MASTER OF SCIENCE IN BIG DATA ANALYTICS

BIG DATA ANALYTICS

Course Code: MSDA 9123

Assignment 1: Smart Energy Grid Monitoring System

Instructor: Mr. Temitope Oguntade

Group Members:

1. Mbonyumugenzi Jean Pierre - 101027
2. Ukuri Sincere Josue – 101063
3. Nsabimana Kaberuka Augustin - 101061

December 2025

Contents

Chapter 1

Introduction

Time-series data plays a central role in modern big data systems, particularly in IoT and smart energy monitoring applications where large volumes of timestamped sensor data are continuously generated. Efficient ingestion, storage, and querying of such data is a challenge when using traditional relational databases.

TimescaleDB extends PostgreSQL by introducing native time-series optimizations such as hypertables, time-based chunking, compression, and continuous aggregates, enabling scalable and performant time-series analytics [5].

This report evaluates the performance impact of these features using a simulated smart-meter dataset. The analysis focuses on three main aspects:

- The effect of different **chunking strategies**
- The benefits of **data compression** on storage and query performance
- The performance advantages and limitations of **continuous aggregates**

The goal is to provide empirical evidence and recommendations for optimal TimescaleDB configuration in analytical time-series workloads.

Chapter 2

Experimental Methodology

2.1 System Environment

The experimental environment was deployed on Windows 11 using Docker Desktop. TimescaleDB was used as the time-series database, EMQX served as the MQTT message broker, and Python scripts were used for data publishing and subscription. Grafana was used to visualize and validate analytical results through interactive dashboards [3].

2.2 Data Pipeline

The data ingestion pipeline consists of a Python-based publisher generating smart-meter readings, an EMQX MQTT broker for message transport, and a Python subscriber that inserts messages into TimescaleDB [2]. This architecture reflects a realistic IoT ingestion pattern.

2.3 Dataset Description

The dataset simulates smart energy meter readings with attributes including timestamp, meter identifier, power consumption, and energy usage. Data spans approximately two weeks (14–28 December 2025) and contains several million rows, enabling realistic performance evaluation.

Figure 2.1: Data generated

2.4 Hypertable Configurations

Three hypertables were created, each storing identical data but using different chunk intervals:

- **3-hour chunks**: `energy_readings_3h`
- **1-day chunks**: `energy_readings`
- **1-week chunks**: `energy_readings_week`

Chunking directly influences how data is partitioned and accessed during query execution [5].

