University of Messina

Department of Mathematics and Computer Sciences,
Physical Sciences and Earth Sciences

Master's Course in Data Science

Querying NGSI-LD Context Data at Scale: A Hybrid Integration of Orion-LD, HBase, and Hive

Professor:

Prof. Antonio Celesti

Student:

Danny De Novi, 561319

ACADEMIC YEAR 2024/2025

Contents

1 Introduction

This project aims to build a complete pipeline that starts from the origin of the data — namely, sensors — and spans all the way to its consumption through analytical queries. The pipeline is designed to handle context data in NGSI-LD format, enabling semantic interoperability and structured metadata management. The data is first published to the Orion-LD context broker, which acts as the central entry point for real-time context information. Through a custom ingestion layer, this data is then persisted into HBase, a scalable NoSQL columnar database suitable for high-throughput writes and efficient time-series storage. Finally, Hive is used to define external tables over the HBase schema, enabling SQL-like querying of the ingested data for analytical and monitoring purposes. The architecture combines semantic context management with big data persistence and queryability, demonstrating how NGSI-LD can be effectively integrated into modern data processing pipelines.

2 Architecture Components and Technologies

The architecture represents a complete data processing pipeline designed to ingest, persist, and analyze sensor-generated context information using a combination of semantic technologies and big data tools. At the edge, various sensors (e.g., temperature, humidity, brightness) are deployed in smart environments such as rooms, and their observations are published as NGSI-LD entities to Orion-LD, the semantic context broker within the FIWARE ecosystem.

Each sensor type corresponds to a specific topic and is managed as a stream of context updates. One or more subscribers act as middleware components that listen to these updates via Orion-LD subscriptions. These subscribers are responsible for transforming and ingesting the received NGSI-LD payloads into Apache HBase, a column-oriented NoSQL database optimized for high-throughput and time-series storage.

To enable SQL-like analytics over this NoSQL data, Apache Hive is used to define external tables that map directly to the HBase storage schema. This integration allows users to perform analytical queries using HiveQL, effectively bridging semantic data ingestion with structured, queryable analytics.

The architecture supports scalability at multiple levels — from sensor sources to distributed storage and multi-subscriber ingestion — and is designed to provide a robust foundation for real-time context management and historical data analysis.

2.1 OrionLD

Orion-LD is a context broker compliant with the ETSI NGSI-LD standard, designed to manage context information structured according to a semantic model based on JSON-LD. It represents the semantic evolution of the classic Orion broker and is developed within the FIWARE ecosystem. Orion-LD enables the registration, update, and subscription of entities that model real-world objects, events, or concepts, enriched with properties, relationships, and temporal metadata in accordance with an implicit RDF graph structure. Unlike its predecessors, Orion-LD supports linked data principles, making it suitable for use cases that require semantic interoperability and fine-grained data integration across heterogeneous systems.

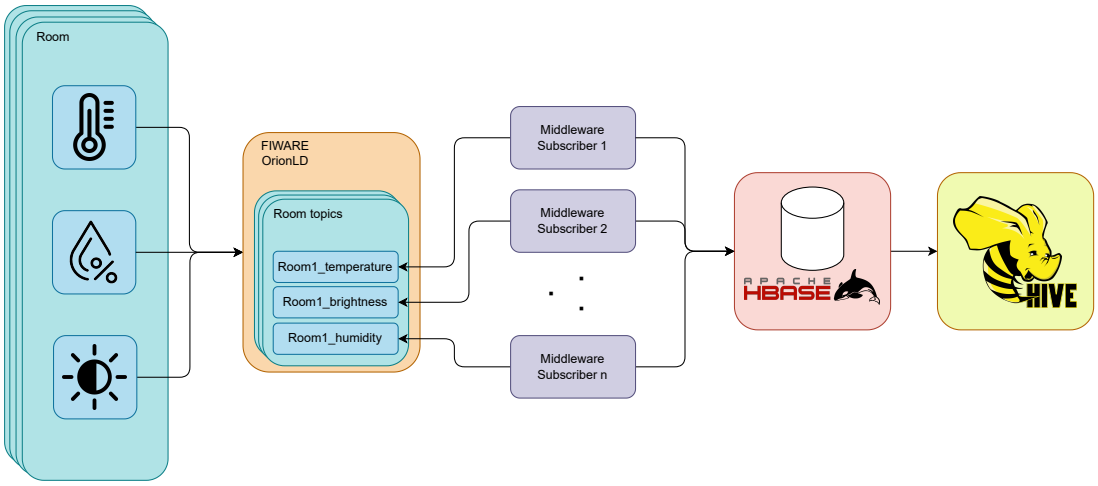

Figure 1: Project architecture

2.2 HBase

Apache HBase is a distributed, scalable, column-oriented NoSQL database built on top of the Hadoop Distributed File System (HDFS). It is designed to handle large volumes of sparse data across commodity hardware with high write throughput and real-time read capabilities. Inspired by Google's Bigtable, HBase organizes data into tables, column families, and rows, where each cell is versioned by timestamp. Unlike traditional relational databases, HBase does not support SQL but provides efficient random access to data using row keys. It is particularly suitable for time-series data, sensor logs, and applications requiring horizontal scalability and strong consistency. In this project, HBase serves as the persistent storage layer for context data ingested from Orion-LD, enabling high-performance write operations and long-term historical data retention.

