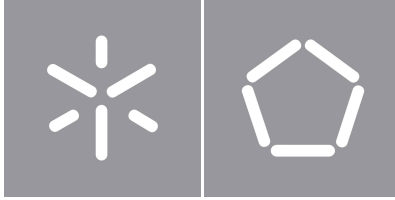


University of Minho
School of Engineering

Pedro Miguel de Leal Meireles Pereira

MulletBench: Multi-layer Edge Time Series Database Benchmark



University of Minho
School of Engineering

Pedro Miguel de Leal Meireles Pereira

MulletBench: Multi-layer Edge Time Series Database Benchmark

Master's Dissertation
Master in Informatics Engineering

Dissertation supervised by
Fábio André Castanheira Luís Coelho (advisor)
João Tiago Medeiros Paulo (co-advisor)
**Luís Manuel Meruje Ferreira (co-advisor in
Research Center)**

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-NC

<https://creativecommons.org/licenses/by-nc/4.0/>

Acknowledgements

This work was only possible due to the support of many people, to whom I would like to express my gratitude.

First and foremost, I would like to thank my supervisors, Professor Fábio Coelho and Professor João Paulo, for their guidance and insight throughout the development of this dissertation. I would also like to express my gratitude to Luís Meruje for his continuous support and help on all aspects of this work. Without their support, it would not have been possible.

To all my colleagues at High-Assurance Software Laboratory (HASLab) who provided much-needed breaks, laughter, and encouragement.

To my family for their support and understanding throughout this academic journey.

To my girlfriend for always being by my side regardless of distance, and for always believing in me.

Finally, I would like to thank INESC TEC that co-funded this work through Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56 (10113/BII-E_B4/2023)

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2023

Assinado por: **Pedro Miguel de Leal Meireles Pereira**
Num. de Identificação: 14673529
Data: 2023.10.31 17:02:35+00'00'

Pedro Miguel de Leal Meireles Pereira

Abstract

Internet of Things (IoT) systems generate massive amounts of time series data that need to be stored for historical analysis. As a result, **Database Management Systems (DBMSs)** for these scenarios have particular requirements in their ability to ingest large amounts of data and to optimise aggregation, filtering and time-ranged queries over this data, which are essential for historical analysis.

Through the use of Fog Computing, combining both Edge and Cloud layers, it is possible to achieve reduced latency and increased scalability, privacy and connectivity through the Edge, while still benefiting from the enhanced computing and storage power of the Cloud. This has led to the development of Fog **DBMSs**.

Database benchmarking allows standardising performance assessment and comparison of different solutions. However, current time series database benchmarking tools are not designed for multi-layer architectures, such as the ones used by Edge-Cloud hybrid **DBMSs**.

This thesis proposes MulletBench, a benchmarking tool that is able to evaluate the internal load-balancing capabilities of a multi-layer **Time Series Database Management System (TSDBMS)**. This is achieved by integrating automated deployment features, per-node and per-layer performance and system resource metrics, allowing for a more detailed analysis of the **SUTs**' performance than previously possible.

The performance of InfluxDB and IoTDB is evaluated using the developed tool, comparing their performance in multiple workloads and deployment scenarios. Results show that the Edge layer can be used to improve performance by distributing the workload over multiple layers and performing downsampling at the Edge layer, increasing overall throughput and reducing latency at the Cloud. These conclusions are enabled by MulletBench's novel features, and would not have been possible with previously existing solutions.

Keywords Benchmark, Internet of Things, Fog, Time Series, Databases

Resumo

Os Sistemas de Internet das Coisas (IdC) geram enormes quantidades de dados de séries temporais que precisam de ser armazenados para análise histórica. Consequentemente, os sistemas de gestão de bases de dados (SGBD) para estes cenários têm requisitos particulares não só para a sua capacidade de ingerir grandes quantidades de dados mas também para a sua capacidade de otimizar queries de agregação ou filtragem e em intervalos de tempo sobre estes dados, que são essenciais para análise histórica.

Através do uso de computação em *Fog*, combinando as camadas de *Edge* e *Cloud*, é possível conseguir latência reduzida e maior escalabilidade, privacidade e conectividade através da *Edge*, beneficiando simultaneamente do superior poder de computação e armazenamento da *Cloud*. Isto levou ao desenvolvimento de sistemas de gestão de bases de dados *Fog*.

O *Benchmarking* de bases de dados permite a standardização do método de avaliação de desempenho e a comparabilidade de resultados de diferentes soluções. Contudo, as atuais ferramentas de benchmarking para bases de dados de séries temporais não estão desenhadas para arquiteturas multi-camada, tais como as utilizadas por SGBDs híbridos *Edge-Cloud*.

A tese propõe o MulletBench, uma ferramenta de benchmarking capaz de avaliar as capacidades de balanceamento interno de carga de um SGBD de séries temporais multi-camada. Esta ferramenta alcança-o através da integração de funcionalidades de automação de *deployment* e métricas de desempenho e de recursos de sistema por nó e por camada, permitindo uma análise mais detalhada do desempenho dos sistemas testados do que era previamente possível.

O desempenho das bases de dados InfluxDB e IoTDB é avaliado usando a ferramenta desenvolvida, comparando o seu desempenho com múltiplas cargas e em múltiplos cenários. Os resultados demonstram que a camada *Edge* pode ser usada para melhorar o desempenho destas, através da distribuição da carga por múltiplas camadas e realização de *downsampling* na camada *Edge*, aumentando o débito total e reduzindo a latência na *Cloud*. Estas conclusões são possíveis graças às funcionalidades inovadoras do MulletBench, e não seriam possíveis com soluções existentes anteriormente.

Palavras-chave *Benchmark*, Internet das Coisas, Fog, Séries Temporais, Bases de Dados

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Objectives and Contributions	3
1.3	Results	3
1.4	Document Structure	4
2	Background and Related Work	5
2.1	IoT and Edge Computing	5
2.2	Time Series Databases	6
2.2.1	Cloud-first Time Series Databases	7
2.2.2	Multi-layer Time Series Databases	8
2.3	Benchmarking	10
2.4	Survey on TSDB Benchmarking Systems	13
2.4.1	Workloads	14
2.4.2	Data	16
2.4.3	Metrics	18
2.4.4	Orchestration	20
2.4.5	Discussion	21
3	MulletBench	23
3.1	Design Principles	23
3.2	Architecture	26
3.2.1	Orchestrator	26
3.2.2	Client	28
3.3	Implementation	31
3.3.1	Orchestrator	31

3.3.2	Client	33
3.3.3	Configurations	34
3.4	Summary	35
4	Evaluation and Results	37
4.1	Methodology	38
4.1.1	Experimental Setup	38
4.1.2	Workloads	38
4.1.3	Data	40
4.2	Feature Validation	40
4.2.1	Control Test	40
4.2.2	Limited Rate	41
4.2.3	Limited Network	43
4.2.4	Limited Resources	45
4.3	Cloud Scenario	49
4.3.1	Insertion Workload	49
4.3.2	Query Workload	52
4.3.3	Mixed Workload	55
4.4	2 Edge + Cloud Scenario	61
4.4.1	Insertion Workload	61
4.4.2	Query Workload	70
4.4.3	Mixed Workload	72
4.5	4 Edge + Cloud Scenario	76
4.5.1	Insertion Workload	76
4.5.2	Query Workload	79
4.5.3	Mixed Workload	80
4.6	Summary	82
5	Conclusions and future work	85
5.1	Future work	86
A	Configuration Options	91
A.1	Deployment Orchestrator Options	91
A.1.1	Shared edgeservers and cloudservers variables	91

A.1.2	Edgeservers variables	92
A.1.3	benchmark_clients	93
A.2	Execution Orchestrator Options	94
A.3	Client Options	95
B	Test Configurations	97
B.1	Cloud Scenario	97
B.1.1	Insertion Workload	97
B.1.2	Query Workload	98
B.1.3	Mixed Workload	98
B.2	2 Edge + Cloud Scenario	99
B.2.1	Insertion Workload	99
B.2.2	Query Workload	100
B.2.3	Mixed Workload	100
B.3	4 Edge + Cloud Scenario	101
B.3.1	Insertion Workload	101
B.3.2	Query Workload	102
B.3.3	Mixed Workload	102

List of Figures

1	MulletBench Overview	26
2	Deployment Flow Diagram	27
3	Execution Flow Diagram	28
4	Timestamp Injection Diagram	29
5	Benchmark Client structure	33
6	Throughput comparison for control test	41
7	Throughput comparison for limited client rate test targeting InfluxDB	42
8	Throughput comparison for limited client rate test targeting IoTDB	43
9	Throughput for limited network test targeting InfluxDB	44
10	Throughput for limited network test targeting IoTDB	45
11	Throughput for limited resources test targeting InfluxDB	46
12	CPU usage for limited resources test targeting InfluxDB	46
13	Memory usage for limited resources test targeting InfluxDB	47
14	Throughput for limited resources test targeting IoTDB	47
15	Insertion latency for limited resources test targeting IoTDB	48
16	CPU usage for limited resources test targeting IoTDB	48
17	Memory usage for limited resources test targeting IoTDB	49
18	Throughput comparison for insertion workload in Cloud-only scenario	50
19	Disk IO usage for insertion workload targeting InfluxDB in Cloud-only scenario	51
20	Disk IO usage for insertion workload targeting IoTDB in Cloud-only scenario	51
21	Memory usage for insertion workload targeting IoTDB in Cloud-only scenario	52
22	Query latency bar chart for query workload in Cloud-only scenario	53
23	Query scatter for query workload targeting InfluxDB in Cloud-only scenario	54
24	Query scatter for query workload targeting IoTDB in Cloud-only scenario	55

25	Throughput comparison for mixed workload in Cloud-only scenario	56
26	Query latency bar chart for mixed workload in Cloud-only scenario	57
27	Query scatter plot for mixed workload targeting InfluxDB in Cloud-only scenario	58
28	CPU usage for mixed workload targeting InfluxDB in Cloud-only scenario	58
29	Query scatter plot for mixed workload targeting IoTDB in Cloud-only scenario	59
30	CPU usage for mixed workload targeting IoTDB in Cloud-only scenario	60
31	Disk IO usage for mixed workload targeting IoTDB in Cloud-only scenario	60
32	Throughput comparison for insertion workload in 2 Edge + Cloud scenario	62
33	Throughput plot for insertion workload targeting InfluxDB with no downsampling in 2 Edge + Cloud scenario	63
34	Edge node disk IO usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	64
35	Edge node memory usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	64
36	Latency scatter plot for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	65
37	Edge node network usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	65
38	Cloud node memory usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	66
39	Edge node memory usage for insertion workload targeting InfluxDB with 2 second down- sampling in 2 Edge + Cloud scenario	67
40	Edge node disk IO usage for insertion workload targeting InfluxDB with 2 second down- sampling in 2 Edge + Cloud scenario	67
41	Edge node disk IO usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario	68
42	Edge node memory usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario	68
43	Edge node network usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario	69
44	Cloud node network usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario	69
45	Query latency for query workload in 2 Edge + Cloud scenario	71
46	Query latency bar chart for mixed workload in 2 Edge + Cloud scenario	73
47	Throughput comparison for mixed workload in 2 Edge + Cloud scenario	73
48	Query latency bar chart for mixed workload targeting InfluxDB in 2 Edge + Cloud scenario	74

49	Layer query latency bar chart for mixed workload targeting IoTDB in 2 Edge + Cloud scenario	75
50	Throughput comparison for insertion workload in 4 Edge + Cloud scenario	76
51	Insertion latency scatter plot for insertion workload targeting InfluxDB with 2 second downsampling in 4 Edge + Cloud scenario	77
52	Edge node disk IO usage for insertion workload targeting InfluxDB with 2 second downsampling in 4 Edge + Cloud scenario	78
53	Query latency bar chart for query workload in 4 Edge + Cloud scenario	79
54	Query latency bar chart for mixed workload in 4 Edge + Cloud scenario	81
55	Throughput comparison for mixed workload in 4 Edge + Cloud scenario	81

List of Tables

1	Workload characteristics presented by surveyed benchmarks	14
2	Data characteristics presented by surveyed benchmarks	16
3	Metrics presented by surveyed benchmarks	18
4	Orchestration presented by surveyed benchmarks	20
5	Raspberry Pi 3 B+ resource limiting profile	32
6	Raspberry Pi 4 resource limiting profile	32
7	Specifications of database machines	39
8	Specifications of Client and Orchestrator machines	39
9	Throughput table for insertion workload in 4 Edge + Cloud scenario	76
10	Shared edgeservers and cloudservers group variables for Deployment Orchestrator	91
11	Network simulation variables for Deployment Orchestrator	91
12	Edgeservers specific group variables for Deployment Orchestrator	92
13	InfluxDB variables for Deployment Orchestrator	92
14	Apache IoTDB variables for Deployment Orchestrator	92
15	benchmark_clients group variables for Deployment Orchestrator	93
16	Deployment variables for Deployment Orchestrator	94
17	Execution Orchestrator options	94
18	Execution Orchestrator client options	94
19	Execution Orchestrator stage options	95
20	Execution Orchestrator node options	95
21	Client options	95
22	Client query options	96
23	Client insertion options	96
24	Client InfluxDB options	96

25	Client IoTDB options	96
26	Client configuration for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	97
27	Client configuration for insertion workload targeting IoTDB in 2 Edge + Cloud scenario .	97
28	Client configuration for query workload Cloud-only scenario	98
29	Pre-population Client configuration for mixed workload Cloud-only scenario	98
30	Insert Client configuration for mixed workload Cloud-only scenario	98
31	Query Client configuration for mixed workload Cloud-only scenario	99
32	Client configuration for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario	99
33	Client configuration for insertion workload targeting IoTDB in 2 Edge + Cloud scenario .	99
34	Query Client configuration for query workload in 2 Edge + Cloud scenario	100
35	Pre-population Client configuration for mixed workload 2 Edge + Cloud scenario	100
36	Insert Client configuration for mixed workload in 2 Edge + Cloud scenario	100
37	Query Client configuration for mixed workload in 2 Edge + Cloud scenario	101
38	Cloud query Client configuration for mixed workload with downsampling in 2 Edge + Cloud scenario	101
39	Client configuration for insertion workload targeting InfluxDB in 4 Edge + Cloud scenario	101
40	Client configuration for insertion workload targeting IoTDB in 4 Edge + Cloud scenario .	102
41	Query Client configuration for query workload in 4 Edge + Cloud scenario	102
42	Pre-population Client configuration for mixed workload 4 Edge + Cloud scenario	102
43	Insert Client configuration for mixed workload in 4 Edge + Cloud scenario	103
44	Query Client configuration for mixed workload in 4 Edge + Cloud scenario	103

Acronyms

DBMS Database Management System.

GAN Generative Adversarial Network.

IoT Internet of Things.

LSM Log-Structured Merge-Tree.

PCP Performance Co-Pilot.

PSoTSDBs Performance Study of Time Series Databases.

SUT system under test.

TSDBMS Time Series Database Management System.

TSM Time-Structured Merge Tree.

WAL Write Ahead Log.

YCSB Yahoo! Cloud Serving Benchmark.

Chapter 1

Introduction

The **Internet of Things (IoT)** concept consists of a network of devices referred to as *Things*, that is, any device that is either a sensor or an actuator in an **IoT** application. Sensors are commonly connected through a *sensor network*, allowing for their collaboration and enabling the collection, processing, analysis and dissemination of the data they produce.

IoT systems have contributed to an exponential increase in the number of connected devices and generated data, resulting in a projection that by 2025 there will be 175 Zettabytes of data, of which 79.4 Zettabytes are to be created by IoT devices (Wang and Wang [2020]). The need for storing such a high volume of data, as well as performing historical time-ranged queries, among other reasons, has led to a rise in popularity of **Time Series Database Management Systems (TSDBMSs)** (dbe [2022b]), which are optimised for high data ingestion rates and timestamped data, suiting **IoT** scenarios in which data is generated by sensors.

A possible solution for storing data generated by **IoT** devices is doing it at the Edge, which consists of computing and storage resources located at the Internet's edge, close to mobile devices and sensors (Satyanarayanan [2017]). This leads to benefits such as lower latency, reduced network strain and improved data privacy, but this solution also suffers from drawbacks, with limited hardware and heterogeneous resource capabilities. Combining both Edge and Cloud allows for the balance of data storage and processing between both layers. This solution also brings additional benefits, enabling the choice of whether data should be kept temporarily or permanently at each layer, and where to perform data processing tasks. This makes use of a concept called Fog or Edge-Cloud hybrid Computing, leveraging both the Edge and Cloud layers, to achieve reduced latency and increased scalability, privacy, and connectivity through the Edge, while still benefiting from the enhanced computing and storage capabilities of the Cloud.

The most common sensor network architecture follows the *sink node* approach, in which sensor-generated data is offloaded to a local, more powerful node called the sink node, which in turn communicates with the Cloud layer, achieving some of the aforementioned benefits (Perera et al. [2013]). There

are several database solutions that make use of both the Edge and Cloud layers, each using different mechanisms to achieve this and consequently having different performance advantages and drawbacks.

Since different database solutions have differing implementations and focus on different aspects of their performance, they show significant performance differences. This results in a need for evaluation tools that provide metrics which allow for informed decision-making, either by database researchers, vendors, or customers. As a result, the establishment of industry-standard benchmarks can offer a means for standardised performance assessment and comparison of competing ideas and implementations (Seltzer et al. [1999]).

While there are benchmarks specifically targetting **TSDBMSs**, there are no current solutions that target multi-layer **TSDBMSs** for sensor networks which distribute storage between the Edge and Cloud layers.

1.1 Problem Statement

There are no current benchmark implementations capable of adequately evaluating **hybrid Edge-Cloud database solutions** for time series data generated in sink node sensor networks.

Benchmarking databases distributed over multiple layers is a complex task, due to the high flexibility of architectures, as well as the need for metrics that can accurately represent not only the database's performance as a whole but also the performance of each layer or even node.

While there are benchmarks that evaluate time series **DBMSs**, none of them are sensitive to important aspects of Edge-Cloud distributed **DBMSs**, such as how the latency of operations is affected by which node replies to the request and how these databases scale with different numbers of both Edge and Cloud nodes, as well as with different distributions of nodes between layers. Metrics that represent these aspects, such as per-node ingestion throughput and query latency, are important to understand how an Edge-enabled sensor network **Database Management System (DBMS)** behaves in real-world scenarios and with real-world workloads, as they enable the analysis of the database's internal load balancing and, consequently, a more in-depth understanding of its behaviour and performance.

Furthermore, features that automate the deployment of database nodes, improving not only the ease of use of the benchmark, but also the reproducibility of its results are uncommon, with no solution offering it for multiple layers. Given that the deployment of a database cluster is a complex task, requiring the installation and configuration of multiple nodes, automation is very useful and eases the evaluation of

different database node configurations.

1.2 Objectives and Contributions

Since a lack of existing benchmarks for this domain has been identified, this thesis presents a benchmarking system capable of providing metrics that accurately measure the performance of a time series **DBMS** deployed in an Edge-Cloud distributed architecture. It focuses on addressing the highlighted features that have not been observed in any existing solution, such as **per-node metrics** and **orchestration features**.

To achieve this, this thesis provides the following contributions:

- An in-depth study and survey of currently existing **TSDBMS** benchmarks. This study includes a new taxonomy of the existing benchmarks, analysing their workloads, data, metrics and orchestration features and concludes that existing solutions lack metrics and orchestration features suitable for the evaluation of Edge-Cloud hybrid **TSDBMSs**.
- A new **TSDBMS** benchmarking tool, named MulletBench, with novel features such as per-node and per-layer metrics, automatic multi-layer database deployment and orchestration of distributed benchmark clients.
- A validation of MulletBench and evaluation of InfluxDB [InfluxData \[2013\]](#) and Apache IoTDB [Wang et al. \[2020\]](#). This evaluation includes a comparison of the targeted databases in multiple workloads and deployment scenarios, and its results show that IoTDB generally presents superior performance to InfluxDB, and that Edge-Cloud hybrid deployments enable significant performance improvements when compared to single Cloud node deployments. Additionally, a flaw in both systems' replication mechanisms is identified when these are subjected to high throughput workloads.

1.3 Results

The development of this dissertation resulted in an open-source **TSDBMS** benchmark providing multiple novel features ¹.

Issues were created in the respective GitHub repositories of the targeted databases documenting the

¹ MulletBench: <https://github.com/pedropereira98/MulletBench>

identified replication flaws ^{2 3 4}.

1.4 Document Structure

This document is organised into four chapters. Chapter 2 gives a more in-depth look at the subjects of **IoT**, **Time Series Database Management System (TSDBMS)** and **DBMS** benchmarking, as well as a survey looking into the existing **TSDBMS** benchmarking solutions. Chapter 3 presents MulletBench, including a specification of its overall features, an architectural overview of the developed system and implementation details. Chapter 4 goes into the evaluation and results, presenting the used methodology, the results of the performed feature validation and an evaluation of multiple databases using several workloads and scenarios. This allowed for comparisons between not only these databases but also their different deployment options. Finally, Chapter 5 presents the final conclusions that could be drawn from the developed work, as well as how it could be further expanded.