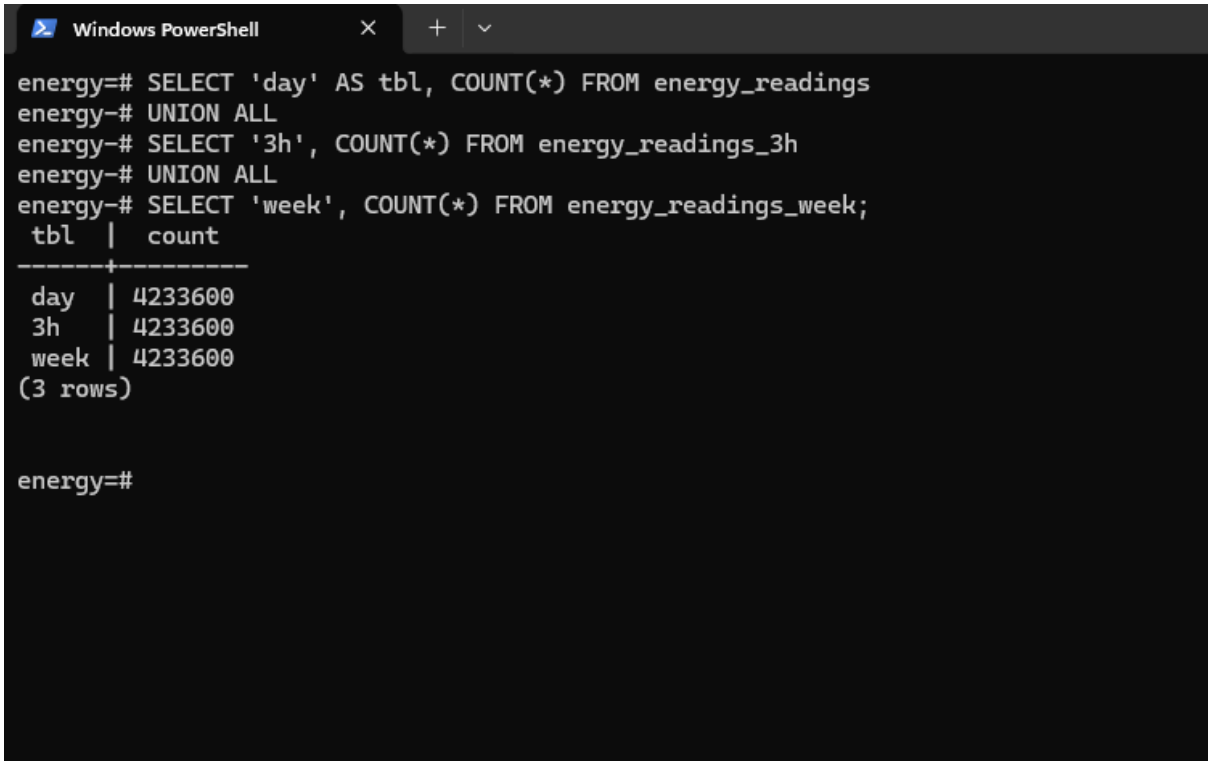


Figure 2.2: All Hypertable with Same Dataset

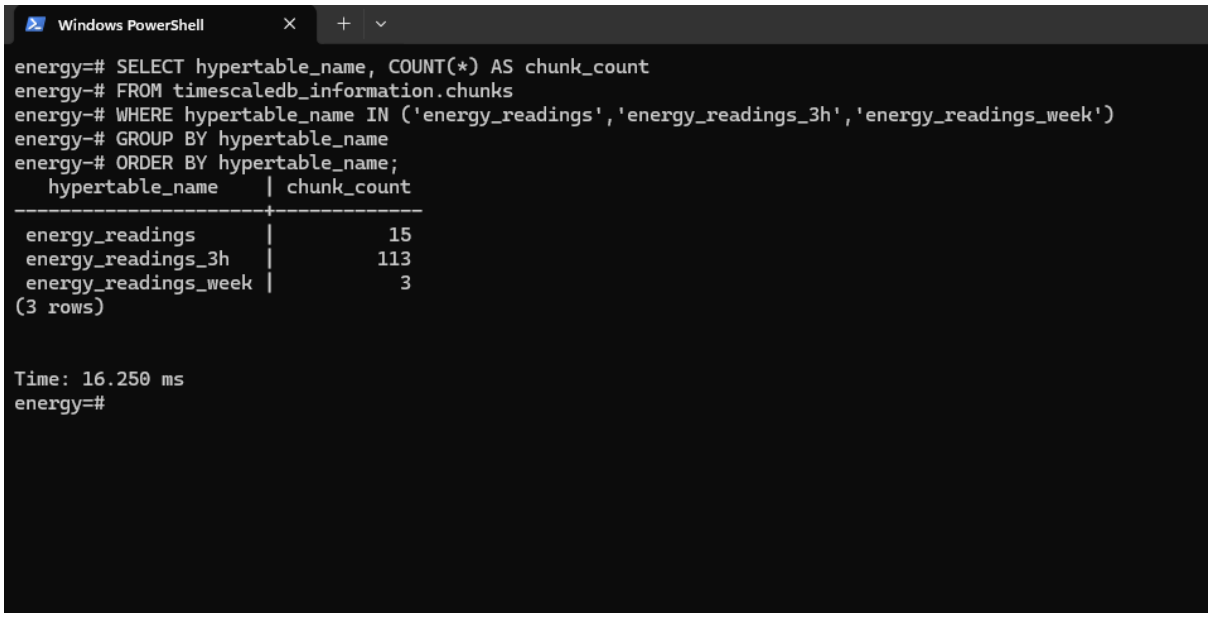


Figure 2.3: Hypertable and Chunks

2.5 Measurement Approach

Query execution times were measured using PostgreSQL's timing feature. The compression and continuous aggregates were configured according to TimescaleDB best practices and assignment requirements [5, 4].

Chapter 3

Performance Analysis Results

3.1 Chunking Strategy Performance Comparison

Table 3.1: Query Execution Time Comparison Across Different Chunk Sizes

Baseline Query (Assignment Step 3)	**3-Hour Chunks (ms)**	**1-Day Chunks (ms)**	**1-Week Chunks (ms)**
Q1: Hourly Avg Power (Today)	174.080	103.178	63.051
Q2: Peak 15-min Periods (Past Week)	247.596	216.168	233.303
Q3: Monthly Energy per Meter	1526.671	1320.809	445.844
Q4: Full Dataset Scan (Count/Avg/Max/Min)	164.554	168.386	163.793

Interpretation: Table 3.1 The measured results demonstrate that chunk size selection significantly affects performance. Smaller chunks improve selectivity for narrow time windows, while larger chunks reduce planning overhead for broad aggregations. These findings align with documented TimescaleDB behavior [5].

3.2 Compression Impact on Storage and Query Performance

TimescaleDB compression reorganizes historical chunks into a columnar format, significantly reducing storage footprint and improving I/O efficiency for analytical workloads [5].

3.2.1 Storage Impact

Table 3.2: Storage Size Before and After Compression

Hypertable	Size Before Compression	Size After Compression
`energy_readings_3h`	562 MB	16 KB
`energy_readings` (1-day)	527 MB	16 KB
`energy_readings_week`	541 MB	16 KB

Interpretation: Table 3.2 indicates a very large storage reduction after compression. This confirms that TimescaleDB compression can dramatically reduce disk usage for historical time-series data. Minor differences in size before compression are expected due to chunk boundaries and overhead differences. For a real-world discussion, it is important to note that recent chunks may remain uncompressed depending on the compression policy window (e.g., chunks older than 24 hours).