2.3 Hive

Apache Hive is a data warehouse system built on top of Hadoop that facilitates querying and analyzing large datasets using a SQL-like language called HiveQL. Originally developed by Facebook, Hive abstracts the complexity of distributed data processing by translating high-level queries into low-level MapReduce, Tez, or Spark jobs. It supports schema-on-read, allowing structured queries over data stored in various formats such as ORC, Parquet, or plain text files in HDFS. While Hive is not designed for low-latency operations, it is well-suited for batch analytics and integration with external storage systems. Through the HBaseStorageHandler, Hive can define external tables mapped to HBase schemas, enabling SQL-style access to NoSQL data. In this project, Hive provides a declarative interface to query the context data stored in HBase, effectively bridging semantic data ingestion with analytical workflows.

3 Setup and Deployment

The software architecture follows a microservice-oriented design and relies on Linux containerization through Docker. Docker provides lightweight, reproducible environments that simplify deployment

and ensure consistency across different systems.

All the relevant source code and configuration files can be found in the project's GitHub repository[1].

The deployment begins by cloning the main repository on the machine designated to act as the Orion-LD server.

A dedicated `docker-compose.yaml` file has been created to streamline the deployment of Orion-LD. Its content is shown below:

```
services:
mongo:
image: mongo:4.4
command:
    -nojournal

orion:
image: fiware/orion-ld
depends_on:
    - mongo
ports:
    - "1026:1026"
command: -dbhost mongo
```

Listing 1: Orion-LD Docker Compose Configuration

To start the Orion-LD services, run the following command:

```
$ docker compose -f orion-compose.yaml up
```

Listing 2: Launching Orion-LD via Docker Compose

On a separate machine (or container host), clone both the OrionLD-to-Hive repository and the Hive-HBase Docker environment that has been made ad hoc for the project[2].

Navigate into the Hive-HBase directory and build the Docker image:

```
$ docker build -t hive-hbase-standalone .
```

Listing 3: Building the Hive-HBase Docker Image

Once the image is built, you can run it using the following command:

```
$ docker run -d
-p 10000:10000 \ # HiveServer2 JDBC/Thrift
-p 9090:9090 \ # HBase Thrift API
-p 16010:16010 \ # HBase Web UI
hive-hbase-standalone
```

Listing 4: Running the Hive-HBase Container

[1] https://github.com/dannydenovi/OrionLD-to-Hive
[2] https://github.com/dannydenovi/hive-hbase-docker

After the container is initialized, execute the `subscriber.py` script using Python 3 on the Orion-LD machine to subscribe to NGSI-LD events and persist them in HBase. You should also run `real_time_data_simulator.py` to simulate sensor data publishing to Orion-LD.

Once the ingestion process is complete, you can query the data stored in HBase through Hive. This can be done either by running the `hive.py` script with Python 3 or interactively using Beeline from within the Hive container.

3.1 Dataset Description

The dataset[3] is composed of 18 CSV files, each representing time-series sensor measurements collected from various rooms in a building. These rooms include Room1, Room2, Room3, the Kitchen, the Bathroom, and the Toilet. For each room, three types of environmental parameters were recorded: temperature (in degrees Celsius), humidity (in percentage), and brightness (unit not specified). Each file is named according to the corresponding room and parameter it contains.

The data are organized as pairs of timestamps and values, indicating the sensor reading at a specific point in time. On average, each file contains between 10,000 and 11,000 entries, highlighting a relatively stable and frequent sampling rate throughout the data collection period.

The CSV structure lacks uniformity, with fields separated by tab characters and no consistent column headers, which suggests the need for a preliminary preprocessing step prior to any analytical use. Despite this inconsistency, the data are rich in temporal and spatial coverage, making them suitable for applications in smart building monitoring, environmental quality control, and spatio-temporal analysis of sensor data.

The underlying data storage is based on Apache HBase, a distributed, column-oriented NoSQL database optimized for high-throughput and sparse time-series data. Each room is represented as a dedicated HBase table, allowing for logical separation and parallel ingestion. Specifically, the following six tables are defined in the schema:

- `kitchen_data`
- `room1_data`
- `room2_data`
- `room3_data`
- `bathroom_data`
- `toilet_data`

Each table stores sensor readings under the column family `cf`. The row key is composed of the entity name, a randomized timestamp (to simulate real-time data), and a row index to ensure uniqueness:

The column qualifiers within the `cf` family include:

- `cf:timestamp` – the time of measurement (as string-encoded datetime)
- `cf:temperature` – the temperature value (if applicable)

[3]https://www.kaggle.com/datasets/claytonmiller/open-smart-home-iotieqenergy-data

- `cf:humidity` – the humidity value (if applicable)
- `cf:brightness` – the brightness value (if applicable)

This schema design allows heterogeneous sensor types to coexist in the same table, with columns being sparsely populated depending on the sensor type.

