **InfluxDB**, and **Apache IoTDB**. The working of these TSDBs are discussed in Chapter 2.4.

4.5 Benchmark Design

This benchmarking framework is based on the TSM Benchmarking framework [23], which is a widely used benchmarking framework for TSDBs. Modifications were made to the original framework to accommodate the specific requirements of this study, such as the data generation process, the selected TSDBs, the queries to be executed, and the performance metrics to be measured for the write and read testing. The ingestion process was modeled to simulate high-frequency, granular, appliance-level energy data. For the query design, six representative SQL queries (Q1–Q6) were selected based on common analytical tasks in time series energy data. These include basic data retrieval, threshold-based filtering, device-level and aggregated averages. These queries are discussed in detail in Section 4.5.3. This structure follows the precedent set by benchmarks like IoTDB-Benchmark [42] and Time Series Benchmark Suite (TSBS) [40], which advocate for a mix of point queries, aggregations, and down-sampling to evaluate TSDB capabilities comprehensively.

This benchmark comprises of three components: scalable pre-defined WLs, a set of

evaluation criteria for write performance testing and read performance testing. The write WLs focus on ingestion performance under varying dataset sizes, while the read WLs evaluate query execution efficiency under common time-series access patterns. These will be discussed in detail in the following sections.

4.5.1 Pre-Defined Benchmark WLs

To test TSDBs' performance under varying load conditions, a set of scalable dataset WL tiers have been defined. These WLs are generated from the same simulator, DGSim, and have the same base structure but differ in size and complexity to simulate different operational scenarios. The WLs are designed to reflect realistic usage patterns. They are structured to allow for scalability to test TSDBs under different conditions, from light to heavy loads, allowing for a comprehensive evaluation of their performance capabilities. WLs increase in size and complexity to simulate realistic deployment scenarios, with each WL representing a different level of complexity and size. The WLs are divided into two categories based on data schema: wide-format and narrow-format. Each category contains identical WLs to ensure a fair comparison between the two schema designs.

The details of each WL are summarized in Table 4.6. Format describes whether the data is in wide or narrow format. 'Duration' indicates the time range of the data generation having one-minute intervals. 'Houses' specifies the number of households included in the dataset. 'Rows' represents the total number of rows in the dataset, while 'Data Points' indicates the total number of individual data points that require storing. Note that 'M' values for the data points represent millions and 'B' values represent billions. 'File Size' indicates the size of the dataset file after preprocessing, MB indicates megabytes and GB indicates gigabytes. The file sizes mentioned are after preprocessing steps have been applied.

Format	WL	Duration	Houses	Rows	Data Points	File Size
Wide (30 columns)	WL1	1 day	100	144 K	4.32 M	25.60 MB
	WL2	1 day	1000	1,440 K	43.20 M	257.40 MB
	WL3	31 days	100	4,464 K	133.92 M	800.90 MB
	WL4	31 days	1000	44,640 K	1.34 B	8.05 GB
	WL5	365 days	100	52,560 K	1.58 B	9.39 GB
Narrow (5 columns)	WL1	1 day	100	3,888 K	19.44 M	136.60 MB
	WL2	1 day	1000	38,880 K	194.40 M	1.40 GB
	WL3	31 days	100	120,528 K	602.64 M	4.24 GB
	WL4	31 days	1000	1,205,280 K	6.03 B	43.59 GB
	WL5	365 days	100	1,419,120 K	7.12 B	51.26 GB

Table 4.6: Summary of Benchmark Workloads

WL1 represents a baseline ingestion test with minimal load, while WL5 simulates a high stress scenario. WL1—WL5 are selected to measure storage performance and the affect of increased size with the same table schema, as they represent a diverse mix of time range and dataset size.

4.5.2 Ingestion Workload Design

To evaluate ingestion performance, a structured ingestion testing framework was designed under realistic WL conditions. This evaluation is conducted on the five WL tiers defined in the previous section, summarized in Table 4.6, which simulate

TSDB	Method	Batch Size	Implementation Details
TimescaleDB	Bulk	N/A	PostgreSQL native COPY command for direct CSV loading
ClickHouse	Bulk	N/A	Full-file HTTP POST via `INSERT INTO ... FORMAT CSVWithNames`
QuestDB	Mixed	1,000,000	Bulk loading for WL1–WL2; chunked streaming for WL3–WL5 via HTTP API
InfluxDB	Batched	10,000	Line Protocol format with batched HTTP writes (recommended practice)
Apache IoTDB	Batched	500,000	Multi-threaded Session API with batched writes following TSM-Bench

Table 4.7: Ingestion Implementation Approaches by TSDB

varying levels of household energy consumption data. The ingestion tests measure how well each TSDB handles high-frequency data ingestion, which is critical for real-time energy monitoring applications.

4.5.2.1 Ingestion Approach

Ingestion methods varied across databases due to differences in client APIs, memory management characteristics, and recommended practices for each system. Table 4.7 summarizes the ingestion approach and batch sizes used for each TSDB.

TimescaleDB and ClickHouse utilized bulk loading approaches throughout all WLs. TimescaleDB employed PostgreSQL's native `COPY` command, which provides optimized bulk insertion directly from CSV files. ClickHouse used full-file HTTP POST

requests via its native protocol, loading entire CSV files in a single operation.

QuestDB employed a mixed approach: smaller WLs (WL1 and WL2) were loaded via bulk HTTP uploads, while larger WLs (WL3–WL5) required chunked streaming with 1,000,000-row batches to manage memory constraints and ensure reliable ingestion.

InfluxDB required batched insertion throughout all WLs. CSV data was first converted to InfluxDB's Line Protocol format, then ingested in batches of 10,000 rows via HTTP writes. This batch size follows common practices documented in InfluxDB guides and was necessary to prevent memory overflow during ingestion.

Apache IoTDB employed batched writes with 500,000-row chunks using its multi-threaded Session API, following the TSM-Bench methodology for parallel ingestion and high throughput.

These implementation differences reflect the operational characteristics and architectural designs of each system. Bulk loading methods typically offer higher throughput than batched approaches, as they minimize network round-trips and API overhead. However, batched approaches provide better memory management and are often necessary for systems with specific protocol requirements (e.g., InfluxDB's Line Protocol) or when working with resource-constrained environments. Ingestion performance comparisons should be interpreted with these distinctions in mind.

4.5.2.2 Performance Metrics

Ingestion testing is conducted on all the WLs mentioned in Table 4.6. For each ingestion run, the following performance metrics are collected:

- **Ingestion Latency:** The total time taken to ingest the entire dataset for a WL tier.

- **Ingestion Throughput:** Calculated as rows inserted per second, measuring how quickly the database can handle incoming data. This is calculated as the total number of rows ingested divided by the total time taken for ingestion.
- **Storage Size:** The total disk space consumed by the database after ingestion.
- **Compression Ratio:** For databases supporting native compression, the ratio of uncompressed storage size to compressed storage size (calculated as uncompressed size ÷ compressed size), indicating the effectiveness of the database's compression algorithms. A higher ratio indicates better compression (e.g., a ratio of 5:1 means the data is compressed to one-fifth of its original size).

4.5.3 Query Workload Design

The query WL is designed to evaluate the read performance of TSDBs for appliance-level energy consumption data. It is structured to test the TSDBs' ability to handle common time-series access patterns, such as consumption queries, detect threshold violations, usage summary, reduce storage, compare appliance behaviors, and perform aggregations. The queries are designed to reflect realistic usage scenarios, such as retrieving energy consumption data for specific appliances, summarizing usage over time, and detecting anomalies in energy consumption patterns. The queries have been divided into two main categories: **point queries** and **analytical queries**. Point queries are designed to retrieve specific data points, while analytical queries are designed to perform aggregations and calculations on the data. This structure follows the precedent set by pre-existing benchmarks, which advocate for a mix of point queries, aggregations, and down-sampling to evaluate TSDB capabilities comprehensively [40, 42]. These queries are executed on all the WLs defined in Table 4.6. They work sightly differently for the narrow-format data in some of the queries, so they have been mentioned as well wherever they are different.

4.5.3.1 Point Queries

Point queries are designed to retrieve specific data points from the TSDBs. They are used to retrieve data for a specific time range, or for a specific appliance. The point queries used in this study are as follows:

- **Query 1 (Data Fetching)**: Retrieve the minute-level peak energy consumption for all appliances during peak usage hours (18.00-21.00).

```
SELECT * FROM workload
WHERE time >= '18:00:00' AND time < '21:00:00';
```

- **Query 2 (Data Fetching with Filters)**: Retrieve the minute-level energy consumption for a specific appliance (HVAC) using `Appliance_ID` across all households, filtering for instances where the consumption exceeds a defined threshold (15kwH). The query plan for both, the wide and narrow format is described.
  Wide-format:

```
SELECT time, house_id, Appliance_23
FROM workload_wide WHERE Appliance_23 > 15;
```

  Narrow-format:

```
SELECT time, house_id, metric, value
FROM workload_narrow WHERE metric = 23 AND value >15;
```

4.5.3.2 Analytical Queries

Analytical queries are designed to perform aggregations and calculations on the data. These queries involve time-based windows, grouping, or multiple appliance usage over

sliced data. The analytical queries used in this study are as follows:

- **Query 3 (Cross-Appliance Average)**: Computes the average value of two or more appliances for a specific household, grouped by distinct timestamps.
  Wide-format:

```
SELECT time, AVG(Appliance_21), AVG(Appliance_23)
FROM workload_wide WHERE house_id = 'x' GROUP BY time;
```

  Narrow-format:

```
SELECT time, AVG(value) FILTER (WHERE metric = 21),
AVG(value) FILTER (WHERE metric = 23)FROM workload_narrow
WHERE house_id = 1 AND metric IN (21, 23) GROUP BY time;
```

- **Query 4 (Average Energy Consumption by House Type)**: Computes the average total energy consumption for each of the existing house types in the dataset. This simulates aggregate-level analytics.
  Wide-format:

```
SELECT house_type, AVG(total_energy_consumption)
FROM workload_wide GROUP BY house_type;
```

  Narrow-format:

```
SELECT house_type, AVG(value) FROM workload_narrow
WHERE metric = 0 GROUP BY house_type;
```

- **Query 5 (Average Energy Consumption by House)**: Computes the average energy consumption per household.

Wide-format:

```
SELECT house_id, AVG(total_energy_consumption)
FROM workload_wide GROUP BY house_id;
```

Narrow-format:

```
SELECT house_id, AVG(value) FROM workload_narrow
WHERE metric = 0 GROUP BY house_id;
```

- **Query 6 (Hourly Down-sampling)**: Down-samples minute-level readings by aggregating them into hourly averages for each appliance and household. This is representative of long-term trend analysis and storage optimization.
  Wide-format:

```
SELECT date_trunc('hour', time) AS hour_slot, house_id,
AVG(Appliance_1), ..., AVG(Appliance_26) FROM workload_wide
GROUP BY hour_slot, house_id ORDER BY house_id, hour_slot;
```

  Narrow-format:

```
SELECT date_trunc('hour', time) AS hour_slot, house_id,
AVG(value) FILTER (WHERE metric = 1),...,
AVG(value) FILTER (WHERE metric = 26)
FROM workload_narrow WHERE metric BETWEEN 1 AND 26
GROUP BY hour_slot, house_id ORDER BY house_id, hour_slot;
```

4.6 Benchmarking Environment Setup

To ensure consistency, all TSDBs are deployed in isolated docker containers with identical hardware configurations, on a local machine. Docker was chosen due to its

Host Machine	macOS (Apple M1 Pro), 10 CPU cores, 32 GB
Docker Version	v28.1.1, build 4eba377
Docker Specs	6 CPU cores; 12 GB memory limit; 2 GB swap; 256 GB disk usage limit
DB Container Versions	TimescaleDB: v2.19.3 (PostgreSQL 14.17) QuestDB: v7.3.2 ClickHouse: v25.5.2.47 InfluxDB: v1.8.10 Apache IoTDB: v2.0.5

Table 4.8: Benchmarking Environment Setup

ability to provide isolated and consistent testing environments across the TSDBs, eliminating system-level dependencies that may affect performance results [12]. The specifications used for the benchmarking tests are summarized in Table 4.8.

Docker images were pinned to specific, ARM64-stable versions to ensure reproducibility on Apple M1 hardware and to avoid compromising results with evolving changes. These versions fully support the features required by the WLs.

4.7 Benchmark Execution

To ensure consistency, all TSDBs are deployed in clean environments with identical hardware configurations. Each test undergoes an initial warm-up run, followed by three measured runs to capture performance. TSDBs are reset between each run. This methodology ensures a fair, controlled, and reproducible comparison of read and write performance across systems.

Ingestion and query testing is performed on the selected TSDBs using the WL tiers defined in Table 4.6. Ingestion used uniformly preprocessed CSV files. For each

TSDB, query testing was executed on all six queries (Q1–Q6) sequentially across all WL tiers (WL1–WL5) before moving to the next database. Docker containers were restarted between database evaluations to ensure a clean environment and minimize cache carryover effects. For each query, an initial warm-up execution was performed and discarded, followed by three measured runs to account for any cold-start effects. Each query was executed three times per WL, with the average latency recorded as the final metric. Query execution order remained consistent across all databases to ensure comparable conditions.

Query correctness was validated by logging and comparing row counts returned by each database for identical queries. This ensured that all TSDBs produced consistent results before performance metrics were analyzed. In cases where queries failed due to memory constraints or timeouts, the failure was recorded and the corresponding performance metric was marked as unavailable (NA) for that WL-database combination.

4.7.1 TimescaleDB

TimescaleDB was deployed within a lightweight Docker container on a macOS host machine. The container was initialized with PostgreSQL version (14.17).

Docker was run with the dataset path added as a volume, ensuring the CSV files were accessible at a known path (`-v /Users/siminshehbaz/datasets:/datasets`) inside the container.

A database `energydb` was created to ensure a testing platform. TimescaleDB was then enabled in the `energydb` database. It only needs to be done once per database, by using the command `CREATE EXTENSION IF NOT EXISTS timescaledb`;

With the initializations complete, tables were individually created for all 10 WLs (5

wide and 5 narrow). To measure hypertable efficiency, tables were created without hypertables for ingestion testing, then the same ingestion tests were repeated for testing with hypertables. The `COPY` function was used to ingest the WLs from the source folder on the host machine, into the DB.

Hypertables were created using the function: `SELECT create_hypertable('WL', 'time', chunk_time_interval => INTERVAL '1 day');` The same was repeated for WLs ingested into hypertables.

To ensure the complete data was copied, the number of rows were recorded post ingestion. Tables were truncated before being re-ingested.

Post ingestion, the storage footprint was recorded. The below measures the size of each chunk and returns the sum of all the chunks of the table.

```
SELECT
sum(pg_total_relation_size(format('%I.%I', chunk_schema,
chunk_name)
::regclass))
AS total_bytes,
pg_size_pretty(sum(pg_total_relation_size
(format('%I.%I', chunk_schema, chunk_name)::regclass)))
AS total_pretty
FROM timescaledb_information.chunks
WHERE hypertable_name = 'WL';
```

Lastly for ingestion, the compression is measured. It is important to note that compression only works on hypertables. Compression needs to be enabled in these tables first. Enabling compression only sets the rules (segment/order-by). It does not compress anything yet. Data gets compressed chunk-by-chunk. One can either

manually compress all the existing closed chunks (everything that is already loaded) or add an automatic compression policy that compresses all the chunks that are older than a certain date or time range. Due to this study ingesting historical data, manual chunking is better for this benchmark. Therefore the compression command used is:

```
ALTER_TABLE public.WL
SET (timescaledb.compress_segmentby = 'house\_id',
    timescaledb.compress_orderby  = 'time DESC');
SELECT compress_chunk(show_chunks('public.WL'));
```

To optimize query performance, composite indexes were created on filtering columns for both wide and narrow formats. For wide-format tables, indexes were created on `(house_id, time DESC)` to support household-specific time-range queries. For narrow-format tables, indexes were created on `(house_id, metric, time DESC)` to enable efficient filtering by both household and appliance type. These indexes were applied consistently across all WL tiers (WL1–WL5) prior to query benchmarking.

```
CREATE INDEX idx_WL_wide_house_time
  ON WL_wide(house_id, time DESC);


CREATE INDEX idx_WL_narrow_house_metric_time
  ON WL_narrow(house_id, metric, time DESC);
```

Queries are run only on the compressed tables. The query mainly measures the time taken and the number of rows returned. To make the reads faster, chunk skipping was switched on, and skipping was enabled on real filter columns. This increases the read speed by skipping irrelevant chunks. It basically adds a tiny "min/max map"

for house_id per chunk so the planner can skip whole chunks when the query filters on house_id. It complements compression (segment_by=house_id, order_by=time) but works at the chunk level, not inside a chunk.

The same is implemented for all the WLs tested in TimescaleDB.

4.7.2 QuestDB

Implementation in QuestDB began with pulling the latest QuestDB version using Docker, and schemas were defined for both wide and narrow formats. The data files were mounted using a volume, allowing the CSV files to be accessible by QuestDB. This DB had an SQL console on port `http://localhost:9000/index.html`, making it more user-friendly. Ingestion was performed through both full CSV uploads (for smaller WLs) and a chunked streaming approach using Python and the /imp HTTP endpoint (for larger WLs). Python scripts were generated to create the tables first in wide format, and then in narrow format. QuestDB has an HTTP API endpoint called `/exec` that takes SQL queries via URL and returns them as A JSON object. Using pythons requests, makes this simple and portable.

After table creation for all their respective WLs, another script was run to ingestion the WLs. For larger datasets (e.g., WL3–WL5), ingestion was completed in batches, with verification of row counts post-upload and measurement of on-disk footprint via `docker exec du -sh`. The chunk sizes used were `1_000_000`, meaning one million rows per chunk. Ingestion leveraged pandas.read_csv with specified chunk sizes to manage memory usage efficiently. For narrow-format datasets, a symbol-indexed schema with timestamp partitioning was used, supporting ingestion of over 600 million rows before reaching memory constraints.

To mitigate ingestion lag and memory pressure, chunk uploads were paced using

polling to confirm row commits between batches. Additionally, an automated benchmarking framework was developed to evaluate six analytical queries (Q1–Q6), capturing metrics like query latency, result size, throughput, CPU usage, and memory change. Each query was executed three times to compute average metrics.

4.7.3 ClickHouse

ClickHouse was containerized using Docker and set up with ports exposed for HTTP (8123), native protocol (9000), and Prometheus-compatible monitoring (9009). As in the case with docker, the data files were mounted using a volume, allowing the CSV files to be accessible by ClickHouse. The original CSV files, which had inconsistencies in timestamp formatting and column naming (e.g., "House ID", "Power Consumption"), were cleaned using Python scripts to ensure compatibility with ClickHouse's ingestion engine. Timestamps were normalized to YYYY-MM-DD HH:MM:SS, and all headers were converted to lowercase with underscores to match SQL naming conventions. A dedicated database `benchmark_db` was created to isolate benchmarking WLs. ClickHouse's equivalent of Hypertables and chunking is done by using `PARTITION BY, ORDER BY` and `MergeTree` tables. MergeTree is the default engine for fast reads and writes. `ORDER BY` defined the primary index key—this is the order in which ClickHouse organizes its rows inside each part of the disk. Index granularity controls how dense the primary index is. Each ORDER BY index is stored for one granule of rows. And one granule holds 8192 rows—this is the index granularity used for table creation. Smaller granularity means more index writes but faster point looks up. After creating the DB, the table is created using time and `Float32 CODEC(Gorilla, ZSTD)` for the float values. The engine used is `MergeTree`. It is partitioned by time and ordered by house_id, time. The `INSERT` function is used to copy the data from the local machine into the database. The size is calculated using:

```
SELECT
  formatReadableSize(sum(bytes_on_disk))
  AS total_size,
  formatReadableSize(sum(data_uncompressed_bytes))
  AS uncompressed_size,
  round(sum(data_uncompressed_bytes) / sum(bytes_on_disk), 2)
  AS compression_ratio
FROM system.parts
WHERE database='energy_test' AND table='wl1_wide' AND active;
```

For larger WLs (from WL3 onward), the index granularity is increased to 16384. For ingestion the metrics logged were time, rows returned, throughput, original size, compression ratio, and compressed size. To ensure accurate storage readings, OPTIMIZE TABLE FINAL was invoked post-ingestion to flush data to disk. For query benchmarking, six WL-specific queries (Q1–Q6) were executed. The metrics measured were latency, rows returned, and throughput.

4.7.4 InfluxDB

InfluxDB v1.8 was installed and deployed within a Docker container. InfluxDB v1.8 was selected because it provides full native support for the InfluxQL query language, whereas newer InfluxDB versions have undergone significant architectural changes in versions 2.x and 3.x and do not offer equivalent first-class support for InfluxQL, making them unsuitable for the query workload evaluated in this study. Results presented here reflect the v1.8 architecture and should not be generalized to newer versions.

InfluxDB does not allow direct ingestion, it needs the data files to be converted to InfluxDB's Line Protocol (LP). To evaluate InfluxDB, the process began with the conversion of WL-specific CSV files (e.g., `wl1_wide`, `wl1_narrow`) into InfluxDB's LP format [17]. This was done using a custom Python script that streamed each row of data into LP-compliant lines, converting timestamps from the original `YYYY-MM-DD HH:MM:SS` format into epoch seconds. The scripts were designed to handle both wide and narrow schema variations and supported chunked processing for scalability with large files.

Influx did not require the files to be mounted. InfluxDB is schemaless so it doesn't actually create tables. InfluxDB stores data in measurements (not tables) inside a database. Each measurement is created automatically when data is inserted and stored inside shards (time-partitioned folders). Inside each shard, `.tsm` files store compressed time series data and `.wal` files temporarily buffer writes. Each `.tsm` file holds one or more measurements—their fields, tags, and timestamps.

Influx does not store each measurement in a separate file—measurements are grouped in shards by time range. Each "table" is identified using a unique measurement. When writing the Python script, this was implemented. As long as the measurement names differ in the lp files, one can safely insert all of the WLs into the same DB. The line protocol stores data in the form of:

```
<measurement>,<tags> <fields> <timestamp>
wl1_wide,house=H1 value=10 1234567890
wl1_narrow,house=H1,appliance=A1 value=5 1234567890
```

The 10 WLs each have a unique measurement name (WL1_wide, WL1_narrow, WL2_wide, etc.), and they will all be stored logically separately, even though they may share the same shard folder.

All ingestion operations were directed at a single database named `energy_data`, with each WL ingested into a separate measurement (e.g., `wl1_wide`, `wl1_narrow`) to allow isolated tracking and evaluation.

Pythons scripts were used to ingest the LP formatted files into InfluxDB via HTTP. The ingestion pipeline was chunked to avoid memory overload. It measured total ingestion time, number of inserted rows, and estimated on-disk storage usage. In InfluxDB v1.8, indexing and partitioning are mostly automatic and fixed. There is limited manual control, but it is considered good enough for most time-series WLs.

Query benchmarking was conducted using a suite of six queries (Q1–Q6) adapted to InfluxQL. Each query was executed three times, and performance metrics logged were latency, rows returned and throughput (rows/sec). However, InfluxDB does not support `date_trunc`. InfluxDB also does not support subqueries inside COUNT(*). Query results were exported to individual Excel files for inspection, and aggregate benchmarking metrics were saved in CSV format. These limitations during execution will be discussed in detail in Chapter 5.

4.7.5 Apache IoTDB

IoTDB 2.0.5 was installed on Docker. Python scripts insert the wide and narrow CSV files into IoTDB using its Session API. Following TSM-Bench, it was executed with batched writes, weekly partitions, and parallel ingestion. It automatically detects CSV headers, chunks the file to control memory use, and leverages multi-threaded sessions for high throughput.