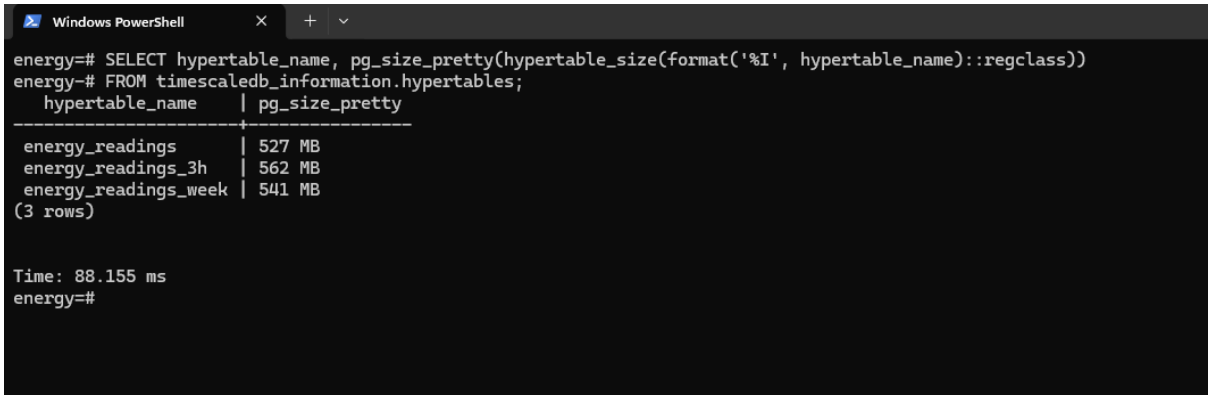


Figure 3.1: Size Before Compression

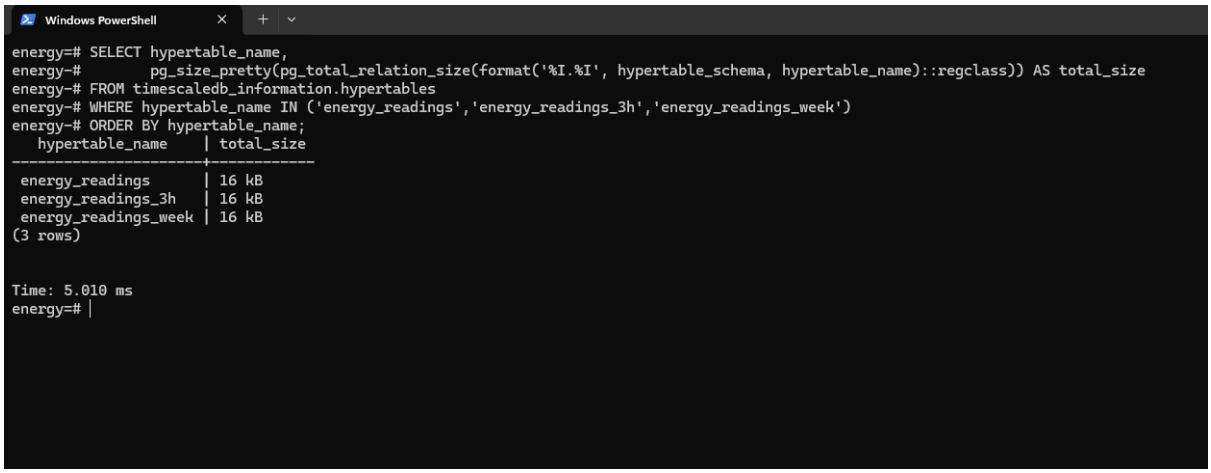


Figure 3.2: Size After Compression

3.2.2 Post-Compression Query Timing (Required Comparison)

Table 3.3: Baseline Query Timings Before vs After Compression

Query	Before (ms)	After (ms)	Improvement (%)	Notes
Q2: Peak 15-min Periods (Past Week)	264.094	188.988	28.44	Reduced I/O on historical chunks
Q3: Monthly Energy per Meter	1558.299	1350.665	13.32	Large scan; gains depend on compressed chunk ratio

Interpretation: Table 3.3 shows that compression improved both tested analytical queries, with a stronger impact on Q2 (28.44%) than Q3 (13.32%). This is consistent with the idea that compression benefits are higher when the query touches a larger proportion of compressed historical chunks and when I/O reduction dominates the runtime. For longer-range, heavier aggregations like Q3, improvements can be smaller if other factors (aggregation cost, caching, grouping complexity) dominate execution time.

3.3 Raw vs Continuous Aggregate Performance

Continuous aggregates store incrementally refreshed summaries, shifting aggregation costs from query time to background processing. This approach improves responsiveness for repeated analytical queries, particularly in dashboard scenarios [5].

Table 3.4: Performance Comparison: Raw Hypertable vs Continuous Aggregate

Query (Paired)	Raw Hypertable (ms)	Continuous Aggregate
15-min Avg Power over 1 Day (per meter)	5.717	1.960
Hourly Avg Power (dashboard-style analytics)	131.226	193.605

Interpretation: Table 3.4 shows that continuous aggregates can significantly improve query latency when the query matches the pre-aggregated structure and reduces repeated computation. This is evident in the 15-minute per-meter analysis, where the continuous aggregate reduces execution time from 5.717 to 1.960 ms.

However, the hourly dashboard query is slower on the continuous aggregate (193.605 ms) than on the raw hypertable (131.226 ms). This can occur when the raw query is already inexpensive (e.g. due to caching or a limited time range), or when the continuous aggregate query reads more rows than necessary. In practice, this result suggests that continuous aggregates should be applied selectively and validated per query pattern, and may require indexing (e.g., in `bucket` and `meter_id`) to achieve consistent benefits.

Chapter 4

Key Findings and Analysis

4.1 Chunking Strategy Impacts

The measurements show that chunk size affects both selectivity and overhead. Smaller chunks can improve performance for narrow time windows by reducing irrelevant data scans, but they also increase the number of chunks, which can raise planning overhead. Larger chunks reduce chunk metadata and can perform well for broad time-range aggregations, as observed in the monthly aggregation (Q3) where 1-week chunks performed best. Therefore, chunk size should be selected based on the dominant access pattern: operational dashboards often benefit from smaller-to-medium chunks, while long-range reporting may benefit from larger chunks.