To enable SQL-based querying, Apache Hive is used to expose each HBase table as an external Hive table. These external tables are defined using the `HBaseStorageHandler`, with a column mapping that links Hive columns to HBase qualifiers. A typical Hive table schema includes:

- `entityid STRING` – mapped to the row key
- `temperature DOUBLE`
- `humidity INT`
- `brightness DOUBLE`
- `ts TIMESTAMP`

The mapping is specified via the SerDe property:

```
‘‘hbase.columns.mapping’’ = ‘‘:key,cf:temperature,cf:humidity,cf:brightness,cf:timestamp’’
```

This design ensures a clear one-to-one correspondence between the HBase backend and the Hive query layer, enabling analytical operations such as filtering, aggregation, joining, and multidimensional grouping over semantically structured IoT sensor data.

4 Execution Time Test on HBase via Hive

A total of four queries were executed on the HBase instance using Hive to evaluate its execution time using the SQL syntax. Each query was tested using 25%, 50%, 75%, and 100% of the dataset. Both the first execution (cold cache) and the subsequent cached execution were measured. The queries were designed to be progressively more computationally complex. To automate the execution of queries, a Python script has been developed. This script calculates the execution time of the initial query execution and stores the subsequent 30 instances for caching. Subsequently, the mean and the 95% confidence interval are computed.

4.1 First Query

This query retrieves the temperature and timestamp values from the kitchen_data table for all records collected within the last hour. It filters data based on the current system time using `unix_timestamp(ts) >= unix_timestamp() - 3600`, effectively returning the most recent readings.

```
SELECT temperature, ts
       FROM kitchen_data
       WHERE unix_timestamp(ts) >= unix_timestamp() - 3600;
```

Listing 5: First Query

Table 1: Performance Metrics – First Query Execution

Dataset %	Warm-up (s)	Mean (s)	Std Dev (s)	95% CI (±s)
25	0.474	0.403	0.060	0.0216
50	0.707	0.521	0.084	0.0302
75	0.826	0.646	0.059	0.0209
100	0.834	0.730	0.049	0.0176

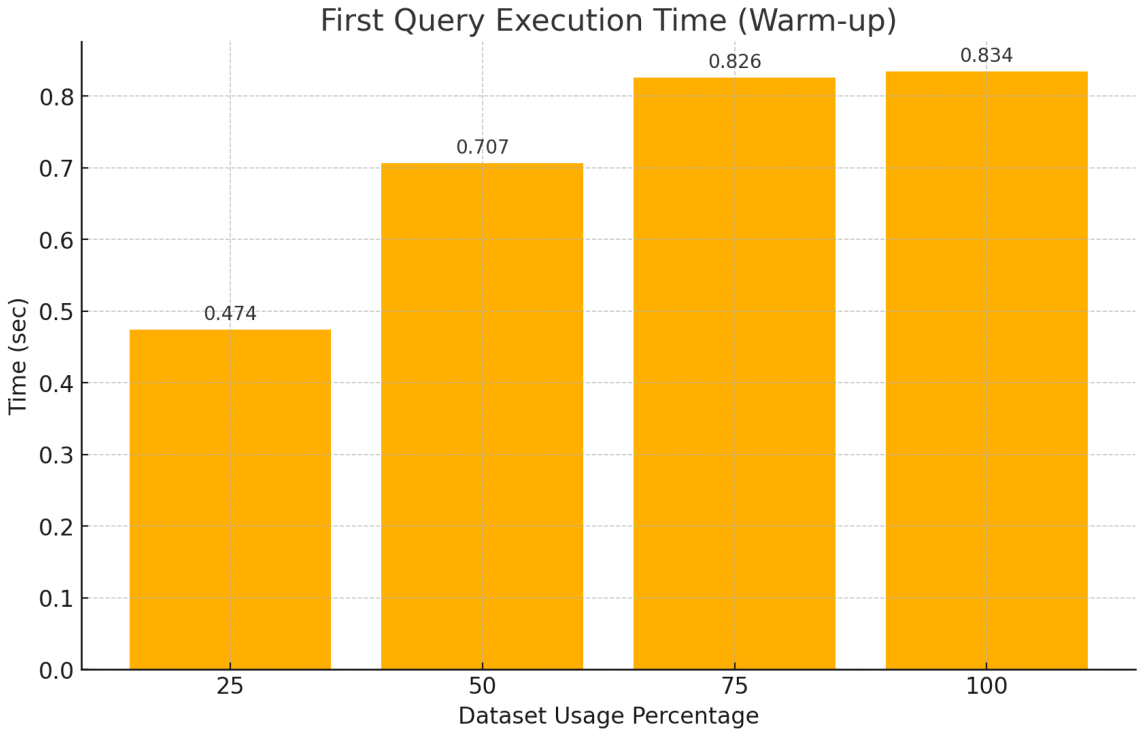

Figure 2: First Query - Warm-up

4.2 Second Query

This query computes the average temperature `(AVG(CAST(temperature AS DOUBLE)))` from the kitchen_data table, grouped by the hour extracted from the timestamp field (hour(ts)). The results are sorted by the hourly time bucket.

```
SELECT hour(ts) as hour_bucket, AVG(CAST(temperature AS DOUBLE)) as avg_temp
       FROM kitchen_data
       GROUP BY hour(ts)
       ORDER BY hour_bucket;
```