² InfluxDB issue: <https://github.com/influxdata/influxdb/issues/24434>

³ IoTDB replication issue: <https://github.com/apache/iotdb/issues/11428>

⁴ IoTDB unsequenced issue: <https://github.com/apache/iotdb/issues/11441>

Chapter 2

Background and Related Work

This chapter introduces relevant background concerning **IoT** and Edge Computing, going into its characteristics as well as the data handled by it, **TSDDBMSs**, what they achieve and how, and **DBMS** benchmarking. Afterwards, a survey analysing the existing **TSDDBMS** benchmarking solutions is presented, from which came the conclusion that there is an opportunity for contributions.

2.1 IoT and Edge Computing

IoT systems can have several applications with some of the most common being smart systems, such as Smart Agriculture, Smart Cities, Smart Healthcare and Smart Industry (also known as Industrial IoT) (Kamienksi et al. [2019]; Wollschlaeger et al. [2017]). The **IoT** attempts to increase the automation processes present in these domains and ultimately achieve higher efficiency.

At the core of the **IoT** are its sensors, which have limited computing power and may have limited energy. These sensors are potentially heterogeneous and communicate among them using wired or wireless technologies, making up sensor networks (Perera et al. [2013]). Although the concept of a sensor network has existed since before the emergence of the **IoT**, it was previously only used to achieve specific purposes in limited domains. The **IoT** can therefore be seen as a general-purpose extension of sensor networks, focusing on the interactions between the users and the physical environment, rather than on sensing and reporting low-level data (Firner et al. [2011]).

As **IoT** devices have limited computing power, they often need to offload tasks to more powerful Cloud nodes, allowing for benefits such as increased computing and storage capacity. New concepts such as Edge and Fog computing attempt to provide better latency and connectivity when compared to systems that offload all data to the Cloud.

Edge computing also enhances the management, storage, and processing power of data generated by connected devices by using small compute and storage nodes at the edge of the network, and as close

as one hop to the **IoT** devices (Reale [2019]), providing better latency than Cloud offloading systems. Fog or Edge-Cloud hybrid computing, on the other hand, consists of deploying nodes located closer to end devices as a way to complement the Cloud, aggregating, processing and/or pre-processing data, essentially extending the Cloud's capabilities and allowing for a balance between processing power and latency (Yousefpour et al. [2019]).

The most common architecture used in sensor networks for Fog computing is the *sink node* sensor network, in which the network's sensor nodes offload their collected data to a local, more powerful node, called the sink node or gateway (Perera et al. [2013]). This central point of the sensor network is often responsible for the pre-processing of the data it receives and stores, before offloading it to a Cloud **TSDBMS** (Buevich et al. [2013]). This architecture is one possible way of achieving a multi-layer data storage and processing solution, making use of both the Edge and Cloud layers, and will for that reason be the one on which this work focuses. Another possible approach is *in-network processing*, in which each sensor node stores its own data locally and queries are disseminated through the network and processed by the sensor nodes. The queried data is sent back to parent nodes where it is combined and once more sent to their parent nodes until the final data reaches the gateway (Diallo et al. [2013]).

2.2 Time Series Databases

The data being generated and used in **IoT** applications needs to be properly understood for it to be correctly stored. It usually consists of time series data, which has a format associating a timestamp for a recorded instant in time with one or multiple values (also sometimes called fields), as well as optionally having tags which give additional information about the measurement, also known as metadata, and are used to more easily categorise the data. **IoT** data can often be categorised as Big Data (Cai et al. [2016]), which is generally considered to have the following characteristics (Han et al. [2017]):

- **Volume:** **IoT** devices generate massive amounts of data, estimated to reach 79.4 Zettabytes by 2025 (Wang and Wang [2020]).
- **Velocity:** **IoT** data is generated at a high rate, with a single device being able to generate hundreds of data points every 20ms, and big deployments reaching the billions of data points generated daily (Hall [2020]). In time series data, this specifically relates to the timestamp interval between samples and the sampling frequency of the sensors.
- **Variety:** As **IoT** devices can be very varied, ranging from environmental sensors to GPSs or cars,

the data and data formats they generate can also vary widely ([Sasaki \[2021\]](#)).

- **Veracity:** Since **IoT** data is often generated by sensors and these are not 100% accurate, this data becomes inherently imprecise, containing noise or errors.

In addition, **IoT** data also has some unique characteristics ([Cai et al. \[2016\]](#)):

- **Multisource:** **IoT** applications acquire data from different sensors with different characteristics and in parallel.
- **Low-Level:** **IoT** data from sensors is low-level and has weak semantics, meaning processing is needed for valuable information to be extracted from it.

All of these characteristics present in **IoT** data introduce challenges for systems that aim to store, process or analyse it. **DBMSs** for these applications need to be able to store massive amounts of data at a very high rate.

On top of being able to withstand the ingestion of this data, these **DBMSs** also need to support and, ideally, optimise certain querying operations, as **IoT** applications often require either real-time or historical analysis. These applications make heavy use of aggregation operations over specific time ranges and hardly ever require data granularity at the individual data point level, allowing for varying query time resolutions or downsampling of data. Monitoring online sensors, incident analysis or long-term trend analysis are examples of common query scenarios with different data freshness requirements and ranging very different time spans ([Karlstetter et al. \[2022\]](#)).

2.2.1 Cloud-first Time Series Databases

The need for these data storage characteristics gave way to the rise in popularity of **TSDBMSs**. The simultaneous emergence of both **IoT** and Cloud Computing resulted in the majority of **TSDBMS** solutions being meant for the Cloud layer, storing all the generated data. These are currently the most popular solutions on the market, with the top 5 **TSDBMSs**, according to DB-Engines Rankings ([dbe \[2022a\]](#)), all being cloud-first databases.

InfluxDB ([InfluxData \[2013\]](#)) is currently ranked as the number one most popular **TSDBMS** and features a storage engine that uses a **Write Ahead Log (WAL)**, cache, a **Time-Structured Merge Tree (TSM)**, as well as a time series index to ensure no data loss, a high ingestion rate and efficient data retrieval. To improve efficiency, the **TSM** only stores deltas between values in a series, while the time series index ensures queries remain fast as data cardinality grows.

Another cloud **TSDBMS** solution, BTrDB [Andersen and Culler \[2016\]](#), achieves relatively fixed response times for analytical queries, regardless of the underlying data size, by using a time-partitioning copy-on-write version-annotated k-ary tree. Its base data points are stored in the leaves, with the depth of the tree defining the interval between data points. Since each internal node holds summaries of the subtrees below it, consisting of statistical aggregates that are computed as nodes are updated, the tree only needs to be traversed to the depth corresponding to the desired resolution for the retrieval of these statistical records. A warm query throughput test (with a preheated cache), which simulates typical tailing-analytics workloads, reached 119 million readings per second. BTrDB also supports out-of-order insertions, which tests have shown to not significantly affect the measured throughput of 53 million inserted values per second, using a four-node cluster.

2.2.2 Multi-layer Time Series Databases

Storage and more complex data analysis have mostly been reserved for the Cloud layer, with Edge devices being, up to this point, mainly used for data processing, such as compression, filtering and aggregation, and no data being stored at this layer. However, as the needs of new **IoT** applications have grown, a demand for increased privacy-sensitivity, lower latencies and enhanced ability to mask connection losses that is not possible to achieve with thus far existing architectures has led to a growth in the need for storage at the Edge layer ([Yousefpour et al. \[2019\]](#)). As a result, multi-layer **TSDBMSs**, supporting both Edge and Cloud deployment, have also become more common. There are 4 main reasons to store data either completely or partially at the edge level ([Satyanarayanan \[2017\]](#)):

- **Latency:** since Edge nodes are located closer to the users, there is a reduction in latency when compared to Cloud databases, allowing them to be able to fulfil the ever-increasing latency requirements of newer applications.
- **Scalability:** by storing data and applying analytics at the Edge level, the amount of data sent to the Cloud can be reduced, allowing for a reduction of its bandwidth usage, and consequently a potential increase in scalability.
- **Privacy:** the use of Edge nodes allows for the storing of sensitive data in a more geographically restricted area, when compared to Cloud storage. Additionally, the processing of data at the Edge enables the denaturing of sensitive data, such as the blurring of faces or coarse aggregation of sensor readings, before it is sent to the Cloud.

- **Connectivity:** storing data at the Edge means that temporary connection losses from the gateway node to the Cloud, either by an outage in the cloud service or the networking infrastructure between them, can be masked.

Apache IoTDB [Wang et al. \[2020\]](#), Storacle [Cejka et al. \[2015\]](#), VergeDB [Paparrizos et al. \[2021\]](#) and Respawn [Buevich et al. \[2013\]](#) are examples of such multi-layer solutions. Additionally, **TSDBMSs** originally intended for Cloud deployment, such as InfluxDB ([InfluxData \[2013\]](#)), have introduced features that enable Edge-Cloud hybrid database clusters.

Apache IoTDB is currently one of the most popular multi-layer **TSDBMSs** according to DB-Engines Rankings ([dbe \[2022a\]](#)). It stores data using a custom file format, TsFile, which organises time series data into chunks and pages, containing not only the raw data but also pre-calculated statistical summary data such as minimum and maximum time and values, counts and averages, allowing for significant acceleration of filtration and aggregation operations, with internal tests measuring latencies in the tens of milliseconds for aggregation queries on billions of data points. It also achieves high write throughput performance, with up to 30 million data points per second on a single node according to internal tests, by featuring in-memory buffers with a **WAL**, as well as vectorized write interfaces for more efficient batch data ingestion. Additional features are also implemented to efficiently handle out-of-order data ingestion.

Storacle is another multi-layer solution, which through the integration of serialisation features as well as managing the lifetime of data at each layer achieves significant results in space reduction and fast response times while being lightweight. By storing only more recent data at the Edge and storing the full data at the Cloud, this solution deals with the limited resources available at the Edge while still enabling access to historical data for analytics.

VergeDB focuses on the Edge layer, and in particular on the compression of **IoT** data for complex analytical tasks and machine learning. By being an Edge-based database, sitting between the device layer and Cloud layer, it aims to alleviate the strain of centralized solutions that can have deteriorating performance as the volume of data increases by being able to rapidly ingest sensor data while optimising for compression, aggregation and filtering. To achieve that, it uses three different compression methods: precision-aware data compression, subsampling compression and similarly-preserving compression. It can execute queries over either the compressed or uncompressed data.

Respawn makes use of a dispatcher front-end that is responsible for directing queries between the Cloud and the Edge through the use of a bloom filter-based caching layer. On Edge-node gateway devices, a multi-resolution storage solution is responsible for down-sampling incoming data and storing copies of the data at differing resolutions, allowing for significant improvements in the performance of range-based

queries. To reduce access latency, the Cloud node serves data summaries collected from the Edge nodes. For this, Respawn makes use of two different types of data migration: periodic migration and proactive migration.

Finally, InfluxDB currently supports deployment at any layer, from sensor to Cloud. It features Edge-to-Cloud data replication at the bucket level, which is a subdivision within a database into which time series data is organised. Although this results in the replication of all data in a given bucket, which is not ideal, this can be worked around by explicitly downsampling data at the Edge node to a bucket that is then replicated to the Cloud node. This means that data can be explicitly downsampled, filtered and pre-processed in whatever way the user intends before being replicated to the Cloud.

2.3 Benchmarking

Benchmarking is a tool that aims to help understand why systems perform as they do and enable the meaningful comparison of competing ideas, designs and implementations.

Benchmarks generally fall into one of three categories: micro benchmarks, which aim to evaluate an individual system component or a specific behaviour; end-to-end benchmarks, designed to evaluate entire systems using application scenarios; and benchmark suites, which combine micro and/or end-to-end benchmarks, aiming to provide a broader benchmarking solution ([Han et al. \[2017\]](#)).

In order for a benchmark to be useful, it should be ([Chen et al. \[2012\]](#)):

- **Representative:** the benchmark should measure performance by subjecting the target systems to typical behaviours and using metrics that are relevant to real-life applications.
- **Relevant:** the benchmark should focus on measuring technologies that are prominent and in need of performance evaluation.
- **Portable:** the benchmark should be able to evaluate different competing systems fairly, being executed on different software systems and architectures.
- **Scalable:** the benchmark should be able to increase or decrease its load, in order to be able to evaluate the performance of systems within different scales while guaranteeing the relevance of its workload.
- **Verifiable:** the benchmarks should produce results that are repeatable and can be independently verified. This characteristic is also commonly referred to as reproducibility.

- **Simple:** the benchmark should be able to be broken down into understandable conceptual elements and abstract away details that do not affect performance.

A benchmark is defined by its **workloads**, **data**, and **metrics**. Its workload and data should simulate real-world scenarios for the results to be representative of real usage, and its metrics should provide meaningful results.

Workloads Workloads are defined by the operations they produce and the patterns in which they are submitted. A submission policy defines a workload execution's input data and arrival pattern, and can be categorised into the following types ([Han et al. \[2017\]](#)):

- **Pre-Specified:** input data, arrival rates, and sequences are pre-specified before their execution.
 - **Synthetic Realistic:** attempt to emulate workloads generated by real systems by generating traces describing the observed mix of operations;
 - **Synthetic Unrealistic:** pre-determine the proportions of operations used, not emulating real systems. These workloads can be useful to evaluate specific aspects of a database's behaviour, e.g., to test how a database would behave when subjected to a sudden spike in ingestion requests.
- **Parameter-Controlled:** allows users to control workload execution by specifying parameters. **Yahoo! Cloud Serving Benchmark (YCSB)** [Cooper et al. \[2010\]](#), for instance, allows the size and contents of input data and workloads to be customized, as well as the workload's submission rate, including the number of clients used.
- **Trace-Driven:** replays a trace of real-world workloads.

Workloads can often be executed with a concurrent multi-client submission approach as it better emulates the loads which databases are subjected to in real-world systems. The throughput of each individual client can be varied in different ways:

- **Maximum:** submits the maximum throughput possible.
- **Static:** submits with a constant rate.
- **Dynamic:** varies the throughput between two values.

Data The data used by workloads can also be categorised in several ways, such as the following:

- **Real Datasets:** since they are obtained from real-world data sources, real datasets have the benefit of having guaranteed veracity, but also have the drawbacks of not being precisely controlled, meaning their volume and timestamp frequency cannot be altered, e.g., for adjusting scale.
- **Generated using synthetic distributions:** these datasets are generated using synthetic distributions, having the benefit of being more easily controlled, allowing for adjustments in their velocity and even allowing for parallelism of data, but since they are synthetic, there is no guarantee of their veracity. These generators can either use random distributions or try to use studied distributions, attempting to generate data that is closer to real-world data.
- **Generated based on real-world data:** data generators that capture and preserve important characteristics of real data, allow for the creation of realistic datasets of variable scale. This is considered to be the best technique currently available to efficiently generate realistic workload inputs with controllable volume and velocity. ([Han et al. \[2017\]](#))

Metrics Performance metrics typically consist of either response time or throughput metrics. Response time or latency metrics measure the time interval between the submission and completion of operations in a workload, and are usually presented as a mean, either of all operations or for each operation type present in the workload. Additionally, benchmarks can also present statistical latency metrics, such as minimum and maximum values, percentile values or standard deviation values, allowing for a more detailed analysis of a database's performance than is possible using only the mean value. Throughput metrics, on the other hand, measure how many operations the **system under test (SUT)** can execute or how much volume of data it can process in a given time frame.

Other performance metrics include reliability metrics, availability metrics, price-performance metrics and energy consumption metrics, whose measurement is less straightforward. Price-performance metrics, for instance, are used by benchmarks such as TPC-C [[TPC](#)] and TPC-DS [[TPC](#)] and evaluate the relation between the price of a **SUT** to its performance. However, these benchmarks only measure the performance aspect, while the price needs to be calculated by the user.

In addition, benchmarks can also feature resource monitoring metrics, representing the resource consumption of the **SUT**, such as CPU, memory, network and disk usage, enabling the analysis of the performance metrics concerning resource consumption, which can be used for finding bottlenecks in the **SUT**.

Orchestration Benchmarks can include orchestration features as a way to simplify their usage as well as attempt to uniformise the way in which they are used across different systems and consequently have more comparable results. Some solutions utilise remote clients to more accurately simulate concurrent workloads. Another possibility is the deployment of additional database nodes to evaluate a **DBMS**'s horizontal scaling performance. However, these features are often neglected, making the accurate evaluation of these aspects more difficult.

2.4 Survey on TSDB Benchmarking Systems

For this section, a survey was conducted as a means to identify what existing Time Series **DBMS** benchmarking solutions achieve and in what way. This included an analysis of the available specifications and documentations, as well as the implementations of said benchmarks when available. The benchmarks analysed were SciTS [Mostafa et al. \[2022\]](#), **Performance Study of Time Series Databases (PSoTSDBs)** [Shah et al. \[2022\]](#), TS-Benchmark [Hao et al. \[2021\]](#), IoTDataBench [Zhu et al. \[2021\]](#), SmartBench [Gupta et al. \[2020\]](#), IoTDB-Benchmark [Liu and Yuan \[2019\]](#), TSDBBench [Bader \[2016\]](#), TSBS [Timescale \[2022\]](#), TPCx-IoT [\[TPC\] Poess et al. \[2018\]](#) and TSM-Bench [Khelifati et al. \[2023\]](#).

Most surveyed benchmarks have published papers and publicly available implementations with exceptions being as follows: **PSoTSDBs** shows a performance study with no available implementation; IoTDataBench is an implemented benchmarking solution with a published paper but no publicly available implementation; TSBS is a publicly available benchmark with no published paper; TPCx-IoT is a publicly available benchmarking specification and implementation.

The available information regarding all the aforementioned benchmarking systems is not documented in a systematic way. Using a consistent set of categories and sub-categories, all the necessary information was compiled, categorising all relevant data in order to make its presentation understandable, intuitive, as well as relevant.

The relevant categories were identified as such: workload characteristics, data characteristics, performance metrics, resource monitoring metrics and orchestration characteristics, as described in subsection [2.3](#).

2.4.1 Workloads

Table 1: Workload characteristics presented by surveyed benchmarks

	Origin			Supported			Client Rate	
	Realistic	Synthetic	Custom	Ingestion	Querying	Mixed	Maximum	Static
SciTS Mostafa et al. [2022]		•	•	•	•		•	
PSoTSDBs Shah et al. [2022]		•		•	•		•	
TS-Benchmark Hao et al. [2021]		•		•	•			•
IoTDataBench Zhu et al. [2021]		•				•		•
SmartBench Gupta et al. [2020]	•			•	•	•	•	
IoTDB-Benchmark Liu and Yuan [2019]	•			•	•	•	•	
TSDBBench Bader [2016]		•	•	•	•	•		•
TSBS Timescale [2022]	•			•	•		•	•
TPCx-IoT [TPC]		•				•		•
TSM-Bench Khelifati et al. [2023]	•			•	•	•	•	•

* using custom workload

Table 1 goes into the sub-categories of the workload origin, supported operations in presented workloads and client rate.

For **workload origin**, the surveyed benchmarks were analysed to be either **synthetic realistic**, **synthetic unrealistic**, and **custom**, which allows for the user to customize the parameters for workload execution, such as which operations to perform, the way they are combined, among other settings. For a **synthetic workload** to be considered realistic, not only does the combination of operations used need to be realistic but also the operations themselves. The following featured operations were considered to determine whether the presented synthetic workloads were realistic: the presence of a mixed workload, out-of-order insertions, batch insertions, time-ranged aggregation queries, downsampling queries, time-ranged queries and out-of-range or filtering queries. Mixed workloads are needed to simulate complex real-world workloads ([Liu and Yuan \[2019\]](#)), out-of-order insertions are inevitable in real-world scenarios and can have performance implications ([Wang et al. \[2020\]](#), [Andersen and Culler \[2016\]](#)), batch insertion is often used by databases to optimise ingestion throughput ([Buevich et al. \[2013\]](#)), aggregation, downsampling, time-ranged and filtering queries are very common operations for IoT scenarios and are often the focus of

TSDBMSs (Andersen and Culler [2016], Paparrizos et al. [2021], Wang et al. [2020]). For the analysed benchmarks to be considered to have realistic workloads, at most one of the listed desired features can be missing.

None of the analysed benchmarks support the use of **workload traces**, possibly due to these not being publicly available. Only IoTDB-Benchmark Liu and Yuan [2019] presents all the desired features for a **realistic synthetic workload**. Benchmarks that were only missing one of the listed features were also considered to be realistic, even while being inferior to IoTDB-Benchmark. These were SmartBench Kamien-ski et al. [2019], TSM-Bench Khelifati et al. [2023], and TSBS Timescale [2022], the latter of which was missing a mixed workload, while the first two were missing out-of-order insertions.