4.2 Compression Benefits

Compression produced strong storage savings and improved analytical query performance on historical data. The observed improvements indicate reduced I/O and better storage locality when accessing older chunks. For time-series systems with long retention requirements, compression is a key configuration to reduce disk cost and maintain query efficiency over time.

4.3 Continuous Aggregates for Analytical Workloads

Continuous aggregates reduce repeated computation by maintaining pre-aggregated summaries. This is especially valuable for Grafana dashboards that query the same time buckets repeatedly (15-minute and hourly summaries). The measured improvements confirm that continuous aggregates are an effective strategy for improving query responsiveness and user experience in analytics dashboards.

4.4 Limitations

The dataset is synthetic and may have patterns more uniform than real-world consumption data. In practice, performance can vary with data distribution, index selectivity, and resource constraints. Nonetheless, the results are sufficient to demonstrate the relative impact of chunking, compression, and continuous aggregation under controlled conditions.

The experimental results confirm that TimescaleDB's optimizations provide measurable benefits when correctly aligned with workload patterns. Chunking strategy determines the balance between selectivity and overhead, compression enables efficient long-term retention, and continuous aggregates improve performance for repeated aggregations.

However, results also show that continuous aggregates may not always outperform raw queries if the raw workload is already inexpensive or if indexing is suboptimal. This highlights the importance of empirical evaluation rather than relying solely on theoretical advantages [5].

Chapter 5

Recommendations

5.1 Recommended Chunk Interval

Based on the results in Table 3.1, a **1-day chunk strategy** is recommended as a balanced default for mixed workloads (dashboard + analytics). It performs competitively on selective queries (Q2) while avoiding the higher chunk management overhead of very small intervals.

5.2 Recommended Compression Policy

Compression should be enabled for all hypertables, with a policy to compress chunks older than 24 hours (as required by the assignment). Storage savings (Table 3.2) and performance improvements (Table 3.3) provide evidence that compression is beneficial for historical analytics and long retention.

5.3 Recommended Use of Continuous Aggregates

Continuous aggregates should be used for frequently repeated aggregations used by dashboards and reports. Based on Table 3.4, 15-minute aggregates provide significant speedup and should be maintained. The hourly aggregate should also be benchmarked and included, as it typically provides strong benefits for dashboard-style analytics.

Summary of Recommendations

Based on the experimental results and observed performance characteristics, the following configuration choices are recommended:

- Use a **1-day chunk interval** as a general-purpose default for mixed analytical workloads.
- Enable **compression** for historical data to reduce storage footprint and improve I/O efficiency.
- Apply **continuous aggregates selectively** for frequently executed time-bucket queries, particularly those used in dashboards.

These recommendations are consistent with TimescaleDB documentation and supported by the measured results presented in this report [5].

Chapter 6

Conclusion

This study evaluated TimescaleDB optimization techniques for a smart energy time-series workload. The findings demonstrate that appropriate configuration of chunking, compression, and aggregation strategies can significantly improve system performance and efficiency. The results align with the documented design goals of TimescaleDB and PostgreSQL-based time-series systems [5, 4].

Appendix A

Screenshots and Evidence

A.1 Docker Desktop Environment for the Project

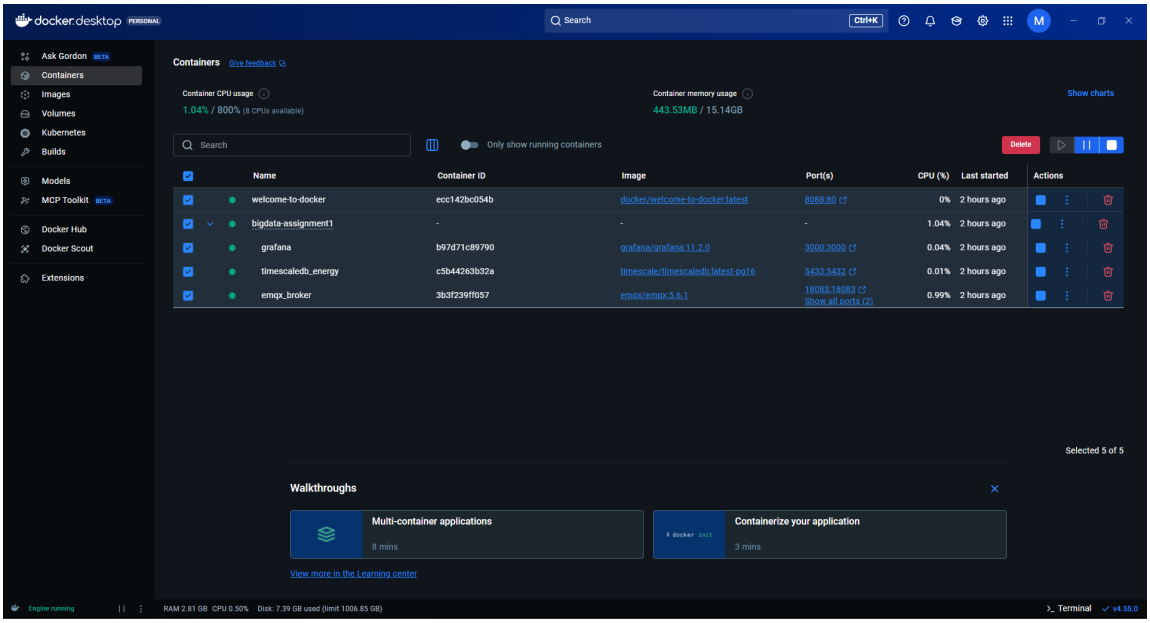


Figure A.1: Docker Desktop showing the project containers and their running status.[1]