IoTDB does not have the same model as relational systems like TimescaleDB or PostgreSQL. It is hierarchical, meaning that data organization depends on how the paths are defined (device and measurement structure). IoTDB uses storage groups,

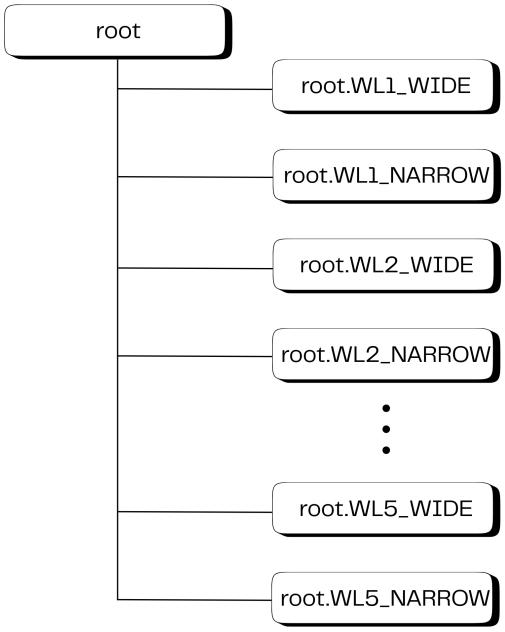


Figure 4.1: Apache IoTDB hierarchical storage structure showing root-level organization of workload storage groups

devices, and measurements instead of databases and tables. For example:

```
root.<storage_group>.<device>
root.WL1.house1
root.WL1.house1
root.WL1.house2
```

For this study, each WL was created as a separate storage group. So, rather than having multiple SQL tables, everything under root belongs to the same storage group, essentially acting as its own database. The logical structure is shown in Figure 4.1. The schema inside the WLs has not been shown in this figure since the data storage would look different for wide and narrow format.

The logical structure for wide-format is shown in Figure 4.2. In this figure, each house

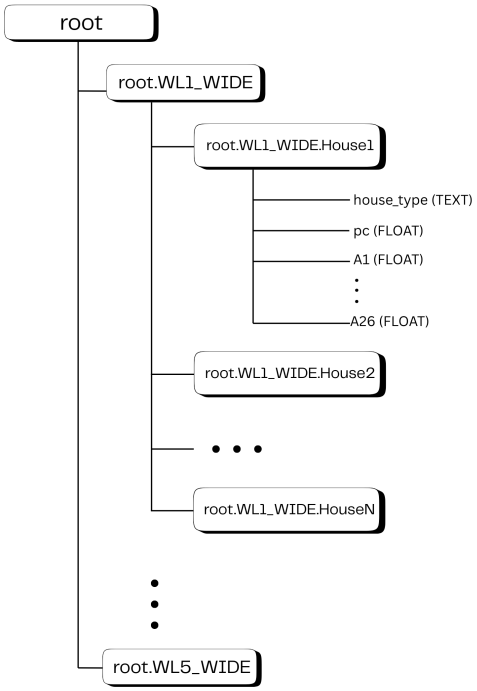


Figure 4.2: Apache IoTDB wide-format structure

is treated as a device. All measurements are stored under `root.WL_wide.HouseX`.

The logical structure for narrow-format is shown in Figure 4.3. As opposed to the wide-format, where all the values are stored under the same measurement, in this narrow-format, there is another layer called 'metric'. Each value is stored under its own metric, and there are 26 such metrics under each 'House' subgroup.

4.8 Summary

This chapter motivated the study and outlined its methodology. It began by describing the data generation process and the resulting dataset characteristics, then compared the wide and narrow data formats. Next, it detailed the data preprocessing steps and the file systems used for initial storage. The selection of time-series databases (TSDBs) was explained, including the inclusion and exclusion criteria.

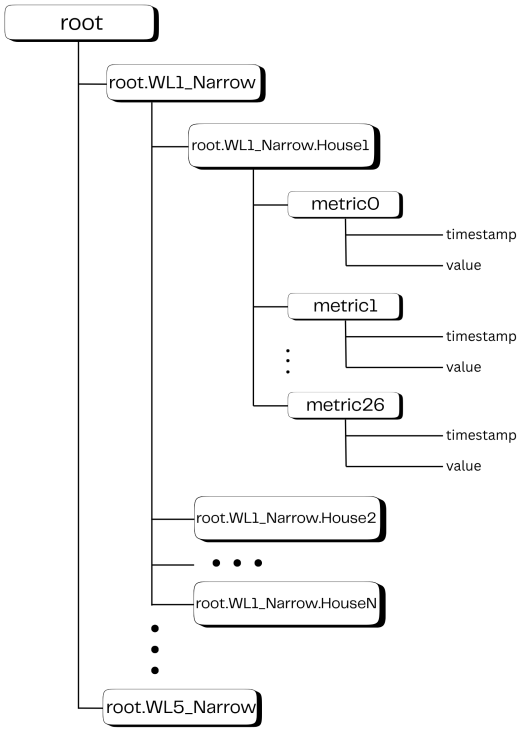


Figure 4.3: Apache IoTDB narrow-format structure

The core of the chapter presented the benchmark design—covering WLs, ingestion strategy, and query design—followed by the execution procedure for each TSDB. The next chapter presents the results and analysis.

Chapter 5

Results and Analysis

This chapter presents comprehensive performance evaluation of the five selected TSDBs. The metrics measured are ingestion throughput, storage efficiency, and query execution latency. Section 5.1 analyzes ingestion performance individually per TSDB followed by a cross-database comparison. Section 5.2 evaluates storage efficiency and compression characteristics. Section 5.3 benchmarks query performance across six representative analytical queries (Q1–Q6), excluding InfluxDB and Apache IoTDB due to fundamental query language limitations documented in Section 5.3.

5.1 Ingestion Performance Analysis

This section analyzes ingestion performance for all five TSDBs–QuestDB, InfluxDB, TimescaleDB, ClickHouse, and Apache IoTDB, across all WLs (WL1–WL5) and both data formats (wide and narrow).

WL	Data Format	Metrics Tested			
		Ingest time (No-HT) [s]	**Ingest time (HT) [s]**	**Rows ingested**	**Throughput**
WL1	Wide	2.692	2.823	144,000	53491.83
	Narrow	5.361	10.389	3,888,000	725237.83
WL2	Wide	26.718	28.476	1,440,000	53896.25
	Narrow	52.644	106.137	38,880,000	738545.70
WL3	Wide	83.036	88.851	4,464,000	53759.82
	Narrow	169.436	438.051	120,528,000	711348.24
WL4	Wide	838.747	951.896	44,640,000	53222.25
	Narrow	1767.679	4523.940	1,205,280,000	681843.25
WL5	Wide	752.711	833.396	52,560,000	69827.60
	Narrow	1629.613	15407.675	1,419,120,000	870832.52

Table 5.1: Ingestion metrics per WL in TimescaleDB

5.1.1 TimescaleDB Ingestion Performance

TimescaleDB's ingestion performance was evaluated, both, with and without hypertables (HT) to isolate the impact of this time-series optimization feature. Throughput (TP) is calculated as the total number of rows ingested divided by the total ingestion time, without HTs. Table 5.1 summarizes the results across all WLs.

5.1.1.1 Wide-Format Performance

Wide-format ingestion demonstrates a remarkably stable TP across WL1–WL4, maintaining 53,000–54,000 rows per second without HTs. This consistency holds regardless of WL size, indicating TimescaleDB's row-oriented PostgreSQL foundation handles wide schemas predictably at moderate scales.

WL5 exhibits anomalous behavior, achieving 69,828 rows/sec—a 31% improvement

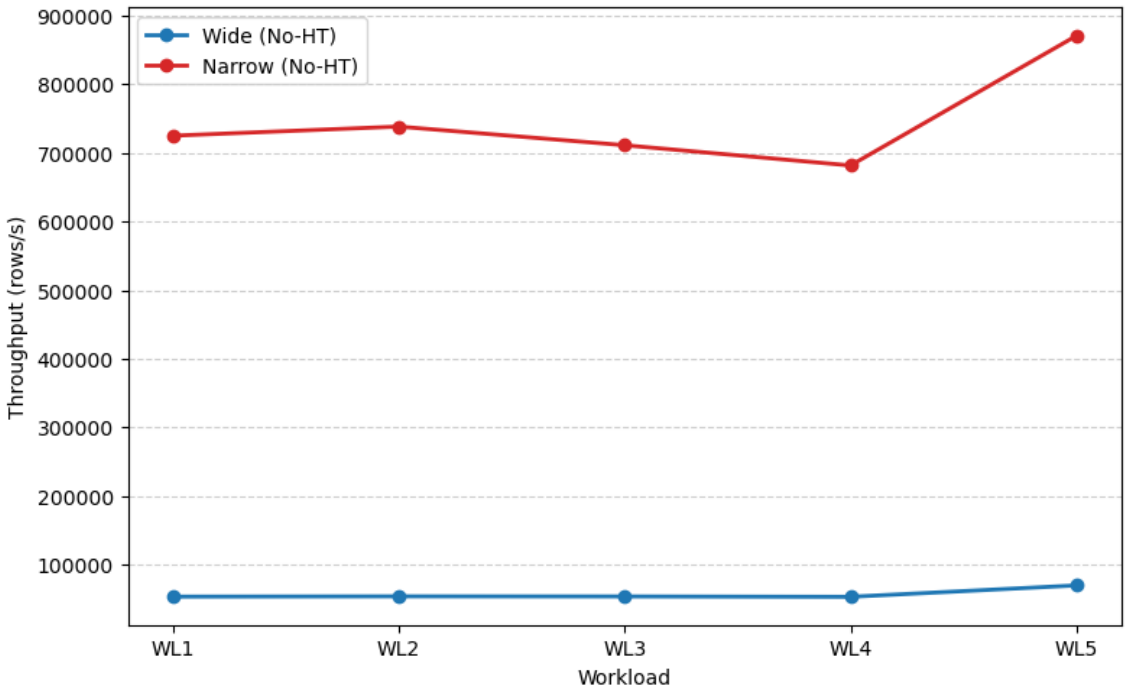


Figure 5.1: TimescaleDB Throughput Comparison

over WL1-WL4. This is notable because WL4 and WL5 contain similar total row counts (44.6M vs 52.6M rows), yet differ in structure: WL4 represents 31 days across 1,000 houses, while WL5 represents 365 days across 100 houses. The performance improvement could be because of differences in data distribution patterns or PostgreSQL's handling of per-house write patterns. However, the exact mechanism remains unclear, warranting further investigation.

5.1.1.2 Narrow-Format Performance

Narrow-format ingestion reports substantially higher TP values: 681,843–870,833 rows per second, appearing 13–16× faster than wide format (Figure 5.1). However, this comparison requires careful interpretation. Each narrow-format row contains a single measurement value, while each wide-format row contains 27 values. The higher row-based TP reflects simpler row structure rather than necessarily superior overall performance.

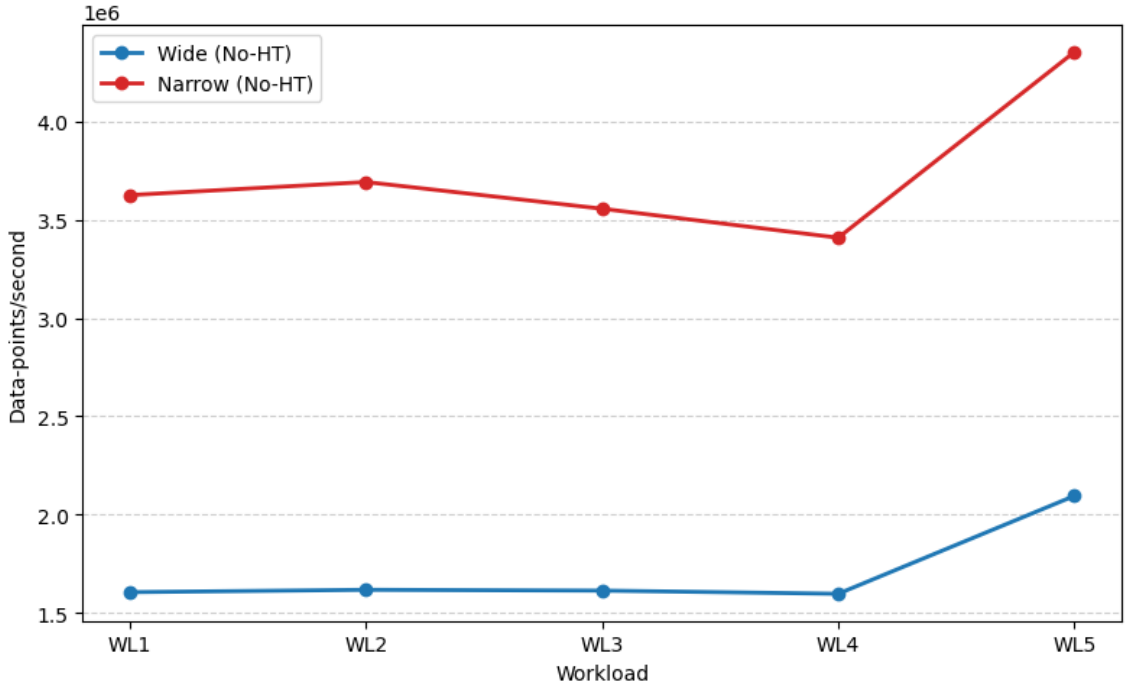


Figure 5.2: TimescaleDB Data-points Ingested

Figure 5.2 provides a normalized view by measuring data-points processed per second (rows × values per row). This metric reveals that narrow format genuinely processes approximately 2× more information per second than wide format. Without HTs, Timescale's write path handles simpler narrow-format rows more efficiently than complex wide-format rows

The narrow format's superior performance, however, comes with a cost: it incurs severe overhead when TimescaleDB's time-series optimizations are enabled.

5.1.1.3 HT Overhead

Figure 5.3 reveals the most significant finding of this analysis–HT overhead varies drastically based on data format and time span.

Wide-Format HT overhead shows minimal, consistent HT overhead across all WLs. The additional time represents the cost of chunk management and metadata main-

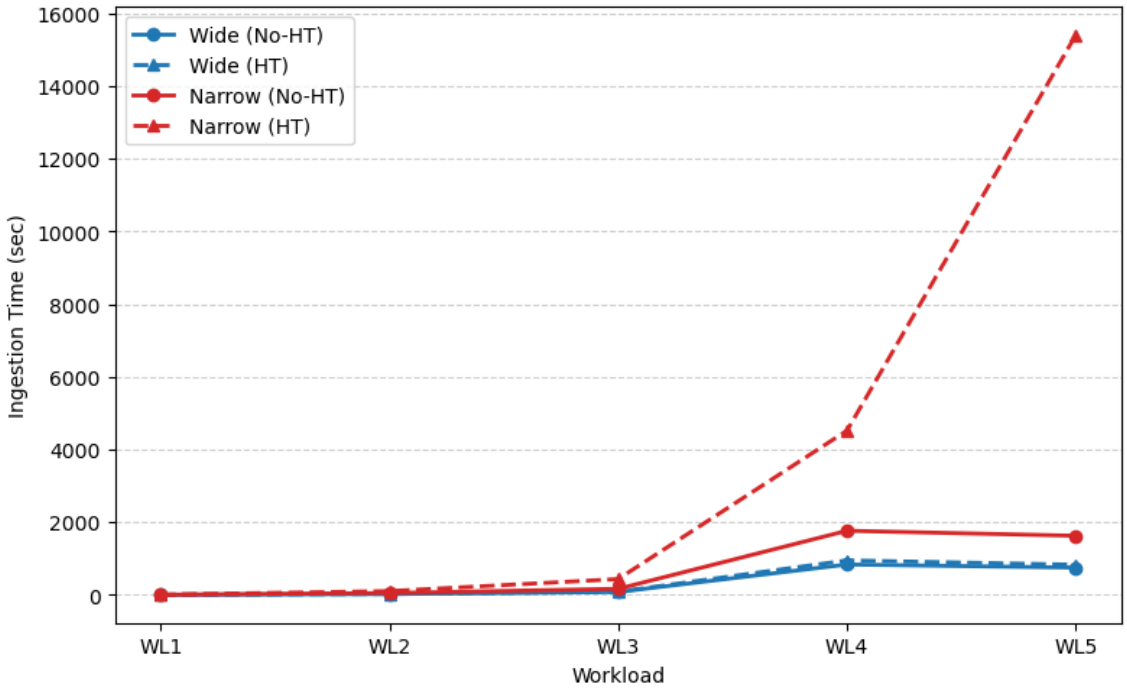


Figure 5.3: TimescaleDB Latency Comparison: With and Without HTs

tenance. This overhead remains manageable and predictable across all scales. WL1-WL4 shows a 5–13% overhead, and WL5 an 11% overhead.

Narrow format exhibits progressively worsening HT overhead. The WL5-narrow result is rather striking–a 4.3-hour ingestion time compared to 27 minutes without HTs, or 14 minutes for wide format with HTs. This represents a complete breakdown of scalability for long-duration, high-cardinality datasets.

5.1.1.4 WL5-Narrow Performance Collapse

The catastrophic WL5-Narrow performance stems from the interaction between chunk interval configuration and data cardinality. Although WL4 and WL5 contain similar total row counts (1.2B vs 1.4B rows), WL5 takes 3.4× longer with HTs enabled. The critical difference lies in their time spans:

- WL4: 31 days of data → 31 HT chunks

- WL5: 365 days of data → 365 HT chunks

This study used 1-day chunk intervals for all workloads. While this configuration works well for wide format (100 time series), it creates severe overhead for narrow format's high cardinality (2,700 time series: 100 houses × 27 metrics). Each chunk creation triggers index maintenance, constraint validation, and stats updates for every time series. At WL5-Narrow's scale, these per-chunk operations dominate ingestion time. This finding highlights the critical importance of chunk interval sizing in TimescaleDB. Larger chunk intervals would likely mitigate this overhead, although testing alternative chunk sizes was outside the scope of this study.

5.1.1.5 Key Findings

TimescaleDB demonstrates a clear architectural preference for wide-format schemas in appliance-level energy monitoring scenarios. While narrow format achieves 2× higher raw data processing rates without HTs, enabling TimescaleDB's time-series optimizations introduces prohibitive overhead for high-cardinality narrow data with fine-grained (1-day) chunk intervals. The following observations were made for this TSDB:

- Wide format is strongly recommended for datasets spanning weeks to years, offering predictable linear scaling and 4–18× faster ingestion with HTs enabled.
- Narrow format should be avoided for long-duration datasets unless chunk intervals can be significantly increased (e.g., weekly or monthly chunks). The schema flexibility comes at severe performance cost with default configurations.
- Chunk interval tuning is critical and represents an important area for future optimization. The WL5 performance collapse demonstrates that default chunk sizing may not be appropriate for high-cardinality time-series data.

WL	Data Format	Ingestion time (s)	Rows ingested	Throughput
WL1	Wide	0.28	144,000	514,285.71
	Narrow	1.47	3,888,000	2,644,897.96
WL2	Wide	3.06	1,440,000	470,588.24
	Narrow	18.07	38,880,000	2,151,632.54
WL3	Wide	11.05	4,464,000	403,981.90
	Narrow	50.72	120,528,000	2,376,340.69
WL4	Wide	149.31	44,640,000	298,975.29
	Narrow	*Failed**	NA	–
WL5	Wide	732.80	52,560,000	71,724.89
	Narrow	*Failed**	NA	–

Table 5.2: Ingestion metrics per WL in QuestDB. *WL4-narrow and WL5-narrow ingestion failed due to memory exhaustion and write back-pressure.

These findings align with TimescaleDB's documented best practices recommending moderate cardinality (<1,000 unique series per chunk) and emphasizing the importance of chunk interval selection for production deployments [39].

5.1.2 QuestDB Ingestion Performance

QuestDB demonstrated strong performance on small-to-medium workloads but encountered critical scalability limitations with large narrow-format datasets. Table 5.2 summarizes the ingestion results.

5.1.2.1 Wide-Format Performance

As visible in Figure 5.4, wide-format ingestion shows progressive TP decline as workload size increases. WL2 has a 8% decrease from WL1. WL3 shows a 14% decrease from WL2. WL4 degrades further, exhibiting a 26% decrease from WL3. WL5 has

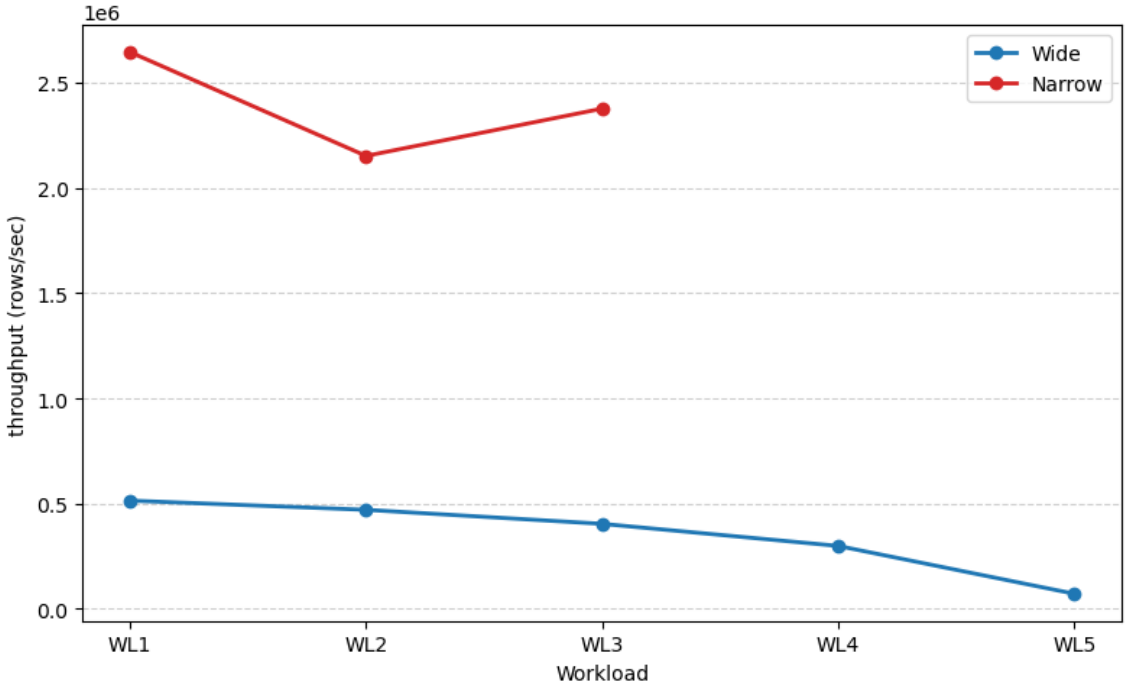


Figure 5.4: QuestDB Throughput Comparison

a sharp TP decline, exhibiting a 76% decrease from WL4.

The dramatic WL5-Wide performance collapse (72K rows/sec) suggests QuestDB is hitting memory or I/O saturation limits within the Docker container's 12GB memory allocation. However, wide-format successfully completes all WLs, demonstrating that QuestDB can handle datasets up to 52.6M rows (9.4 GB) for 30-column wide schemas, albeit with degraded performance at scale.

Figure 5.5 shows the corresponding increase in ingestion time. Wide-format latency grows from 0.28 seconds (WL1) to 732.8 seconds (WL5)—a 2,617× increase for a 365× increase in data volume. QuestDB's columnar write path encounters increasing bottlenecks as dataset size grows, even with only 30 columns.