The remaining benchmarks presented **unrealistic synthetic workloads**. SciTS Mostafa et al. [2022], **PSoTSDBs** Shah et al. [2022] and TS-Benchmark Hao et al. [2021] were missing mixed workloads. Out-of-order insertion was the most absent feature with only the aforementioned IoTDB-Benchmark and TSBS supporting it. The other considered ingestion feature, batch ingestion, was missing from **PSoTSDBs**, IoTDataBench Zhu et al. [2021], TSDBBench Bader [2016] and TPCx-IoT [TPC]. **PSoTSDBs** and TPCx-IoT were also missing downsampling and out-of-range or filtering queries from their workloads, with TSDBBench also missing the latter.

The **operation types supported** by the supported workloads were: **ingestion**, **querying**, or a **mix**, in which ingestion and querying operations are used concurrently. Both data loading and appending workloads were categorised as ingestion workloads.

The vast majority of the surveyed benchmarks support workloads consisting of either only **ingestion** or only **querying** operations. TPCx-IoT [TPC] and IoTDataBench were the only benchmarks missing these pre-defined workloads, opting only for **mixed workloads**. IoTDataBench, SmartBench, IoTDB-Benchmark, TSDBBench, TPCx-IoT, and TSM-Bench all support mixed workloads, concurrently performing ingestion and querying operations, and achieving a more demanding workload.

The **client rate** category analyses the way workloads scale by varying each client's throughput rate. This can fall into one of three categories: **maximum**, meaning the workload tries to increase the client's throughput rate until the database is stressed; **static**, meaning the workload keeps a constant throughput rate for each client; and **dynamic**, in which the workload tries to vary the throughput rate of each client between a minimum and maximum value.

Most surveyed benchmarks **maximise** the client's throughput. IoTDataBench, TSDBBench and TPCx-IoT use a **static** client rate since they are all built on top of **YCSB** Cooper et al. [2010] which throttles the operation throughput, while TSBS supports both a **maximum** throughput mode and **static** throughput.

TSM-Bench maximises throughput when running its insertion workload and uses a static throughput rate when running its mixed workload. Moreover, none of the benchmarks use a dynamic client rate strategy.

2.4.2 Data

Table 2: Data characteristics presented by surveyed benchmarks

	Origin			Datasets	
	Real	Realistic	Unrealistic	Multiple	Scalable
SciTS Mostafa et al. [2022]			●		●
PSoTSDBs Shah et al. [2022]	●		●	●	
TS-Benchmark Hao et al. [2021]		●		●	
IoTDataBench Zhu et al. [2021]	●*	●*	●		●
SmartBench Gupta et al. [2020]		●			●
IoTDB-Benchmark Liu and Yuan [2019]			●		●
TSDBBench Bader [2016]			●		●
TSBS Timescale [2022]			●		●
TPCx-IoT [TPC]			●		●
TSM-Bench Khelifati et al. [2023]		●		●	●

* supports user-provided

Table 2 presents the observed data characteristics sub-categorised into the data's origin and data scaling options

For the **origin** of the datasets used for benchmarking, three options were observed. **Real data**, meaning data extracted from real-world scenarios; **synthetic data**, consisting of either purely random data, or random data generated by using probabilistic distributions which can be more realistic but were still categorised as purely synthetic; and **data based on real data**, which can be generated by a tool that receives real data and uses it to generate data that approximates the input.

Only two of the analysed solutions, **PSoTSDBs** [Shah et al. \[2022\]](#) and IoTDataBench [Zhu et al. \[2021\]](#) support the use of **real datasets**, with the former featuring them for testing, while the latter supports user-provided real datasets.

TS-Benchmark [Hao et al. \[2021\]](#), TSM-Bench [Khelifati et al. \[2023\]](#), and SmartBench [Kamienski et al. \[2019\]](#) both provide **realistic synthetic data** by using data generation tools that were trained

using real data. The first two make use of trained **Generative Adversarial Network (GAN)** models, while the latter uses temporal, speed, device, and user scaling to generate realistic data. In addition, IoTDataBench also supports the training of a data generation model using a user-provided dataset, allowing for customisable realistic synthetic data generation.

Overall, the vast majority of the surveyed benchmarks allow using **purely synthetic data**, with five out of the ten surveyed solutions supporting only the use of synthetic data, and supporting no other data sources.

The scaling capabilities of the analysed data solutions were evaluated by considering whether **multiple** pre-made datasets were provided or if the data generation tools allowed for the volume of generated data to be **scaled**.

PSoTSDBs, TS-Benchmark and TSM-Bench provide **multiple** dataset volume options for the scaling of tests, while the remaining provide a way of **scaling** the data for testing, usually by having the user be able to decide the volume of data to be generated. TSM-Bench provides multiple datasets with different volumes as well as providing a data-generation tool, allowing for scaling of the generated dataset.

2.4.3 Metrics

Table 3: Metrics presented by surveyed benchmarks

	Performance								Resource monitoring				
	Throughput		Latency										
	Sampling		Target		Readings								
	Continuous	Average	Inserts	Queries	Average	Percentiles	Min/Max	Standard Deviation	CPU	Memory	Storage (I/O)	Network (I/O)	Storage Quota
SciTS Mostafa et al. [2022]	●		●	●	●	●	●	●	●	●	●	●	
PSoTSDBs Shah et al. [2022]		●		●	●								●*
TS-Benchmark Hao et al. [2021]	●			●	●				●*	●*			●
IoTDataBench Zhu et al. [2021]	●												●
SmartBench Gupta et al. [2020]		●		●	●				●*				
IoTDB-Benchmark Liu and Yuan [2019]	●	●	●	●	●	●	●		●	●	●	●	●
TSDBBench Bader [2016]				●	●	●	●						
TSBS Timescale [2022]	●			●	●	●	●	●					
TPCx-IoT [TPC]	●												
TSM-Bench Khelifati et al. [2023]		●		●	●								

Table 3 organises the observed performance metrics in throughput and latency sub-categories.

Throughput metrics represent a very important aspect of a **TSDBMS**'s performance since these solutions are subjected to a high volume of data ingestion.

Overall, the analysed benchmarks mostly measure the throughput of **inserts**, with TSDBBench [Bader \[2016\]](#) being an exception as it does not measure throughput of any kind, and TPCx-IoT [\[TPC\]](#) and IoT-DataBench [Zhu et al. \[2021\]](#) being the remaining exceptions by measuring throughput for **all operations**. Both data loading measurements and data appending measurements were considered to be inserts.

Apart from **PSoTSDBs** [Shah et al. \[2022\]](#), which presents the **time** taken to load a certain amount of data in seconds, all other benchmarks present throughput in **operations per second** (also mentioned as points or rows per second), with TS-Benchmark [Hao et al. \[2021\]](#) also producing throughput for loading performance in data volume per second.

Finally, the throughput values produced by these benchmarks are either presented or logged in a **continuous** way throughout its execution or simply calculated as an **average** result and presented at the

end of the execution. Benchmarks such as TPCx-IoT, continuously present a rolling overall average, while others, such as SciTS [Mostafa et al. \[2022\]](#), continuously present the calculated average observed over regular intervals; both of these were considered to be continuous.

Two sub-categories were considered for **latency** metrics: the **type of operation** for which latency is measured, as well as what **type of readings** are presented. Latency metrics can be measured for both insert and query operations, with queries being considered any data fetching operation ranging from simple reads to aggregation and ranged queries, which are very common operations for the analysis of **IoT** sensor data.

Both TPCx-IoT and IoTDataBench [Zhu et al. \[2021\]](#) do not present latency metrics, being the only analysed benchmarks to not do so, while SciTS and IoTDB-Benchmark [Liu and Yuan \[2019\]](#) were the only surveyed benchmarks to present latency metrics for **inserts**. SciTS used insert latency for different batch sizes to study how these affected performance. All benchmarks that measured latency did so for **queries**, either displaying latency metrics for individual queries or all queries.

While benchmarks such as **PSoTSDBs**, TS-Benchmark, SmartBench, and TSM-Bench [Khelifati et al. \[2023\]](#) present only **average** values for latency, most surveyed benchmarks present more useful statistics, which allow for more in-depth analysis of the performance of the **SUT**. These included several **percentile** values, **minimum** and **maximum** values and **standard deviation**. All these statistics are produced by SciTS and TSBS [Timescale \[2022\]](#), with IoTDB-Benchmark and TSDBBench presenting all but the standard deviation values.

Table 3 also presents the resource monitoring metrics measured by each benchmark. Resource monitoring was considered as it allows for a better understanding of a **SUT**'s behaviour and performance.

Most surveyed benchmarks include little to no resource monitoring, with **PSoTSDBs** [Shah et al. \[2022\]](#) and SmartBench [Kamienski et al. \[2019\]](#) including some monitoring only during testing, without it being integrated into the benchmark. TS-Benchmark [Hao et al. \[2021\]](#) also uses **CPU** and **Memory** monitoring in testing, and **storage** monitoring integrated into the benchmark.

While both SciTS [Mostafa et al. \[2022\]](#) and IoTDB-Benchmark [Liu and Yuan \[2019\]](#) feature most considered resource monitoring (with SciTS missing **storage quota** monitoring), SciTS does so by integrating an external tool (Glances), while IoTDB-Benchmark implements its system monitoring.

Storage quota monitoring usually consists of simply measuring the final storage used by the database, and is mostly used to evaluate compression, by comparing the measured value to the expected storage usage by considering how much data was inserted.

2.4.4 Orchestration

Table 4: Orchestration presented by surveyed benchmarks

	Multi-client		Database node scaling*
	Local	Remote	
SciTS Mostafa et al. [2022]	●		
PSoTSDBs Shah et al. [2022]			
TS-Benchmark Hao et al. [2021]	●		
IoTDataBench Zhu et al. [2021]		●	●
SmartBench Gupta et al. [2020]			
IoTDB-Benchmark Liu and Yuan [2019]	◐		
TSDBBench Bader [2016]	◐		●
TSBS Timescale [2022]	◐		
TPCx-IoT [TPC]		◐	
TSM-Bench Khelifati et al. [2023]			

● Automatic

◐ Manual

* Single-layer only

Table 4 details whether the surveyed benchmarks support orchestration, i.e., the automatic deployment of multiple clients and database nodes.

Support for the deployment of multiple clients was categorised in two different ways. First, whether the analysed benchmarks support multiple **local clients**, multiple **remote clients** or do not support multiple clients at all. Second, the benchmarks that support multiple clients were categorised by whether the deployment of additional clients is done **automatically** or **manually** through a set number of clients. Ideally, a benchmark should support the automatic deployment and usage of multiple remote clients, for a more realistic simulation of a distributed workload as well as ease of use.

SciTS [Mostafa et al. \[2022\]](#) supports automatically testing over a set of numbers of clients defined by the user through a parameter, and such was considered to be **automatic**, while TS-Benchmark [Hao et al. \[2021\]](#) automatically increases the number of threads in its continuous injection workload to simulate a growing workload. In addition, IoTDB-Benchmark [Liu and Yuan \[2019\]](#), TSDBBench [Bader \[2016\]](#) and TSBS [Timescale \[2022\]](#) support multiple **local clients** with the use of parameters. IoTDataBench [Zhu et al. \[2021\]](#) and TPCx-IoT [\[TPC\]](#) are the only surveyed systems that support usage of **remote clients**,

with the former doing so automatically to increase the load on the **SUT**, and the latter allowing the user to define which remote machines to use for a certain benchmark run.

Analysed benchmarks only feature **single-layer** database node scaling, since none of them consider **multi-layer TSDBMSs**. IoTDataBench and TSDBBench are the only benchmarks that feature **database node scaling**. IoTDataBench uses database commands to increase the number of database nodes while simultaneously adding corresponding clients for load generation during execution of its scalability test. However, it does not seem to take into account any form of connection between database nodes. On the other hand, TSDBBench uses exploits Python and Vagrant ([HashiCorp \[2010\]](#)) scripts in its *Elastic Infrastructure* to enable the automatic setup and deployment of database nodes in virtual machines in a range of both public and private cloud solutions.

2.4.5 Discussion

Some of the analysed benchmarks were capable of producing realistic workloads, without which it is impossible to accurately assess a **SUT**'s performance when deployed in a real system. Through the use of concurrent insert and querying operations, multiple solutions achieve realistic workloads that simulate demanding real-world scenarios in which data is being inserted while historical queries are being performed.

Several of the existing solutions provide either real or realistic data, essential for the proper evaluation of the targeted databases, since these systems may feature mechanisms that make use of patterns present in real time series data, such as InfluxDB [InfluxData \[2013\]](#) which stores deltas between sequential values. By using unrealistic data, a benchmark may not be able to accurately access the impact of these mechanisms in a particular system's performance. In addition, ways to scale the volume of data were also present. These are needed for the benchmarking tools to stay relevant as either the **SUTs** or the hardware they run on evolves to be capable of handling more data.

For throughput metrics, benchmarks should ideally continuously track both operation and volume rates for inserts, with one achieving that and several other existing solutions only missing the volume rates, meaning there are existing solutions that are satisfactory in this regard. By providing continuous metrics, benchmarks enable the analysis of a system's continuous behaviour, such as how the insertion throughput or query latency values vary throughout a test's execution, e.g., identifying a severe performance degradation halfway through a test, and which would not be obvious by looking at a single value.

The analysed benchmarks focus mostly on having latency metrics for querying operations, and several solutions present statistical latency results, which can be very useful to assess query performance. Often times, mean latency values do not tell the whole story about a system's performance. Two systems

with identical mean latencies may in practice perform very differently if one has significantly higher 90th percentile latencies, which can affect certain applications' performance.

Resource monitoring metrics can help understand a **SUT**'s behaviour but were mostly neglected by existing benchmarking solutions, with only a couple featuring extensive monitoring. This kind of metric can be used to identify system bottlenecks, which can allow the benchmark users to build more efficient deployments, such as in an Edge-Cloud deployment in a situation where the edge nodes are the system's bottleneck and in which case adding a cloud node would not improve its overall performance.

None of the surveyed benchmarks present metrics at the node level, measuring only the database's overall performance. This makes the existing solutions suboptimal for the assessment of the multi-layer **DBMSs** as their query performance can vary depending on not only in which node but also in which layer the necessary data is stored.

Orchestration features are not commonly found, with none of the surveyed benchmarks supporting multi-layer database node scaling, which is essential for the proper evaluation of Edge-cloud sink node **TSDBMSs** as it allows the assessment of how the **SUT**'s performance varies depending on how many nodes are present on each layer or how they are configured. In addition, the automatic deployment of both multiple benchmark clients and database nodes can greatly increase a benchmark's ease of use by not only allowing the user to easily deploy a database cluster several nodes but also to more easily scale the benchmark's generated load.

In summary, while there are good existing solutions for benchmarking **TSDBMSs**, all of them lack features that are essential for the evaluation of multi-layer Edge systems.

Chapter 3

MulletBench

This chapter presents an overview of the MulletBench benchmarking tool, as well as its architecture, features and general characteristics, including its usage and configurations.

MulletBench operates through two distinct phases: deployment and execution. Its two components, the Orchestrator and the Client, are responsible for the entire process. During deployment, the Orchestrator configures database nodes, simulates resource and network limitations, and deploys Clients, enabling the testing and simulation of a multi-layer **TSDBMS** deployment. In the execution phase, Clients are triggered to run in sequential stages, logging all operations and resource usage, and the Orchestrator aggregates all results. These stages can be configured to accommodate various workloads, such as including pre-population stages, or having several consecutive stages of varying load. By providing aggregated metrics per node, layer, Client or stage, as well as overall results, MulletBench allows for a fine-grained analysis of the **SUT**'s performance.

3.1 Design Principles

MulletBench consists of two main components: the Orchestrator which is responsible for the installation of all tools necessary for the entire benchmarking process, running Client instances, orchestrating the Clients' execution and finally retrieving and aggregating all generated metrics from each Client; and the Client, responsible for submitting the specified load to the database and registering all submitted operations. Each Client can perform either insertion or query operations, and multiple instances of each can be run simultaneously to generate mixed workloads.

Overall, the benchmark features the following characteristics:

Workload Workloads can be defined by their type, the operations composing these and what rate is used to submit these operations. MulletBench includes:

- Synthetic workloads: Through the composition of both insert and query Clients, different kinds of workloads can be defined. The following are examples of workloads that were defined and used in our experiments.
 - Ingestion workload: Simple insertion of a set amount of data into an empty database. This workload allows for a synthetic analysis of a **DBMS**'s ability to handle large amounts of data, an important aspect for **TSDDBMSs**;
 - Query workload: Performing a limited rate of queries over data previously loaded into the database. Measures query latency as well as query throughput, and allows for the measuring of a database's isolated query performance;
 - Pre-populated mixed workload: This workload constitutes a realistic workload as per the parameters defined in [subsection 2.4.1](#). An initial stage of insertion is performed, followed by the insertion of a set amount of data while simultaneously performing a limited rate of queries. Measures both insert and query latency and throughput, including for the initial insertion stage. This subjects the **SUT** to a realistic workload, allowing also for the analysis of how a database performs when it contains higher volumes of data.
- Operations: To compose the desired workloads, Clients should be configured to perform either insertion or query operations.
 - Inserts: perform the insertion of data using batches with configurable size. This allows for the evaluation of a database's ability to ingest data, including how it performs with different batch sizes;
 - Queries: perform time-ranged aggregations, downsampling, and filtering queries, which are the most common query types for historical analysis. These can be used to create a query workload that can be representative of the real usage.
- Client Rate: The rate at which operations are submitted can be explicitly defined for both insertions and queries, and remains constant for the whole execution of a Client instance. A single test can be defined to have an increased rate of operations through the use of multiple stages. This allows for the simulation of a scenario where a constant amount of analysis is done on an increasing amount of data.

Data The characteristics of the data used in a benchmark can have a significant impact on the results. Using a real dataset allows MulletBench to benefit from the advantages of using real data, containing real

data patterns. By looping over the entire dataset, the benchmark can be configured to insert as much data as desired, regardless of the used dataset's size.

Metrics Relevant metrics enable the accurate evaluation of a **TSDBMS**'s performance, as well as the comparison of different **TSDBMSs** or deployment configurations. MulletBench includes the following metrics:

- Throughput: continuously logging operation timestamps enables the computing of both insert throughput (in operations and volume inserted per second) and query throughput (in operations and volume received per second) per node, allowing for later analysis of throughput over time;
- Latency: continuously logging operation latency enables the computing of query latency (per query type) as well as insertion latency, per node, allowing for later aggregation into average, median, percentiles, min, max and standard deviation, and analysis of query latency behaviour;
- Resource monitoring: CPU, RAM, storage I/O, as well as network throughput (per node/layer), allowing the identification of potential bottlenecks.
- Aggregating metrics per database node, layer, Client and stage, as well as overall results, allowing for a more detailed analysis of the **SUT**'s performance.

Orchestration Orchestration features provide the automation of tasks simplifying the benchmark's usage and increasing its reproducibility. MulletBench supports the following features:

- Deployment of remote benchmark Clients, enabling a more realistic scaling of load;
- Deployment of a remote multi-layer database cluster, increasing the consistency of results as well as the ease of use;
- Simulated edge node resources, limiting CPU, RAM and disk IO usage, allowing for the evaluation of how a database performs in Edge nodes without having access to the hardware;
- Simulated network limitations can be used to increase latency of operations, limit bandwidth as well as reorder network traffic, which leading to out-of-order operations, can potentially impact performance;
- Configured by the user (number of Benchmark Clients, database nodes, IPs).

This feature set has been carefully chosen based on the analysis of previously existing time series benchmarks as well as considering the feasibility of what can be achieved in the scope of this dissertation. It allows MulletBench to be the first **TSDBMS** benchmark to feature per layer metrics, edge node and network simulation, and the automated deployment of a multi-layer database cluster, while also including resource monitoring metrics and the ability to generate distributed workloads, features which are often missing from other benchmarks.

3.2 Architecture

The overview diagram in Figure 1 presents a general view of MulletBench's deployment. The Orchestrator deploys database nodes on servers, being able to simulate both edge and cloud environments, and coordinates the Clients' execution which submit a workload to any node in the cluster. The Orchestrator can collect resource monitoring logs from all nodes, as well as operation logs from Clients.