Listing 6: First Query

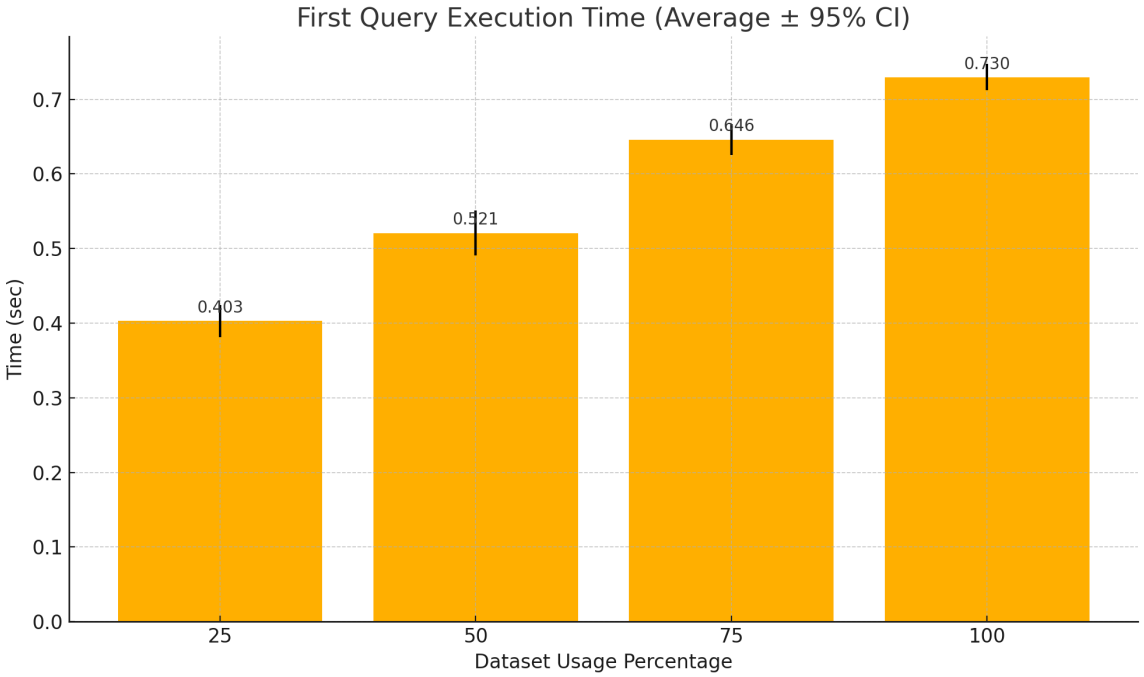

Figure 3: First Query - Average

Table 2: Performance Metrics – Second Query Execution

Dataset %	Warm-up (s)	Mean (s)	Std Dev (s)	95% CI (±s)
25	4.735	4.721	0.769	0.275
50	6.643	5.251	0.796	0.285
75	4.279	5.081	0.336	0.120
100	4.500	5.064	0.303	0.108

4.3 Third Query

This is the most complex query, performing a union of all six sensor tables: kitchen_data, room1_data, room2_data, room3_data, bathroom_data, and toilet_data. Each record is tagged with a room name. The resulting dataset is then grouped using the CUBE operator over both room and hour(ts), and for each group, the following statistics are calculated: Average temperature, Minimum humidity, Maximum brightness, Count of total records.

```
SELECT
  tab.room,
  HOUR(tab.ts) AS hour,
  AVG(CAST(tab.temperature AS DOUBLE)) AS avg_temperature,
  MIN(CAST(tab.humidity AS INT)) AS min_humidity,
  MAX(CAST(tab.brightness AS DOUBLE)) AS max_brightness,
  COUNT(*) AS count_readings
FROM (
  SELECT 'kitchen' AS room, entityid, temperature, humidity, brightness, ts FROM
      kitchen_data
  UNION ALL
```