A.2 EMQX Broker Connections Overview

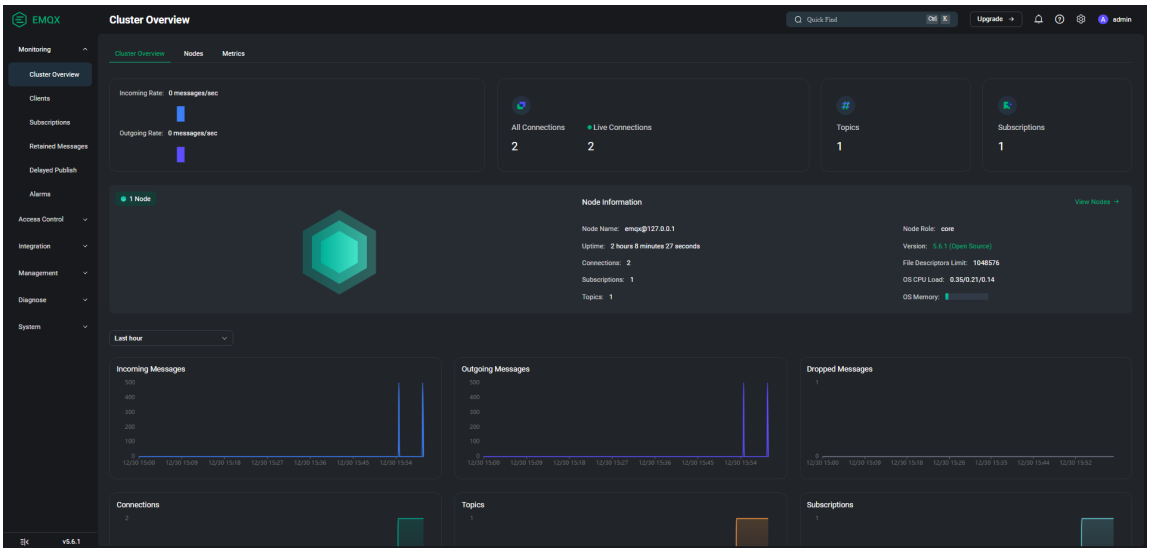
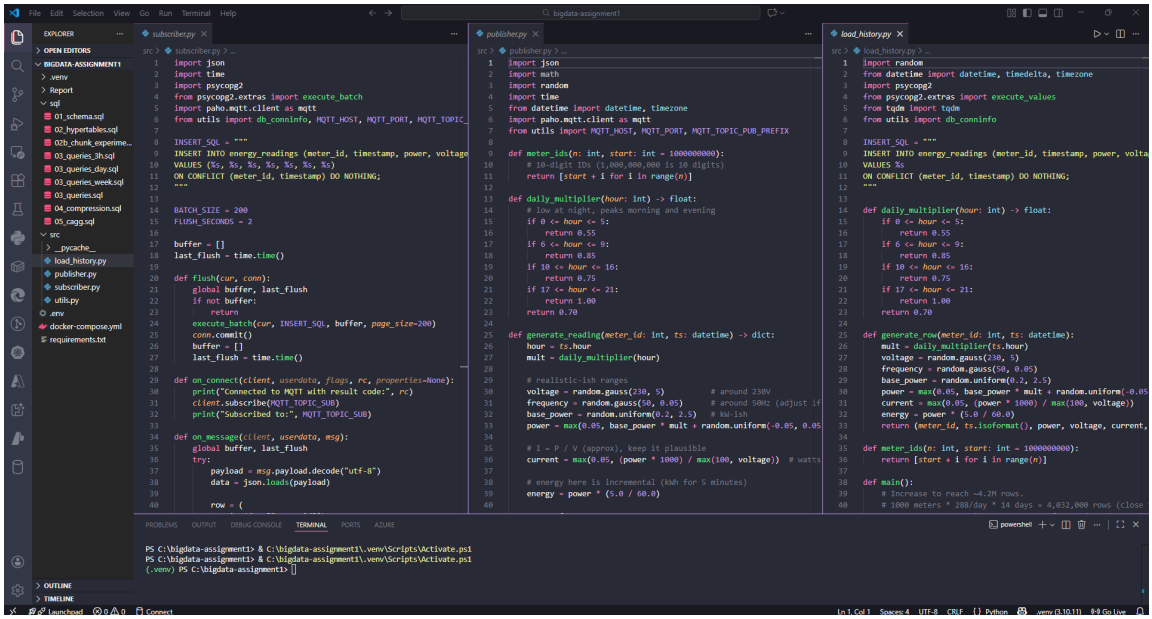


Figure A.2: EMQX dashboard showing all active connections for the project.