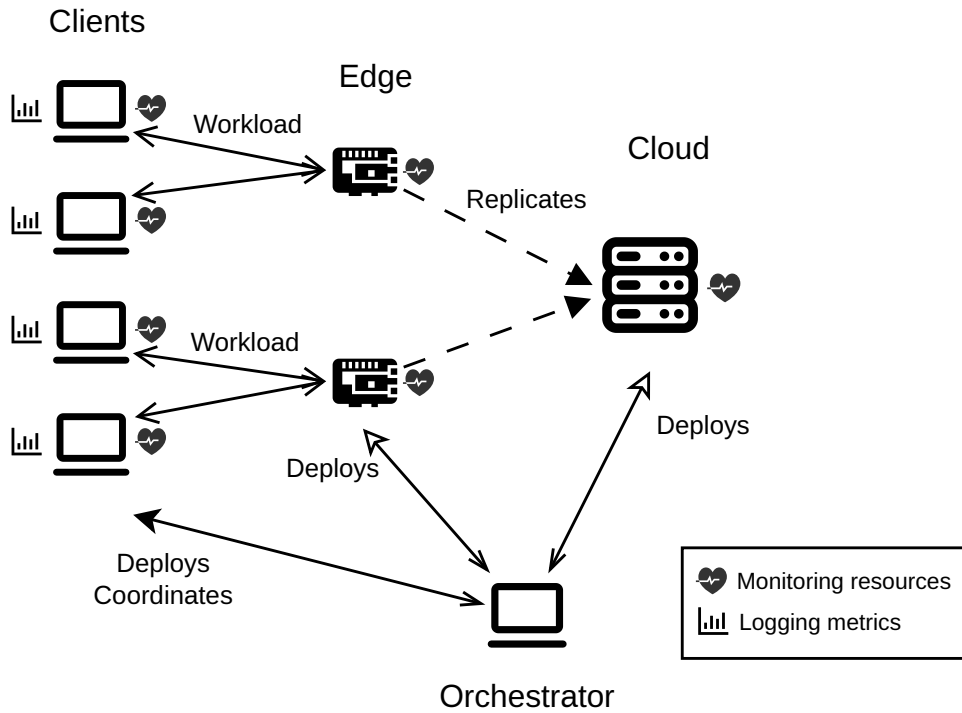


Figure 1: MulletBench Overview

3.2.1 Orchestrator

In the deployment phase, the Orchestrator is responsible for provisioning the database host machines and deploying all database nodes (step 1 and 2 in Figure 2). This includes setting up their configuration,

such as enabling replication between nodes or downsampling tasks (in step 2). This can also ensure each repetition of a test uses the same exact database configuration, increasing the benchmark’s reproducibility. In addition, it also simplifies its usage since the user does not need to deploy and configure the nodes themselves.

This phase is also when the Orchestrator sets resource limiting for edge nodes such as limited CPU usage, RAM usage and disk I/O usage, if needed. It can also control network usage, limiting bandwidth, introducing latency and reordering packets. Thus, the benchmark can simulate both edge nodes and the entire network conditions between all nodes. This can be useful when a distributed setup needs to be assessed and there is no way to deploy a real distributed setup.

Finally, the deployment phase launches as many Client instances as specified in the configuration file (step 3).

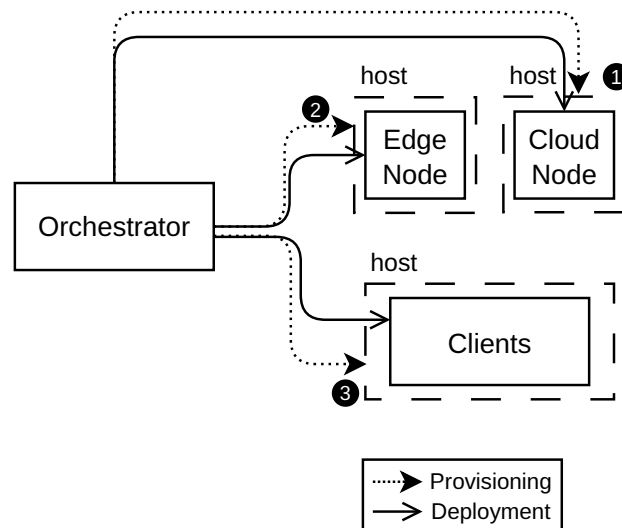


Figure 2: Deployment Flow Diagram

The Orchestrator is responsible for remotely monitoring database nodes (step 5 in Figure 3) and Clients (step 6). During test execution, its main function is connecting to all Clients (step 4) and triggering the start of their workload execution in sequential stages (step 7), ensuring all Clients of each stage start simultaneously and test stages are run sequentially. During the test, the Clients and database nodes will exchange data through insertion and query operations (step 8). At the end of the test's execution, the Orchestrator also collects all results from the Clients (step 9), and displays results by layer, by individual database node and by Client, as well as overall results for the test. For all these features to be possible, the Orchestrator must be aware of all Clients as well as all database nodes involved in the test. It also locally stores all collected results (both execution logs and monitoring logs).

Since the Orchestrator needs to know about all actors present in a particular test, both for Clients and database nodes, it requires a configuration file detailing all necessary information.

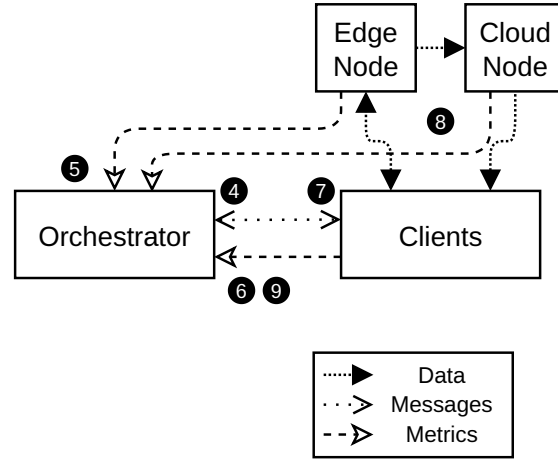


Figure 3: Execution Flow Diagram

3.2.2 Client

The Client is responsible for performing any local processing and ultimately submitting a workload to a particular database node. Clients can be configured to perform either insertion or query operations, and multiple instances of each can be run simultaneously to generate mixed workloads.

Insert

The Insert Client can either operate in an original time insertion mode, in which the dataset's original timestamps are used, inserting past data, or a real-time insertion mode, in which current timestamps are injected into the dataset, simulating data being generated during execution.

In the real-time insertion mode, there is the problem of how timestamps should be assigned to the dataset in a realistic way. Since insertion operations are performed in batches, the timestamps of each entry in a batch need to be distributed throughout the time interval between the previous insertion operation and the current one. For instance, when inserting a batch of 10 entries every 100 milliseconds with a current timestamp of 1100, the entries in the inserted batch would have the timestamps 1010, 1020, 1030, 1040, 1050, 1060, 1070, 1080, 1090 and 1100. This means that the first entry in following batch would have the timestamp 1110, ensuring entries in consecutive batches keep the same interval as each entry in the same batch. Figure 4 presents a visualization of this mechanism.

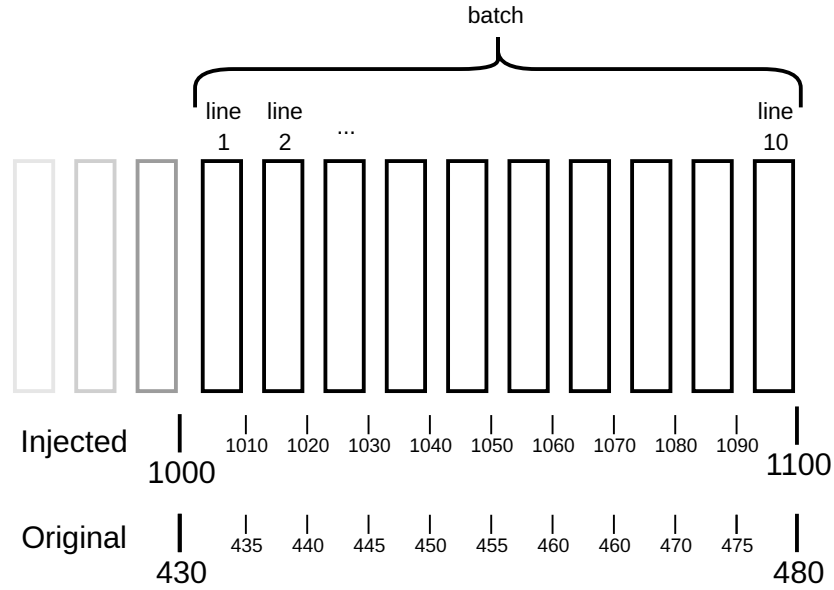


Figure 4: Timestamp Injection Diagram

In the original time insertion mode, the existing dataset is used with its real timestamps. Since it might be desirable to insert more entries than are present in a dataset, it is necessary to loop over the existing data as many times as needed. This way, the first loop over a dataset will use its real, pre-existing timestamps. However, each subsequent loop will need to calculate a new timestamp in a way that ensures a consistent interval between entries, including between the end of a loop and the start of the next. For this, the interval between the first entry in the dataset and the first entry in the next loop can be calculated by subtracting the first timestamp in the dataset from the last timestamp, and adding the interval between the first and second timestamps, assuming the dataset uses a regular interval between entries. The algorithm 1 shows how this can be achieved in a simplified scenario.

For the sake of example, assuming a dataset has a first timestamp of 10000, a second timestamp of 10100, and a last timestamp of 19900, this interval between consecutive loops comes out to 10000. When looping over this dataset, the first timestamp in the second loop would become 20000, 10000 (the original timestamp) + 10000 (the calculated interval), the second one 20100, and so on, thus keeping the constant interval of 100.

Query

The Query Client performs a certain number of randomly generated queries at a constant rate.

Since the Query Client needs to perform relevant queries for a particular dataset, it needs to initially

Algorithm 1 Dataset looping with original timestamps

```
dataset_length  $\leftarrow$  dataset[dataset.length - 1].timestamp - dataset[0].timestamp
    + (dataset[1].timestamp - dataset[0].timestamp)
num_loops  $\leftarrow$  0
while inserted < target_insertions do
    for line  $\in$  dataset do
        line.timestamp  $\leftarrow$  line.timestamp + (dataset_length * num_loops)
        insert(line)
        inserted  $\leftarrow$  inserted + 1
    end for
    num_loops  $\leftarrow$  num_loops + 1
end while
```

perform an analysis of the dataset, extracting average, standard deviation, minimum and maximum values for each column. The generation of queries can be performed either before execution, minimizing overhead during execution, or performed during execution, which incurs a higher performance overhead but enables queries to take into account current time during execution. Since the Client can't know before execution how long the insertion process will take, it can only pre-generate queries if a dataset has been inserted with its original timestamps, in which case it can calculate what the minimum and maximum timestamps will be based on how much data was inserted.

The Query Client is able to generate filter, aggregation and downsampling queries. All of these are time-ranged, meaning they search for values between two timestamps.

Aggregation queries use time ranges between 5 seconds and 20 minutes. They choose a random time interval in which data has been inserted and perform a randomly selected aggregation (chosen between minimum value, maximum value, count, sum, or average of values) in which all values for a randomly selected field in that time range are reduced to a single value.

Downsampling queries use a randomly selected function from the same set of possible functions, aggregating data from a randomly selected field in intervals of 15, 30, 60 or 120 seconds, and using time ranges from 1 to 120 minutes.

Outlier filter queries use z-scores to find values that lie outside certain ranges. This can be done using the computed average (μ) and standard deviation (σ) values for each field in the used dataset, and being given a target z-score (z). The value (x) can thus be calculated through the following equation:

$$x = \sigma z + \mu \quad (3.1)$$

This value is used as a minimum value for a filter query with the purpose of finding maximum values. A value for finding minimum values can easily be calculated by using negative z in the previous equation.

For every successfully performed query, the Client registers both the timestamp in which the query was submitted, the timestamp in which the result was received, the size of the received result (how many entries it contains) and the type of query performed, allowing for query-type specific results.

3.3 Implementation

3.3.1 Orchestrator

The deployment component of the Orchestrator was implemented through the use of Ansible¹, an open-source tool that simplifies all kinds of automation and deployment. Through the implementation of an Ansible playbook, composed of several plays specifying which tasks to perform, a single configuration file (commonly called inventory file) allows for the definition of all hosts and variables needed for the execution of a wide range of tests. This allows for the installation of all tools needed on all machines, along with the running of any program needed for the system.

Docker² and Docker Compose are both installed by this component on each machine as both the execution component of the Orchestrator and the Clients are run in Docker containers, with each of these components having their own image. Additionally, supported databases should also run in Docker containers.

For the automatic installation of database nodes as well as setting up replication, database-specific plays were implemented. Support for other databases can be added by implementing additional plays for the following: starting and stopping the instance, starting and stopping replication, starting downsampling (if supported).

This component is also responsible for setting limited resources on database instance containers by altering the used Docker Compose files through templating. Docker limits disk IO usage through the use of blkio, when using the blkio_config option in a Compose file. In practice, the container's use of the storage device is limited, which does not necessarily exactly match the application's use of disk IO, due to the usage of buffered IO. CPU and RAM usage are limited in the Compose files through the deploy-resources-

¹ Ansible: <https://www.ansible.com/>

² Docker: <https://www.docker.com/>

Table 5: Raspberry Pi 3 B+ resource limiting profile

Variable	Value
limited_resources_cpu	4
limited_resources_mem	1024M
limited_resources_read_bps	22M
limited_resources_write_bps	16M
limited_resources_read_iops	2700
limited_resources_write_iops	1200

Table 6: Raspberry Pi 4 resource limiting profile

Variable	Value
limited_resources_cpu	4
limited_resources_mem	4096M
limited_resources_read_bps	44M
limited_resources_write_bps	40M
limited_resources_read_iops	2700
limited_resources_write_iops	1200

limits keys *cpus* and *memory*, respectively. These resource limits can be defined with each individual parameter, but some pre-defined profiles that emulate common edge devices such as the Raspberry Pi 4 have already been defined and can be used directly. The CPU and memory values were based on each device's specifications and the IO limits were based on benchmarks performed on SD cards found on PiBenchmarks³. The resource limiting profiles for the Raspberry Pi 3 B+ and Raspberry Pi 4 can be found in Table 5 and 6, respectively.

The Orchestrator also achieves network limitations through the use of tcconfig⁴, an open-source wrapper for tc commands. **Performance Co-Pilot (PCP)**⁵ is also installed and run if monitoring is desired.

The execution component of the Orchestrator component was implemented using Java, with Maven for build automation. This component is provided all relevant information about the test configuration through a YAML file generated in the deployment phase. Since the orchestrator needs to perform resource monitoring of both database and Client nodes through **PCP**, it runs multiple *pmrep* processes and, when a test's execution is completed, kills them.

³ PiBenchmarks: <https://pibenchmarks.com/>

⁴ tcconfig: <https://github.com/thombashi/tcconfig>

⁵ Performance Co-Pilot: <https://pcp.io/>

3.3.2 Client

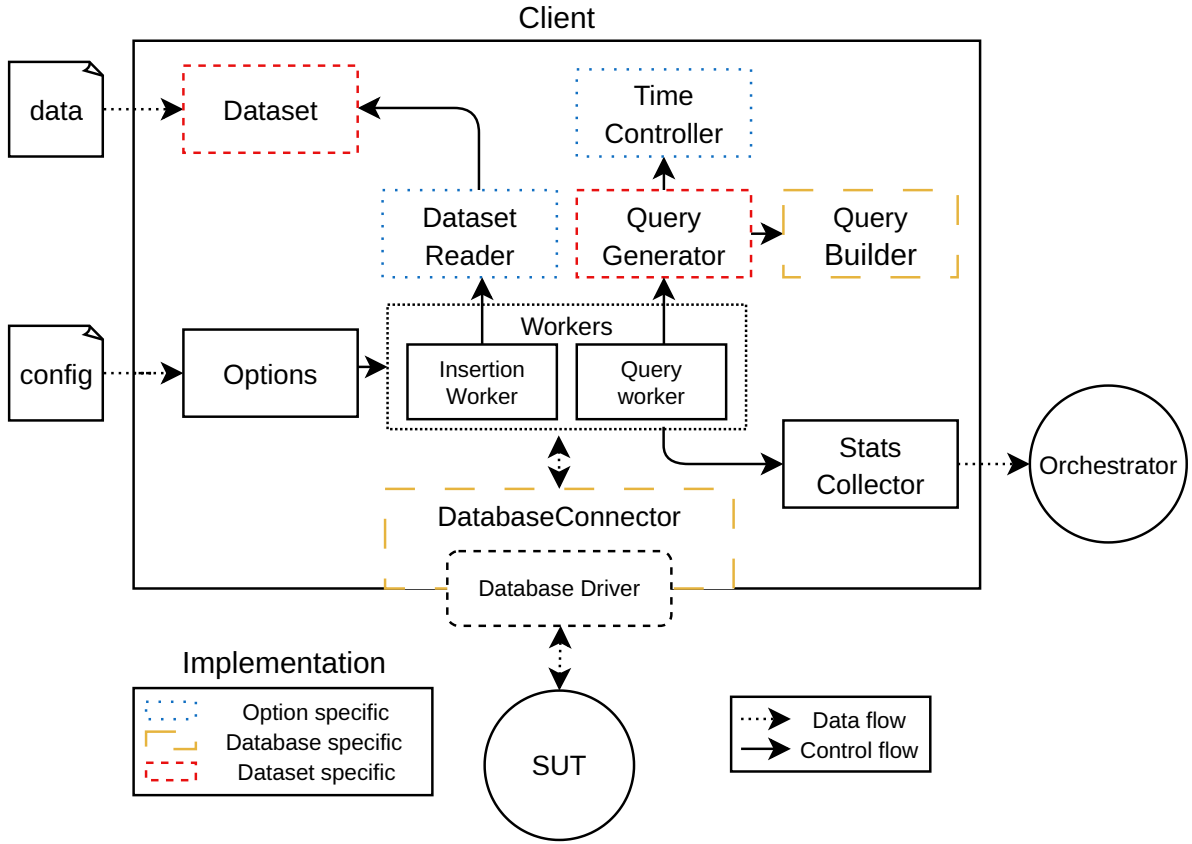


Figure 5: Benchmark Client structure

The Client component was also implemented using Java. Diagram 5 presents the main classes used in the Client component, with options, database, and dataset specific classes being omitted for simplicity, showing only respective interfaces or abstract classes. Since Clients should be abstracted from which database they are evaluating, database connectors must be implemented for each supported database, using a common interface that can be used by the workers. Any database-specific classes, such as *Measurement* classes for InfluxDB or *Schema* classes for Apache IoTBD, should be implemented and used exclusively by that database connector implementation. These will need to implement each dataset's parsing, so that the original string data can be correctly parsed into each database's preferred representation.

Dataset classes are alternative ways data can be stored and used, implementing the *Dataset* interface to standardize access to any dataset representation's data. In cases where the dataset used for an insertion workload can fit entirely into memory, it can be loaded into memory only once and subsequently accessed by all workers simultaneously using the *SharedDataset*. Otherwise, *IteratorDataset* is used, avoiding loading the whole dataset into memory, but resulting in higher runtime overhead since it

the dataset needs to be read from disk during execution.

On the other hand, *DatasetReader* classes implement parsing for each specific dataset to be used as well as provide a standardized way for workers to access data from a used dataset, and make use of *Dataset* classes to access that data in any representation.

An implementation of *Dataset* was created for a Passive Vehicular Sensors Dataset, in particular the `gps_mpu` dataset ([Menegazzo and von Wangenheim \[2020\]](#)). This dataset contains inertial sensor data combined with GPS data in a comma-separated values (CSV) format, containing 32 columns including the timestamp and 144,036 lines or lines, totalling 4,609,152 values, and with a frequency of 100 samples per second.

QueryGenerator classes also need to be implemented for each supported dataset since these need to know the name of each column and how to process the values in the dataset. Any dataset containing only float values can simply extend *FloatsQueryGenerator* and needs only to implement the *process* method, meaning the way queries are generated can be consistent for different datasets.

Each supported database should also implement a *QueryBuilder*, allowing the *QueryGenerators* to be abstracted from which database they are generating queries for.

The *TimeController* classes allow for there to be multiple timestamp options. *CurrentTimeController* allows for datasets to be manipulated by injecting the current time, while *DatasetTimeController* allows for datasets to be inserted with their original timestamp, and for their original time sequence to be extended when the dataset is being looped over.

A mechanism for rate limiting is used in cases where a lot of submitted operations are pending a response. Since operations are submitted through a thread pool, this mechanism does not impact how many operations are actually submitted to the database since that is already limited by the size of the pool, instead limiting only how many operations are in memory ready to be submitted. This avoids scenarios where very slow **SUTs** result in increasingly high memory usage on the Client through an increasing number of pending operations.

3.3.3 Configurations

For the configuration and executions of tests, the user needs to specify an Ansible inventory file, containing all the necessary information for the deployment of the database nodes and the benchmark Clients, as well as for the execution of each Client's workload. All the available options for this configuration file are listed in [A.1](#). Since the execution Orchestrator and the Clients can also be run manually, as well as for the understanding of each component's specific options, the available options for the execution of the

execution Orchestrator and the Clients are listed in [A.2](#) and [A.3](#), respectively.

Usage

To specify a test configuration, the user needs only to configure the aforementioned Ansible inventory file, specifying hosts and their specific variables, both for the database nodes and the benchmark Clients. Several examples of pre-defined tests have already been specified, and these can also be used as a guide for the user to more easily understand how this file should be organized. To perform a test run, the user should run the Ansible playbook, specifying the inventory file with the configured hosts and variables.

After a run has been performed, its logs are stored on the `/tmp/` directory in the machine where the Orchestrator was run. These logs contain the benchmark execution logs, as well as the monitoring logs for each Client and database node. These can be used by the included plot generation script to generate graphs of the results.