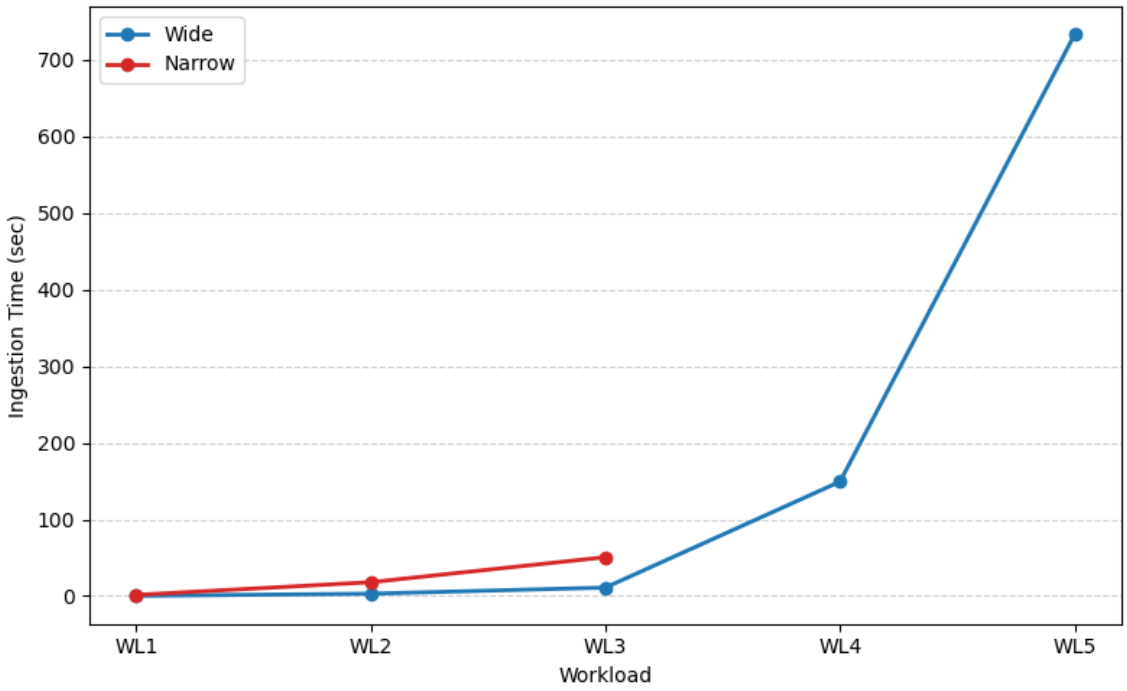


Figure 5.5: QuestDB Latency Comparison

5.1.2.2 Narrow-Format Performance

Narrow-format ingestion maintains strong performance through WL3 but fails at WL4 and WL5. Figure 5.4 illustrates this collapse. QuestDB's TP remains strong through WL3 (2.2–2.6M rows/sec, approximately 5× faster than wide-format). However, this performance advantage is negated by QuestDB's inability to complete large narrow-format ingestion operations.

WL4-Narrow ingestion failed after processing 684 million rows—57% of the 1.2 billion row target. The ingestion process exhibited increasing write back-pressure and commit delays as row count approached approximately 600M rows. Monitoring revealed memory usage climbing to 11.8 GB (98% of 12GB Docker limit), sustained high I/O wait times (over 40%),increasing latency between batch commits (from 1s to 30s), and the process eventually hanging indefinitely (suspended state).

The failure pattern points to memory exhaustion and I/O subsystem saturation.

QuestDB's columnar storage engine must maintain in-memory structures for symbol indexing and write buffers across 2,700 narrow-format time series. At WL4's scale (43.6 GB raw data, 1.2B target rows), these structures exceeded available memory, triggering cascading performance collapse. WL5-Narrow exhibited similar failure patterns at early stages of ingestion, confirming that the scalability cap lies well below 1.4B rows for narrow-format data in this configuration.

5.1.2.3 Key Findings

QuestDB demonstrates strong ingestion performance for small-to-medium scale workloads (WL1–WL3) but encounters critical limitations beyond 100M rows for narrow-format data:

- Wide format is viable through WL5 despite 7× performance degradation in TPs from WL1 to WL5. All wide-format workloads completed successfully, making QuestDB suitable for wide-schema appliance monitoring up to 52.6M rows (year-long, 100-house deployment).

- Narrow format is limited to approx. 684M rows in Docker-constrained environments. The combination of high cardinality (2700) and columnar symbol indexing exhausts available memory, causing complete ingestion failure for workloads exceeding WL3's scale (120.5M rows).

- The 12GB Docker limit proved insufficient for narrow-format ingestion beyond 684M rows. Larger memory allocations (32GB+) or bare-metal deployments might extend scalability, but this was outside the scope of this study.

- Configuration tuning offers limited benefit. Batching and chunking strategies did not mitigate the failure, indicating the limitation is architectural rather

WL	Data Format	Ingestion time (s)	Rows ingested	Throughput
WL1	Wide	0.290	144,000	496,551.72
	Narrow	0.394	3,888,000	9,868,020.30
WL2	Wide	1.172	1,440,000	1,228,668.94
	Narrow	3.373	38,880,000	11,526,830.71
WL3	Wide	2.862	4,464,000	1,559,748.43
	Narrow	14.187	120,528,000	8,495,665.05
WL4	Wide	29.991	44,640,000	1,488,446.53
	Narrow	146.992	1,205,280,000	8,199,629.91
WL5	Wide	45.235	52,560,000	1,161,932.13
	Narrow	200.880	1,419,120,000	7,064,516.13

Table 5.3: Ingestion metrics per WL in Clickhouse

than configurational. While alternative batch sizes or commit pacing strategies might offer marginal improvements, they are unlikely to fundamentally extend the scalability ceiling beyond WL3's successful completion, and such optimization was outside the scope of this study.

These findings indicate that QuestDB is best suited for wide-format schemas or narrow-format deployments with row counts below 120M when operating under typical containerized resource constraints.

5.1.3 ClickHouse Ingestion Performance

ClickHouse demonstrated exceptional ingestion performance across all WLs, successfully completing all tests without failures. Table 5.3 summarizes the results.

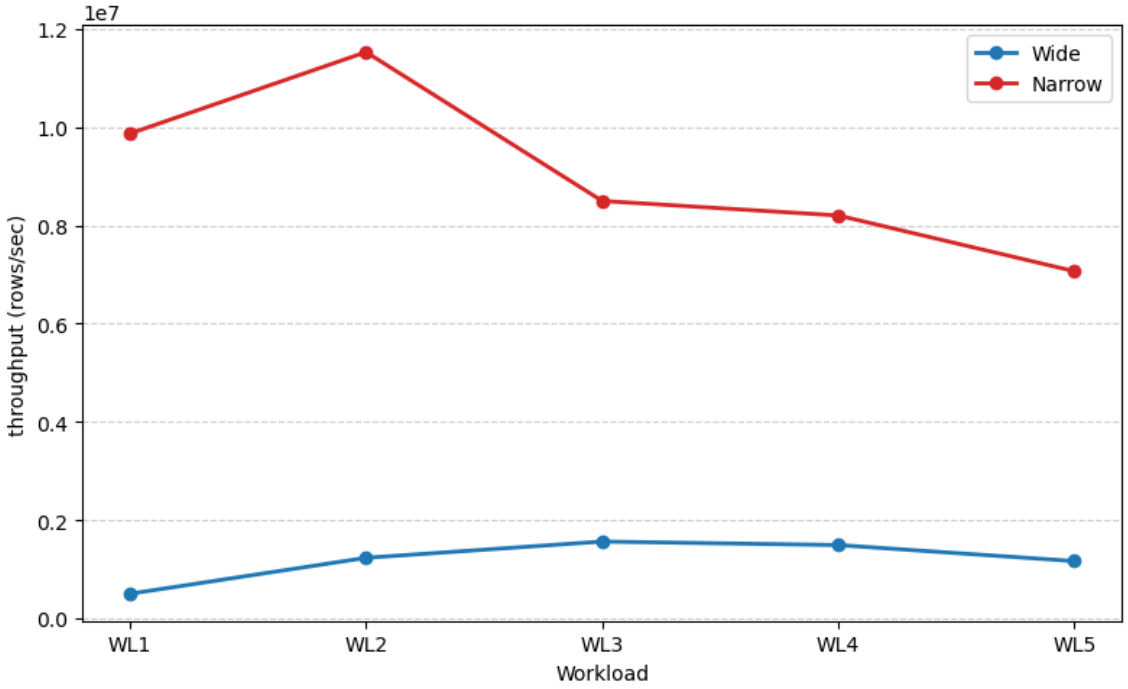


Figure 5.6: ClickHouse Throughput Comparison

5.1.3.1 TP Performance

As shown in Figure 5.6, ClickHouse achieved the highest ingestion TP of all evaluated TSDBs, processing 7–11.5 million rows per second for narrow format and 0.5–1.6 million rows per second for wide format. All WLs completed successfully without failures, memory exhaustion, or performance degradation, demonstrating excellent scalability from 144K rows (WL1-Wide) to 1.42 billion rows (WL5-Narrow). Unlike TimescaleDB and QuestDB, ClickHouse maintained functional performance across all scales, with no catastrophic degradation or hard scalability caps within the tested range. For wide, TP increases from WL1 to WL3. The modest decline at WL4–WL5 (1.5M $\rightarrow$ 1.2M rows/sec) represents only a 20% performance reduction despite a 365$\times$ increase in data volume from WL1, indicating excellent scaling characteristics. Narrow-format ingestion exhibits more complex behavior, with peak performance at WL2 followed by progressive decline.

This pattern can be explained by the interaction between index configuration and data scale. For WL1–WL2, ClickHouse used `index_granularity=8192` (more granular indexing). From WL3 onward, granularity was manually increased to `16384` to reduce index maintenance overhead for larger datasets. TP declined after this optimization. While the coarser granularity (16384) reduces index write overhead, the benefit was outweighed by the dramatic increase in data volume and narrow format's increased cardinality. The net effect: TP peaked at WL2 (11.5M rows/sec with fine-grained indexing) and declined to 7.1M rows/sec at WL5 despite the optimization attempt.

5.1.3.2 Ingestion Latency

Wide-format ingestion demonstrates consistent, predictable performance across all workloads as shown in Figure 5.7. Wide-format ingestion times remain remarkably low–0.29 seconds for WL1–scaling to just 45 seconds for WL5's 52.6M rows, demonstrating ClickHouse's efficiency with moderate-cardinality, wide-schema data.

5.1.3.3 Data-Point Processing Rate

Figure 5.8 normalizes performance by measuring data-points per second (rows $\times$ values per row), revealing actual information processing capability.

Wide format peaks at WL3–46.8M data-points/sec, and declines to 34.8M at WL5 (25.6% reduction).

Narrow format peaks at WL2–57.6M data-points/sec, and declines to 35.3M at WL5 (39% reduction).

Notably, narrow and wide formats converge at WL5 (approx. 35M data-points/sec each), indicating that at billion-row scales, ClickHouse's columnar storage processes

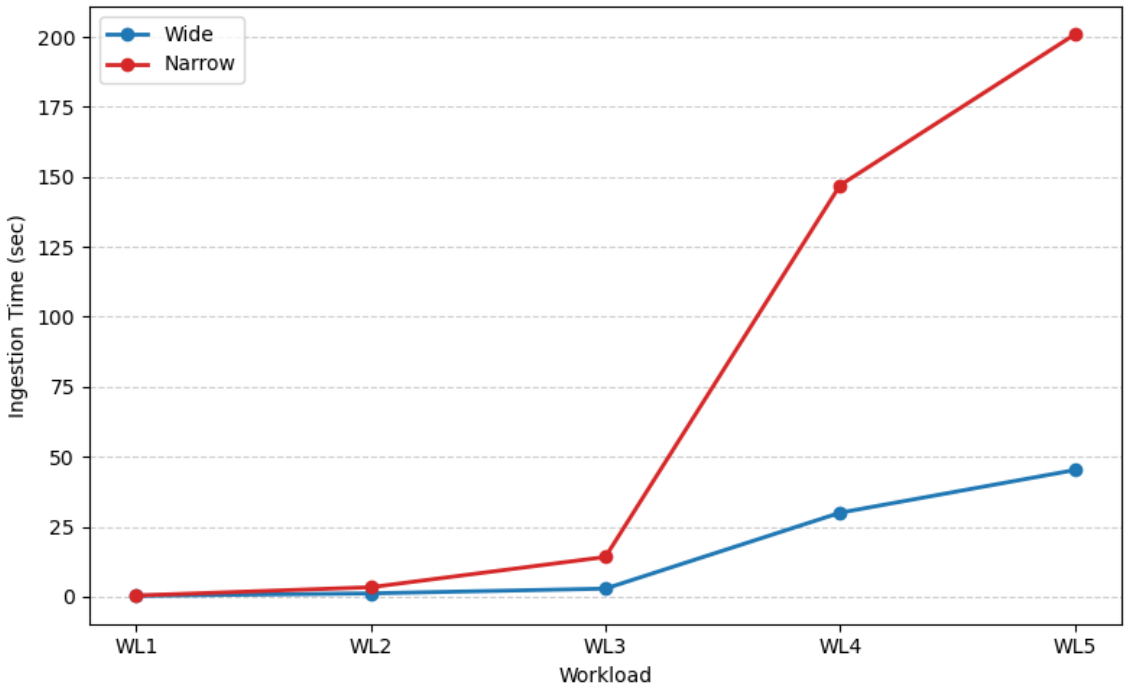


Figure 5.7: ClickHouse Latency Comparison

information at similar rates regardless of schema design. The narrow format's initial advantage (2× faster at WL1–WL2) diminishes as cardinality and index maintenance costs accumulate.

The data-points metric also reveals that wide format's apparent lower TP (1.2M rows/sec vs 7.1M for narrow at WL5) masks comparable actual performance, as each wide row contains 27× more information than a narrow row.

5.1.3.4 Scaling Behavior

ClickHouse exhibits steady scaling for wide format and sub-linear scaling for narrow format, but maintains functionality across the entire tested range. Narrow format's scaling is unsteady likely due to cardinality effects, however, the absolute performance remains exceptional. ClickHouse ingested 1.4 billion narrow-format rows in 3.3 minutes—unmatched by any other database in this study.

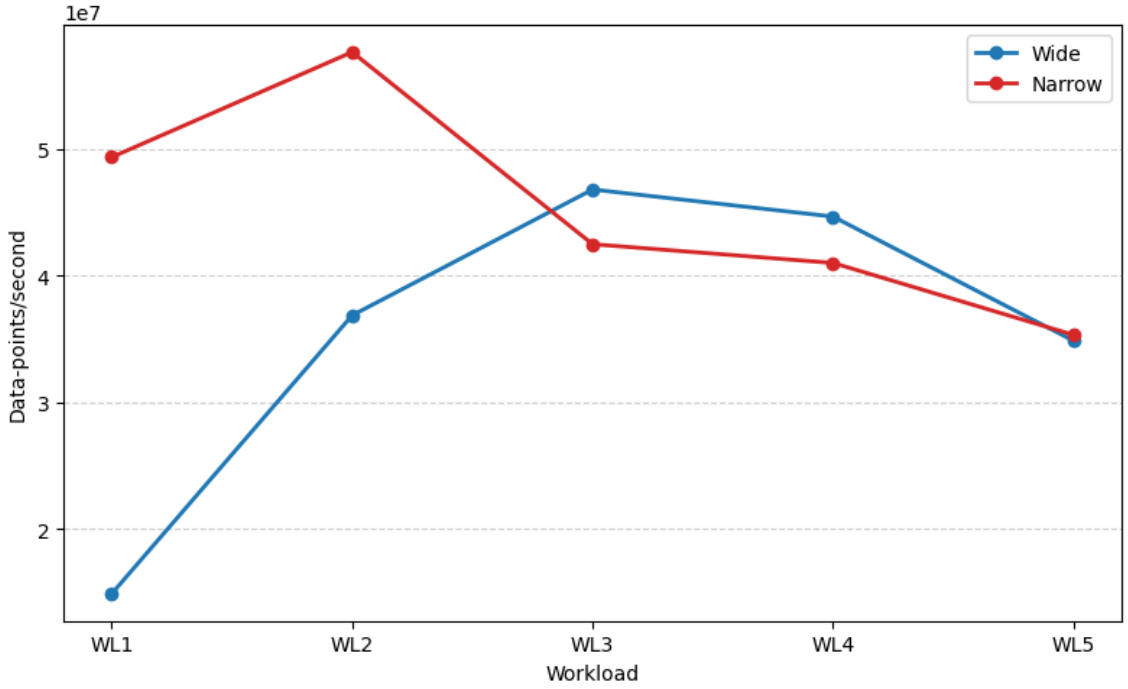


Figure 5.8: ClickHouse Data-points Ingested: Wide vs. Narrow Comparison

5.1.3.5 Key Findings

ClickHouse demonstrates superior ingestion performance for appliance-level energy monitoring regardless of schema designs:

- Highest TP achieved so far: 7–11.5M rows/sec for narrow format, 0.5–1.6M rows/sec for wide format—the fastest of all evaluated databases by significant margins.
- No scalability-related failures: All workloads completed successfully, including WL5-Narrow (1.4B rows, 51.3 GB) where QuestDB failed and TimescaleDB required 4.3 hours.
- Index granularity matters: The configuration change from 8192 to 16384 at WL3 impacted narrow-format performance, demonstrating the importance of index tuning for high-cardinality workloads.

WL	Data Format	Ingest time (s)	Rows Ingested	Throughput
WL1	Wide	2.43	144,000	59,259.26
	Narrow	11.96	3,888,000	325,083.61
WL2	Wide	23.07	1,440,000	62,418.73
	Narrow	102.35	38,880,000	379,872.98
WL3	Wide	68.54	4,464,000	65,129.85
	Narrow	260.86	120,528,000	462,040.94
WL4	Wide	681.82	44,640,000	65,471.83
	Narrow	3,078.62	1,205,280,000	391,500.09
WL5	Wide	827.51	52,560,000	63,515.85
	Narrow	3,527.07	1,419,120,000	402,350.50

Table 5.4: Ingestion metrics per WL in InfluxDB

- Unlike TimescaleDB (which strongly favors wide) or QuestDB (which fails at narrow), ClickHouse handles both schemas efficiently. Wide format offers more consistent performance; narrow format achieves higher peak TP but with gradual degradation at scale.

- Production-ready at tested scales: ClickHouse completed WL5 (year-long, 100-house deployment) in 45 seconds for wide format and 201 seconds for narrow format, making it suitable for production appliance-level energy monitoring deployments without architectural concerns.

5.1.4 InfluxDB Ingestion Performance

InfluxDB v1.8 demonstrated consistent but comparatively modest ingestion performance across all workloads. All tests completed successfully, although with significantly longer ingestion times than other evaluated databases. Table 5.4 summarizes the results.

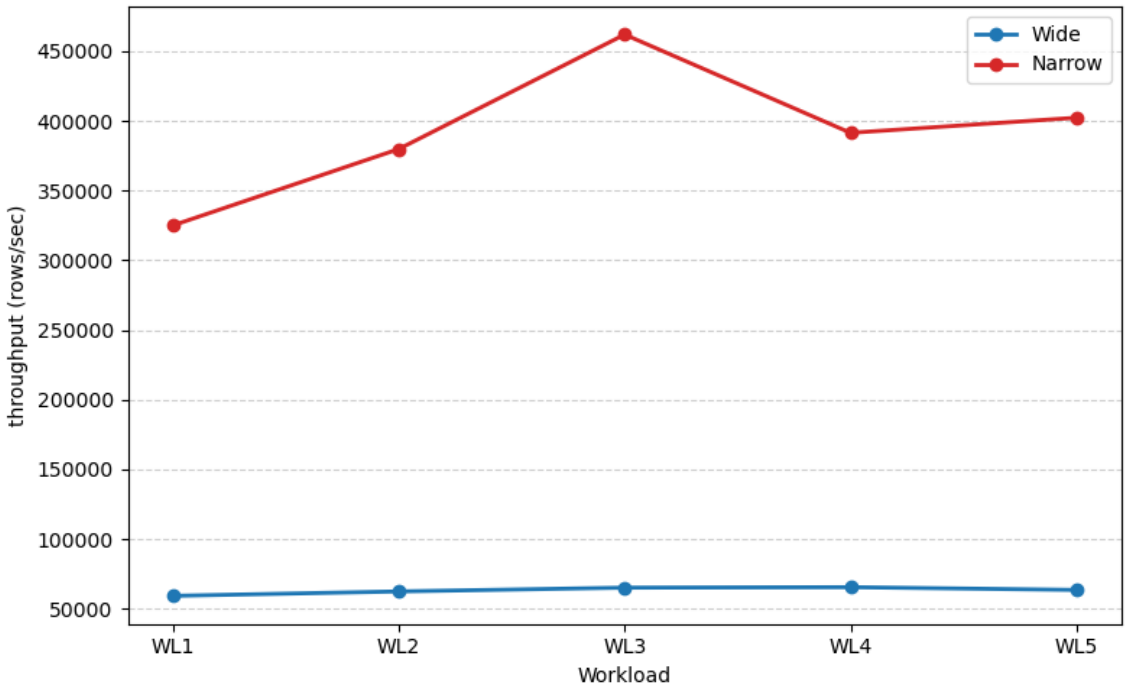


Figure 5.9: InfluxDB Throughput Comparison

5.1.4.1 Implementation Context

Ingestion was performed, using batched HTTP writes, with 10,000 row chunks. This batch size is smaller than those used for other DBs (1M rows for QuestDB, 500K for Apache IoTDB), introducing additional HTTP request overhead. Data was converted from CSV to InfluxDB's LP format prior to ingestion, with each workload stored as a separate measurement within a single database to enable isolated tracking. The smaller batch size was selected based on InfluxDB v1.8 documentation recommendations for avoiding write queue saturation. However, this configuration may have introduced performance penalties compared to larger batch sizes, as discussed in the analysis below.

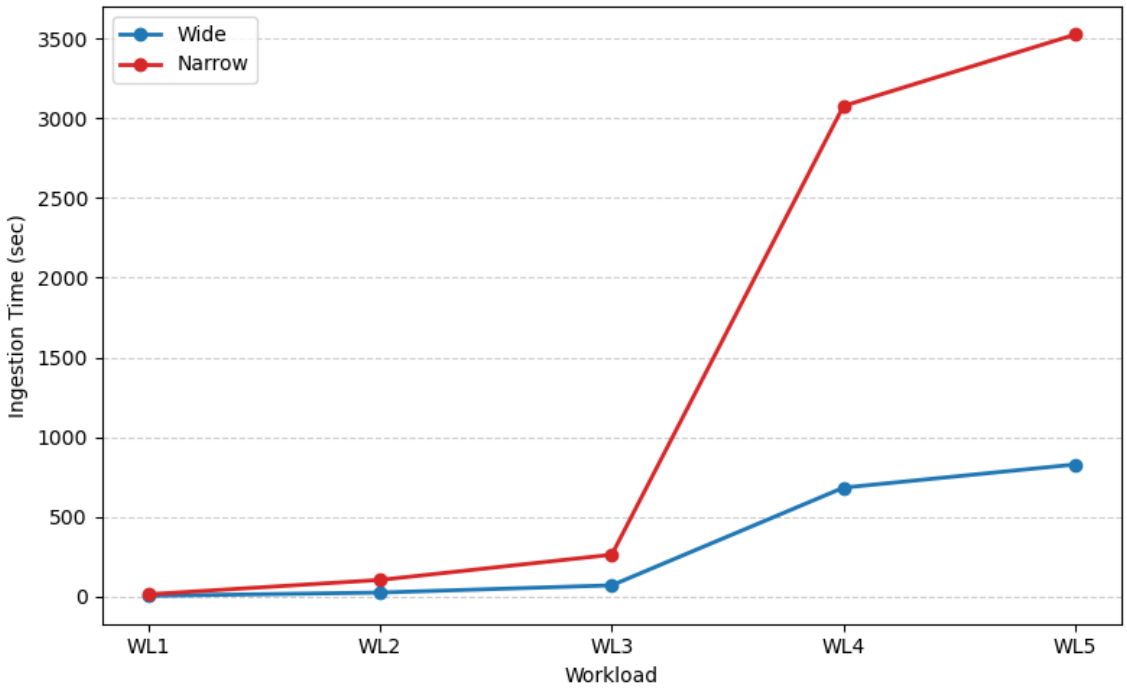


Figure 5.10: InfluxDB Latency Comparison

5.1.4.2 Wide-Format Performance:

Wide-format ingestion shows a remarkably stable TP across all workloads as shown in Figure 5.9. WL2 exhibits a modest 5% increase over WL1. WL3 achieves peak TP (65,130 rows/sec), a 10% improvement over WL1. WL4 maintains this peak performance, and WL5 shows a slight 3% decrease from WL4.