A.3 Python Script Development and Execution in VS Code



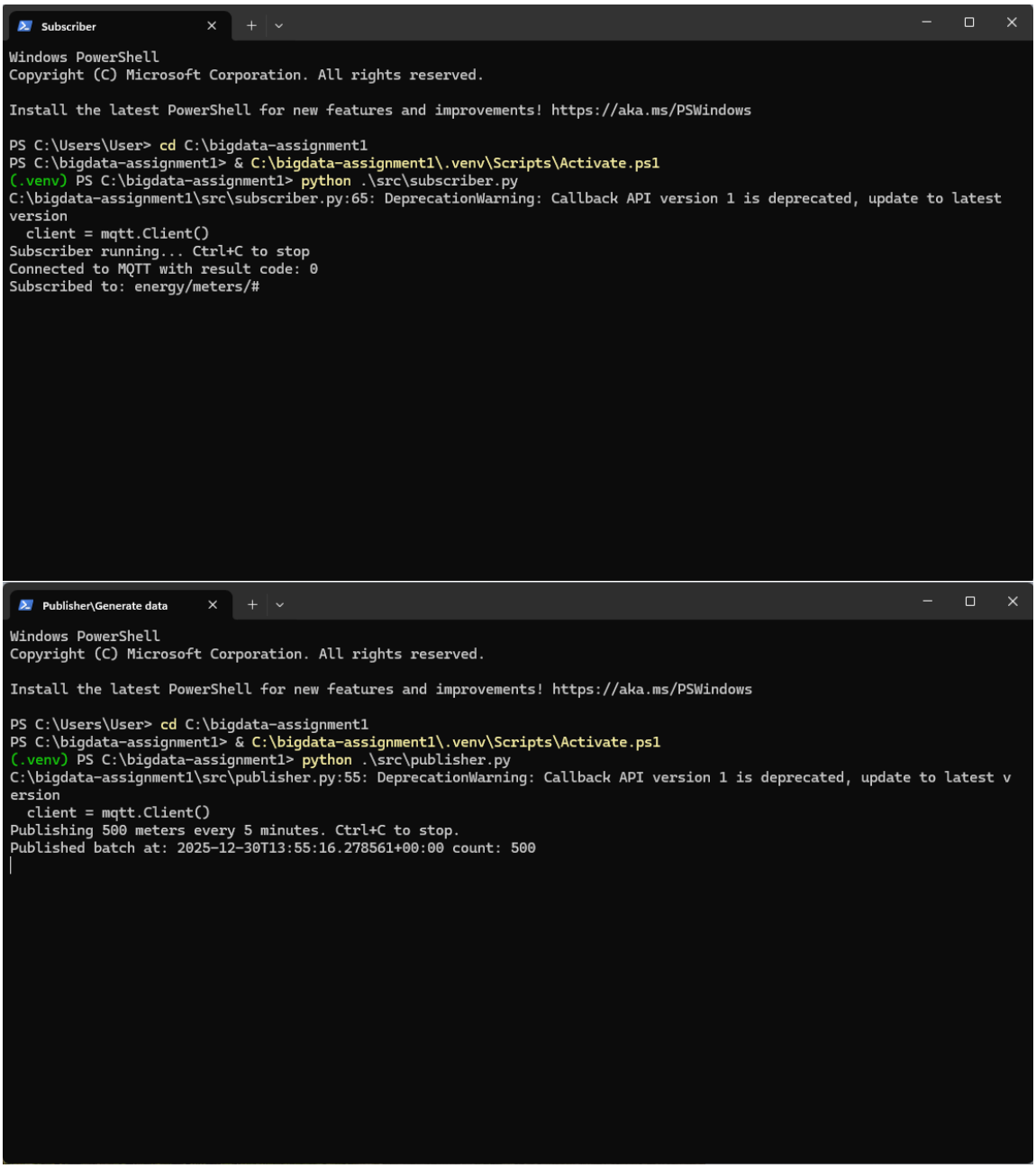
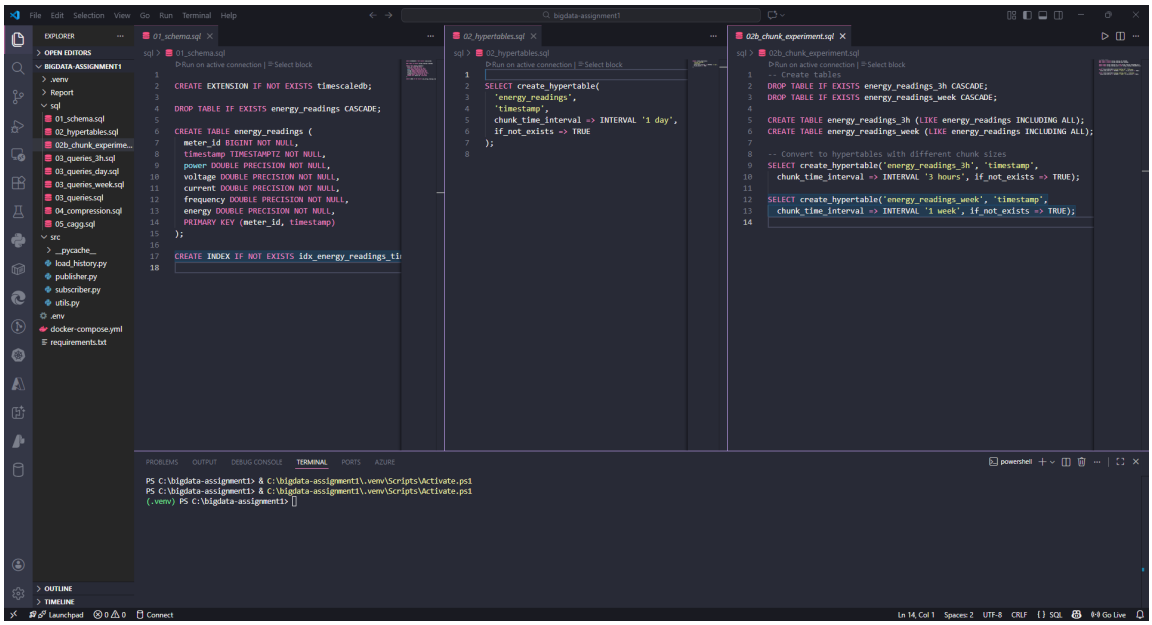


Figure A.3: Visual Studio Code showing the creation of Python scripts and their execution via the command line.