There is also a provided script for the automatic execution of multiple runs and generation of result graphs, with optional arguments which can be used to specify the number of runs, number of the starting run, whether to skip cleaning or whether to only perform cleaning.

3.4 Summary

MulletBench is a **Time Series Database Management System (TSDBMS)** benchmark specifically designed to evaluate how these systems perform when deployed in hybrid Edge-Cloud deployments and how their replication mechanisms perform. By featuring user-configurable workloads, MulletBench can be used to generate a wide range of realistic scenarios.

Its deployment features allow for not only in ease-of-use but also consistent results. In addition, the orchestration of several Client instances, allows for loads to be generated in a distributed way, more accurately representing real scenarios while simultaneously enabling the generation of more intensive loads by leveraging multiple machines.

While MulletBench's per-node, per-layer and per-stage metrics give relevant values for comparison and assessment of a system's performance, its continuous logging of performance and resource monitoring metrics enable a detailed analysis of a system's continuous performance throughout the whole test, more easily identifying problems or bottlenecks that may occur in any node present in the **system under test (SUT)**.

Chapter 4

Evaluation and Results

This chapter presents results from experiments performed using the specified and developed benchmark, including its 2 main components: the Orchestrator, and Client. Testing scripts were used to help automate the testing as well as generating all figures presented in this chapter, except for bar charts. These tests allow for the validation of some of the benchmark's mechanisms, such as its limitation features and observing the benchmarks general functioning. Then, the results from the tests are used to not only compare the two **DBMSs**, InfluxDB and Apache IoTDB, but also to compare different deployments and data retention strategies using downsampling. Using the benchmark's capabilities allows for the detailed analysis of performance metrics and resources during the test's execution.

This chapter aims to answer the following questions:

- Can MulletBench limit the rate at which its Clients submit a workload?
- Can MulletBench limit the resources of the edge database nodes?
- Can MulletBench limit the network conditions of its testing environment?
- How does the performance of InfluxDB and Apache IoTDB compare in the same deployment scenario for each workload type?
- How does the performance of each database vary in different deployment scenarios for each workload type?

Section 4.2 presents the results from the tests performed to validate the benchmark's limiting features, including the Client's rate limiting, the limiting of edge node resources and the limiting of network conditions. Section 4.3 presents the results from the tests performed in a cloud scenario, Section 4.4 presents the results from the tests performed in an edge-cloud hybrid scenario with 2 edge nodes, and Section 4.5 presents the results from the tests performed in an edge-cloud hybrid scenario with 4 edge

nodes. Throughout these sections, comparisons are made between both databases and between the different deployment scenarios. Finally, Section 4.6 presents the main points that can be drawn from the presented results.

4.1 Methodology

Each test was run 3 times as a way to ensure both the consistency of the benchmark and the conclusions taken from analysing the results.

Between each run, all database nodes were reset, removing all directories generated by the database's execution as well as removing the used Docker container instances.

For all tests, the Clients' resources were also monitored to ensure that they were never the limiting factor.

4.1.1 Experimental Setup

A total of 8 machines were used for testing, 5 of which were used for all Clients and the Orchestrator, sharing the same specifications. The remaining 3 machines were used for the database nodes, also sharing the same specifications between them. The specifications and software versions for the Client and Orchestrator machines are as listed in Table 7, while those of the database machines are listed in Table 8.

All Client and database machines are connected through a switched Gigabit network at a max distance of 1 hop.

4.1.2 Workloads

The testing considers 3 main workload categories: insertion or ingestion, query and mixed workloads.

Insertion The insertion workload aims to evaluate a database's pure insertion performance using batch insertions, enabling the analysis of a database's behaviour when submitted to a controlled insertion load. All insertion tests were configured with multiple Client instances and each with multiple workers. In each test, each worker inserts a batch of 100 lines every second. The scaling of this test is achieved by increasing the number of workers and Client instances. The target runtime length for each test was chosen to be 16 minutes, which means each worker inserts 96,000 lines, at the aforementioned rate. For this workload, the main metric is the number of lines per second successfully inserted by all Client instances, corresponding

Table 7: Specifications of database machines

Item	Description
CPU	Intel Core i5-9500
RAM	16 GB DDR3 2666MT/s
Storage	240 GB NVME SSD
Operating System	Ubuntu 20.04
Docker	version 24.0.4
Docker Compose	version v2.11.1
IoTDB	version 1.1.1
InfluxDB	version 2.7

Table 8: Specifications of Client and Orchestrator machines

Item	Description
CPU	Intel Core i3-4170
RAM	16 GB DDR3 1600MT/s
Storage	120 GB SATA SSD
Operating System	Ubuntu 20.04
Docker	version 24.0.4
Docker Compose	version v2.11.1
Ansible	version 2.15.1
Python	version 3.10.12

to how many lines from the selected dataset are being inserted, regardless of how many columns each line contains.

Query The configured query workloads aim to evaluate a database’s query performance by submitting common **IoT** queries such as outlier value filters, aggregations and downsampling queries, which are generated based on an analysis of the used dataset. These enable the analysis of a database’s ability to handle each type of query individually, without having the resulting metrics affected by its ingestion mechanism. For this workload, the main metrics are the latencies for each query type. In this workload, the dataset is loaded with its original timestamps due to the benchmark’s mechanism for knowing the time range for queries. The used query distribution is 40% for outlier filters, 40% for aggregations and 20% for downsampling queries, representing a plausible distribution, in which downsampling queries are less common since they correspond to continuous aggregations.

Mixed Finally, mixed workloads combine the previous types, performing the insertion of a dataset while submitting relevant queries for that dataset, to simulate a workload which the database might be subjected to when deployed in the real-world. For this workload, the main metrics are both the number of lines per second successfully inserted by all Client instances and the latencies for each query type. For tests using the replication of downsampled data to a cloud node, using the outlier filter query for that node no longer makes sense, since extreme values have been smoothed out. Therefore, in tests including downsampling, the cloud layer are subjected only to downsampling and aggregation queries. The total amount of queries is the same while keeping the proportion between downsampling and aggregation queries. The distribution

of query types is the same as in the query workload, except for the cloud layer when downsampling is used, where the distribution is 66.7% aggregations and 33.3% downsampling queries.

4.1.3 Data

All tests used the Passive Vehicular Sensors gps_mpu dataset [Menegazzo and von Wangenheim \[2020\]](#). This dataset is approximately 81 MB, containing 144,036 lines with 30 values each. As mentioned before in Section [3.2.2](#), the size of the used dataset is not a limitation since the Clients automatically loop over when the end of the dataset is reached, injecting the correct timestamps. This way, the used dataset can essentially be inserted as many times as necessary, each time with updated timestamps which follow the same pattern as the original dataset.

4.2 Feature Validation

This section presents a series of tests performed to demonstrate MulletBench's limiting capabilities. These tests were performed using a single database node without any resource limiting (except when stated). By validating each limiting feature individually, it is possible to ensure that the benchmark is capable of limiting the resources of a database node, or network connection between node, as well as the rate at which data is submitted to it, something which is necessary for the validity of the tests that follow.

4.2.1 Control Test

First, as control, a standard insertion test was conducted, showing what throughput each database can achieve without any limiting. For this test, the target throughput was controlled by varying the number of Client instances as well as the number of workers submitting data in each Client instance.

For InfluxDB, this test used 8 Client instances configured as listed in Table [26](#), resulting in a target throughput of 64,000 lines per second. The average throughput is 41,094.186 lines per second, showing the database was overloaded, and not able to handle the requests at the rate they were submitted. This resulted in the tests taking on average roughly 24.9 minutes.

For IoTDB, as initial runs show it could handle higher throughputs, the number of clients was increased, with 8 Client instances configured as listed in Table [27](#), totalling a target throughput of 360,000 lines per second. The average throughput for this test is 320,640.8 lines per second, a significantly higher value than what InfluxDB was able to achieve, which is also clearly visible by looking at the comparison of both database's throughput in Figure [6](#).

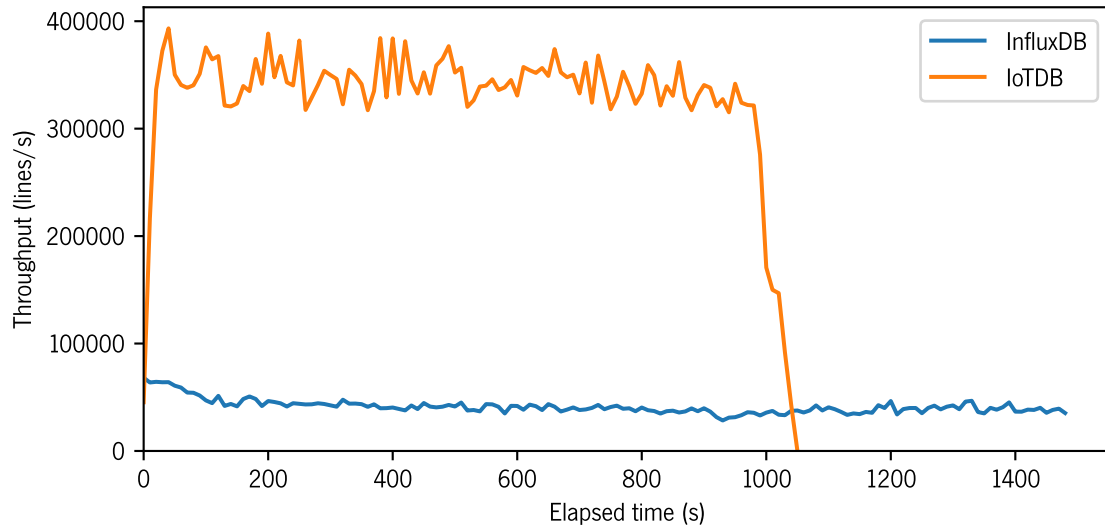


Figure 6: Throughput comparison for control test

4.2.2 Limited Rate

The rate limitation test shows how MulletBench can limit its Clients' rate and submit a consistent workload. The test setup is essentially the same as the control test, with the number of workers halved for each database, resulting in a target throughput of 32,000 lines per second for InfluxDB and of 180,000 lines per second for IoTDB. Note that, since each worker submits the same amount of data at the same rate, the expected runtime length of the test is the same.

InfluxDB

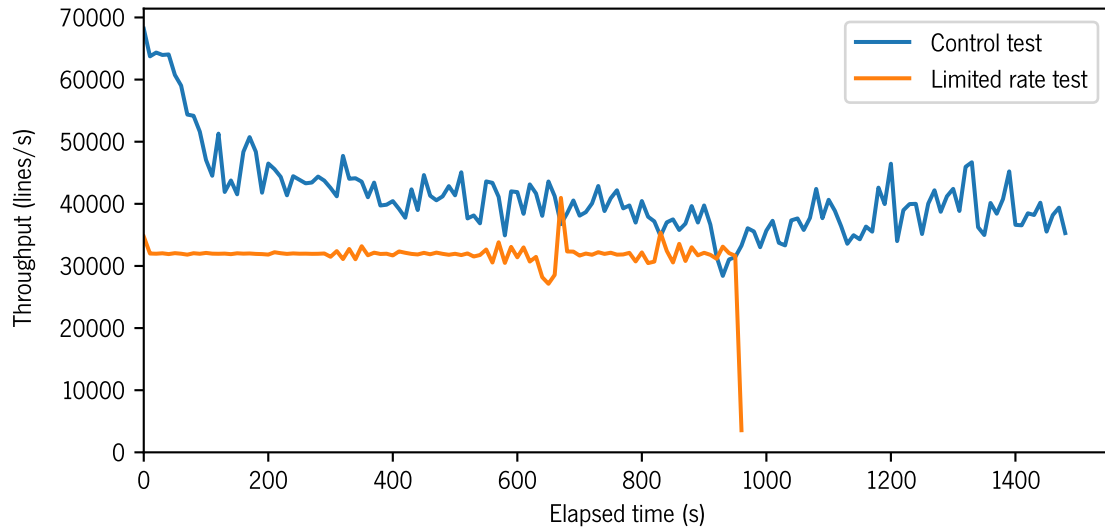


Figure 7: Throughput comparison for limited client rate test targeting InfluxDB

In Figure 7 the measured throughput is more stable than in the previous test, and stays around 32,000 lines per second, which is the target throughput for this test. This shows that the benchmark's rate limiting is working properly, and that the database is no longer overloaded. The submitted throughput can temporarily go over the target whenever drops in throughput from the database side causes a backlog of pending operations waiting for an available thread in the Client.

Although the achieved throughput does not exactly equal the targeted value, the run with the lowest average throughput has only a difference of 0.42% relative to the targeted value of 32,000 lines per second. This small difference from the target value can be attributed to overhead from the Client's execution.

Apache IoTDB

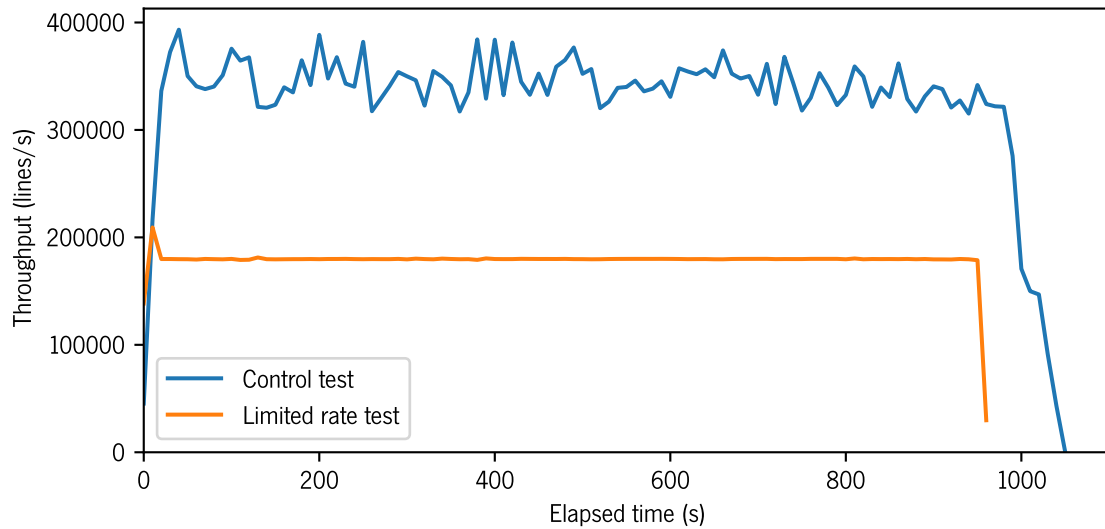


Figure 8: Throughput comparison for limited client rate test targeting IoTDB

For IoTDB, the observed average throughput is 178,544.95, only 0.81% away from the targeted throughput, and indicating the database is capable of handling the submitted load. Once again, the difference to the target throughput can be attributed to overhead from the Client's side. As Figure 8 shows, the measured throughput remains stable throughout the duration of the test showing that the Client's rate limiting is working as expected, being able to accurately control the load submitted to the tested database.

4.2.3 Limited Network

This test aims to demonstrate the effects of the network limiting capabilities of the benchmark. The test setup is similar to the rate limited test, except for the added network limiting, consisting of 50ms of added latency and 5ms of latency variation between each Client instance and the database node.

InfluxDB

As expected, this resulted in an observable latency increase, with similar throughput values. The usage of a request pool in the Client, as mentioned in Section 3.3.2, means that unless the pool is completely used, the rate at which requests are submitted will not change significantly. As a result, the increased latency is absorbed by the usage of this pool, resulting in a similar throughput value.

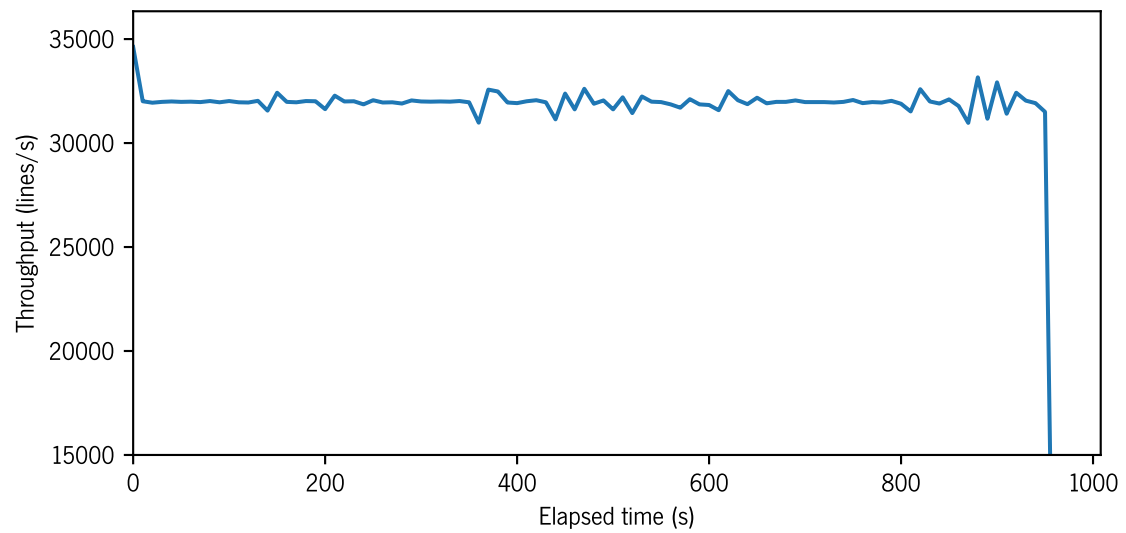


Figure 9: Throughput for limited network test targeting InfluxDB

Figure 9 shows that there is no observable decrease in throughput. The average latency for this test is 283.66 milliseconds, 129.66 milliseconds higher than without the added network limitations. This is higher than the expected 50 ms increase, possibly due to multiple roundtrips being required for each operation.

IoTDB

For IoTDB, the throughput values also stay similar to the test without added latency, with the observed average throughput being 178,522.16 lines per second.

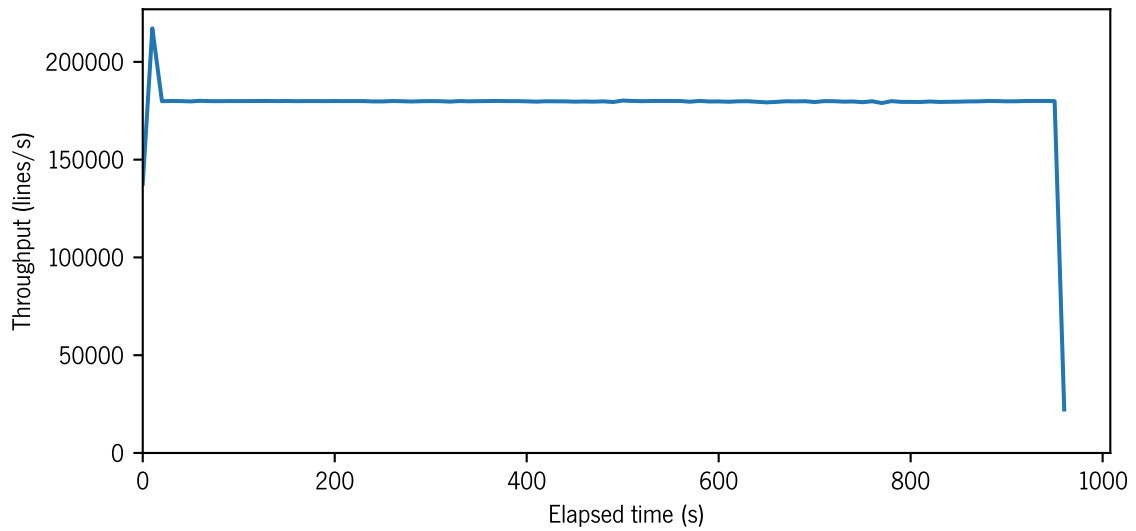


Figure 10: Throughput for limited network test targeting IoTDB

The throughput plot in Figure 10 shows that the added latency did not result in an observable decrease in throughput. The observed average latency is 112 ms, 94.67 ms higher than without added latency. This may be due to multiple roundtrips being required for each insertion operation.

These results show that the Orchestrator can in fact induce latency in the connections between the Clients and the database nodes.

4.2.4 Limited Resources

For the final feature validation test, the system resource limiting capabilities of the benchmark were tested. The database container's resources were limited to 3 CPU cores and 8 GB of memory, corresponding to half of the host machine's resources. The generated load was reduced for each database, to avoid tests with unnecessary test length due to the database being overloaded. The target throughputs were therefore reduced to 19,200 lines per second for InfluxDB and 96,000 lines per second for IoTDB.