InfluxDB's write path handles moderate-cardinality wide-format data predictably. TP remains within a narrow 59K-65K rows/sec range despite a 365× increase in data volume. However, this stable performance comes at the cost of speed. InfluxDB's wide-format TP is 18× slower than ClickHouse (1.2M rows/sec) and comparable to TimescaleDB's wide-format performance (70K rows/sec).

Figure 5.10 shows the linear scaling of ingestion time with data volume. WL5-Wide requires 828 seconds (13.8 minutes) to ingest 52.6M rows, compared to ClickHouse's 45 seconds for the same workload—an 18× difference in wall-clock time.

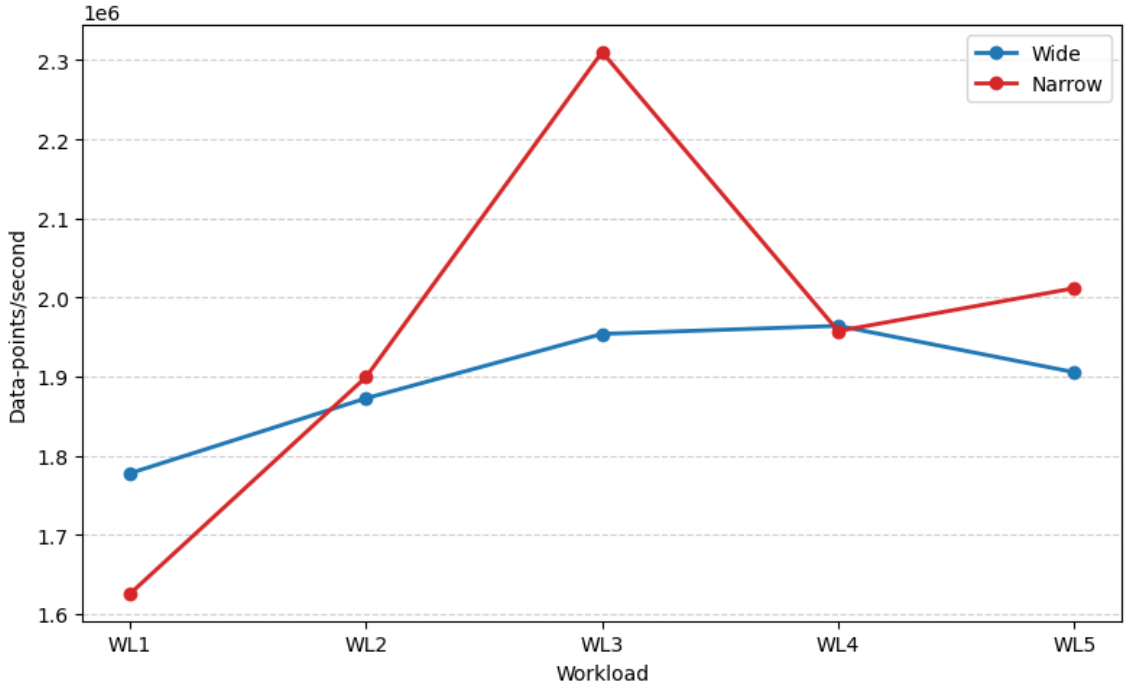


Figure 5.11: InfluxDB Datapoint Comparison

5.1.4.3 Narrow-Format Performance

Narrow-format ingestion exhibits unexpected behavior, with peak performance at WL3 followed by degradation at larger scales (Figure 5.9). The WL3 peak may reflect an interaction between InfluxDB's internal buffering, the 10K batch size, and the specific scale of WL3 (120.5M rows). At this scale, the write path may achieve optimal efficiency before encountering bottlenecks at WL4-WL5 scales. More critically, narrow-format ingestion time explodes at WL4. This worse-than-linear scaling suggests that InfluxDB v1.8 encounters increasing overhead as narrow-format cardinality (2,700 time series) and row count grow. The 51-minute WL4 ingestion time is particularly concerning for production deployments, especially compared to ClickHouse's 2.4-minute completion of the same workload.

5.1.4.4 Data-point Processing Rate

Figure 5.11 reveals interesting convergence behavior when normalizing by data-points per second (rows × values per row). Wide-format peaks at WL3-WL4 (1.96M data-points/sec), and decreases slightly to 1.90M at WL5. Narrow format peaks at WL3 (2.31M data-points/sec) and declines to 2.01M at WL5 (13% reduction). The narrow schema spike at WL3 possible occurs due to smaller writes with fewer fields per measurement being more efficient to process and index, hitting an optimal throughput point before other system bottlenecks (disk I/O, CPU) start dominating at higher workloads. Wide schemas suffer from greater indexing overhead and memory pressure, preventing them from achieving the same peak performance.

Like ClickHouse, InfluxDB shows near-parity between formats at WL4-WL5 (approx. 2M data-points/sec each). This suggests that InfluxDB v1.8's TSM storage engine processes individual values at similar rates regardless of schema design, with both formats converging at the same fundamental write TP caps.

Notably, narrow format's initial advantage (2.3M vs 1.8M data-points/sec at WL1-WL2) diminishes as scale increases, indicating that high-cardinality overhead accumulates with data volume.

5.1.4.5 Key Findings

InfluxDB v1.8 demonstrates functional but comparatively slow ingestion performance for appliance-level energy monitoring:

- Slowest overall performance: 60-65K rows/sec for wide format, 325K-462K rows/sec for narrow—substantially slower than all other evaluated databases.
- Batch size likely constrained performance: The 10K batch size (100× smaller

than other databases) introduced significant HTTP overhead. A 10K batch size at 1.2B rows means 120,000 HTTP requests. At a 1M batch size, it would require only 1,200 requests (100× less). Each HTTP request incurs network latency, header parsing, line protocol parsing, and write queue insertion overhead. Testing with larger batches (100K-1M rows) would likely improve TP substantially.

- WL4-Narrow's 51-minute ingestion time (11.8× increase for 10× data growth) indicates architectural limitations at billion-row scales for high-cardinality data in v1.8.

- For deployments that can tolerate 14-minute ingestion times for year-long datasets, InfluxDB v1.8 remains functional.

- These findings reflect v1.8's 2020-era architecture and should not be extrapolated to InfluxDB 2.x or 3.x, which feature fundamentally different storage and ingestion systems.

The combination of small batch sizes and v1.8's architectural limitations positions InfluxDB as the slowest option among evaluated databases for high-volume appliance energy data ingestion. However, the impact of batch size optimization and modern InfluxDB versions remains an open question for future investigation.

5.1.5 Apache IoTDB Ingestion Performance

Apache IoTDB demonstrated moderate ingestion performance with notable variability across workload scales. All tests completed successfully, though with longer ingestion times than ClickHouse and TimescaleDB. Table 5.5 summarizes the results.

Apache IoTDB ingestion utilized the Session API with 500,000-row batches. Data

WL	Data Format	Ingest time (s)	Rows Ingested	Throughput
WL1	Wide	8.46	144,000	17,021.28
	Narrow	36.65	3,888,000	106,084.58
WL2	Wide	71.83	1,440,000	20,047.33
	Narrow	365.59	38,880,000	106,348.64
WL3	Wide	168.90	4,464,000	26,429.84
	Narrow	815.58	120,528,000	147,781.95
WL4	Wide	1,685.78	44,640,000	26,480.32
	Narrow	8,005.71	1,205,280,000	150,552.54
WL5	Wide	2,102.31	52,560,000	25,001.07
	Narrow	14,126.15	1,419,120,000	100,460.49

Table 5.5: Ingestion metrics per WL in Apache IoTDB

was inserted via RPC calls using `insert_aligned_records()`. A key implementation constraint affected performance: IoTDB's aligned timeseries do not fully support TEXT data types (e.g., `house_type`). While numeric columns (FLOAT, INT32) were inserted as aligned timeseries, the TEXT metadata column was included in the same `insert_aligned_records()` call, introducing additional per-record overhead for type handling and storage management.

5.1.5.1 Wide-Format Performance

Wide-format ingestion shows consistent TP across most WLs as shown in Figure 5.12. Performance stabilizes at WL3-WL4 (26K rows/sec) and maintains this level through WL5. The slight WL5 dip suggests minor overhead accumulation. However, TP remains modest compared to ClickHouse (1.2M rows/sec for wide) and comparable to InfluxDB v1.8 (64K rows/sec). Figure 5.13 shows WL5-Wide requiring 2,102 seconds (35 minutes) to ingest 52.6M rows—47× slower than ClickHouse's 45-second

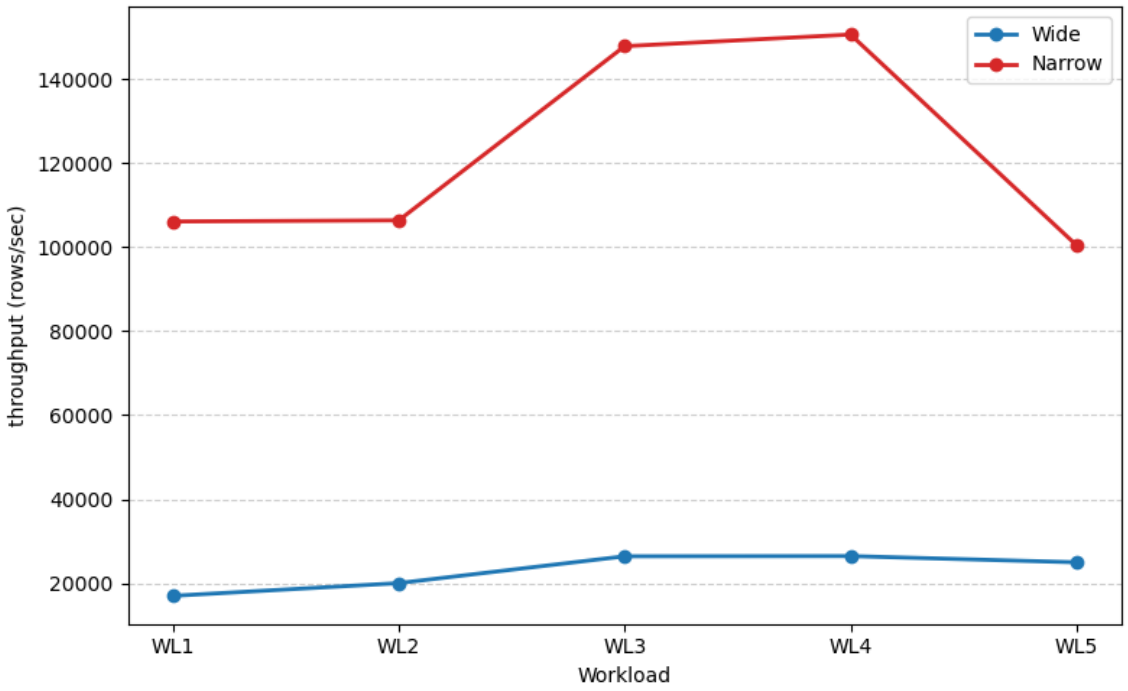


Figure 5.12: Apache IoTDB Throughput Comparison

completion for the same WL.

5.1.5.2 Narrow-Format Performance

Narrow-format ingestion exhibits unsteady behavior, with TP performance peaking at WL3-WL4 before collapsing at WL5 as shown in Figure 5.12. IoTDB's architecture achieves optimal efficiency at the 120M-1.2B row scale for high-cardinality narrow data. However, WL5's 1.4B rows triggers severe performance collapse, requiring 14,126 seconds (3.9 hours) compared to WL4's 8,006 seconds (2.2 hours)—a 1.76× time increase for only 1.18× more data, indicating architectural bottlenecks. Figure 5.13 visualizes this collapse.

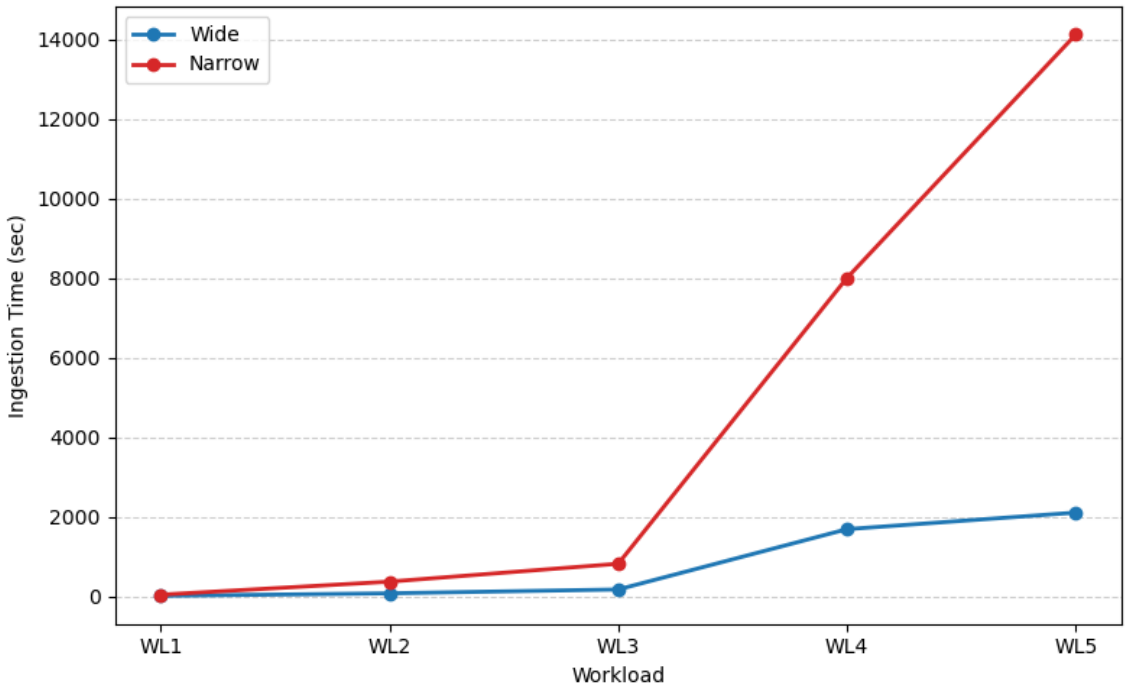


Figure 5.13: Apache IoTDB Latency Comparison

5.1.5.3 Datapoint Processing Rate

Figure 5.14 reveals convergent behavior between formats when normalized to data-points per second (rows × values per row). Wide format peaks at WL3-WL4 (793-796K data-points/sec), and maintains 750K at WL5 (6% decline).

Narrow format exhibits at WL1-WL2: 530K data-points/sec (baseline). At WL3-WL4: 739-754K data-points/sec (40% increase—peak), and at WL5: 502K data-points/sec (33% collapse).

At peak performance (WL3-WL4), both formats converge near 750-800K data-points/sec, suggesting IoTDB's fundamental information processing cap. Wide format maintains this rate at WL5, while narrow format collapses below baseline—a 50% gap emerges (750K vs 502K). Narrow format's high cardinality combined with WL5's 365-day time span overwhelms IoTDB's storage engine. Wide format stays within manageable complexity.

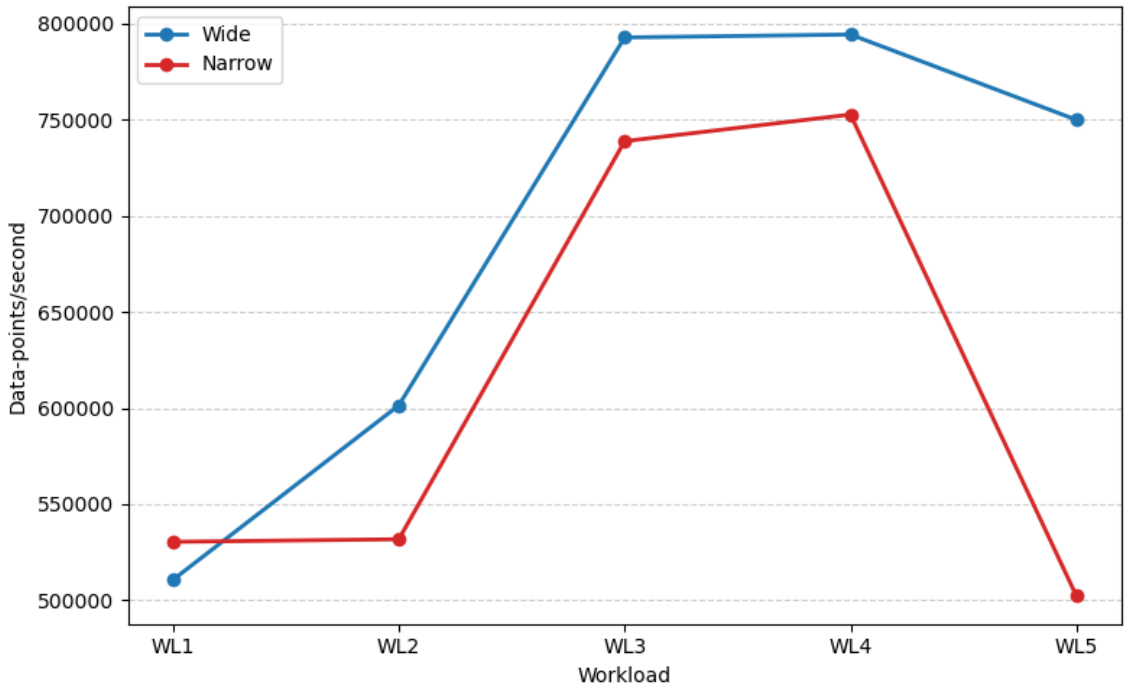


Figure 5.14: Apache IoTDB Datapoint Comparison

5.1.5.4 Key Findings

Apache IoTDB demonstrates functional but architecturally constrained ingestion performance for appliance-level energy monitoring:

- *Moderate TP:* 17-26K rows/sec for wide format, 100-151K rows/sec for narrow—substantially slower than ClickHouse but functional across all tested scales.
- *WL3-WL4 optimization zone:* Narrow format peaks at 148-151K rows/sec for 120M-1.2B row workloads, suggesting IoTDB's architecture achieves optimal efficiency in this range.
- *WL5 performance collapse:* Narrow format degrades 33% at WL5 (1.4B rows), taking 3.9 hours compared to ClickHouse's 3.3 minutes for the same workload. Wide format remains stable, indicating cardinality-driven bottleneck rather

than absolute scale limitation.

- *RPC and TEXT overhead:* IoTDB's Session API uses RPC calls for each batch, introducing per-batch serialization, network transmission, and server-side processing overhead. Even with 500K-row batches, WL5-Narrow requires approximately 2,838 RPC calls (1.4B rows ÷ 500K), each incurring latency. Per-batch RPC calls and TEXT column handling introduce measurable overhead compared to bulk-loading databases. Optimizations like larger batch sizes (1M+ rows) or pre-conversion of TEXT to numeric codes could improve performance.

- *Hierarchical Storage Model:* IoTDB's device-centric hierarchy creates management overhead for high-cardinality narrow format. Each of 2,700 device-metric combinations requires independent TsFile partitions and indexing structures. At WL5's scale (365 days × 2,700 series = 985,500 device-day combinations), this complexity likely triggers index fragmentation and I/O amplification, explaining the WL5 collapse.

- *Time-Order Integrity Guarantees:* IoTDB enforces strict time-order insertion and WAL durability per batch, ensuring data consistency at the cost of write TP. Each RPC call triggers Timestamp validation and ordering checks, WAL flush to disk before acknowledgment, and TsFile indexing and metadata updates. These guarantees provide robustness but reduce raw ingestion speed.

- *Scalability demonstrated:* Successfully completed all workloads including WL5-Narrow (1.4B rows), where QuestDB failed. While slow, IoTDB proves viable for billion-row appliance monitoring deployments when 3-4 hour ingestion windows are acceptable.

These results position Apache IoTDB as a robust but slower option for appliance-level

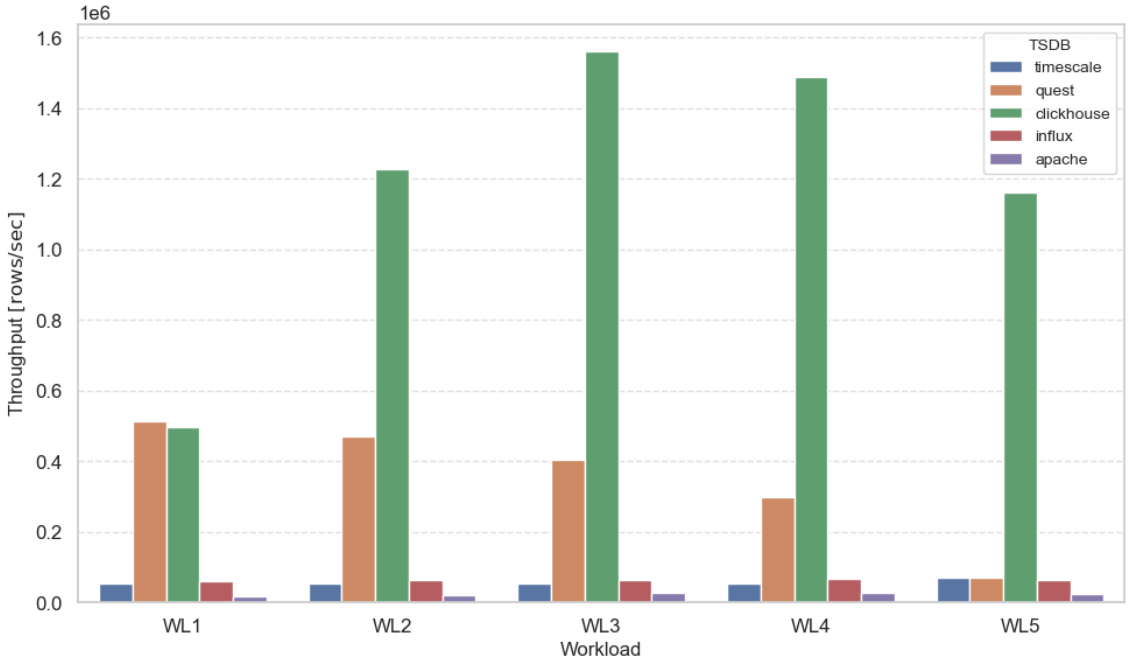


Figure 5.15: Wide-Format Ingestion Throughput Comparison

energy data, suitable for deployments prioritizing data consistency and hierarchical device management over raw ingestion speed.

5.1.6 Cross-Database Ingestion Performance Comparison

Having analyzed each database individually, this section provides comparative analysis across all five TSDBs to identify overall performance leaders, scaling characteristics, and schema design implications for appliance-level energy monitoring.

5.1.6.1 Wide-Format Ingestion Throughput Comparison

Figure 5.15 presents wide-format TP across all TSDBs and WLs. ClickHouse demonstrates overwhelming dominance in wide-format ingestion, achieving 10–20× higher TP than all competitors. Its 1.56M rows/sec peak at WL3 represents 24× faster TP than InfluxDB's 65K rows/sec and 59× faster than IoTDB's 26K rows/sec.