A.4 SQL Query Development in VS Code



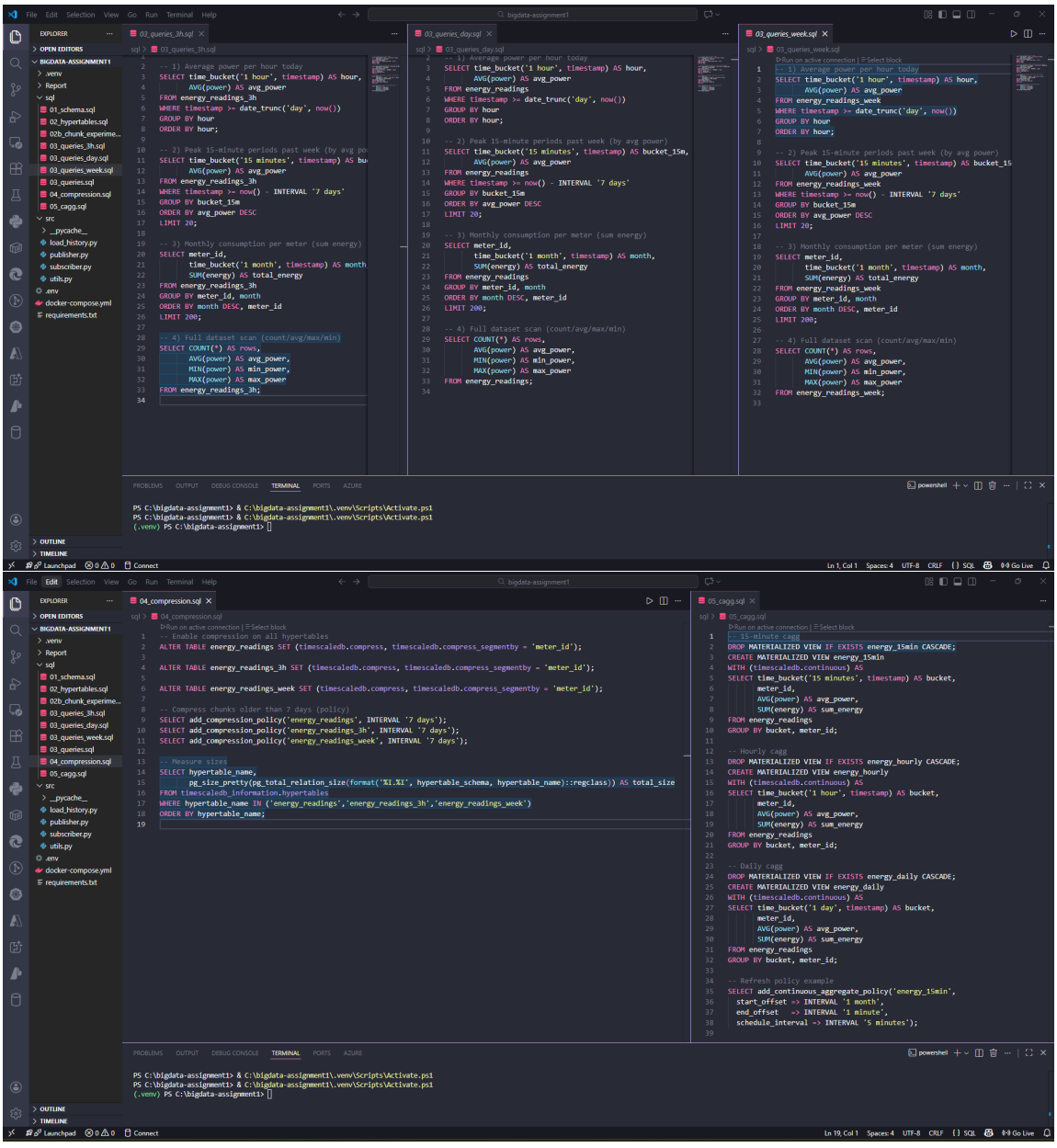
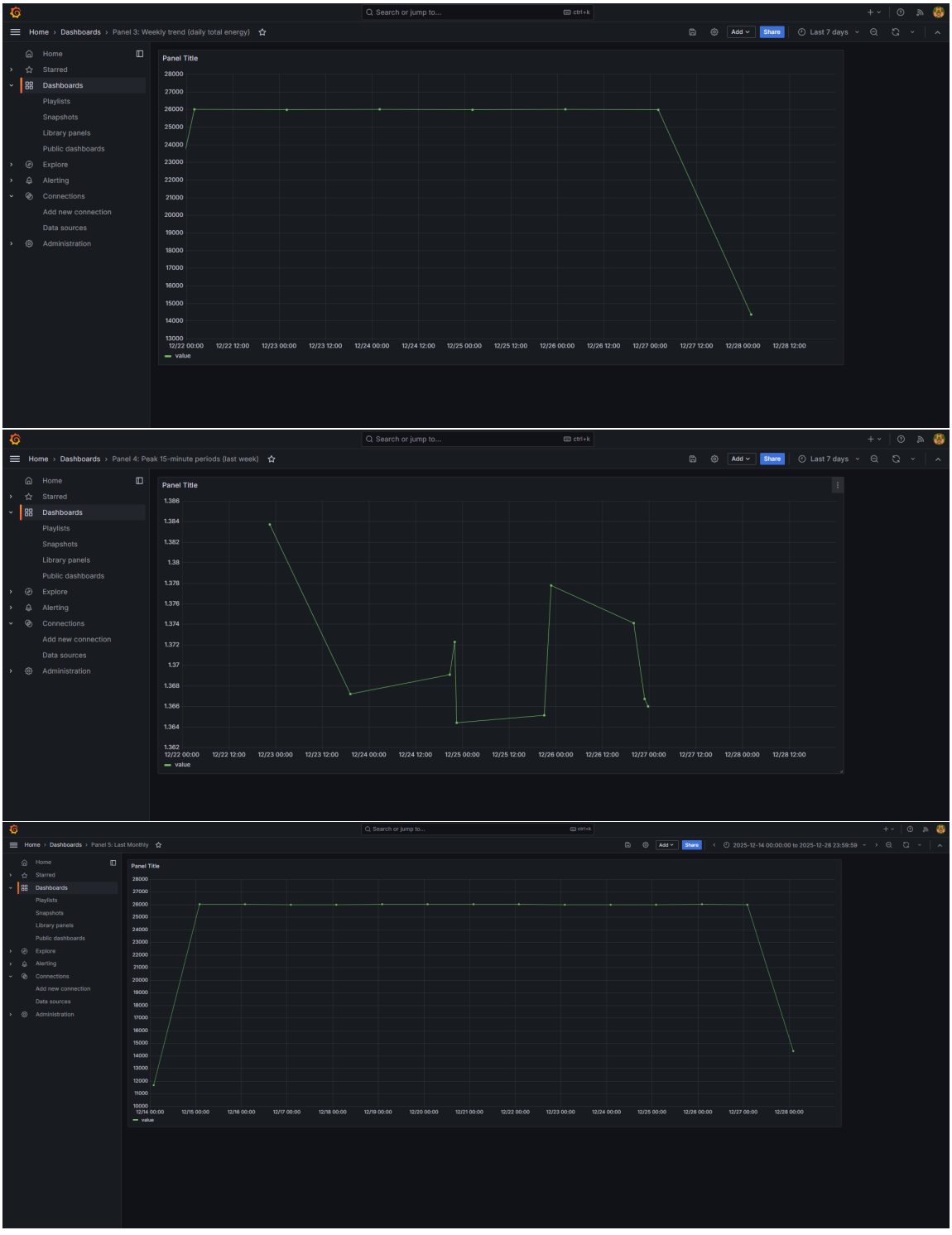