InfluxDB

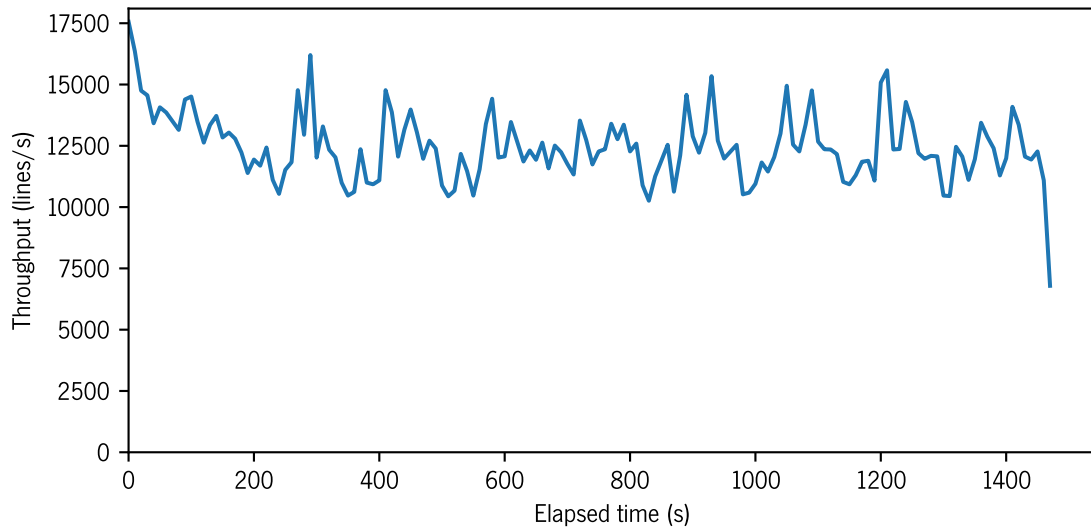


Figure 11: Throughput for limited resources test targeting InfluxDB

For InfluxDB, the observed average throughput is 12,530.66 lines per second, and the resource bottleneck can immediately be attributed to the CPU, which remains at maximum usage through the entirety of all test runs, as can be seen in Figure 12. Throughout the test, memory usage does not reach the maximum of 8 GB, as seen in Figure 13.

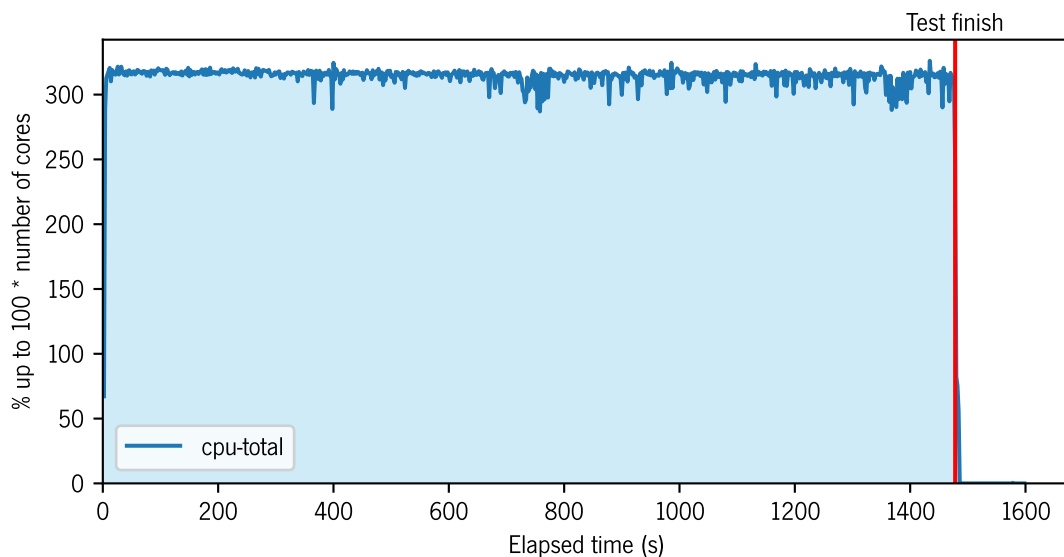


Figure 12: CPU usage for limited resources test targeting InfluxDB

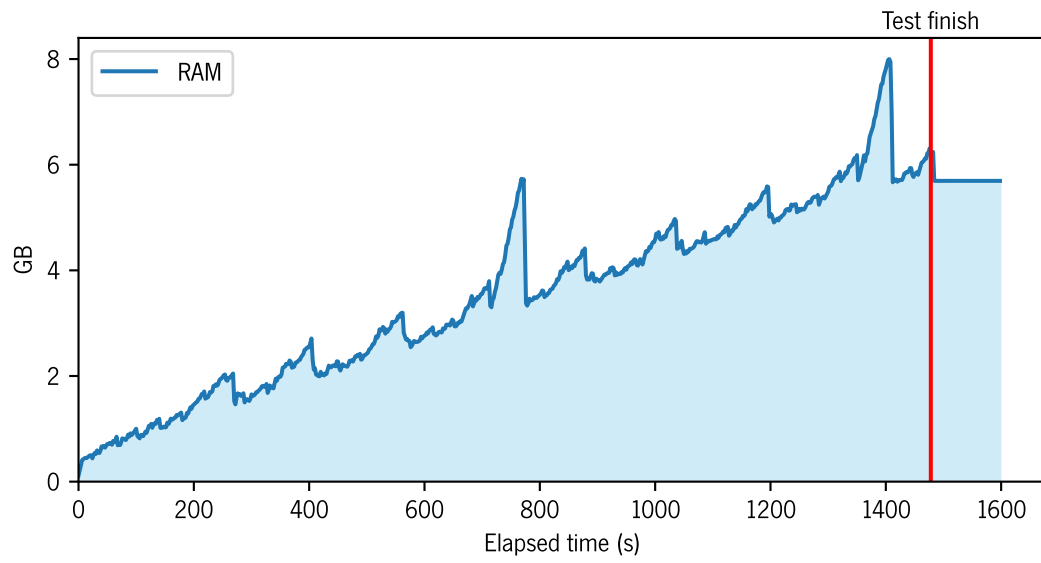


Figure 13: Memory usage for limited resources test targeting InfluxDB

Apache IoTDB

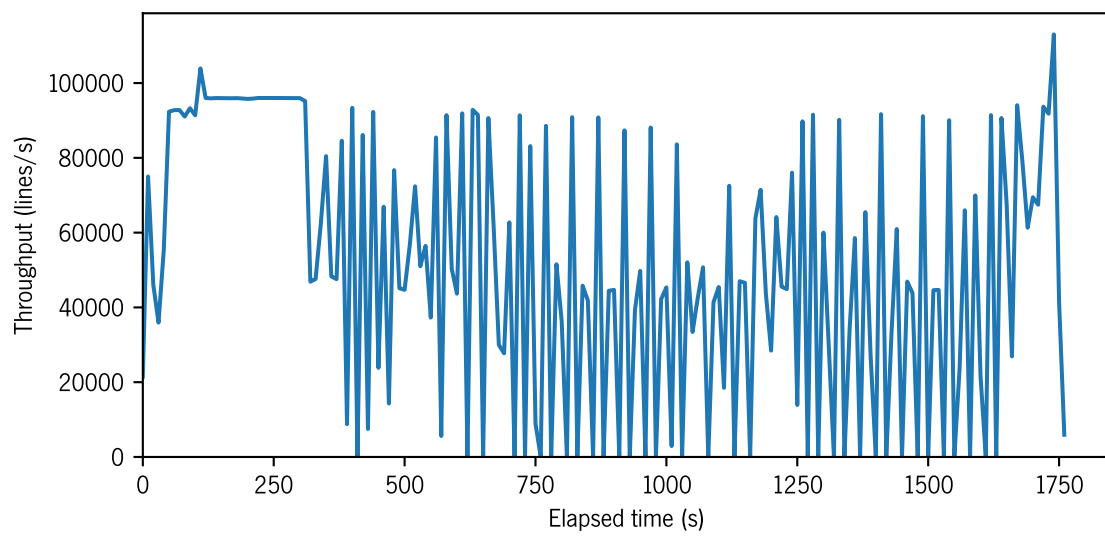


Figure 14: Throughput for limited resources test targeting IoTDB

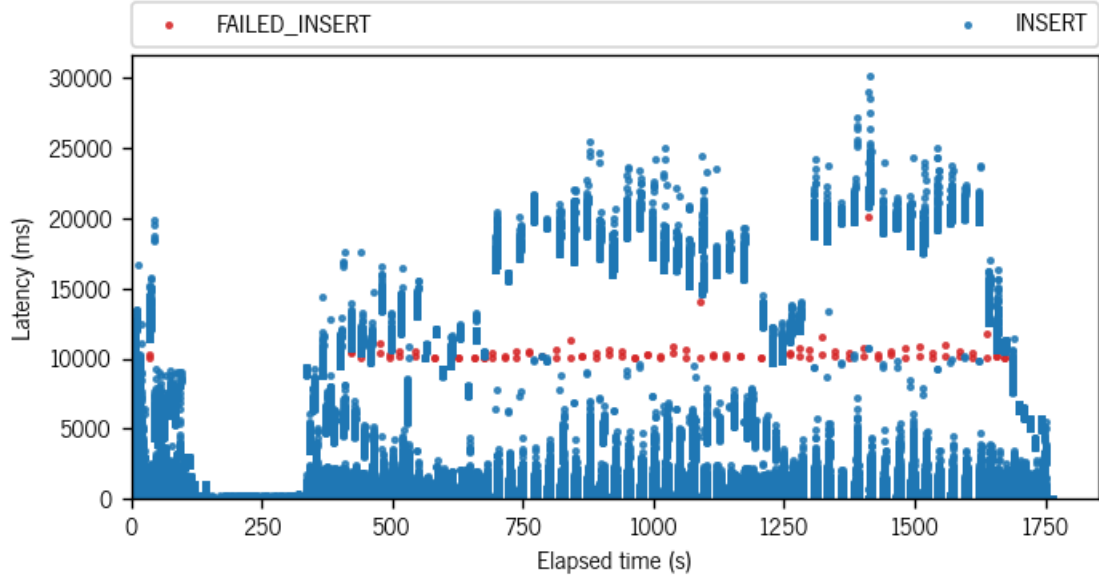


Figure 15: Insertion latency for limited resources test targeting IoTDB

For IoTDB, the observed average throughput is 49,677.15 lines per second. Looking solely at this value, one might assume that the database's performance is normal, with an expected reduction in performance. A closer look reveals all runs have over 100 failed insertion operations and the throughput plot in Figure 14 and latency scatter plot in Figure 15 show rather unusual behaviours. This is most likely due to a severe CPU bottleneck, evidenced by the almost constant 100% CPU usage as seen in Figure 16. Figure 17 shows that memory usage is also a possible limiting factor, staying at its maximum of 8 GB throughout the test.

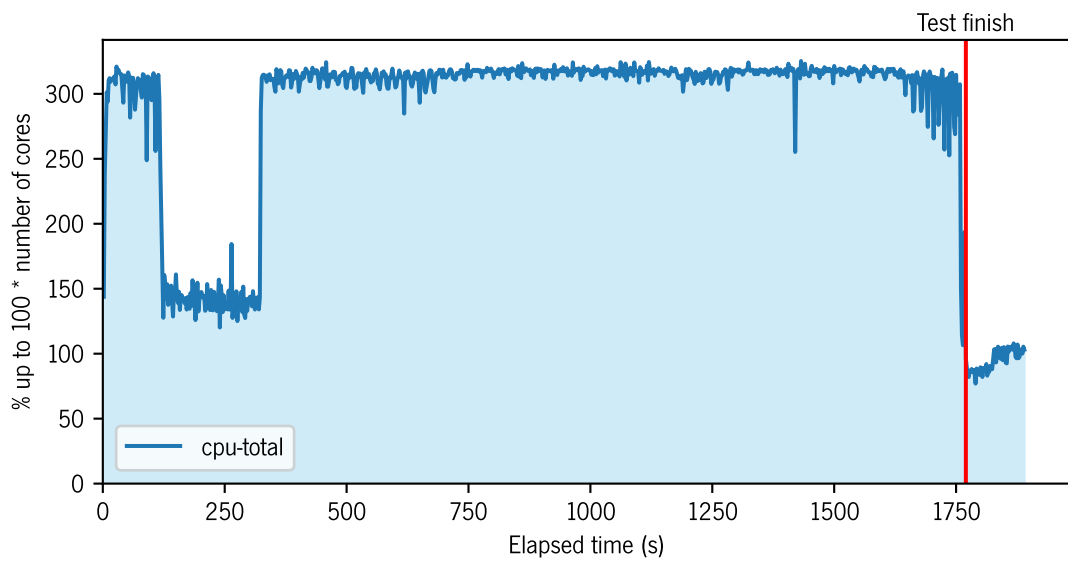


Figure 16: CPU usage for limited resources test targeting IoTDB

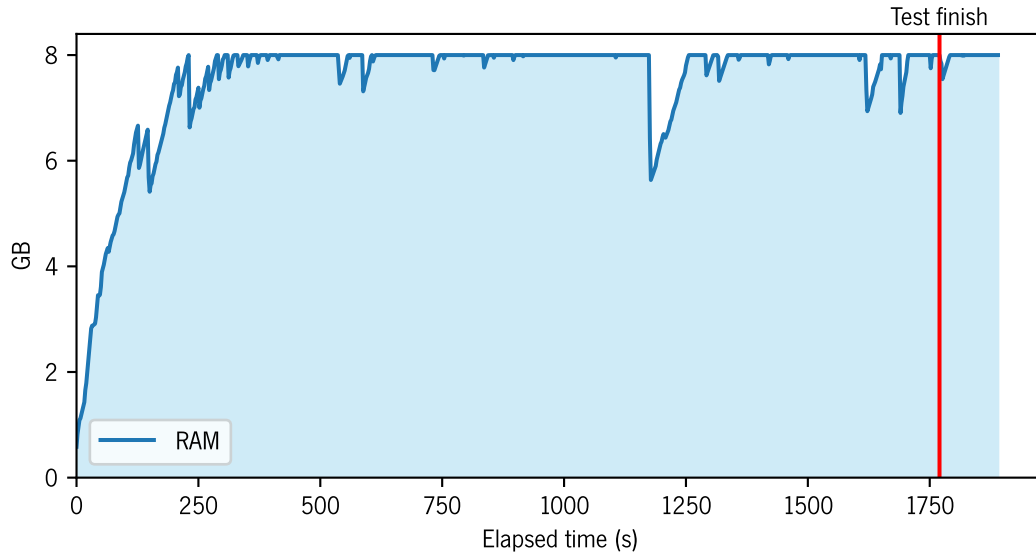


Figure 17: Memory usage for limited resources test targeting IoTDB

These results show that MulletBench is capable of limiting the resources available to a deployed database node, as well as submitting an expressive workload, allowing for a fine-grained analysis of the **SUT**'s behaviour.

4.3 Cloud Scenario

All the tests performed in this section use a cloud scenario configuration, meaning a single database node is used and all generated load is submitted to that node. This allows for the evaluation of each system in a scenario where data is offloaded directly to the cloud, without the intervention of any edge nodes. Tests were run for each of the workloads mentioned in Section 4.1.2.

4.3.1 Insertion Workload

This workload's goal is to evaluate a database's ability to handle an insertion-only scenario in which data is concurrently inserted from several sources and no queries are performed. This test corresponds to the control test previously presented. For both databases, all workers insert the same amount of data at the same rates. To vary the generated load, the number of workers is varied, simulating different amounts of devices accessing and inserting data.

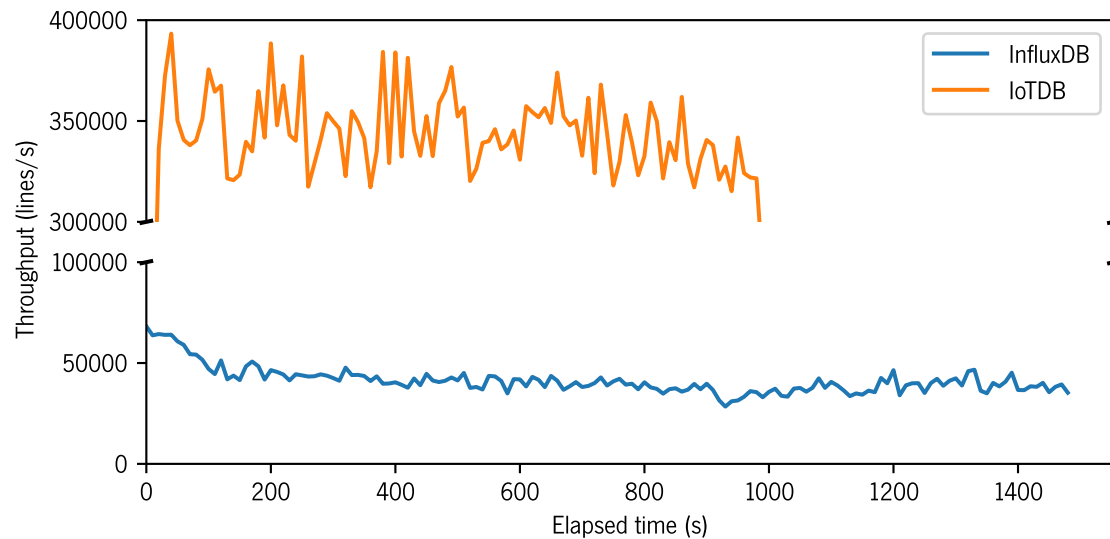


Figure 18: Throughput comparison for insertion workload in Cloud-only scenario

The throughput comparison in Figure 18 shows that IoTDB achieves a significantly higher throughput than InfluxDB, with an average of 320,640.8 lines per second compared to 41,094.19 lines per second, corresponding 7.8x the throughput. InfluxDB's test has a longer duration since its average throughput is further from its target throughput.

InfluxDB

Influx's throughput plot in Figure 18 shows that the target throughput is matched for a few seconds before dropping to around 40,000 lines per second, remaining there with some variation until the end of the test. Although the presented figure pertains to a single run, the same behaviour was present in all runs.

Looking at the resource monitoring graphs, it is not easy to tell which is the limiting factor for the database's performance. Since the IO usage monitoring in Figure 19 shows high IO throughput, the limit may lie in the number of IOps.

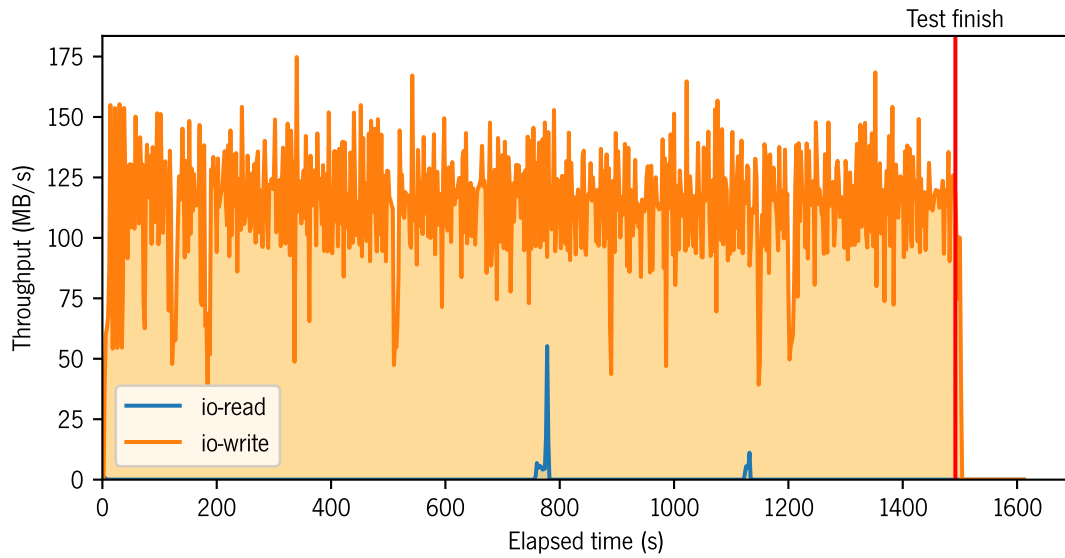


Figure 19: Disk IO usage for insertion workload targeting InfluxDB in Cloud-only scenario

IoTDB

IoTDB's disk IO usage in Figure 20 and memory usage in Figure 21 reveal that the memory limit is quickly reached and that there are constant disk reading operations, even though this is an insertion workload. This indicates the performance limitation may be coming from the limited memory, likely resulting in the use of swap memory.