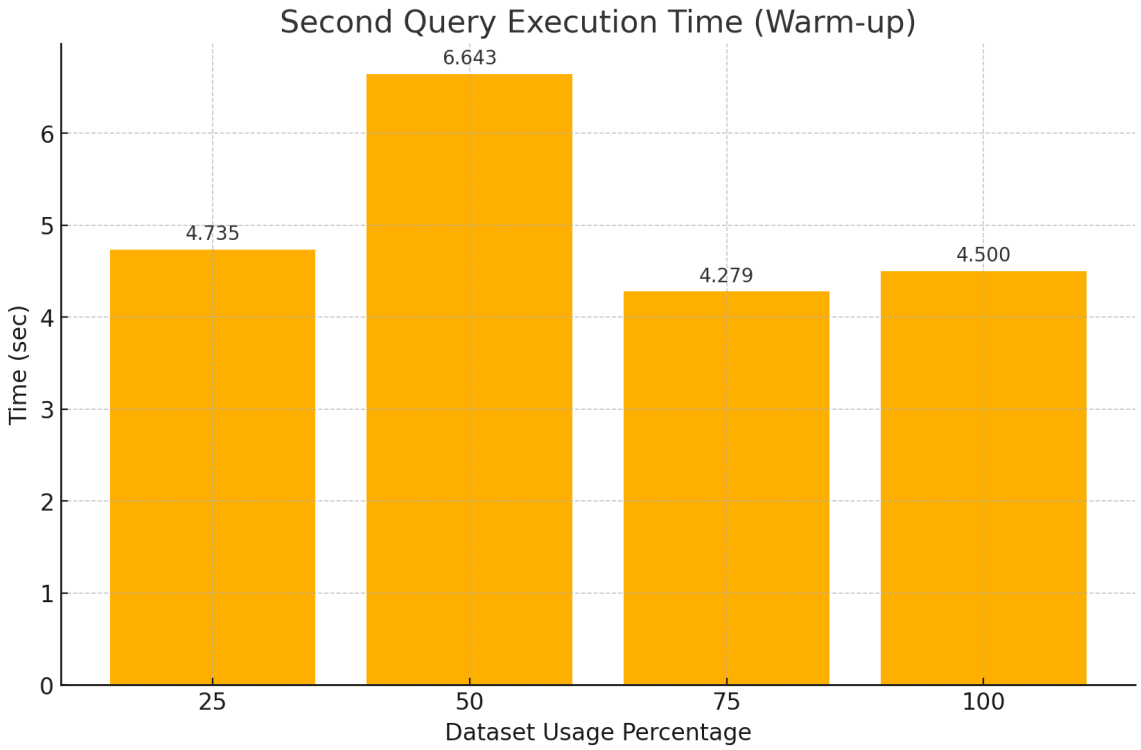

Figure 4: Second Query - Warm-up

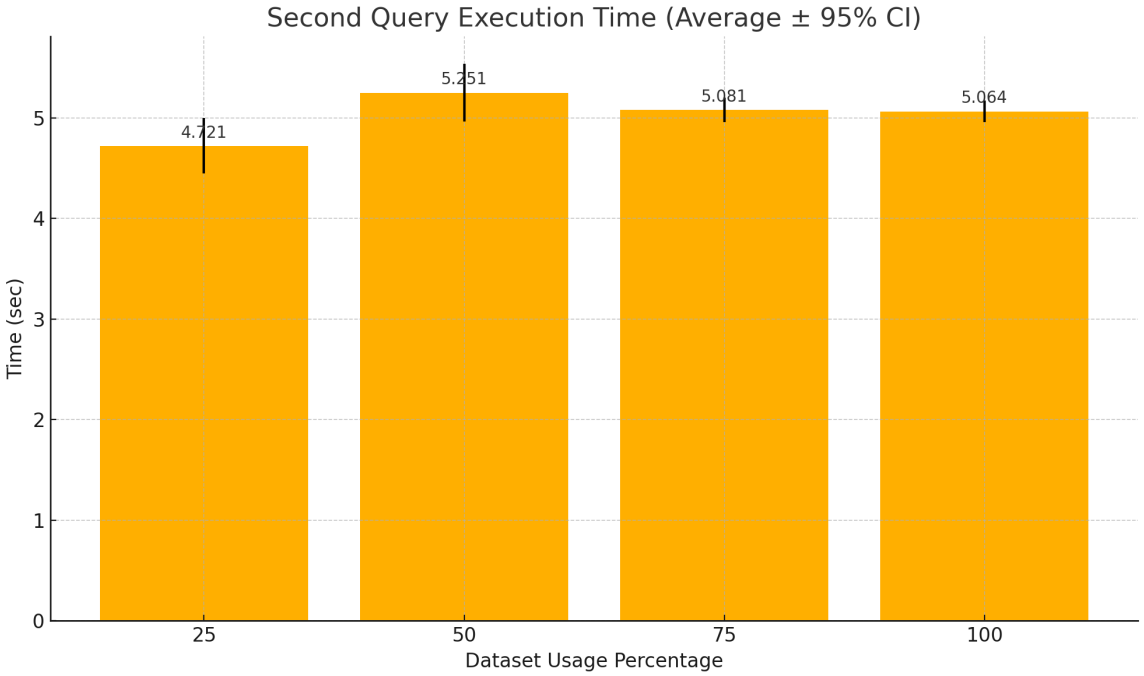

Figure 5: Second Query - Average

```
SELECT 'room1', entityid, temperature, humidity, brightness, ts FROM room1_data
UNION ALL
```

```
  SELECT 'room2', entityid, temperature, humidity, brightness, ts FROM room2_data
  UNION ALL
  SELECT 'room3', entityid, temperature, humidity, brightness, ts FROM room3_data
  UNION ALL
  SELECT 'bathroom', entityid, temperature, humidity, brightness, ts FROM bathroom_data
  UNION ALL
  SELECT 'toilet', entityid, temperature, humidity, brightness, ts FROM toilet_data
) tab
GROUP BY CUBE (tab.room, HOUR(tab.ts));
```

Listing 7: First Query

Table 3: Performance Metrics – Third Query Execution

Dataset %	Warm-up (s)	Mean (s)	Std Dev (s)	95% CI (±s)
25	7.736	6.660	0.957	0.342
50	7.720	7.002	0.609	0.218
75	7.718	6.906	0.446	0.160
100	7.620	8.091	2.233	0.799