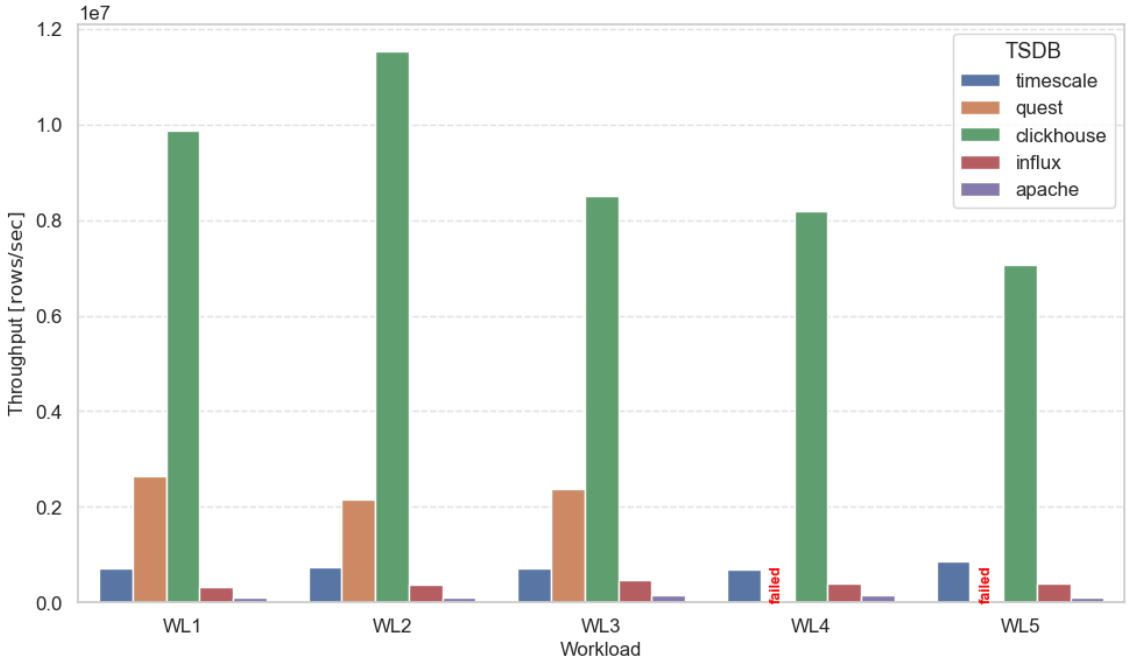


Figure 5.16: Narrow Format Ingestion Throughput Comparison

QuestDB shows competitive performance at small scales (514K rows/sec at WL1) but exhibits severe degradation at WL5 (72K rows/sec), an 86% TP loss. This collapse drops QuestDB below TimescaleDB and InfluxDB at large scales, despite strong initial performance.

TimescaleDB and InfluxDB maintain a stable and comparable TP at 53K–70K rows/sec across all WLs. Apache IoTDB's 17K–26K rows/sec represents the slowest absolute performance, constrained by RPC overhead and per-record durability guarantees.

5.1.6.2 Narrow-Format Ingestion ThroughputComparison

Figure 5.16 presents narrow-format TP, revealing more complex competitive dynamics. ClickHouse maintains clear leadership, though with reduced advantage compared to wide format.

Critical Finding: QuestDB's complete failure at WL4–WL5 represents a hard scalability ceiling. Despite competitive performance through WL3 (2.4M rows/sec), QuestDB could not complete WL4-Narrow (failed at 684M of 1.2B rows) or WL5-Narrow (unable to begin meaningful ingestion). This failure eliminates QuestDB from consideration for billion-row, high-cardinality narrow-format deployments.

ClickHouse's 7.1M–11.5M rows/sec TP represents 8–35× advantage over competitors. However, the margin is narrower than in wide format (10–20×), suggesting narrow format's high cardinality (2,700 series) introduces overhead even for ClickHouse's optimized architecture.

TimescaleDB demonstrates stable performance (681K–871K rows/sec) when HTs are *disabled*. However, enabling HTs (required for TimescaleDB's time-series optimizations) causes catastrophic degradation at WL5. This fundamental architectural limitation effectively disqualifies TimescaleDB for large-scale narrow-format deployments with 1-day chunk configurations.

InfluxDB and IoTDB show modest but functional performance. Both successfully ingested all WLs despite slow TP, demonstrating that billion-row ingestion is achievable, but at the cost of time.

5.1.6.3 WL5 Ingestion Time Comparison

WL5 is used for comparison because it represents the most demanding workload. Figure 5.17 shows the wall-clock ingestion time comparison for the largest workload, WL5 (52.6M rows wide, 1.4B rows narrow), using logarithmic scale to accommodate the 342× range from fastest to slowest.

Wide Format WL5 (52.6M rows): ClickHouse completed the workload in **45 seconds**, making it the clear winner. QuestDB followed at 733 seconds (12.2 min-

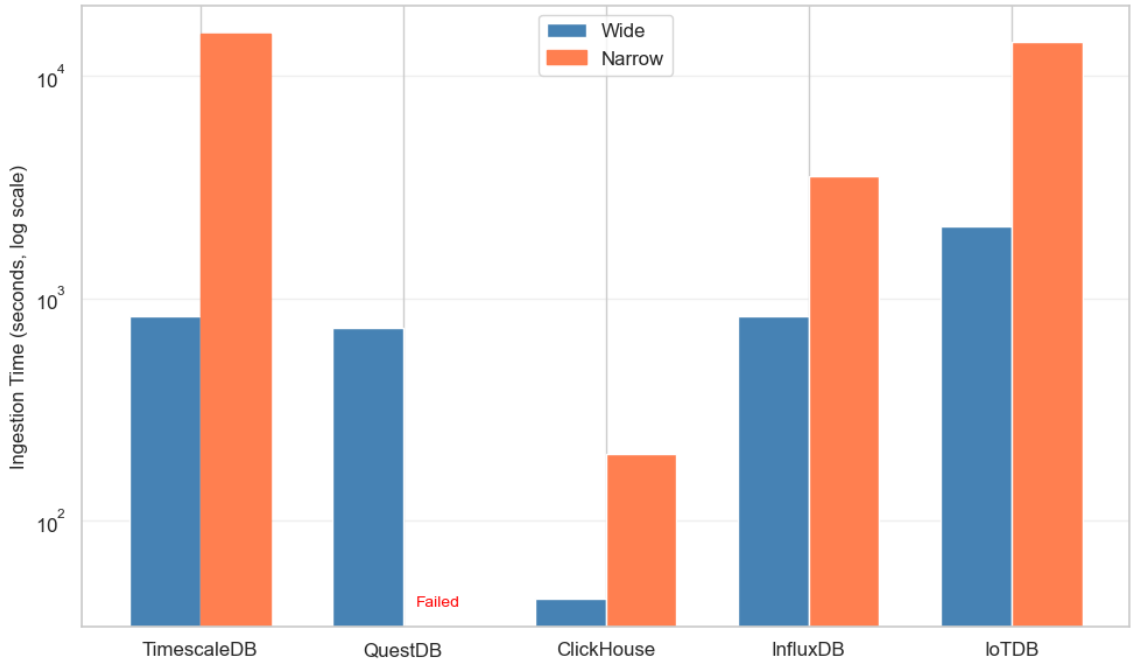


Figure 5.17: WL5 Ingestion Time Comparison (Log Scale)

utes), while InfluxDB and TimescaleDB showed similar performance at 828 seconds (13.8 minutes) and 833 seconds (13.9 minutes) respectively. IoTDB had the slowest performance at 2,102 seconds (35 minutes).

ClickHouse completes wide-format WL5 in 45 seconds—16–47× faster than competitors. The clustering of QuestDB, InfluxDB, and TimescaleDB at 12–14 minutes demonstrates comparable practical performance despite different architectures. IoTDB's 35-minute ingestion reflects its RPC-based per-batch overhead.

Narrow Format WL5 (1.4B rows): ClickHouse achieved the best performance at **201 seconds (3.3 minutes)**. InfluxDB completed the workload in 3,527 seconds (59 minutes), while IoTDB took significantly longer at 14,126 seconds (3.9 hours). TimescaleDB with hypertables showed **catastrophic performance** at 15,408 seconds (4.3 hours), and QuestDB **failed to complete** the workload entirely.

ClickHouse's 3.3-minute completion for 1.4 billion rows represents exceptional engineering. InfluxDB requires 18× longer (59 minutes), while IoTDB and TimescaleDB

take 70–77× longer (approaching 4 hours).

TimescaleDB's 4.3-hour narrow-format ingestion (with HTs enabled) represents complete architectural breakdown, taking 18.5× longer than wide format (833s) for the same database. QuestDB's outright failure further emphasizes the architectural challenges of billion-row, high-cardinality narrow data.

5.1.6.4 Schema Design Impact

The following findings on schema design impact were made from this ingestion study:

- *Wide format is universally faster:* Every TSDB ingests wide-format data faster than narrow, with penalties ranging from 4.3× (InfluxDB, ClickHouse) to 18.5× (TimescaleDB) to complete failure (QuestDB).
- *ClickHouse handles both formats well:* The 4.5× penalty is among the smallest, and absolute performance remains exceptional. ClickHouse is the only TSDB where narrow format remains practical for production billion-row deployments.
- *Cardinality is the primary cost driver:* Narrow format's 27× row increase does not translate to proportional time increase (which would be 27×). Instead, penalties of 4–7× suggest the true cost is cardinality-driven index and metadata management, not raw row processing.

5.1.6.5 Summary Rankings

Table 5.6 summarizes overall ingestion performance rankings.

- **ClickHouse is the clear winner** for ingestion-intensive appliance energy monitoring, offering 10–77× faster performance than competitors.

Rank	Wide Format	Narrow Format	Overall
1	ClickHouse (1.2M rows/s)	ClickHouse (7.1M rows/s)	**ClickHouse** (dominant both)
2	QuestDB (514K → 72K)*	TimescaleDB (871K rows/s, no HT)	TimescaleDB (w/o HT only)
3	TimescaleDB (70K rows/s)	InfluxDB (402K rows/s)	InfluxDB (slow but stable)
4	InfluxDB (64K rows/s)	IoTDB (100K rows/s)	IoTDB (slowest viable)
5	IoTDB (25K rows/s)	QuestDB (FAILED WL4+)	QuestDB (fails at scale)
TimescaleDB: Only without HTs; with HT drops to rank 5 *QuestDB: Fast initially but severe WL5 degradation (86% loss)			

Table 5.6: TSDB Ingestion Performance Rankings

- Wide format is strongly recommended for all TSDBs except ClickHouse.
- QuestDB is unsuitable for billion-row deployments, despite strong small-scale performance.
- For TimescaleDB, narrow format with HTs is effectively unusable (4.3-hour WL5 ingestion). Wide format or disabled HTs are necessary for acceptable performance.

These findings provide clear guidance for TSDB selection based on ingestion requirements, with ClickHouse representing the performance benchmark and other TSDBs

offering trade-offs between speed, features, and architectural constraints.

5.2 Storage Performance

This section evaluates the storage efficiency and compression characteristics of the five TSDBs across all WLs. Storage performance is critical for long-term appliance-level energy monitoring deployments, where years of minute-resolution data can accumulate rapidly. Three databases (TimescaleDB, ClickHouse and Apache IoTDB) provide explicit compression mechanisms, while two (QuestDB and InfluxDB) utilize automatic compression integrated into their storage engines.

5.2.1 TimescaleDB

Table 5.7 presents TimescaleDB's storage and compression characteristics across all WLs. File Size is the original CSV file size before database ingestion. Table Size is the PostgreSQL table storage size after ingestion. Table Size Post Compression is the storage size after running TimescaleDB's native compression with `SEGMENT BY house_id, ORDER BY time DESC`. Compression Ratio (CR) is calculated as uncompressed size ÷ compressed size, indicating compression effectiveness.

Wide-Format Storage Observations: TimescaleDB exhibits minimal compression and consistent 1.15-1.20× CRs across all WLs. WL5-Wide requires 14,833 MB (14.5 GB) before compression, reducing to 12,335 MB (12.0 GB) after compression—a modest 17% size reduction. This limited compression effectiveness reflects the architectural challenge of compressing wide-format, sparse appliance data. With 30 columns per row, TimescaleDB's compression algorithm encounters difficulty achieving substantial gains.

WL	Data Format	File Size (GB)	Table Size (MB)	Size Post Compression (MB)	CR
WL1	Wide	0.03	38.76	33.82	1.15
	Narrow	0.14	269.88	42.39	6.37
WL2	Wide	0.26	405.22	337.05	1.20
	Narrow	1.40	2,603.48	411.30	6.33
WL3	Wide	0.80	1,259.86	1,047.57	1.20
	Narrow	4.24	8,366.21	1,312.56	6.37
WL4	Wide	8.05	12,561.32	10,446.16	1.20
	Narrow	43.59	80,705.82	13,093.27	6.16
WL5	Wide	9.39	14,832.88	12,334.88	1.20
	Narrow	51.26	98,502.35	15,467.99	6.37

Table 5.7: TimescaleDB Table Storage and Compression Metrics Across WLs

Compressed wide-format storage is larger than the original CSV (9.4 GB → 12.3 GB), representing a 1.31× expansion. This overhead stems from PostgreSQL's storage model, which includes metadata and B-tree index structures for the `(house_id, time)` composite index created for query optimization.

Narrow-Format Compression Strengths: Narrow-format demonstrates exceptional compression, achieving consistent CRs of 6.16-6.37× across all WLs (Figure 5.18), showing an average reduction of 84%. The consistency of storage reduction demonstrates that TimescaleDB's compression algorithm scales linearly. This predictability enables accurate capacity planning for production deployments. The dramatic compression stems from narrow format's columnar-friendly structure. TimescaleDB's configuration groups all measurements for a given house together and orders them chronologically, enabling the compression algorithm to exploit both temporal patterns and value similarities within each house's time series.

For example, a refrigerator's power consumption exhibits stable patterns with periodic cycling between on/off states. When 1,440 consecutive minute-level mea-

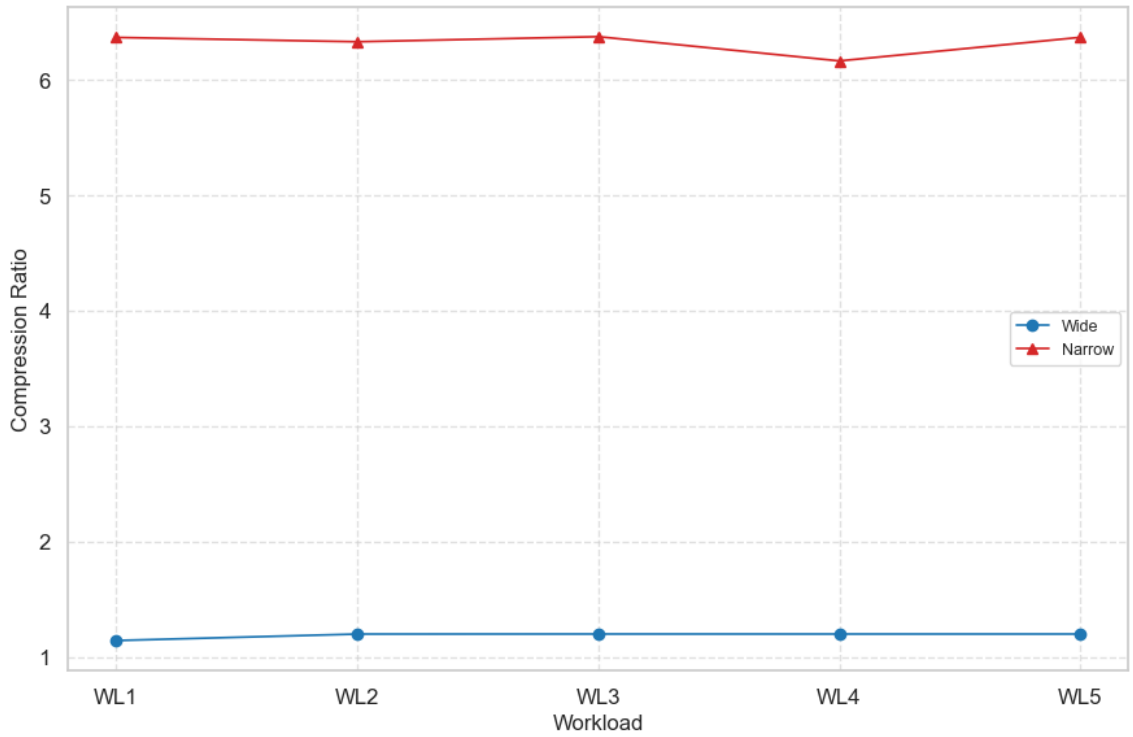


Figure 5.18: TimescaleDB - Compression Ratio Comparison

surements for a single house's refrigerator are stored sequentially, the compression algorithm can efficiently encode the repetitive patterns using back-references rather than storing each value explicitly.

Wide is better Despite Poor Compression: Despite narrow format's superior 6× compression ratio versus wide format's minimal 1.2× ratio, wide format remains more storage-efficient in absolute terms. WL5-Wide compressed to 12.3 GB for 52.6M rows, while WL5-Narrow compressed to 15.1 GB for 1.42B rows—27× more rows. Although narrow format stores 27× more rows, it requires only 23% more space. For storage-constrained deployments, this finding strongly favors wide format, which delivers faster ingestion and an 18% smaller storage footprint than narrow format.

CSV vs. TimescaleDB Storage: TimescaleDB storage for wide and narrow format tables is larger than the original CSV files. However, compressed TimescaleDB

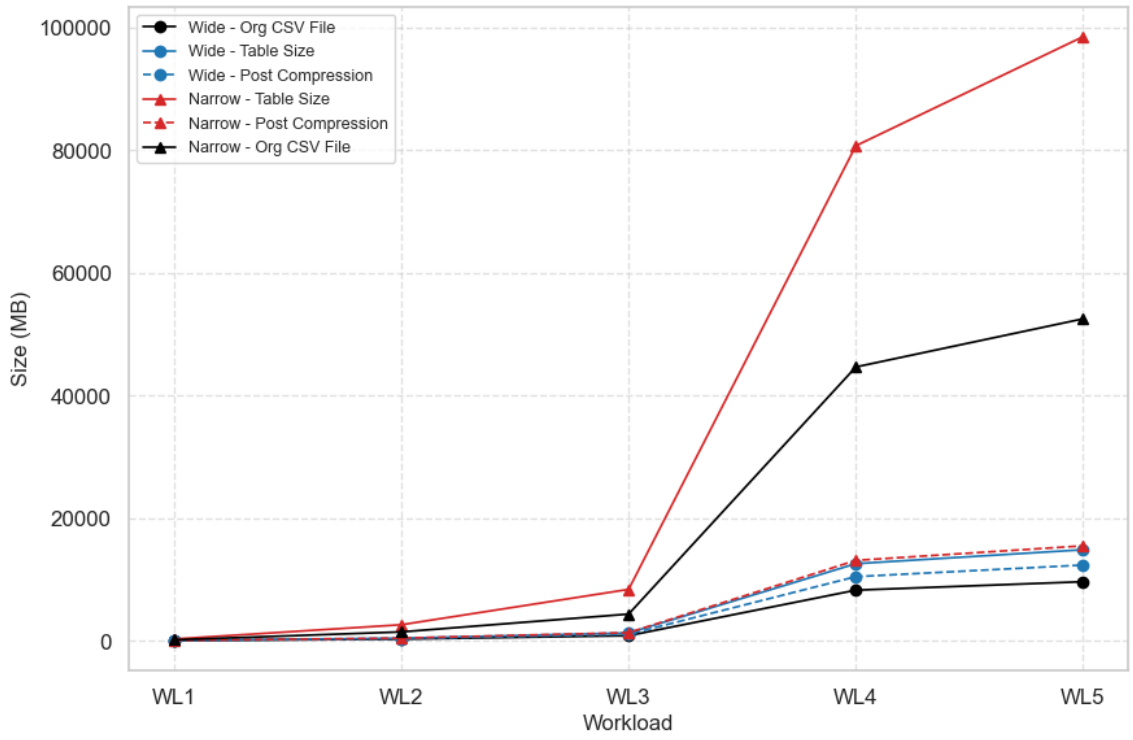


Figure 5.19: TimescaleDB–Storage Performance Comparison

storage is smaller than the original CSV files for narrow-format because of the aggressive compression done by the TSDB, showing a compression of 71% (Figure 5.19), while providing query capabilities, indexing, and ACID guarantees. The wide-format compressed table is bigger than the CSV even after compression and shows an overhead of 28%, likely due to PostgreSQL's indexing and row-oriented storage structures.

5.2.2 QuestDB

Table 5.8 presents QuestDB's storage characteristics across all the WLs.

QuestDB employs automatic compression integrated into its columnar storage engine, providing no manual compression controls. The storage sizes reported represent default compression behavior using symbol indexing for categorical data and colum-

WL	Data Format	CSV File Size (GB)	Table Size (MB)
WL1	Wide	0.03	34.72
	Narrow	0.14	140.28
WL2	Wide	0.26	345.82
	Narrow	1.40	1,402.18
WL3	Wide	0.80	1,071.74
	Narrow	4.24	4,346.63
WL4	Wide	8.05	10,716.52
	Narrow	43.59	N/A
WL5	Wide	9.39	13,140.93
	Narrow	51.26	N/A

Table 5.8: QuestDB Storage Metrics Across Workloads

nar compression for numeric values.

Wide-Format Storage: Wide-format storage demonstrates efficient compression relative to the original CSV file, as shown in Figure 5.20. QuestDB maintains consistent 33-40% storage overhead across all scales, indicating that the columnar storage format, symbol tables for `house_type`, and indexing structures add approximately one-third overhead compared to raw CSV files. This overhead is reasonable given that QuestDB provides indexed queries, columnar compression, and memory-mapped I/O optimizations.

Narrow-Format Storage: QuestDB achieves remarkable storage efficiency for narrow format at small-to-medium scales, storing data at nearly 1:1 ratio with CSV files. This exceptional efficiency stems from QuestDB's columnar architecture, which stores the homogeneous `value` column (all FLOAT) with aggressive compression while using symbol indexing to avoid repetitive storage. However, this efficiency could not extend to WL4-WL5 scales. QuestDB's narrow-format ingestion failed at WL4 (partial completion at 684M rows) and WL5 (complete failure) due to memory

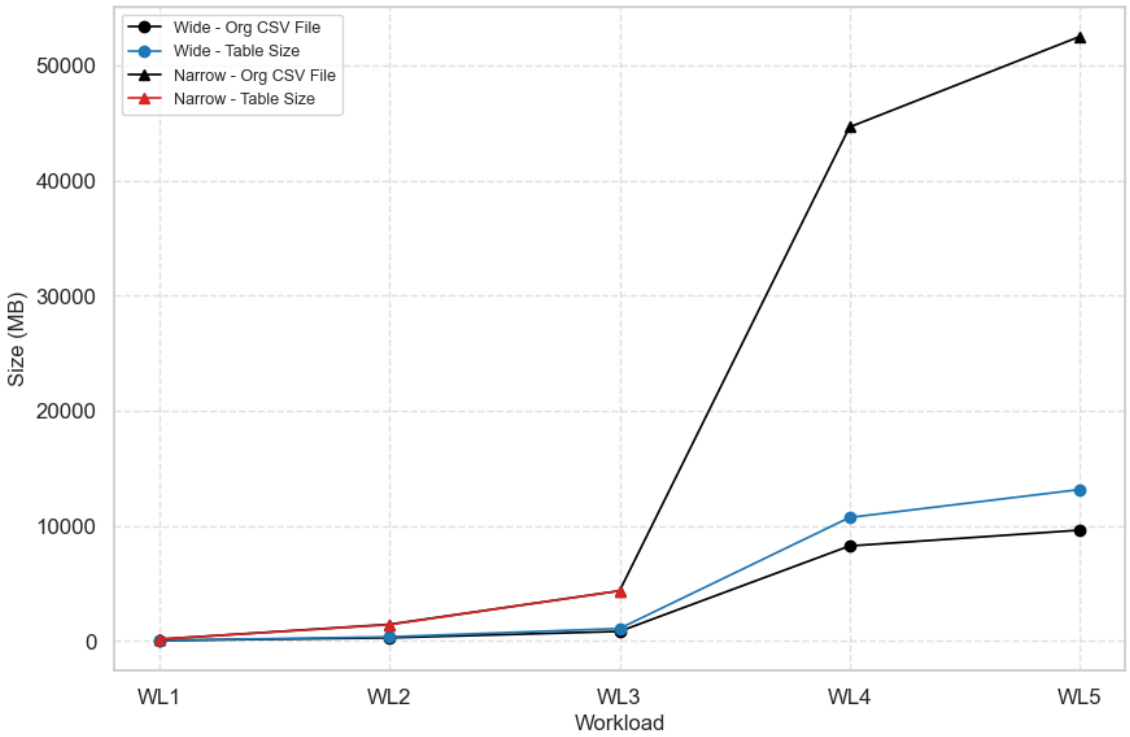


Figure 5.20: QuestDB - Storage Performance Comparison

exhaustion and symbol table saturation. The storage system's inability to handle this represents a fundamental architectural limitation rather than a configuration issue.