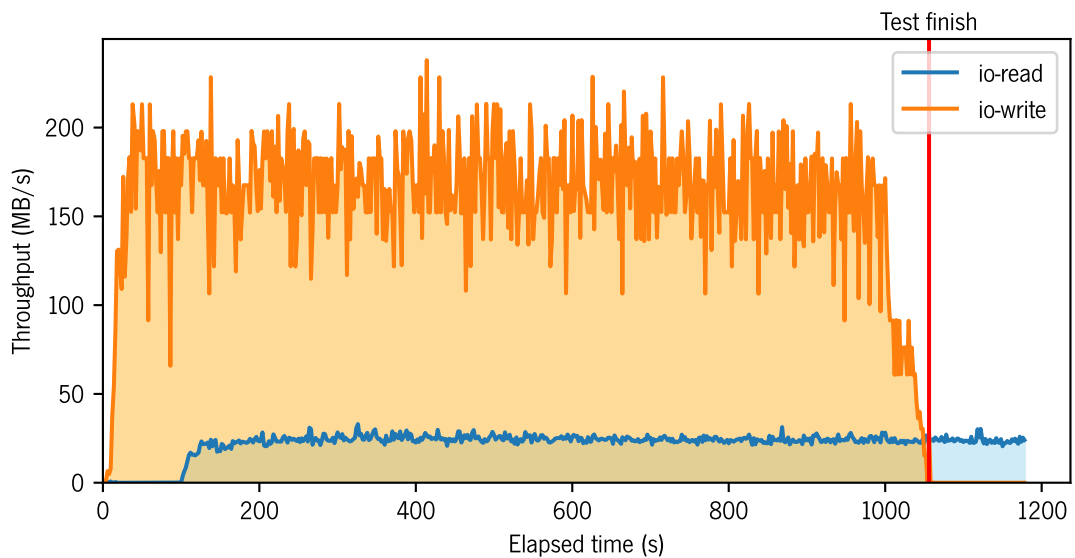


Figure 20: Disk IO usage for insertion workload targeting IoTDB in Cloud-only scenario

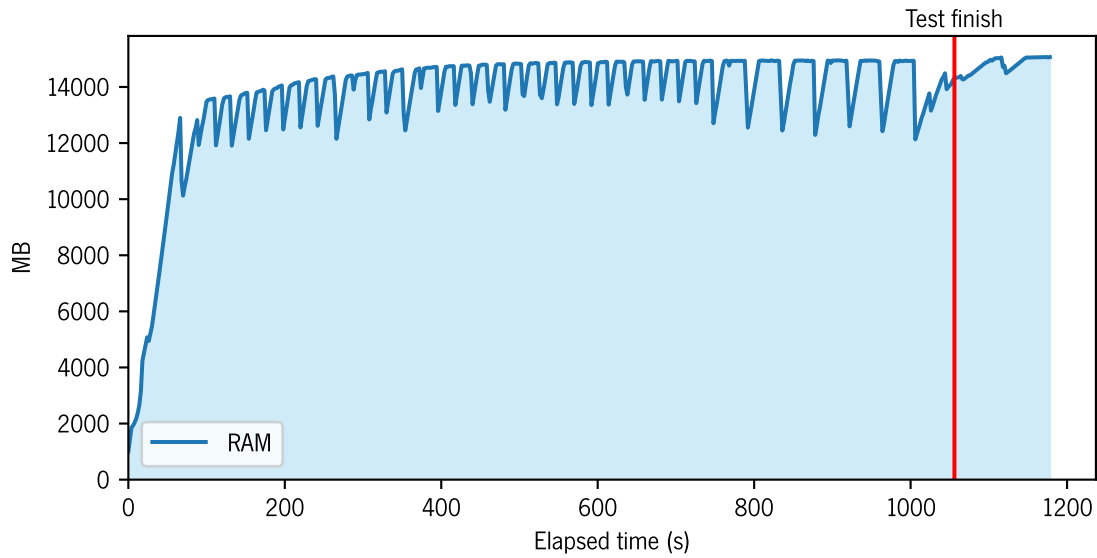


Figure 21: Memory usage for insertion workload targeting IoTDB in Cloud-only scenario

This workload shows that IoTDB is capable of handling a significantly higher insertion throughput than InfluxDB, in a single node deployment scenario.

4.3.2 Query Workload

The query workload aims to evaluate a database's ability to handle an analytical scenario in which previously inserted data needs to be analysed using common time series queries such as aggregation, down-sampling and filtering queries. For both databases, the same amount of data is initially ingested, ensuring the observed performance is dependant solely on the database's querying performance. This initial ingestion has a volume close to 120 GB assuming the original dataset's text format. In total, 207.4 million lines are inserted, corresponding to 6,222 million values. For this workload, the database is loaded once and the 3 tests were performed in sequence without reloading the database.

During test execution, queries were submitted at the same rate and with the same distribution for both databases. The used test configuration consisted of a single Client with a configuration as listed in Table 28.

This configuration results in a query being submitted by the Client every 2.5 seconds and for an estimated length of 16 minutes. With this workload, the analytical performance of a database can be measured by the latency with which it responds to each type of query, meaning a database's performance for a particular kind of query type can be measured and compared.

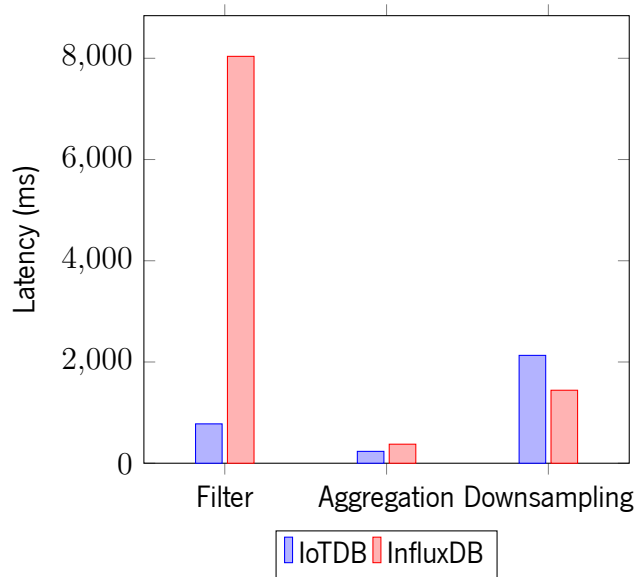


Figure 22: Query latency bar chart for query workload in Cloud-only scenario

Figure 22 shows that InfluxDB has higher average latency for filter queries, while being only slightly worse for aggregations and slightly better for downsampling, when compared to IoTDB. Although InfluxDB's outlier filtering queries take on average more than 10x longer than IoTDB's (8038 ms compared to 778 ms), looking at the median and 10th percentile values reveals something different, with these being, respectively, only 80% and 28% higher for InfluxDB than IoTDB (1433.99 ms compared to 798.50 ms and 818.33 ms compared to 638.67 ms). This indicates that some queries with very high latency are skewing the average values.

Another interesting observation from the results is that, although InfluxDB performed worse in outlier filter and aggregations, results show on average 32% lower latency on downsampling queries.

InfluxDB

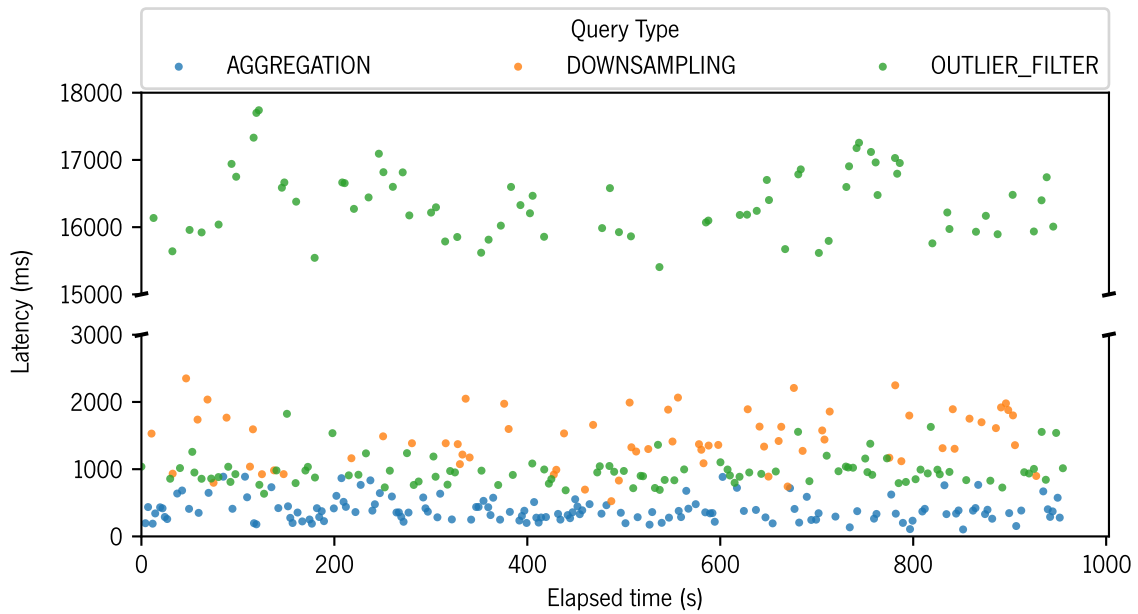


Figure 23: Query scatter for query workload targeting InfluxDB in Cloud-only scenario

The query scatter plot in Figure 23 shows how InfluxDB's outlier filtering queries are clearly divided into two layers by their latency, one of them having significantly higher latencies than any other queries.

IoTDB

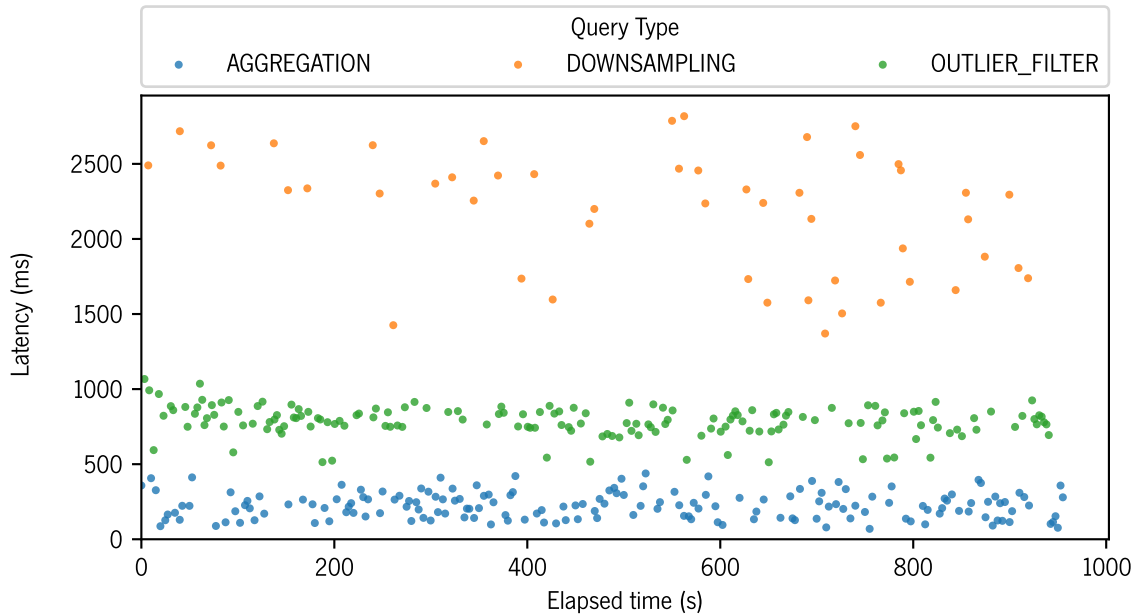


Figure 24: Query scatter for query workload targeting IoTDB in Cloud-only scenario

IoTDB's query scatter plot is also presented in Figure 24. It shows that all query types consistently have similar latencies, with no clear outliers. Unlike InfluxDB, there is no clear split in outlier filtering query latencies.

This workload shows that IoTDB has a significant lower average latency for filter queries and a slight advantage for aggregations, while InfluxDB has slightly lower downsampling query average latency. Results also show that both databases should be able to deal with a higher rate of queries at the cost of higher latencies as they both were capable of responding to the submitted query rate.

4.3.3 Mixed Workload

This workload corresponds to a realistic scenario in which queries are being performed in parallel with new data being inserted. The workload also includes an initial population stage, consisting of insertion only, and allowing for the database to already populated when the mixed stage starts and queries start being performed.

The same exact configuration was used for both databases, both in volume of data and in rate of insertion and querying. Since the benchmark submits a constant rate of insertions, the database's performance can be measured by whether it can successfully handle that insertion throughput. At the same

time, its performance can also be evaluated by the latency with which it responds to querying operations. Since the benchmark allows the analysis of querying latency per query type, a database may be better in certain types of queries while being worse in others.

The preload stage consists of 4 Client instances with a configuration as listed in Table 29. The mixed stage consists of 4 insert Clients and 1 query Clients. The insert Clients in the mixed stage use a configuration as listed in Table 30. The query Client uses a configuration as listed in Table 31.

The throughput comparison in Figure 25 shows that both databases are able to handle the constant data throughput during the initial population stage, but InfluxDB is unable to do so during the mixed stage.

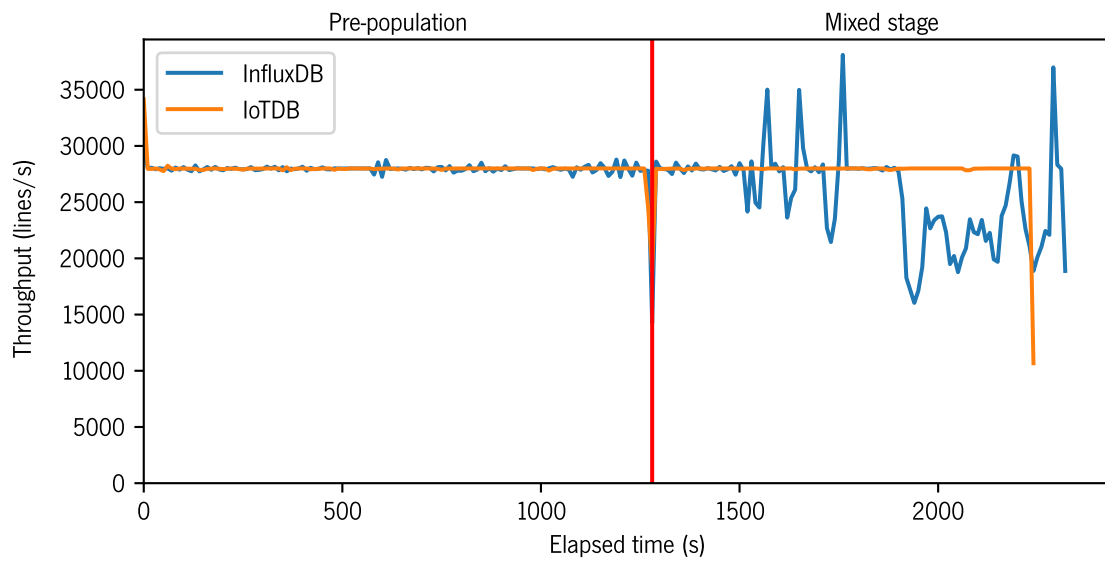


Figure 25: Throughput comparison for mixed workload in Cloud-only scenario

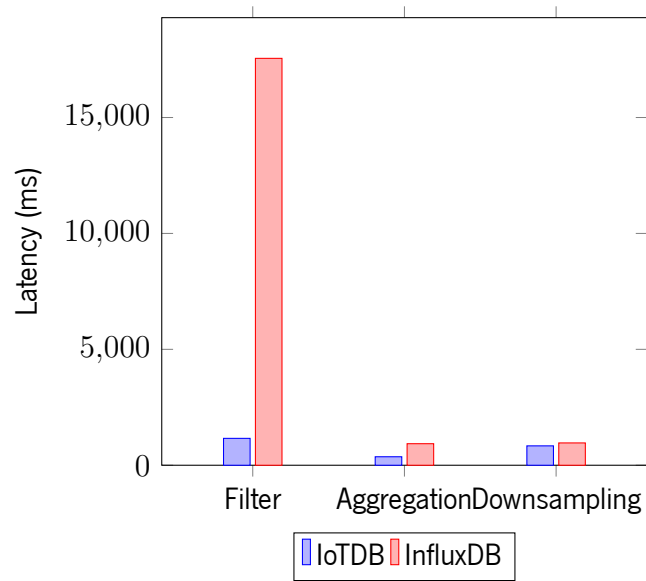


Figure 26: Query latency bar chart for mixed workload in Cloud-only scenario

InfluxDB

Although data insertion is generally stable throughout the preload stage, during the mixed load it drops to an average of 26,036.47 lines per second, below the targeted 28,000.

For InfluxDB, the average latency of downsampling operations is very similar to the average latency of aggregations. This is counterintuitive since a single downsampling operation essentially consists of multiple aggregation operations. Although the average filter latency is 17,552.36 ms, the median is 4567.29 ms, indicating the presence of high values skewing the average. This is also evidenced by the 90th percentile values, and can be seen in the query scatter plot in Figure 27. Note that this plot in particular uses a logarithmic scale on the y-axis to increase its readability.

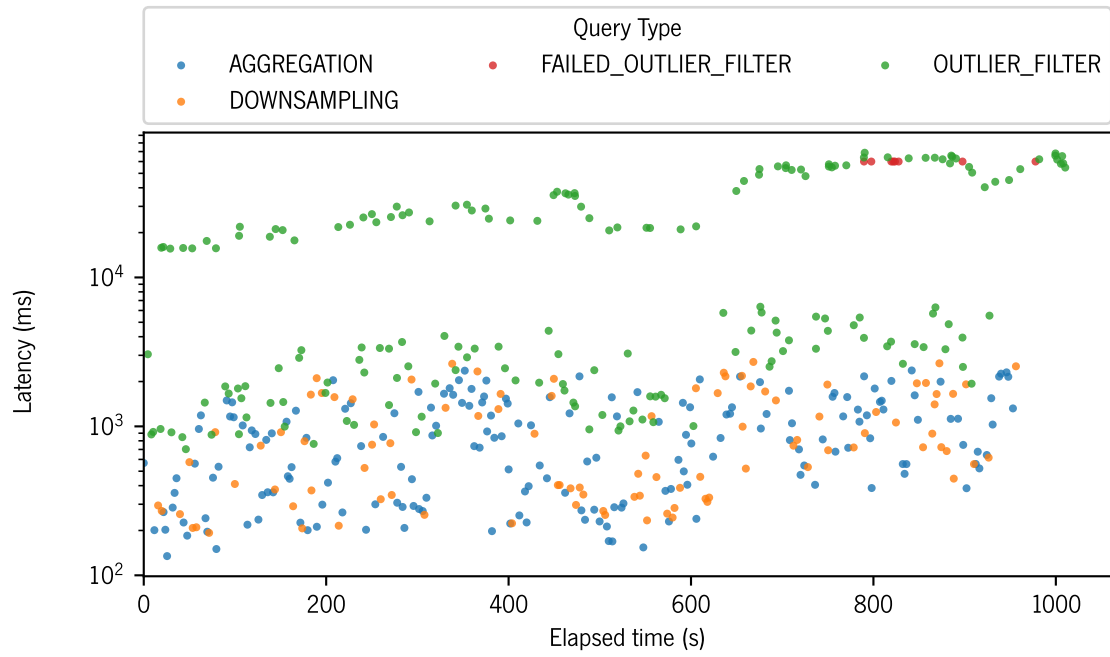


Figure 27: Query scatter plot for mixed workload targeting InfluxDB in Cloud-only scenario

CPU monitoring in Figure 28 shows very high CPU usage, getting up to over 83% usage (5 core usage out of 6 cores).

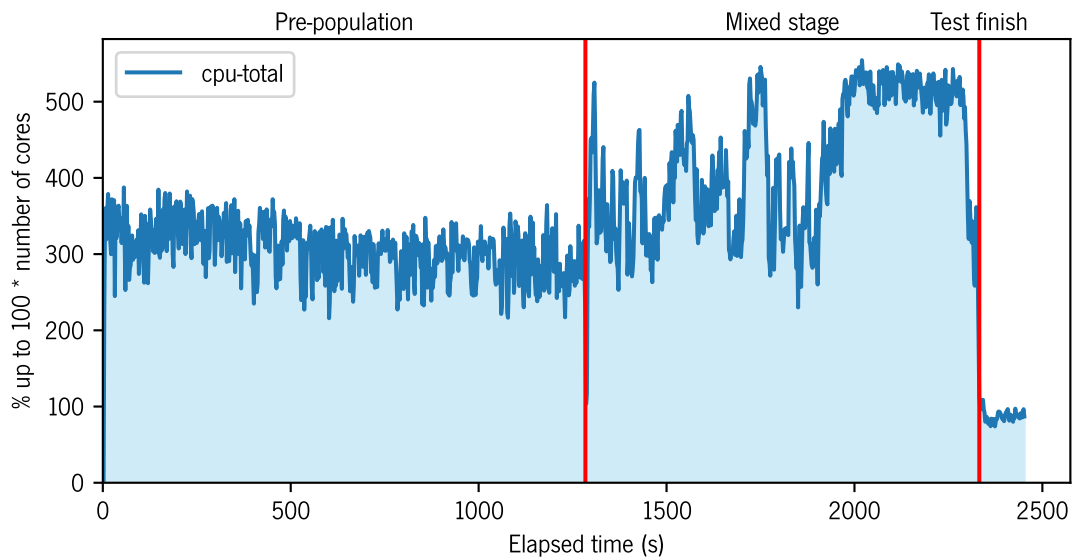


Figure 28: CPU usage for mixed workload targeting InfluxDB in Cloud-only scenario

Apache IoTDB

Results for IoTDB showed a much more stable throughput, remaining pretty much constant throughout both stages. Note that the dip halfway through the test is due to the switch from one stage to the next.