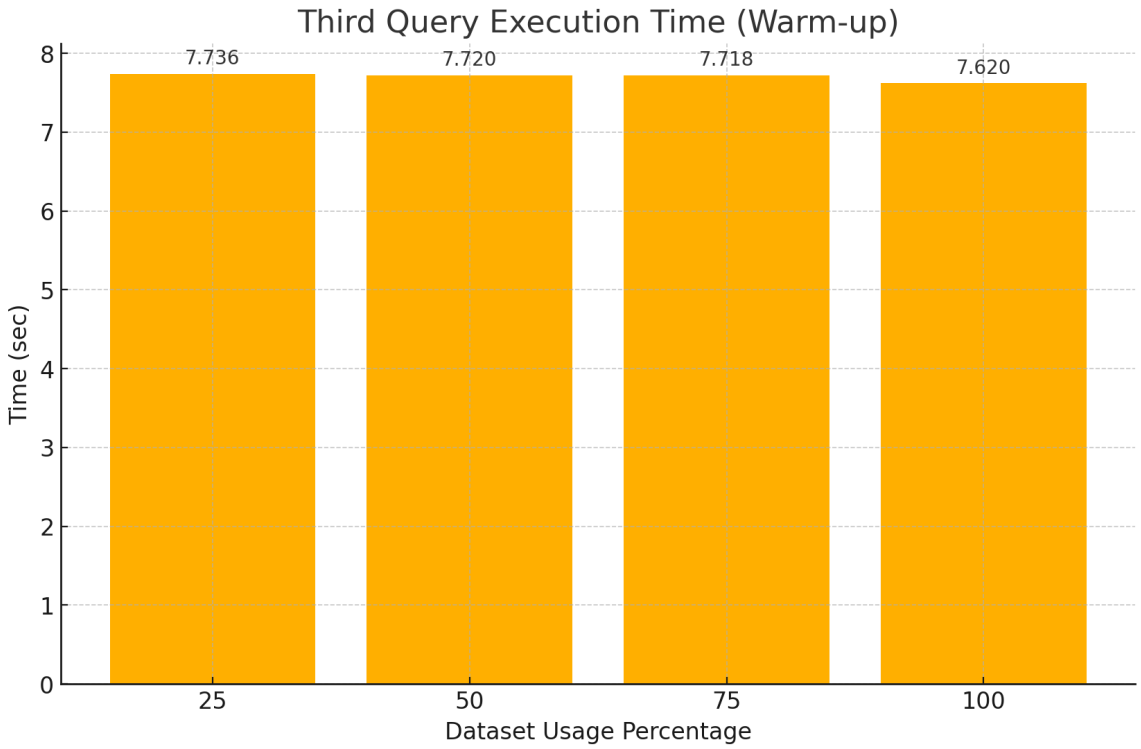

Figure 6: Third Query - Warm-up

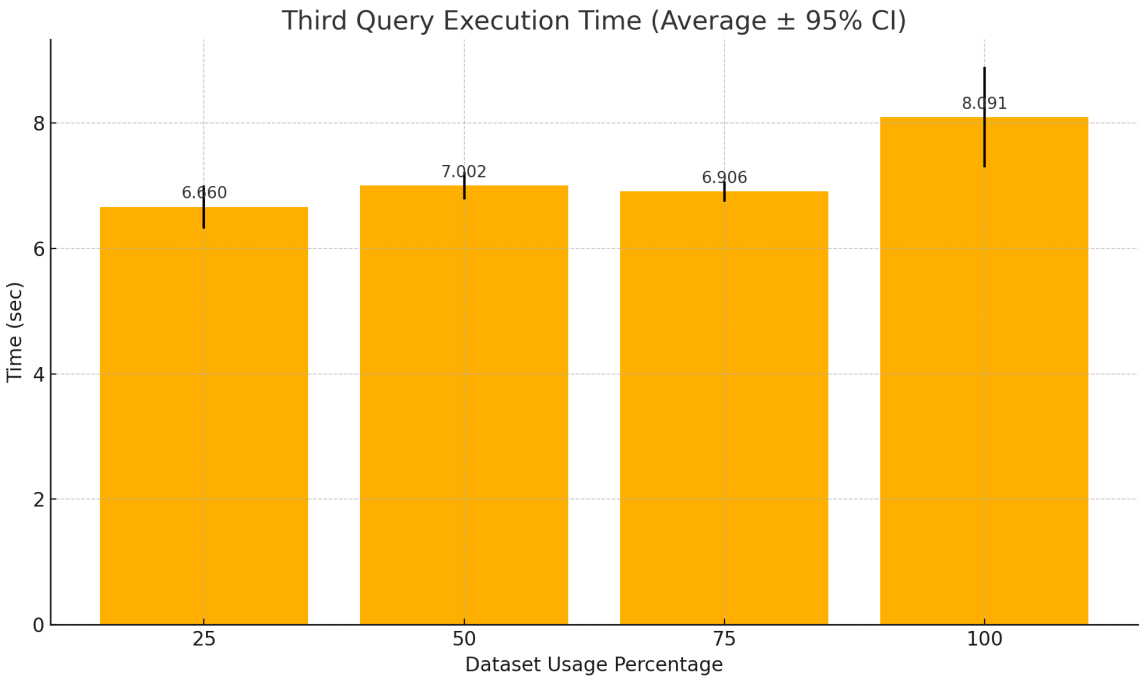

Figure 7: Third Query - Average

4.4 Fourth Query

This query computes hourly average temperatures for Room 1 and Room 2 independently and then joins the results on the hour field to compare the two rooms' temperatures side by side.

```
SELECT r1.hour, r1.avg_temp as room1_temp, r2.avg_temp as room2_temp
       FROM (
           SELECT hour(ts) as hour, AVG(CAST(temperature AS DOUBLE)) as avg_temp
           FROM room1_data
           GROUP BY hour(ts)
       ) r1
       JOIN (
           SELECT hour(ts) as hour, AVG(CAST(temperature AS DOUBLE)) as avg_temp
           FROM room2_data
           GROUP BY hour(ts)
       ) r2
       ON r1.hour = r2.hour;
```

Listing 8: First Query

Table 4: Performance Metrics – Fourth Query Execution

Dataset %	Warm-up (s)	Mean (s)	Std Dev (s)	95% CI (±s)
25	14.027	10.230	1.079	0.386
50	14.038	10.569	0.807	0.289
75	15.490	10.172	0.806	0.288
100	12.991	10.533	0.721	0.258