Storage Efficiency Comparison: At WL3 (the largest successful narrow workload), QuestDB demonstrates superior storage efficiency compared to TimescaleDB. Despite appearing more efficient relative to CSV, QuestDB's absolute storage is 3.3× larger than TimescaleDB's compressed narrow format. Revealing that QuestDB's compression is less aggressive than TimescaleDB's explicit compression algorithms.

5.2.3 ClickHouse

Table 5.9 presents ClickHouse's storage and compression characteristics across all workloads.

WL	Data Format	CSV File Size (GB)	Table Size (MB)	Size (MB) Post Compression	CR
WL1	Wide	0.03	16.85	15.52	1.09
	Narrow	0.14	54.44	17.27	3.15
WL2	Wide	0.26	168.49	155.17	1.09
	Narrow	1.40	544.45	170.66	3.19
WL3	Wide	0.80	522.31	497.68	1.05
	Narrow	4.24	1685.78	966.78	1.74
WL4	Wide	8.05	6152.56	5851.92	1.05
	Narrow	43.59	16857.81	9667.80	1.74
WL5	Wide	9.39	6152.56	5851.92	1.05
	Narrow	51.26	19875.04	11381.71	1.75

Table 5.9: ClickHouse Table Compression Metrics Across Workloads

Wide-Format Storage: Wide-format data exhibits minimal compression across all WLs. The result stems from ClickHouse's already-efficient columnar storage in the MergeTree engine, making explicit compression provide minimal benefit.

Compared to CSV files, ClickHouse achieves moderate compression. WL5-Wide goes from 9,390 MB CSV to 5,852 MB database (0.62× compression and a 38% reduction), as shown in Figure 5.21. This shows that ClickHouse's columnar storage, even without aggressive compression, stores data more efficiently than CSV's row-oriented text format.

Narrow-Format Storage: Narrow-format compression exhibits interesting scale-dependent behavior, exhibited in Figure 5.22. WL1-WL2 show a 3.15-3.19× compression. Whereas, WL3-WL5 show only 1.74-1.75× compression (degraded by 45%). This indicates that ClickHouse's compression effectiveness decreases as narrow-format data scales.

The cause likely relates to index granularity changes implemented at WL3. Index granularity increased from 8,192 to 16,384 at WL3 to reduce index maintenance

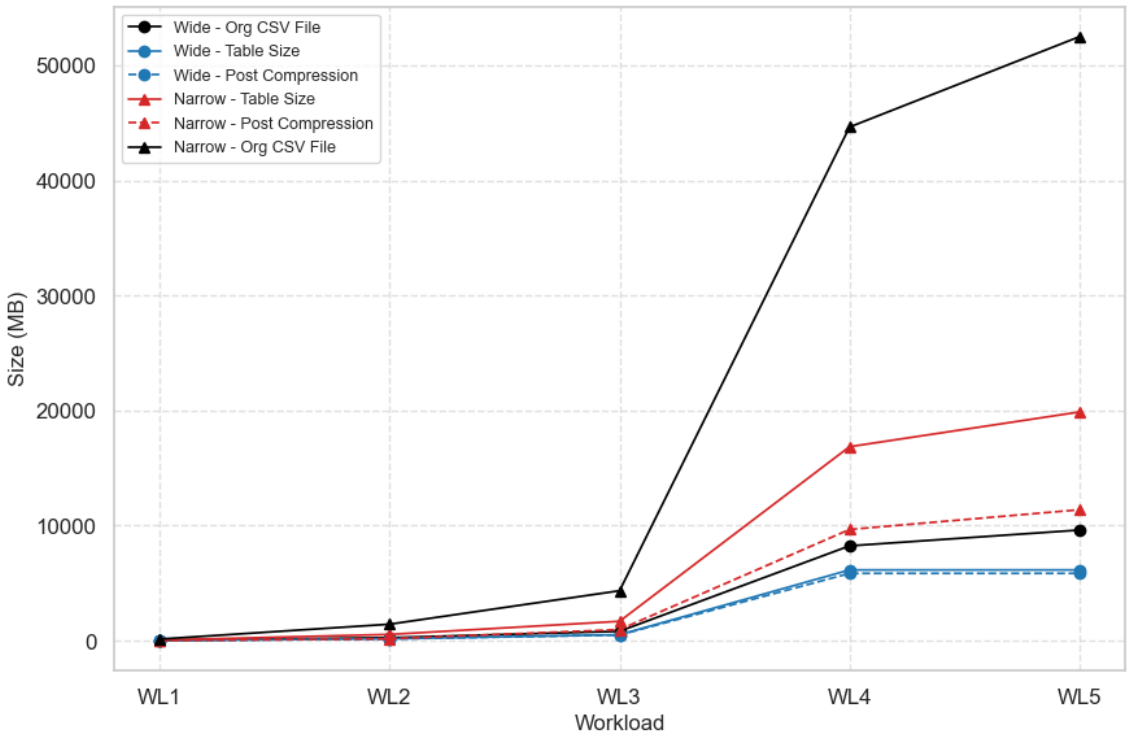


Figure 5.21: ClickHouse–Storage Performance Comparison

overhead. Coarser granularity means larger data blocks between index markers, potentially reducing compression efficiency as the compression algorithm must work with larger, more heterogeneous data segments. Despite this degradation, narrow format still achieves meaningful compression.

5.2.4 InfluxDB

Table 5.10 presents InfluxDB v1.8's storage characteristics across all workloads. Note: InfluxDB v1.8's storage engine includes automatic compression using Snappy and other algorithms, with no manual compression controls available.

Wide-Format Storage: Wide-format storage demonstrates moderate efficiency. InfluxDB's TSM storage maintains 20-62% overhead compared to CSV files, with variability across scales. Interestingly, overhead decreases at larger scales (WL4-

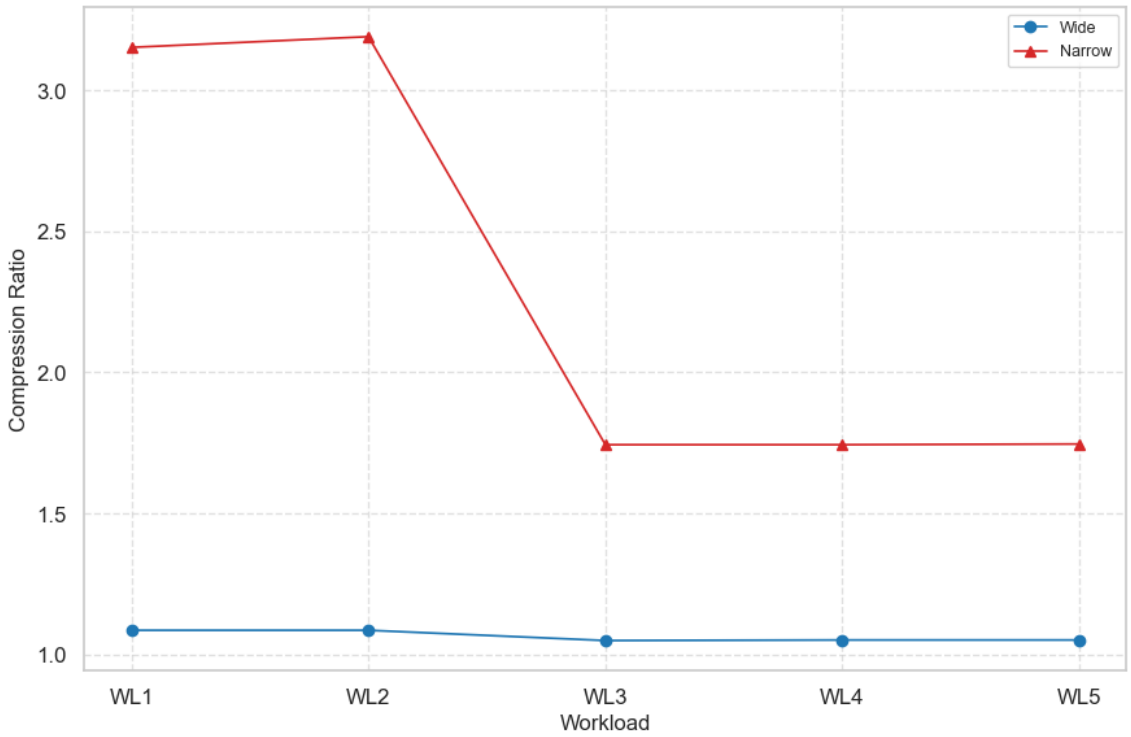


Figure 5.22: ClickHouse - Compression Ratio Comparison

WL5: 1.35× and 1.20×), suggesting that InfluxDB's compression becomes more effective as shard sizes grow. TSM's block-based compression (4KB-64KB blocks) likely achieves better compression ratios when shards contain more data, enabling the Snappy algorithm to exploit larger-scale redundancies.

Narrow-Format Storage: Narrow-format storage exhibits unexpected behavior at WL3 where storage inexplicably expands 3.26× compared to CSV. The WL3 anomaly (13.8 GB for 120M rows) is also inconsistent with WL4-WL5, suggesting that WL3 data was not fully compacted, or suffered from TSM shard fragmentation. Excluding the WL3 anomaly, WL4-WL5 narrow format achieves impressive compression, from a 51.3 GB CSV to 15.8 GB database (69% reduction).

WL	Data Format	CSV File Size (GB) (Before Ingestion)	Table Size (MB)
WL1	Wide	0.03	25.91
	Narrow	0.14	174.76
WL2	Wide	0.26	331.28
	Narrow	1.40	1290.10
WL3	Wide	0.80	1299.85
	Narrow	4.24	13829.00
WL4	Wide	8.05	10877.62
	Narrow	43.59	14248.00
WL5	Wide	9.39	11223.00
	Narrow	51.26	15780.00

Table 5.10: InfluxDB Table Size Comparison Across Workloads

5.2.5 Apache IoTDB

Table 5.11 presents Apache IoTDB's storage characteristics across all WLs. Apache IoTDB's TsFile columnar format includes built-in compression (GORILLA for floats, LZ4 for general compression) applied automatically during writes, with no manual compression controls.

Apache IoTDB emerges as the clear winner in storage efficiency, achieving the smallest footprint among all evaluated databases. Remarkably, both formats converge at nearly identical storage sizes across most workloads, as shown in Figure 5.24. This convergence is extraordinary—despite narrow format containing 27× more rows than wide format (1.42B vs 52.6M at WL5), both formats occupy virtually identical storage (5.6 GB). This indicates that IoTDB's TsFile columnar compression is so effective that the row count becomes nearly irrelevant—what matters is the total information content, which is equivalent between formats.

Storage Trade-Off Analysis: IoTDB's storage efficiency demonstrates a classic

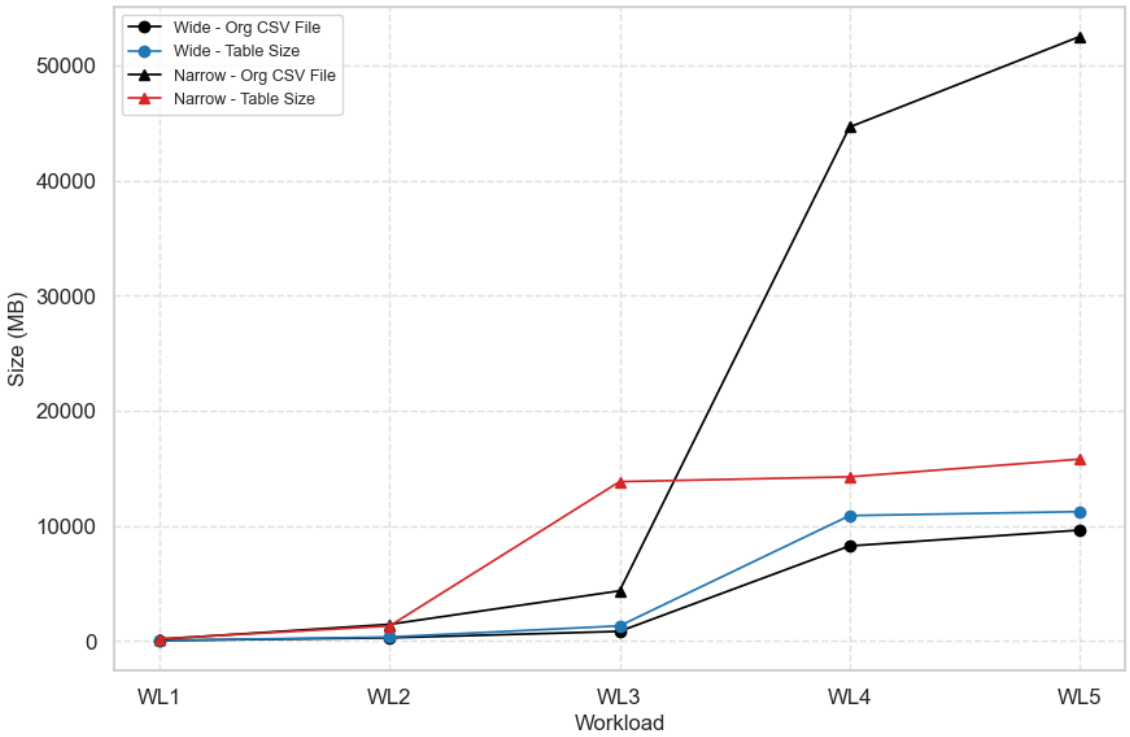


Figure 5.23: InfluxDB - Storage Performance Comparison

performance trade-off. Having the best storage (5.6 GB for WL5) but also having the slowest ingestion (3.9 hours for WL5-Narrow). This implies that IoTDB optimizes for long-term storage density at the expense of write TP. For deployments where storage costs dominate operational expenses and batch ingestion windows are acceptable, IoTDB offers compelling value.

5.2.6 Cross-Database Storage Comparison

Table 5.12 summarizes storage efficiency across all databases at WL5.

5.2.6.1 Compression Effectiveness Comparison

TimescaleDB's 6.37× narrow-format compression vastly exceeds ClickHouse's 1.75×, yet ClickHouse's absolute storage (11.4 GB) remains smaller than TimescaleDB's

WL	Data Format	CSV File Size (GB) (Before Ingestion)	Table Size (MB)
WL1	Wide	0.03	16.00
	Narrow	0.14	16.00
WL2	Wide	0.26	159.00
	Narrow	1.40	148.00
WL3	Wide	0.80	485.00
	Narrow	4.24	481.00
WL4	Wide	8.05	4800.00
	Narrow	43.59	4700.00
WL5	Wide	9.39	5600.00
	Narrow	51.26	5600.00

Table 5.11: Apache IoTDB Table Size Comparison Across Workloads

(15.1 GB). This reveals that ClickHouse's pre-compression columnar storage is inherently more efficient—storing data in 19.9 GB before compression compared to TimescaleDB's 98.5 GB uncompressed starting point.

ClickHouse's efficient default storage reduces the marginal benefit of additional compression, while TimescaleDB's row-oriented PostgreSQL foundation creates significant initial bloat that compression must overcome.

TSDB	Wide (GB)	Narrow (GB)	Wide/CSV	Narrow/CSV
Apache IoTDB	**5.6**	**5.6**	0.60×	**0.11×**
ClickHouse	5.9	11.4	0.62×	0.22×
InfluxDB	11.2	15.8	1.20×	0.31×
TimescaleDB	12.3	15.1	1.31×	0.29×
QuestDB	13.1	Failed	1.40×	—
CSV sizes: WL5-Wide = 9.4 GB, WL5-Narrow = 51.3 GB				

Table 5.12: WL5 Storage Efficiency Rankings

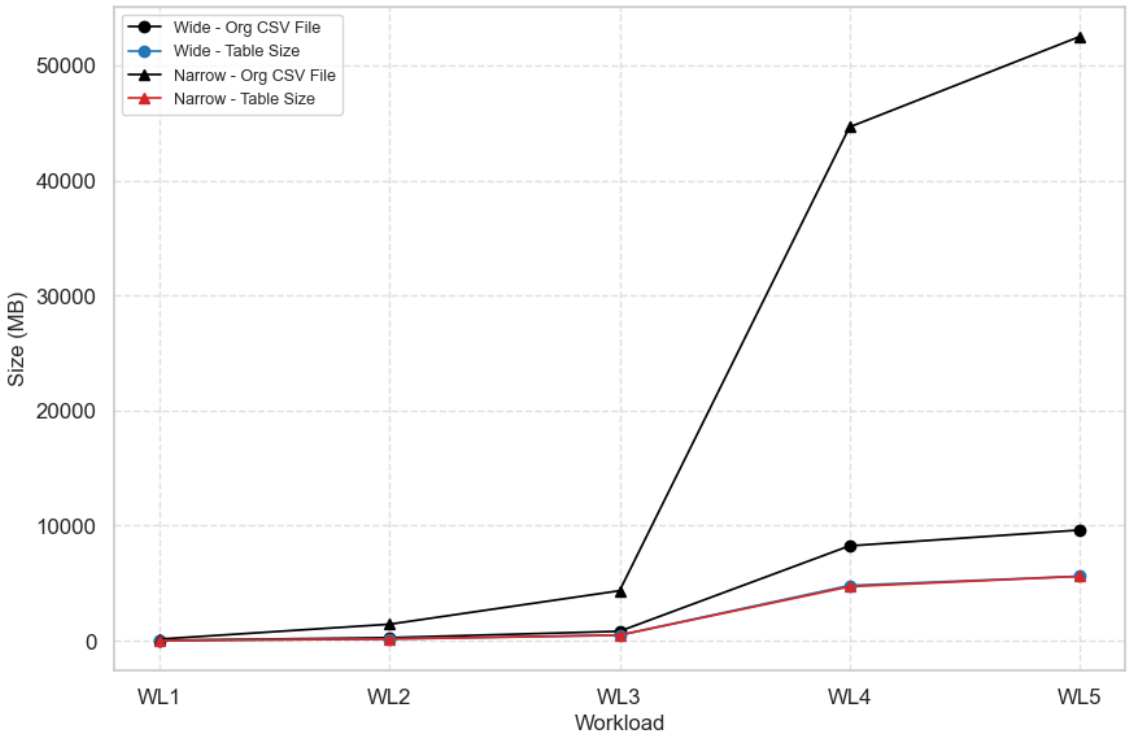


Figure 5.24: Apache IoTDB - Storage Performance Comparison

5.2.6.2 Storage Performance Summary

Apache IoTDB and ClickHouse show the best storage performance. Both databases leverage columnar storage with aggressive compression to achieve 40-89% storage savings compared to CSV files. Apache IoTDB achieves smallest footprint, showing 5.6 GB for both formats, representing 89% storage savings. ClickHouse offers excellent efficiency with speed with 10-77× faster ingestion than IoTDB, making it the best balance of storage efficiency and ingestion performance.

Wide format is universally more efficient. All TSDBs store wide format in less space than narrow format, despite narrow format's superior compression ratios in some cases. Wide format's lower starting size before compression enables smaller final storage even with minimal compression gains.

QuestDB's failure eliminates it from consideration. Despite competitive WL1-WL3

efficiency (1.03× CSV size for narrow format), inability to complete WL4-WL5 disqualifies QuestDB for billion-row deployments.

The storage analysis, combined with ingestion performance findings, positions ClickHouse as the overall leader for appliance-level energy monitoring deployments where both storage efficiency and ingestion speed matter.

5.3 Query Performance

This section evaluates query performance across six queries (Q1-Q6) designed to test common appliance-level energy monitoring analytical patterns. Query performance is assessed across TimescaleDB, QuestDB, and ClickHouse using wide and narrow formats for all WLs.

For all successfully executed queries across TimescaleDB, QuestDB, and ClickHouse, result correctness was verified by comparing row counts returned by each database for identical queries. All three databases returned identical row counts for every query execution.

Two databases are excluded from query performance benchmarking due to fundamental limitations–InfluxDB and Apache IoTDB.

InfluxDB v1.8 was excluded entirely from query benchmarking due to InfluxQL language limitations that prevented execution of the standardized benchmark query suite. Key incompatibilities included:

- Timestamp format incompatibility: Data ingested using epoch seconds (e.g., `1704132000`) could not be queried using standard ISO 8601 time strings.
- No subquery support: InfluxQL does not support SQL subqueries, preventing queries like Q3 (compute average power across all houses) and Q6 (pre-

aggregate then filter).

- InfluxQL cannot filter on aggregated results, requiring workarounds that alter query semantics and performance characteristics.

- In InfluxQL, SELECT * behaves differently from SQL because it returns only fields plus time, never tags, which must be explicitly included. Queries also depend on time filtering. So omitting a 'WHERE' time clause can lead to empty or inconsistent results. Unlike SQL, InfluxDB does not allow SELECT * with GROUP BY unless aggregates are used, and wildcard rules require tags to appear before *. It also enforces strict field type consistency, meaning SELECT * can fail if fields were written with mixed types. Overall, full-measurement scans are slower in InfluxDB because it's optimized for time-bounded, field-specific queries rather than SQL-style table-wide selection.

Given these fundamental incompatibilities with the SQL-based benchmark suite, InfluxDB results are excluded from query performance comparisons. Detailed documentation of attempted queries is provided in Appendix A.

Apache IoTDB exhibited query execution failures across both data formats and all WLs, rendering systematic performance comparison infeasible. Figure 5.25 illustrates the extent of failures.

Only 34 of 60 possible query executions (10 WLs $\times$ 6 queries) completed successfully—a 43% failure rate. Failure patterns include:

- Q1 (SELECT *) memory exhaustion: Point queries retrieving all columns for a time range caused out-of-memory failures for both formats. IoTDB's query execution appears to buffer entire result sets in memory before streaming to clients, causing failures when result sets exceed the available RAM.