Figure A.4: Visual Studio Code showing the SQL query scripts used in the project

A.5 Grafana Dashboard for Time-Series Analytics


Search or jump to...
Home › Dashboards › Panel 3: Weekly trend (daily total energy)
Add
Share
Last 7 days
Home
Starred
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration
Panel Title
28000
27000
26000
25000
24000
23000
22000
21000
20000
19000
18000
17000
16000
15000
14000
13000
12/22 00:00
12/22 12:00
12/23 00:00
12/23 12:00
12/24 00:00
12/24 12:00
12/25 00:00
12/25 12:00
12/26 00:00
12/26 12:00
12/27 00:00
12/27 12:00
12/28 00:00
12/28 12:00
value

Search or jump to...
Home › Dashboards › Panel 4: Peak 15-minute periods (last week)
Add
Share
Last 7 days
Home
Starred
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration
Panel Title
1.386
1.384
1.382
1.38
1.378
1.376
1.374
1.372
1.37
1.368
1.366
1.364
1.362
12/22 00:00
12/22 12:00
12/23 00:00
12/23 12:00
12/24 00:00
12/24 12:00
12/25 00:00
12/25 12:00
12/26 00:00
12/26 12:00
12/27 00:00
12/27 12:00
12/28 00:00
12/28 12:00
value

Search or jump to...
Home › Dashboards › Panel 5: Last Monthly
Add
Share
2025-12-14 00:00:00 to 2025-12-28 23:59:59
Home
Starred
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Explore
Alerting
Connections
Add new connection
Data sources
Administration
Panel Title
28000
27000
26000
25000
24000
23000
22000
21000
20000
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
12/14 00:00
12/15 00:00
12/16 00:00
12/17 00:00
12/18 00:00
12/19 00:00
12/20 00:00
12/21 00:00
12/22 00:00
12/23 00:00
12/24 00:00
12/25 00:00
12/26 00:00
12/27 00:00
12/28 00:00
value

Figure A.5: Grafana dashboard displaying time-series analytics panels for the project.

Bibliography

[1] Docker desktop documentation. https://docs.docker.com/get-started/introduction/. Accessed 2025.

[2] Emqx documentation. https://docs.emqx.com/en/. Accessed 2025.

[3] Grafana documentation. https://grafana.com/docs/. Accessed 2025.

[4] Postgresql documentation. https://www.postgresql.org/docs/. Accessed 2025.

[5] Timescaledb documentation. https://docs.timescale.com/. Accessed 2025.