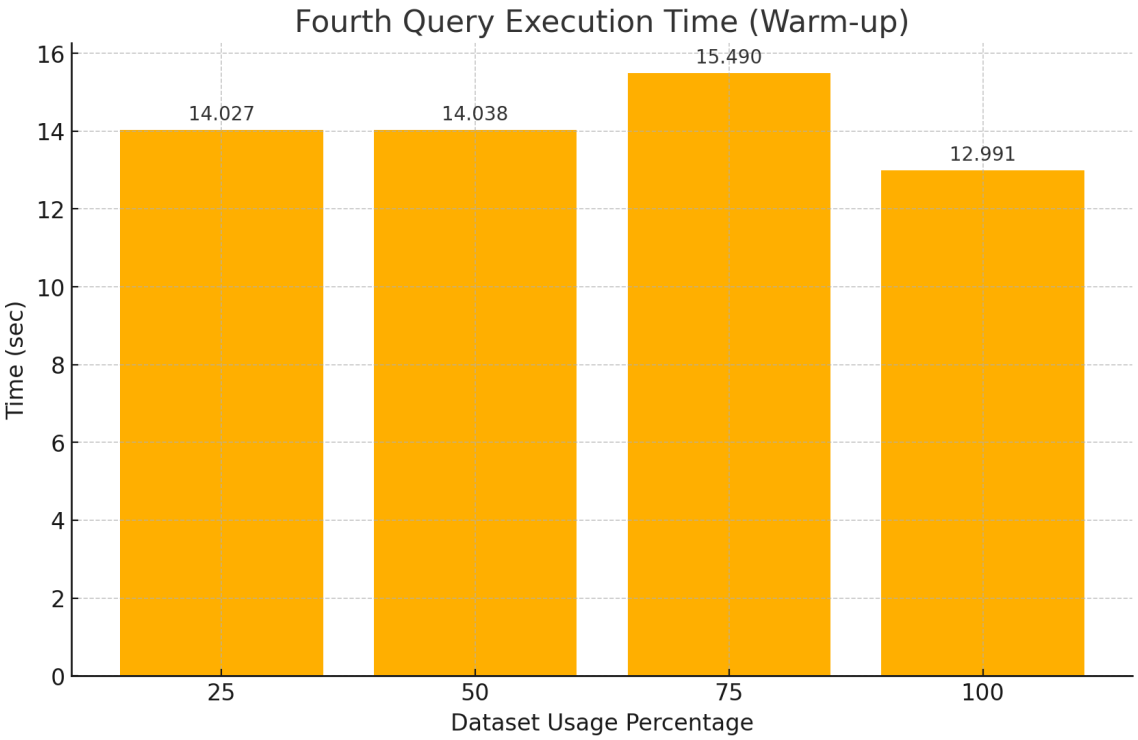

Figure 8: Fourth Query - Warm-up

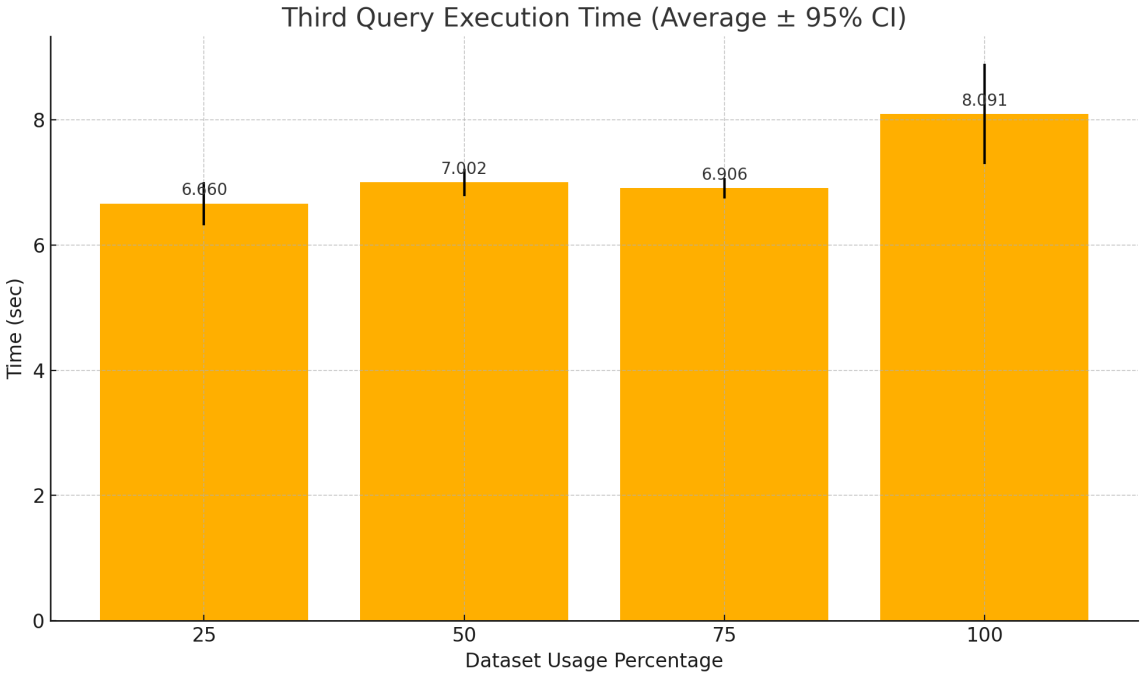

Figure 9: Fourth Query - Average

Table 5: Query Performance Summary (All Queries)

Query	Dataset %	Warm-up (s)	Mean (s)	Std Dev (s)	95% CI (±s)
1	25	0.474	0.403	0.060	0.0216
1	50	0.707	0.521	0.084	0.0302
1	75	0.826	0.646	0.059	0.0209
1	100	0.834	0.730	0.049	0.0176
2	25	4.735	4.721	0.769	0.275
2	50	6.643	5.251	0.796	0.285
2	75	4.279	5.081	0.336	0.120
2	100	4.500	5.064	0.303	0.108
3	25	7.736	6.660	0.957	0.342
3	50	7.720	7.002	0.609	0.218
3	75	7.718	6.906	0.446	0.160
3	100	7.620	8.091	2.233	0.799
4	25	14.027	10.230	1.079	0.386
4	50	14.038	10.569	0.807	0.289
4	75	15.490	10.172	0.806	0.288
4	100	12.991	10.533	0.721	0.258

5 Overall Values

The measured execution times across the four query types reflect the increasing complexity of the SQL logic and the underlying operations performed on the data. Each query introduces additional computational effort, which explains the observed performance trends.

Query 1 is a simple time-filtered `SELECT` on a single HBase-backed table (e.g., `kitchen_data`). It retrieves temperature values from the last hour using a `WHERE` clause on a timestamp. This query is lightweight and involves a linear scan with no aggregations or joins. As expected, its execution time is the lowest across all queries and grows linearly with the dataset size.

Query 2 groups the data by hour and computes an average temperature. Although it still operates on a single table, the `GROUP BY` clause introduces data shuffling and aggregation, which require additional processing. This results in higher execution times and warm-up latency compared to Query 1.

Query 3 executes a `GROUP BY CUBE` across six different entity tables (`kitchen`, `room1`, `room2`, `room3`, `bathroom`, `toilet`). The `UNION ALL` merges all readings into a single logical stream, and the cube operation computes grouped aggregations over both `room` and `hour`. Although `GROUP BY CUBE` is typically