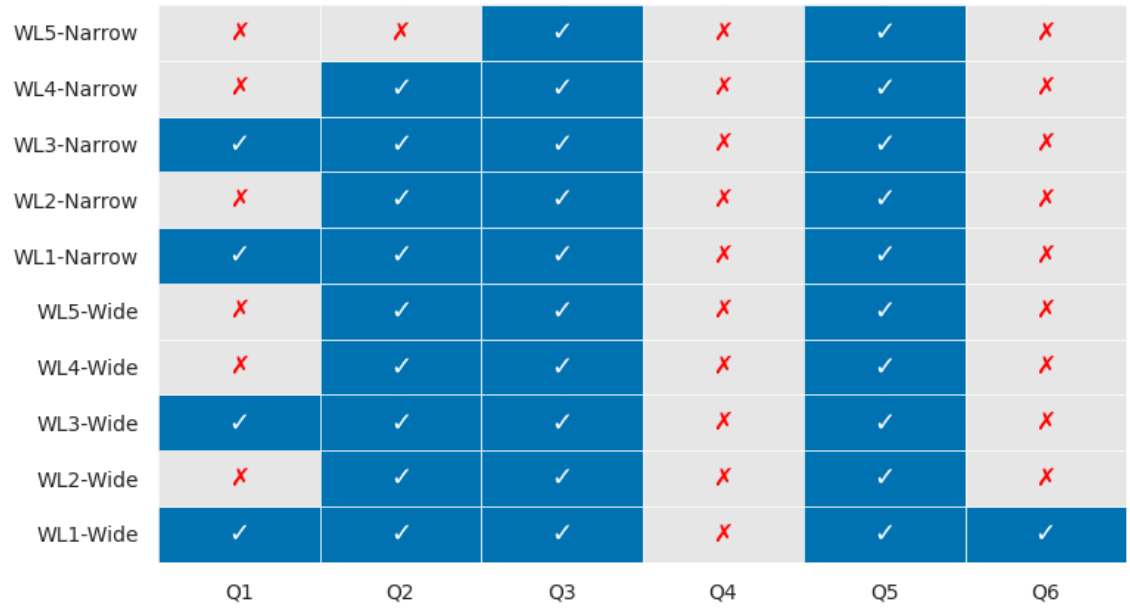


Figure 5.25: Apache IoTDB Query Success/Fail Matrix (Q1–Q6)

- Q4 (GROUP BY) failures: Per-house aggregation queries failed across all WLs in both formats. Error messages suggested query parser or planner bugs rather than resource exhaustion, as even tiny workloads failed consistently. This represents a fundamental limitation in IoTDB's cross-device aggregation engine.

- Q6 (complex multi-dimensional aggregation) failure: Hourly aggregation with house-level grouping failed on 9 of 10 WLs (90% failure rate), succeeding only on WL1-Wide, indicating IoTDB's aggregation engine cannot handle fine-grained temporal grouping combined with dimensional grouping at scale.

5.3.1 Query Execution Success Rates

Figure 5.26 presents query execution success rates for Q1-Q5 across all.

TimescaleDB exhibits a 100% success rate for Q1-Q5 across all WLs, demonstrating robust PostgreSQL query executor scalability.

QuestDB has an 80% success rate (40/50 executions), failing only on WL4-WL5

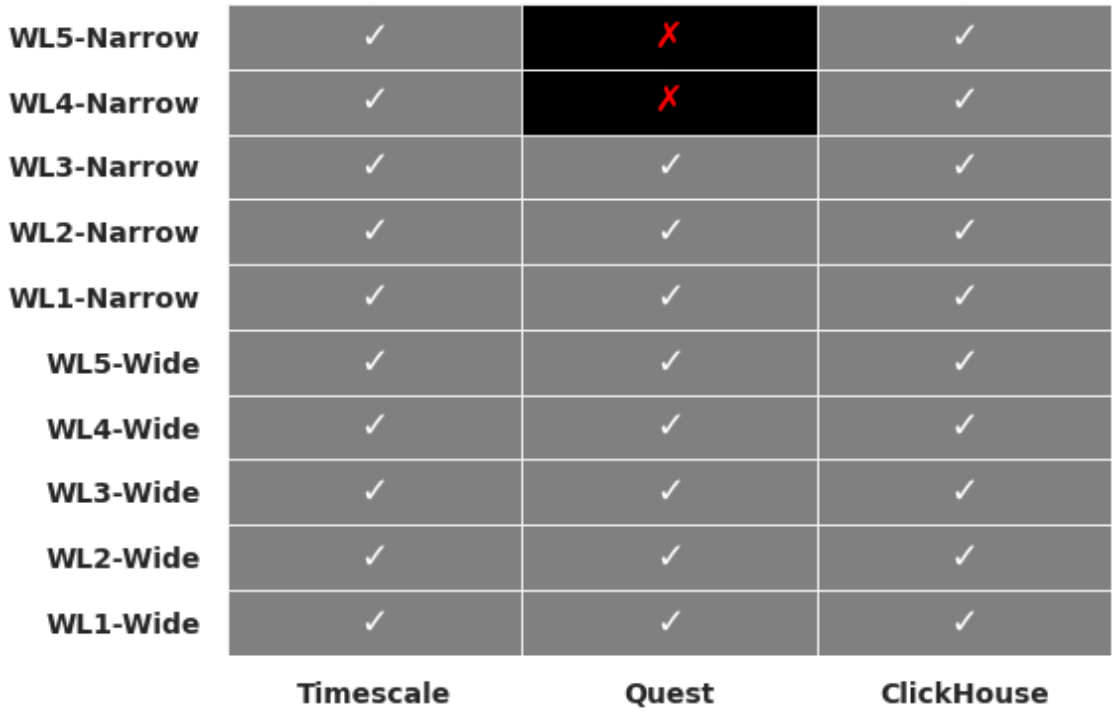


Figure 5.26: Query Success/Fail Matrix (Q1–Q5)

narrow format due to ingestion failures documented earlier. All ingested WLs by Quest were executed successfully.

ClickHouse has a 100% success rate (50/50 executions), matching TimescaleDB for Q1-Q5.

Q1-Q5 exhibit identical success patterns across all configurations, hence a single success matrix represents all five queries. Q6's distinct failure pattern is shown separately in Figure 5.27.

Q6, which is the hourly aggregation with house-level grouping, introduces additional failures. TimescaleDB shows an 80% success rate (8/10) for Q6, failing on WL4-Narrow and WL5-Narrow due to resource exhaustion. Q6's narrow-format rows triggered CPU saturation and memory exhaustion (>12 GB RAM consumption). Q6 completed successfully on WL5-Wide, indicating the failure is specific to high-cardinality narrow format at billion-row scales.

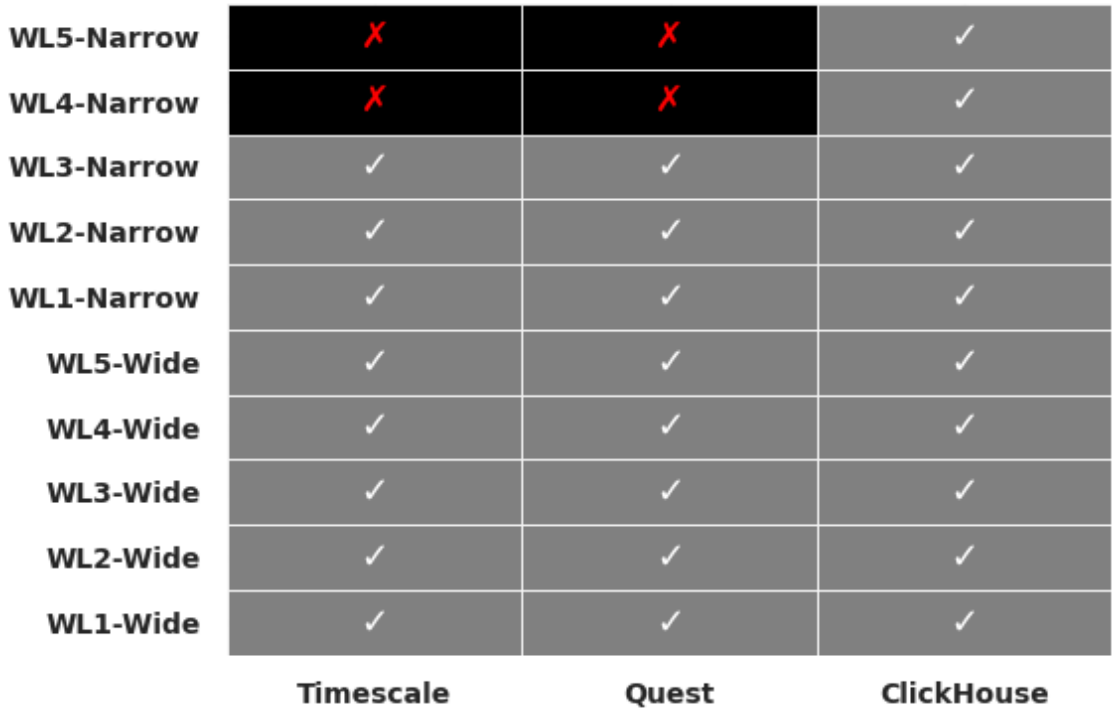


Figure 5.27: Query 6 Success/Fail Matrix

There was no change to QuestDB since it was not able to ingest for WL4-Narrow and WL5-Narrow, therefore also failing for these while successfully executing the rest.

ClickHouse has a 100% success rate for even Q6, demonstrating superior scalability for complex aggregations.

Overall query reliability rankings:

1. ClickHouse: 100% (60/60 total query executions)
2. TimescaleDB: 96.7% (58/60)
3. QuestDB: 80% (48/60)

ClickHouse's perfect reliability across all query types and scales positions it as the most robust choice for production analytical WLs.

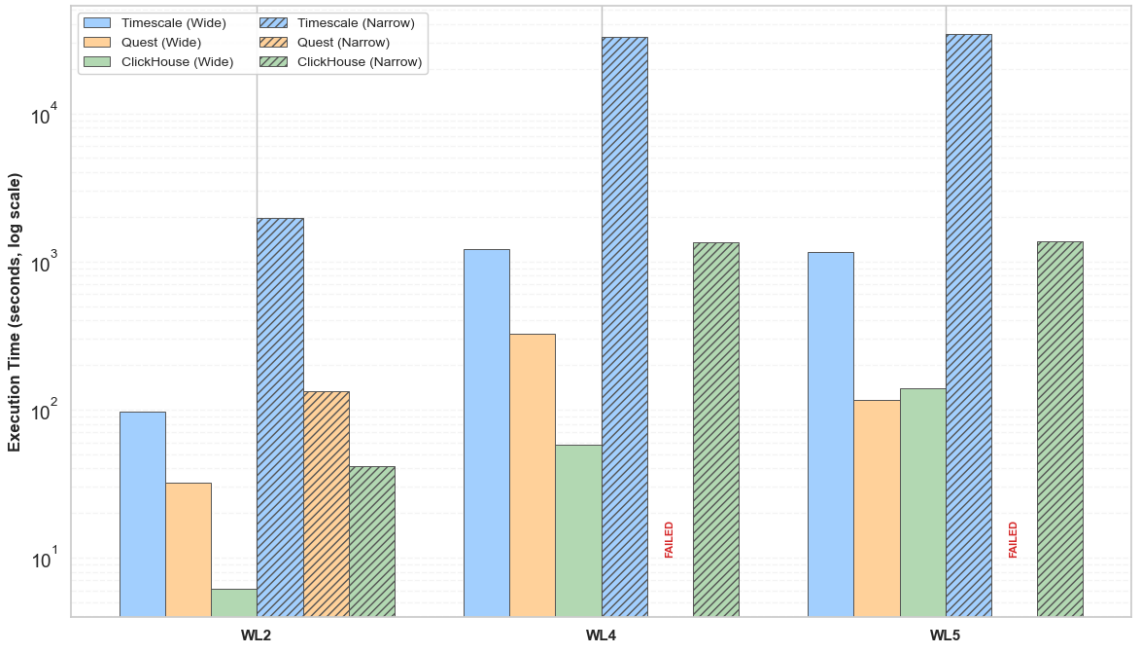


Figure 5.28: Query 1 Performance: Wide (Solid) vs. Narrow (Striped)

5.3.2 Cross-Query Performance Comparison

This section analyzes execution time patterns across all six queries, identifying database-specific strengths and scalability characteristics.

5.3.2.1 Per-Query Analysis

The following figures and analysis evaluate the performance of the three TSDBs across six query types using daily (WL2), monthly (WL4), and annual (WL5) workloads.

Across all six queries, ClickHouse consistently demonstrated superior scalability and efficiency. As illustrated in Figures 5.29 and 5.31, it maintained the lowest execution times for the annual wide workload (WL5-Wide), ranging from 33.45 seconds in Q4 to 241.69 seconds in Q2.

The narrow schema format imposed a significant computational penalty across all

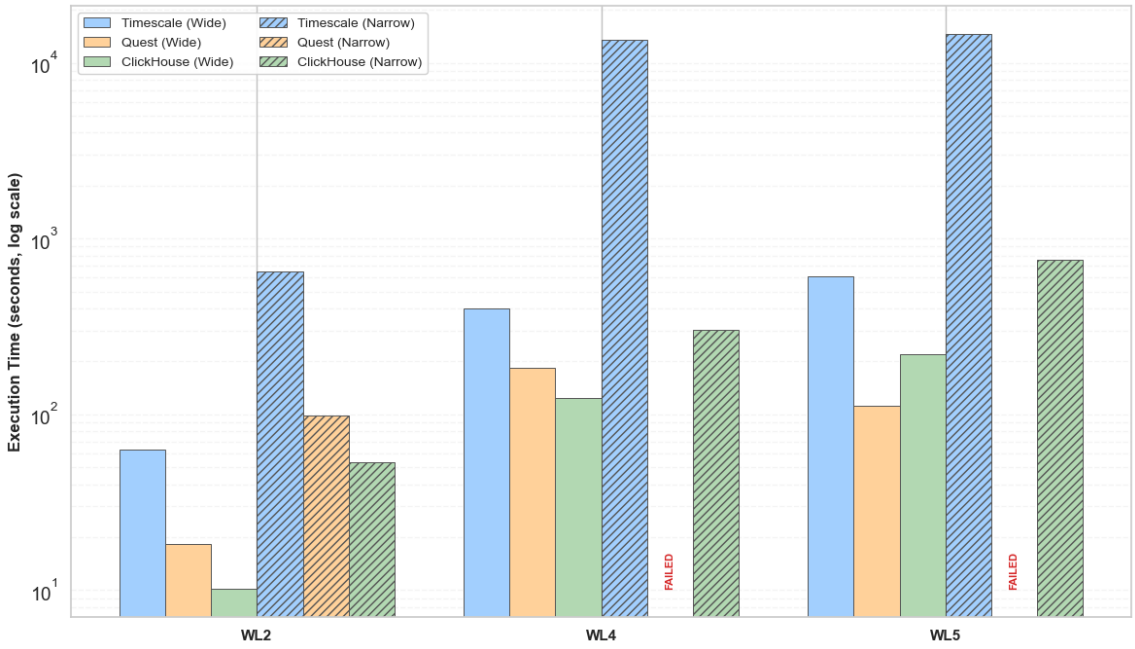


Figure 5.29: Query 2 Performance

engines, primarily driven by a massive expansion in row volume—from 6.57 million rows in WL5-Wide to 177.39 million rows in WL5-Narrow. TimescaleDB exhibited the most extreme sensitivity to this schema shift. In Figure 5.28, its execution time surged 30x from 1,152.7 seconds to 34,698.26 seconds, while Figures 5.31 and 5.32 show narrow schema latencies exceeding 10,000 seconds.

QuestDB, while competitive in wide schema configurations, consistently failed to complete narrow schema queries for monthly (WL4) and annual (WL5) workloads, as marked in Figures 5.28 through 5.33. This suggests a critical processing bottleneck for high-cardinality narrow datasets. Figure 5.33 highlights the limits of wide schema scalability for TimescaleDB and QuestDB as data volumes grew, with QuestDB recording a peak latency of 7,123.45 seconds for WL5-Wide. For Q6 in narrow-format, TimescaleDB fails completely due to memory constraints and is unable to execute the query for larger WLs (WL4-Narrow and WL5-Narrow).

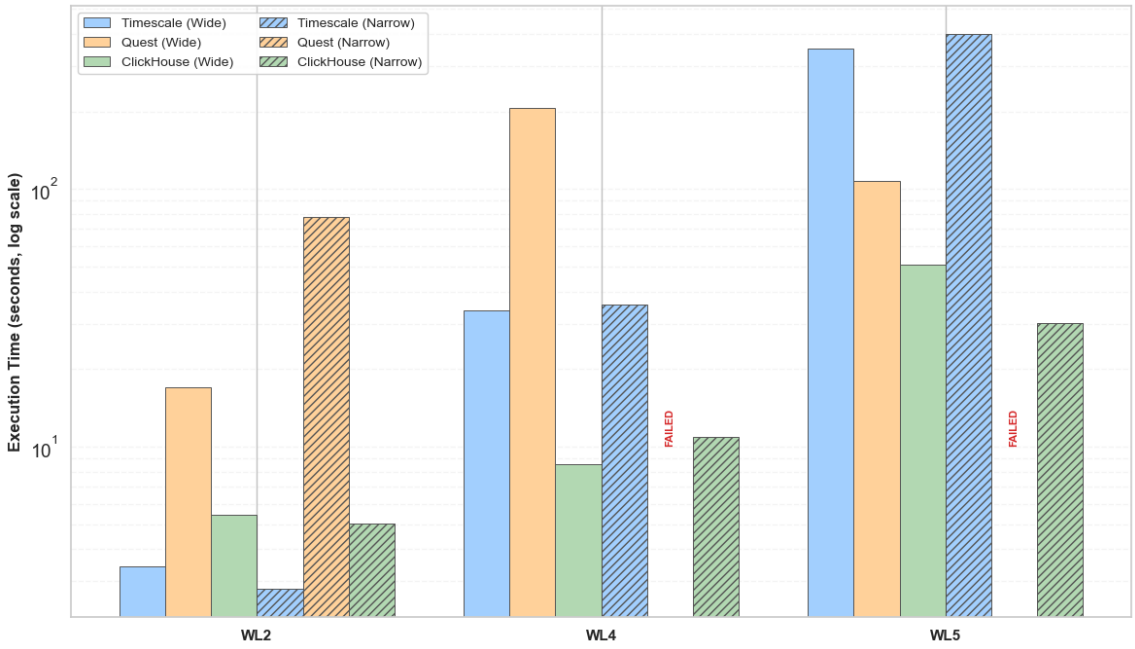


Figure 5.30: Query 3 Performance

5.3.2.2 Per-Query Speedup Analysis

In this study, the Speedup Ratio was utilized as a normalized metric to evaluate how schema design specifically impacts the performance of each individual database across different WLs. Figure 5.34 presents speedup ratios for each query, calculated as:

$$\text{Speedup Ratio} = \frac{\text{Narrow Format Execution Time}}{\text{Wide Format Execution Time}} \tag{5.1}$$

Ratio >1.0 means narrow format is *slower* than wide;

Ratio <1.0 means narrow format is *faster* than wide;

and Ratio = 1.0 means both formats perform equally well.

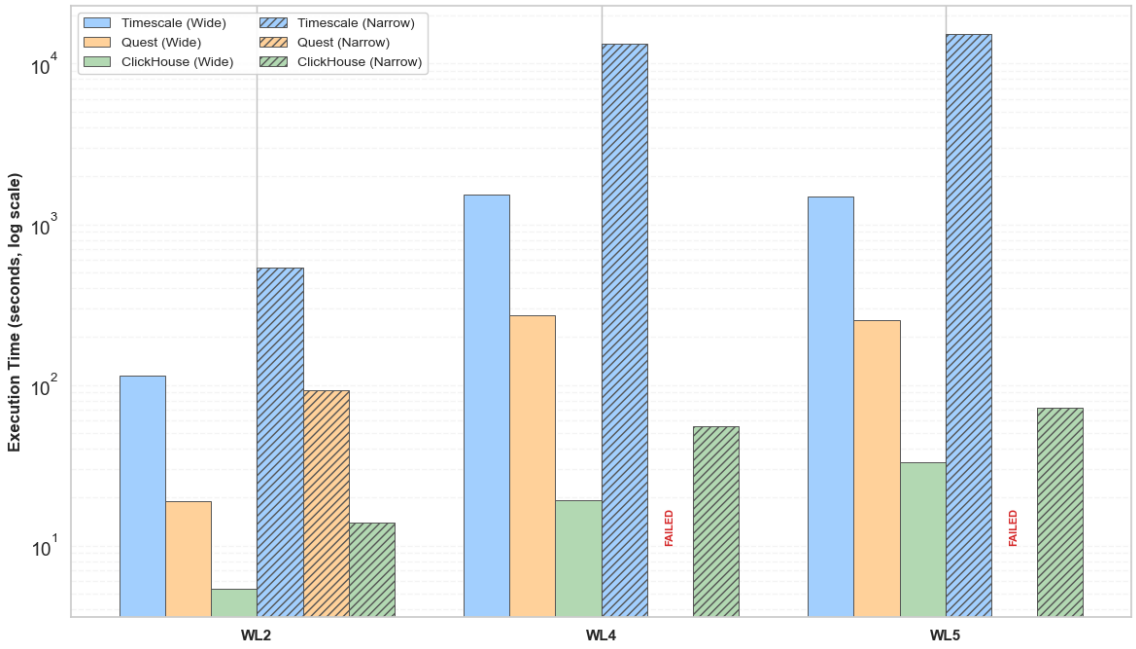


Figure 5.31: Query 4 Performance

5.3.2.3 TimescaleDB Speedup Patterns

TimescaleDB exhibits dramatic, scale-dependent format sensitivity. Q1, shows a slowdown for narrow-format (WL3-WL5), indicating severe index inefficiency for high-cardinality narrow format. Q2 also shows a similar performance. However, it decreases from WL4 to WL5 indicating that the year long day performs with lesser discrepancy or gap than the month long WL. It performs well for Q3. There is a catastrophic degradation for Query 4, from WL3 to WL4. This represents complete architectural breakdown for GROUP BY operations on billion row narrow format.

TimescaleDB's speedup ratios reveal that narrow format becomes progressively less viable as scale increases, with GROUP BY queries (Q4, Q5) suffering 100-700× performance penalties at WL4-WL5. The root cause is the HT chunking–WL5-Narrow creates 98,500 chunks (1-day intervals × 2,700 series), forcing PostgreSQL's planner to evaluate tens of thousands of partition scans for GROUP BY operations.

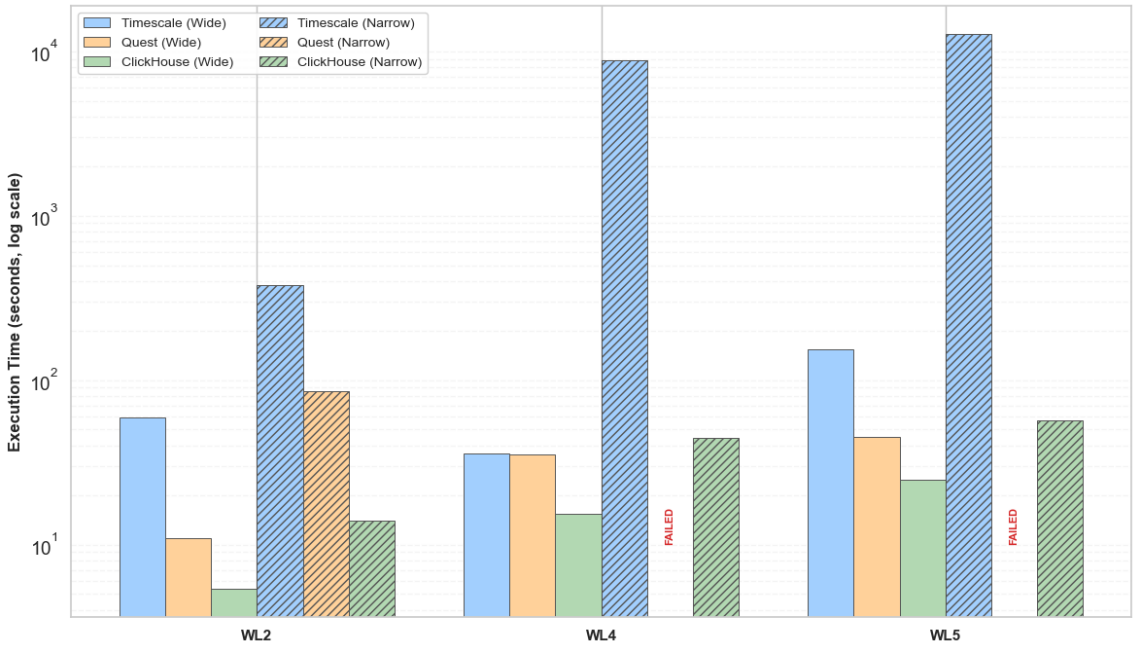


Figure 5.32: Query 5 Performance

5.3.2.4 QuestDB Speedup Patterns

QuestDB maintains relatively stable speedup ratios across successful workloads, indicating predictable format performance characteristics. However, the abrupt failures at WL4-WL5 prevent assessment of large-scale behavior.