less expensive than joins, in this case, the cost stems from the volume of data being aggregated and the number of dimensions being expanded. Additionally, the necessity to scan and merge six HBase tables amplifies the I/O and coordination overhead. This results in consistently high warm-up and execution times, though with slightly lower variability compared to Query 4, since no join keys are evaluated.

Query 4 performs a join between two aggregated subqueries (`room1_data` and `room2_data`), each grouped by hour. While the grouped views reduce the row count, Hive must still evaluate a distributed join over HBase-backed tables. This involves separate scans, grouping operations, and key matching on the `hour` attribute. The join introduces significant overhead and variability, particularly at the 100% dataset level, where the execution time shows a noticeable increase and a higher standard deviation. In this context, joins over HBase tables are particularly costly due to

the lack of native join optimization and indexing.

6 Conclusions

This project demonstrated the design and implementation of a complete data pipeline that integrates semantic context management with scalable storage and querying technologies. Starting from real-time sensor data published to Orion-LD in NGSI-LD format, the system enables persistent storage in HBase and structured analytical access via Apache Hive.

The architecture proved to be modular, portable, and aligned with modern microservice principles thanks to Docker-based deployment. The use of external Hive tables mapped over HBase made it possible to query live and historical data using SQL-like constructs while retaining the scalability and throughput of a NoSQL backend.

Overall, the system shows that combining semantic interoperability (NGSI-LD) with big data technologies (HBase + Hive) is feasible and effective, although some queries—especially those involving joins—require careful consideration of performance implications in production-grade deployments.