5.3.2.5 ClickHouse Speedup Patterns

ClickHouse's speedup ratios cluster near 1.0-10× across most queries, demonstrating that its columnar architecture performs well regardless of schema design. Notably, for Q3, narrow format is actually *faster* than wide, and Q4 and Q5 are similar in performance regardless of schema-a unique result attributable to ClickHouse's per-column storage granularity and predicate pushdown optimizations. ClickHouse dominates across all queries, achieving 1.5-768× faster execution. On Q4 (GROUP BY), ClickHouse completes in 2.5 seconds what takes TimescaleDB 32 minutes.

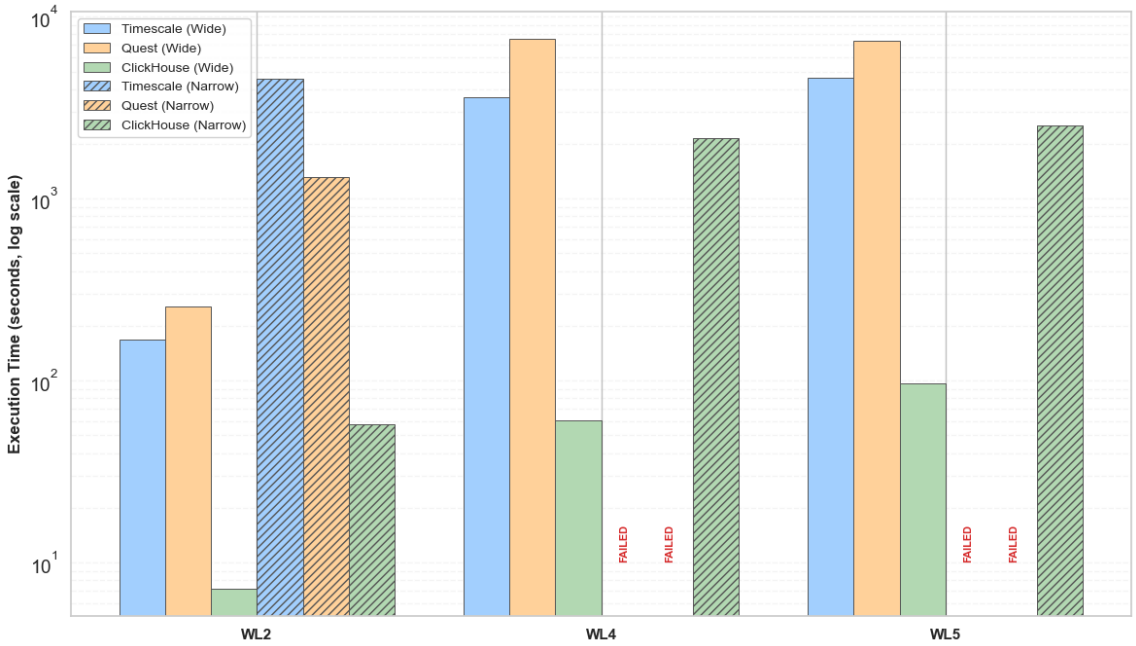


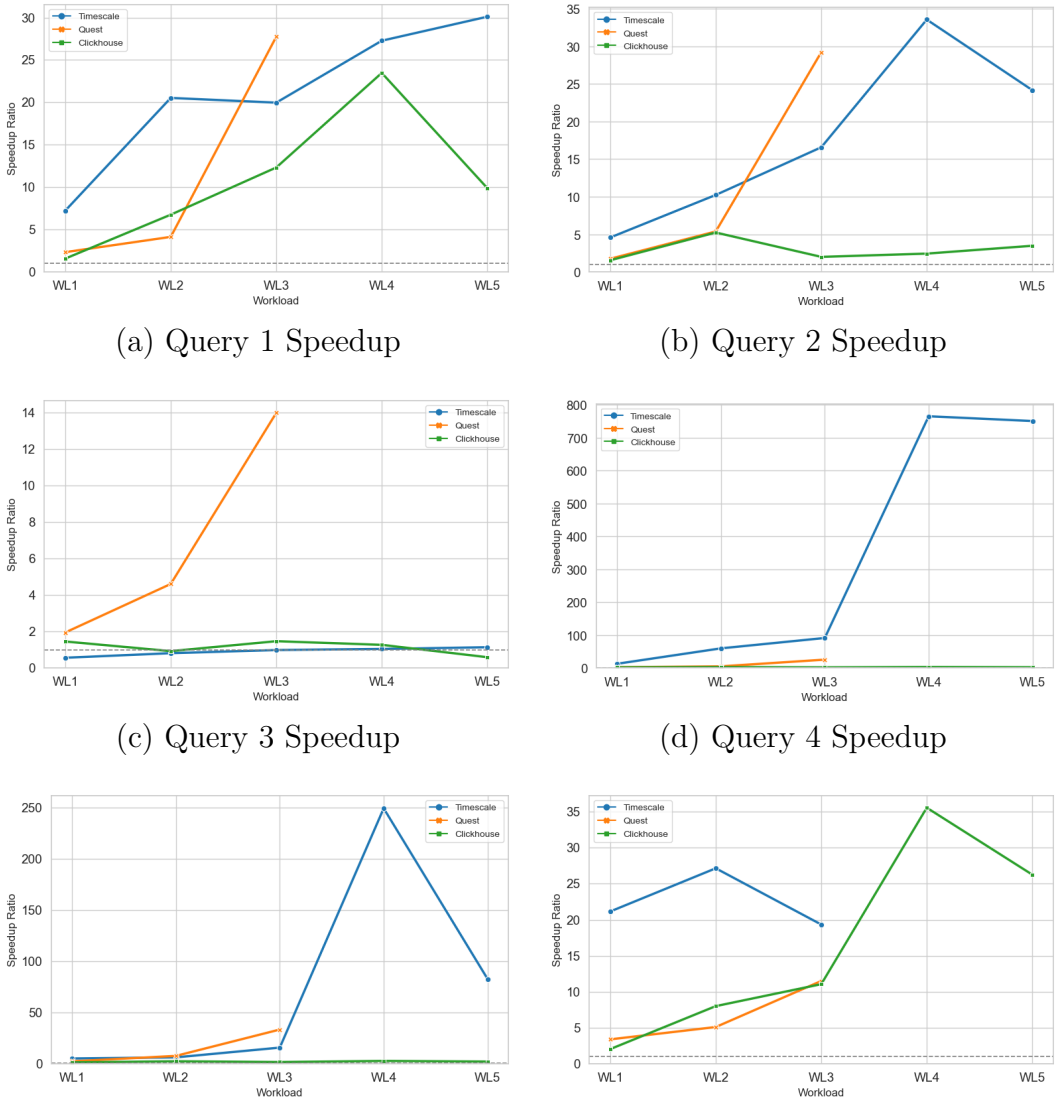
Figure 5.33: Query 6 Performance

5.3.3 Key Findings from Query Performance Analysis

The query performance analysis reveals several key insights:

- *TimescaleDB and QuestDB strongly favor wide format* across nearly all query types (11-768×). Only Q3 shows narrow-format benefit for TimescaleDB.
- *ClickHouse uniquely works well with both wide and narrow format.* This result stems from ClickHouse's columnar storage: narrow format stores each metric as an independent column, enabling aggressive predicate pushdown and column-level compression. Wide format requires scanning 27 appliance columns even when queries target specific metrics, leading to similar performance.

ClickHouse emerges as the clear query performance leader, offering best-in-class execution speed, reliability, and format flexibility. TimescaleDB provides adequate performance for wide-format deployments but suffers narrow-format penalties. QuestDB's narrow-format failures eliminate it from consideration for large-scale deployments.



(a) Query 1 Speedup

(b) Query 2 Speedup

(c) Query 3 Speedup

(d) Query 4 Speedup

(e) Query 5 Speedup

(f) Query 6 Speedup

Figure 5.34: Per-Query Speedup Ratios for Q1–Q6. Ratios above 1.0 indicate that the narrow format is slower than the wide format.

Chapter 6

Conclusion and Future Work

Cross-database ingestion evaluation revealed that ClickHouse dominated performance across all workload scales and schema formats. It successfully completed all WLs without failures, memory exhaustion, or catastrophic performance degradation across the tested range. ClickHouse exhibited the smallest wide-to-narrow performance penalty (4.5×), making it the only system where narrow-format remains practical for billion-row deployments. Other databases exhibited significant limitations, which have been summarized in Chapter 5.

Storage evaluation revealed that Apache IoTDB and ClickHouse achieve superior compression efficiency, though with different trade-offs.

For production-level deployments requiring real-time dashboards, ad-hoc analytics, and complex aggregations, ClickHouse is the only viable option offering sub-second query latency at billion-row scales with perfect reliability. TimescaleDB provides adequate wide-format query performance but becomes unusable for narrow-format billion-row aggregations. InfluxDB v1.8 and Apache IoTDB are unsuitable for analytical workloads requiring SQL-like query flexibility, though InfluxDB v2/v3 with Flux language may address these limitations.

6.1 Conclusion

This thesis presented a benchmarking study of five TSDBs—TimescaleDB, ClickHouse, QuestDB, InfluxDB v1.8, and Apache IoTDB—tailored to appliance-level residential energy monitoring. By extending the TSM-Bench methodology, this research evaluated ingestion throughput, storage efficiency, and query performance across five workload scales spanning from 144,000 to 1.4 billion rows. This dual-format approach addressed a critical gap in existing literature by evaluating both wide-format and narrow-format schemas under identical hardware constraints.

The findings of this study directly address the RQs established in Chapter 4:

RQ1: ClickHouse dominated ingestion performance across all scales, processing up to 11.5 million rows per second for narrow-format and 1.6 million for wide format. In contrast, QuestDB failed catastrophically at the 684-million-row mark for narrow-format data, while TimescaleDB and Apache IoTDB exhibited significant ingestion latency, taking approximately 4 hours to complete the largest narrow-format workload. While Apache IoTDB was slower, it achieved the highest storage efficiency, reducing the 51.3 GB narrow-format CSV to just 5.6 GB—an 89% savings.

RQ2: Every tested TSDB ingested wide-format data faster than narrow-format data, with performance penalties for narrow schemas ranging from 4.3× in ClickHouse to 18.5× in TimescaleDB. Wide-format schemas were universally more storage-efficient. ClickHouse demonstrated the highest format flexibility, with narrow and wide formats converging to similar data-point processing rates at scale.

RQ3: ClickHouse is the only viable option for production-level deployments requiring real-time dashboards and complex aggregations, offering sub-second query latency at billion-row scales with perfect reliability. While TimescaleDB provides adequate performance for wide-format deployments, it becomes unusable for high-cardinality

narrow-format aggregations. Apache IoTDB and InfluxDB v1.8 remain unsuitable for analytical workloads requiring SQL-like query flexibility due to fundamental language and execution engine limitations.

In summary, for smart-home energy monitoring requiring billion-row scalability and query reliability, ClickHouse represents the performance benchmark, while Apache IoTDB offers a robust, storage-optimized alternative where ingestion speed is not a priority.

6.2 Limitations

TSDB configurations followed vendor documentation recommendations and common deployment practices, revealing several limitations:

- **TimescaleDB chunk intervals:** Fixed 1-day chunk intervals across all workloads; weekly or monthly chunks would likely improve narrow-format performance but were not tested due to scope limitations–similar chunking would be needed for other TSDBs as well, which would require additional configuration.
- **InfluxDB batch sizes:** Conservative 10,000-row batches following v1.8 documentation; larger batches (100K–1M) would likely improve throughput but risk write queue saturation per vendor warnings.
- **ClickHouse index granularity:** The index granularity was manually adjusted from 8,192 to 16,384 at WL3 based on ClickHouse's documentation [5]. Finer-grained tuning may optimize performance further.
- **Apache IoTDB batch sizes:** Apache IoTDB used Fixed 500,000-row batches. Testing alternative batch sizes and configurations may reduce overhead.

6.3 Future Work

The benchmarking framework and findings presented in this thesis provide a foundation for several promising avenues of research. This work can be extended in three primary directions to further refine the selection and optimization of time-series databases for energy monitoring.

First, an evaluation of database-specific configuration parameters is necessary to determine the limits of system-level performance. This includes investigating alternative TimescaleDB chunk intervals. Specifically moving from the fixed one-day interval used in this study to weekly or monthly chunks, to determine if such adjustments can mitigate the severe hypertable penalty observed in high-cardinality narrow-format workloads. Similarly, InfluxDB batch size optimization (ranging from 10,000 to 1,000,000 rows) should be explored to assess whether larger batches can eliminate the ingestion throughput gap identified in this research. For Apache IoTDB, future experiments should vary batch sizes and explore alternative RPC configurations to identify if current ingestion bottlenecks are architectural or merely configurational.

Second, the benchmark query suite should be expanded to reflect more advanced analytical use cases. While this study focused on core temporal aggregations and point queries, future work will integrate machine learning (ML) workloads, such as real-time anomaly detection and predictive load forecasting. These additions would test the TSDBs' ability to support complex temporal operations required for automated demand-response systems.

Third, future research must address sparse and irregular data patterns representative of real-world IoT deployments. Residential energy data is often incomplete due to sensor failures or variable reporting frequencies. Evaluating such datasets would

determine whether the performance advantages of wide-format schemas persist when handling large volumes of null values, or if the flexibility of narrow-format schemas (such as those managed by ClickHouse or InfluxDB) provides a superior trade-off in heterogeneous environments.

Collectively, these investigations will quantify critical configuration trade-offs, expand the applicability of the benchmarking framework to modern smart-grid requirements, and establish the definitive hardware and software requirements for billion-row scalability in production environments.

Bibliography

[1] B. S. P. Addala, M. M. Mohammadi, and K. B. Kent, *DGSim: A Scalable Framework for Simulating Energy Consumption of Household Appliances*, Proceedings of the 39th ECMS International Conference on Modelling and Simulation (Catania, Italy), June 2025, pp. 562–569.

[2] Apache IoTDB, *Query Language, Indexing, and Metadata Management — Apache IoTDB Documentation*, Apache Software Foundation, 2025, Accessed 2025-10-19.

[3] ———, *Storage Engine and TsFile Structure — Apache IoTDB Documentation*, Apache Software Foundation, 2025, Accessed 2025-10-19.

[4] Martin F. Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver, *IoTAbench: an Internet of Things Analytics Benchmark*, Proc. of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE), ACM, 2015, pp. 133–144.

[5] ClickHouse, Inc., *Codecs and Compression — ClickHouse Documentation*, 2025, Accessed 2025-10-19.

[6] ———, *Indexes, Data Skipping, and Vectorized Execution — ClickHouse Documentation*, ClickHouse, Inc., 2025, Accessed 2025-10-19.

[7] ———, *LowCardinality Data Type — ClickHouse Documentation*, 2025, Accessed 2025-10-19.

[8] ———, *MergeTree Tables — ClickHouse Documentation*, ClickHouse, Inc., 2025, Accessed 2025-10-19.

[9] ClickHouse Team, *ClickBench: Benchmarking Analytical Databases*, `https://github.com/ClickHouse/ClickBench`, 2021, Accessed: 2025-06-19.

[10] CrateDB Team, *CrateDB Documentation: Data Model and Storage*, 2024, Accessed: 2025-06-19.

[11] Christos Faloutsos, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, and Yuyang Wang, *Forecasting Big Time Series: Theory and Practice*, Companion Proceedings of the Web Conference 2020 (New York, NY, USA), WWW '20, Association for Computing Machinery, 2020, p. 320–321.

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio, *An Updated Performance Comparison of Virtual Machines and Linux Containers*, 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2015, pp. 171–172.

[13] Peeyush Gupta, Majid Niknami, Danica Dimitrova, Magdalena Balazinska, and Luis Ceze, *SmartBench: A Benchmark for Data Management in Smart Spaces*, Proc. VLDB Endow. **13** (2020), no. 11, 1807–1820.

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

[15] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton, *Architecture of a Database System*, Foundations and Trends in Databases **1** (2007), no. 2, 141–259.

[16] Stratos Idreos, Manos Athanassoulis, Rasmus Pagh, and Martin Zukowski, *Designing Access Methods: The RUM Conjecture*, EDBT, 2016, pp. 461–466.

[17] InfluxData, *Line Protocol — InfluxDB OSS 1.8 Documentation*, `https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/`, 2024, Accessed: 2025-10-08.

[18] InfluxData, Inc., *Measurements, Series, and Indexing — InfluxDB Documentation*, InfluxData, 2025, Accessed 2025-10-19.

[19] ———, *Storage Engine and Shard Management — InfluxDB Documentation*, InfluxData, 2025, Accessed 2025-10-19.

[20] ———, *TSM Compression and Retention — InfluxDB Documentation*, InfluxData, 2025, Accessed 2025-10-19.

[21] Mostafa Jalal, Wojciech Gawronski, and Ulf Leser, *SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial IoT*, Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2022.

[22] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen, *Time Series Management Systems: A Survey*, IEEE Transactions on Knowledge and Data Engineering **29** (2017), no. 11, 2581–2600.

[23] Abdelouahab Khelifati, Mourad Khayati, Anton Dignös, Djellel Difallah, and Philippe Cudré-Mauroux, *TSM-Bench: Benchmarking Time Series Database*

Systems for Monitoring Applications, Proceedings of the VLDB Endowment **16** (2023), no. 11, 3363–3376.

[24] Rui Liu and Jun Yuan, *IoTDB-Benchmark for IoT Scenarios*, `https://arxiv.org/abs/1901.08304`, 2019, arXiv:1901.08304.

[25] Iain L. MacDonald, W. Zucchini, and Roland Langrock, *Hidden Markov Models for Time Series: An Introduction Using R*, 2nd ed., CRC Press, 2016.

[26] Syeda Noor Zehra Naqvi and Sofia Yfantidou, *Time Series Databases and InfluxDB*, Technical report, Université Libre de Bruxelles, Advanced Databases (Winter Semester 2017-2018), 2017, Supervisor: Dr. Esteban Zimanyi — this is a student project report.

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil, *The Log-Structured Merge-Tree (LSM-Tree)*, Acta Informatica **33** (1996), no. 4, 351–385.

[28] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan, *Gorilla: a fast, scalable, in-memory time series database*, Proc. VLDB Endow. **8** (2015), no. 12, 1816–1827.

[29] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang, *Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression*, FAST, 2022.

[30] QuestDB, Ltd., *Architecture and Query Execution — QuestDB Documentation*, QuestDB, 2025, Accessed 2025-10-19.

[31] ———, *Storage Model, Partitioning, and Retention — QuestDB Documentation*, QuestDB, 2025, Accessed 2025-10-19.

[32] ———, *Table Partitioning and Storage Layout — QuestDB Documentation*, QuestDB, 2025, Accessed 2025-10-19.

[33] QuestDB Team, *QuestDB Documentation: Architecture Overview*, 2024, Accessed: 2025-06-19.

[34] Stefano Rinaldi, Federico Bonafini, P. Ferrari, Emiliano Sisinni, and Devis Bianchini, *Impact of Data Model on Performance of Time Series Database for Internet of Things Applications*, 05 2019, pp. 1–6.

[35] Stefano Rinaldi, Federico Bonafini, Paolo Ferrari, Alessandra Flammini, Emiliano Sisinni, and Devis Bianchini, *Impact of Data Model on Performance of Time Series Database for Internet of Things Applications*, IEEE Int. Instrumentation and Measurement Technology Conf. (I2MTC), 2019, pp. 1–6.

[36] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed., McGraw–Hill, 2019.

[37] TigerData, *Compression on Continuous Aggregates — TimescaleDB Documentation*, TigerData, 2025, Accessed 2025-10-19.

[38] ———, *Continuous Aggregates — TimescaleDB Documentation*, TigerData, 2025, Accessed 2025-10-19.

[39] ———, *Hypertables — TimescaleDB Documentation*, TigerData, 2025, Accessed 2025-10-19.

[40] Timescale Team, *Time Series Benchmark Suite (TSBS)*, `https://github.com/timescale/tsbs`, 2020, Accessed: 2025-06-19.

[41] Xinle Wu, Dalin Zhang, Chenjuan Guo, Chaoyang He, Bin Yang, and Christian S. Jensen, *AutoCTS: Automated Correlated Time Series Forecasting – Extended Version*, 2021.

[42] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang, *Time Series Data Encoding for Efficient Storage: A Com-*

parative Analysis in Apache IoTDB, Proc. VLDB Endow. **15** (2022), no. 10, 2148–2160.

[43] Y. Yuan, Y. He, et al., *TsFile: An Efficient Storage Format for Industrial Time-Series Data*, IEEE International Conference on Big Data (BigData), 2019, pp. 2714–2723.

Appendix A

InfluxDB Queries

- Query 1:

  SELECT * FROM WL

  WHERE $time$ >=1704132000000000000

  AND $time$ <=1704142799000000000

- Query 2:

  SELECT COUNT(A23) FROM WL

  WHERE $A23$ >15

- Query 3:

  SELECT MEAN(A21), MEAN(A23) FROM WL

  WHERE house_id = '1'

  GROUP BY time(1m)

- Query 4:

  SELECT MEAN(pc) FROM WL GROUP BY house_type

- Query 5:

```
SELECT MEAN(pc) FROM WL GROUP BY house_id
```

- Query 6:

```
SELECT MEAN(A1), MEAN(A2), MEAN(A3), MEAN(A4), MEAN(A5),
MEAN(A6),MEAN(A7), MEAN(A8), MEAN(A9), MEAN(A10),
MEAN(A11), MEAN(A12),MEAN(A13), MEAN(A14), MEAN(A15),
MEAN(A16), MEAN(A17), MEAN(A18),MEAN(A19), MEAN(A20),
MEAN(A21), MEAN(A22), MEAN(A23), MEAN(A24),
MEAN(A25), MEAN(A26)
FROM WL GROUP BY time(1h), house_id
```

Vita

Candidate's full name: Simin Shehbaz

University attended: Bachelors in Technology (Computer Science and Engineering), MAHE, Dubai, United Arab Emirates, 2021

Publications:

S. Shehbaz, M. Mehabadi, & K. Kent, "Benchmarking and Evaluation of Time Series Databases for Appliance-Level Energy Consumption Data," in Proceedings of the 2025 74 Technical Papers Track, CASCON, 2025.

M. Mehabadi, S. Shehbaz, B. S. Prasad Addala & K. Kent, "Pattern-Driven and Stochastic Generation of Energy Time Series via Differentiable Simulation," in Proceedings of the Technical Papers Track, CASCON, 2025.

S. Shehbaz, M. Mehabadi, & K. Kent, "Evaluating Time Series Databases Across Wide and Narrow Schema Designs for Appliance-Level Residential Energy Monitoring," SoutheastCon, 2026.

Conference Presentations:

S. Shehbaz, M. Mehabadi, and K. Kent, "Benchmarking and Evaluation of Time Series Databases for Appliance-Level Energy Consumption Data," short-paper talk, CASCON 2025 – 74th Technical Papers Track (Data Analysis), York University, Toronto, ON, Canada, November 12, 2025.

Posters (Non-refereed):

Simin Shehbaz, Ravali Santi, Kenneth. B. Kent - Optimizing Household Energy Simulation with Weather-Based Insights, Annual Research Exposition of the UNB Faculty of Computer Science, Fredericton, Canada, April 2024

Simin Shehbaz, Kenneth. B. Kent, Mohammad Mehabadi, Bhavani Sai Prasad Addala - Benchmarking and Optimizing TSDBs forAppliance-Level Energy Consumption Data, Annual Research Exposition of the UNB Faculty of Computer Science, Fredericton, Canada, April 2025