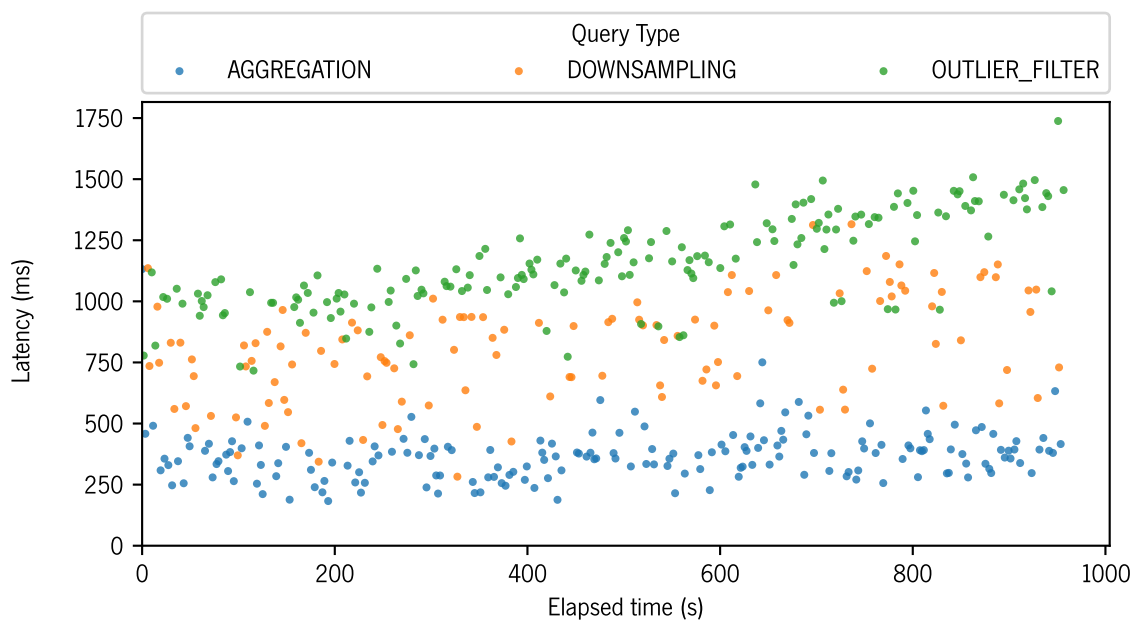


Figure 29: Query scatter plot for mixed workload targeting IoTDB in Cloud-only scenario

In addition, the query scatter plot in Figure 29 also shows more consistent results in comparison to InfluxDB, without significant outliers in any query type. Overall, IoTDB achieves 93.4% lower filter latency, 60.6% lower aggregation latency and 13.2% lower downsampling latency when compared to InfluxDB.

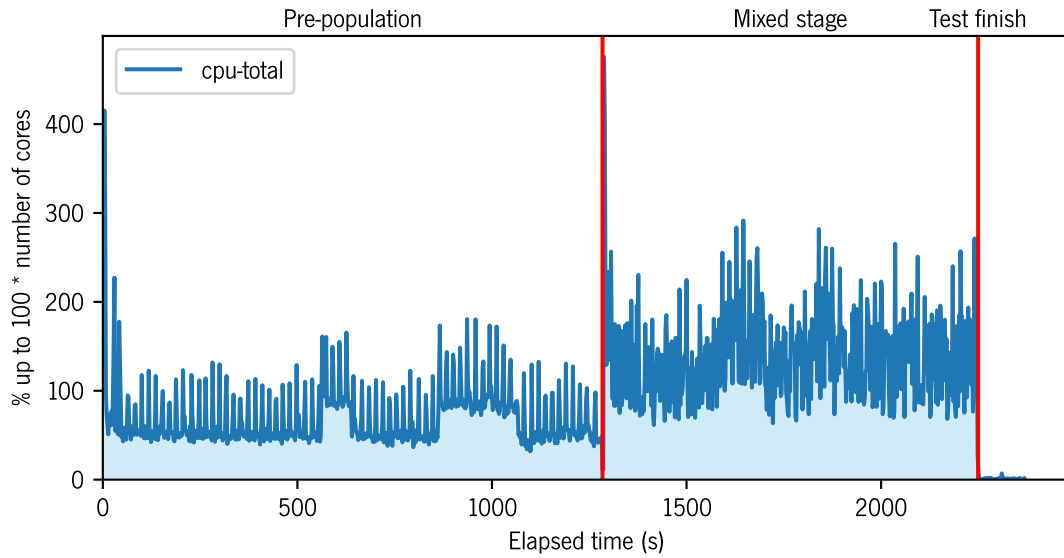


Figure 30: CPU usage for mixed workload targeting IoTDB in Cloud-only scenario

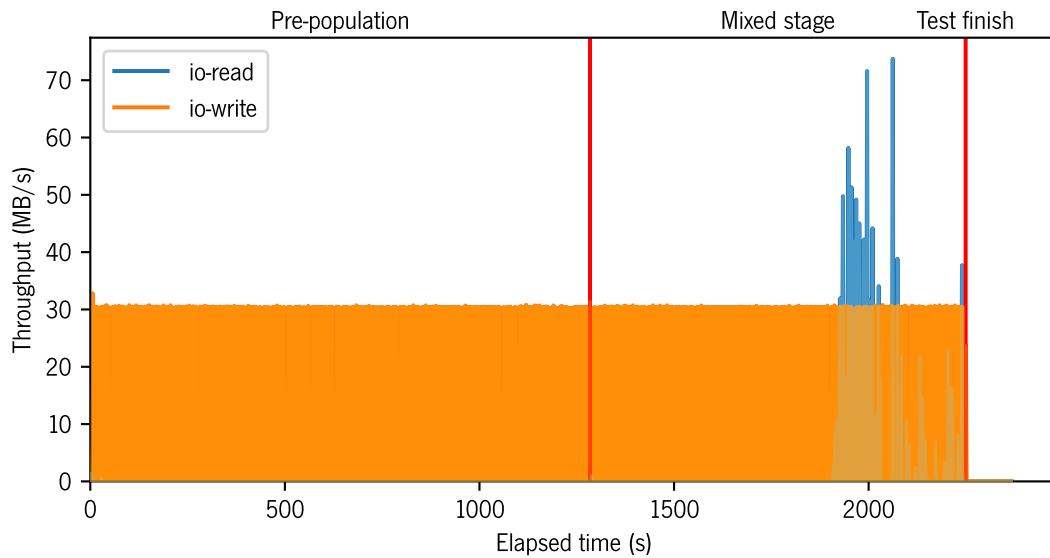


Figure 31: Disk IO usage for mixed workload targeting IoTDB in Cloud-only scenario

An analysis of CPU usage in Figure 30 and disk IO usage in Figure 31 indicates that, for this scenario, IoTDB still remains far from a significant bottleneck, and could be able to handle more data insertion and/or a higher query rate.

This workload shows that InfluxDB struggles to handle the targeted throughput and performs worse than IoTDB for all query types, including 1413.6% higher average filter latency than IoTDB. Detailed query statistics allow for the conclusion that this is due to very high latency queries skewing the average. IoTDB,

on the other hand, is able to keep a constant throughput under the used query rate.

4.4 2 Edge + Cloud Scenario

The 2 Edge + Cloud attempts to emulate a typical multi-layer solution, in which multiple edge nodes, located closer to the sensors generating data, are responsible for storing each area's sensor data, and replicate it to a single central cloud node. The tests for this scenario use 2 edge nodes with a resource limiting profile configured to emulate a Raspberry Pi 4, with parameters as listed in Table 6, and a single cloud node, without resource limiting.

Each of these edge nodes connects and sets up replication to the cloud node. The end result is that data inserted through the edge nodes and is then replicated to the cloud node, using each database's replication mechanism.

The edge nodes can also perform downsampling, reducing the amount of data that needs to be replicated to the cloud node. InfluxDB supports continuous downsampling through the use of periodic tasks, and so tests were performed in 3 possible downsampling scenarios: no downsampling, replicating all raw data to the cloud node, downsampling once per second (100 to 1 downsampling ratio) and downsampling once every two seconds (200 to 1 downsampling ratio). Although IoTDB supports continuous queries, these cannot currently be used to perform aggregations on all available measurements without explicitly listing them individually. This is due to the fact that, when using *SELECT INTO* operations, if a query uses an aggregation function, its target measurement cannot use variable placeholders, and can therefore not be generalized. As a result, tests for IoTDB were all performed without downsampling.

4.4.1 Insertion Workload

Like with the cloud scenario, the insertion workload's purpose is to evaluate a database's pure insertion performance. In this scenario, the replication process might also have an effect on results if it incurs a high performance cost. Multiple concurrent Clients connect to and submit data into each of the edge nodes, with each Client containing multiple workers inserting the same amount of data at the same rates. Both edge nodes are subjected to the same load.

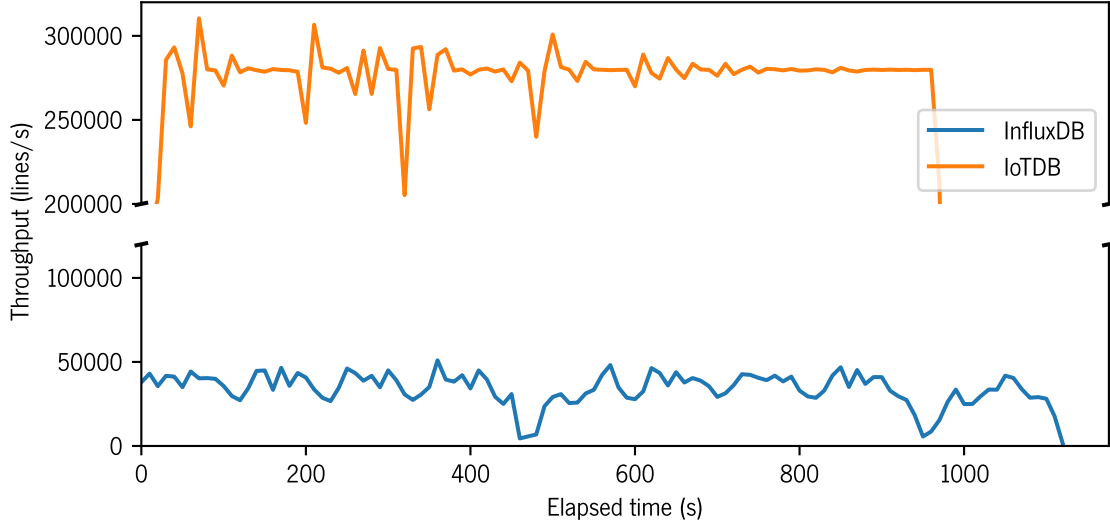


Figure 32: Throughput comparison for insertion workload in 2 Edge + Cloud scenario

Figure 32 shows the comparison of the throughput's aggregate sum with the downsampling configuration that results in the highest throughput for each database. For InfluxDB, this corresponds to 2 second downsampling, meaning that for each sensor, all readings within every 2 second interval are downsampled into a single reading. This downsampled data is then replicated to the cloud, greatly reducing the amount of replicated data. Using 2 second downsampling, InfluxDB achieves an average throughput of 33,921.17 lines per second, while IoTDB manages to reach 272,602.7 lines per second without downsampling.

InfluxDB

For InfluxDB the test configuration is composed of 4 Clients with configuration as listed in Table 32, totalling a target throughput of 32,000 lines per second. Since InfluxDB supports continuous downsampling, tests were performed with 3 different downsampling configurations: no downsampling, 1 second downsampling and 2 second downsampling. Tests with 1 and 2 second downsampling showed the latter results in 12.5% higher throughput. Although this is not an insignificant difference, both tests result in essentially the same behaviour and so only the results for the no downsampling and 2 second downsampling configurations are presented.

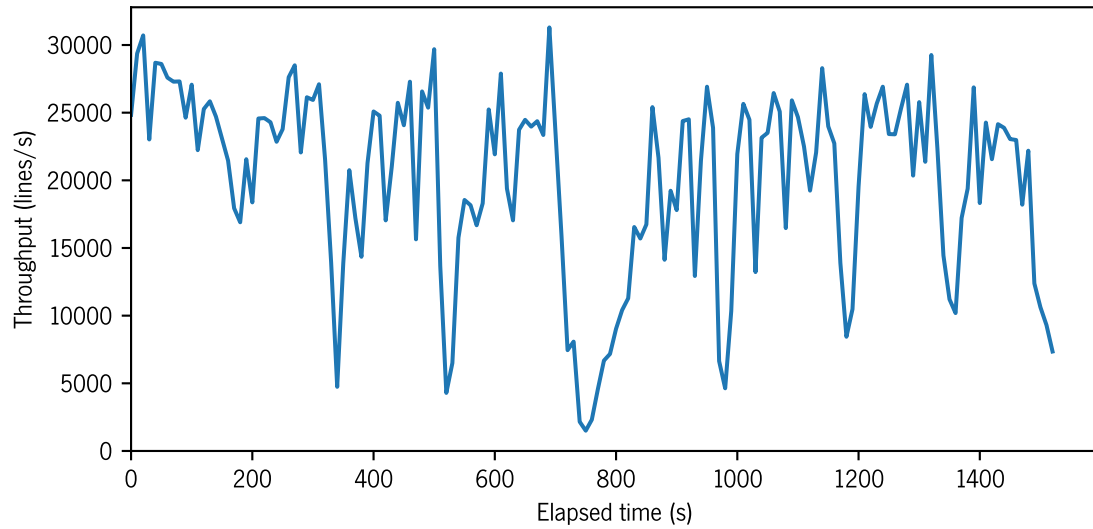


Figure 33: Throughput plot for insertion workload targeting InfluxDB with no downsampling in 2 Edge + Cloud scenario

No downsampling Replicating raw data instead of downsampled data, InfluxDB manages to achieve a throughput of 20,129.40 lines per second. Looking at the disk IO usage in Figure 34 and memory usage in Figure 35 allows us to clearly identify the **TSM** compactions InfluxDB performs during execution, as they coincide with the spikes in latency in Figure 36. These compactions are a mechanism used by InfluxDB (and other databases using **Log-Structured Merge-Tree (LSM)** tree-based storage engines) to free memory or disk by converting write-optimized files to compressed and read-optimized files in disk. This can be seen through the regular interval spikes in IO writes. These spikes can have different lengths depending on the compaction levels, which are increasingly more demanding, since the higher the compaction level, the more data they need to compact. The relation between compactions in **LSM**-based stores and throughput latency spikes has already been widely studied (Balmau et al. [2019]).

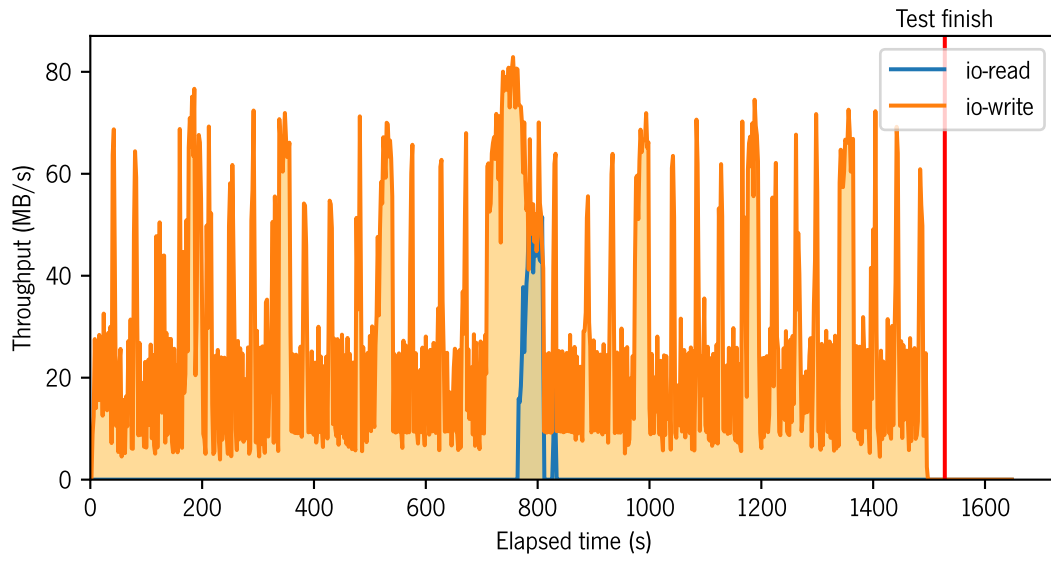


Figure 34: Edge node disk IO usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

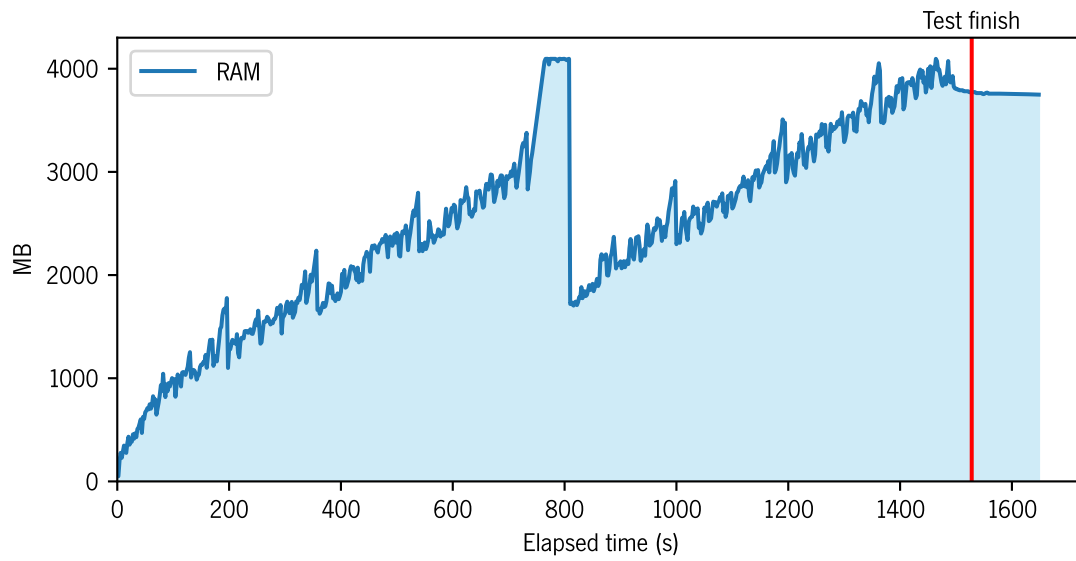


Figure 35: Edge node memory usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

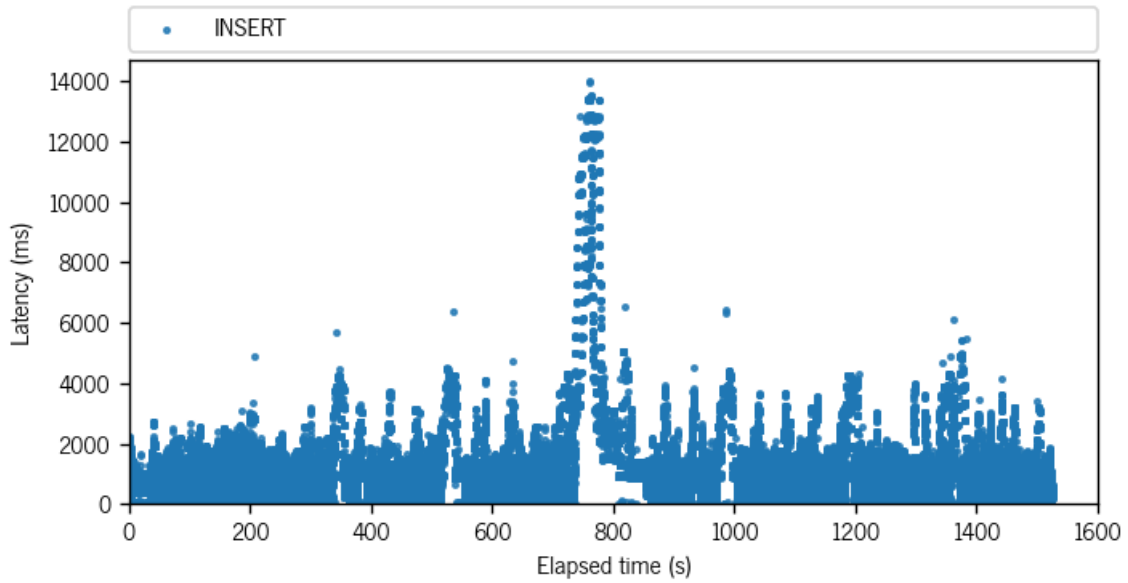


Figure 36: Latency scatter plot for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

Looking at the network usage for an edge node in Figure 37 and for the cloud node in Figure 38 on the section after the test execution has finished shows there is still activity between the edge and cloud nodes.

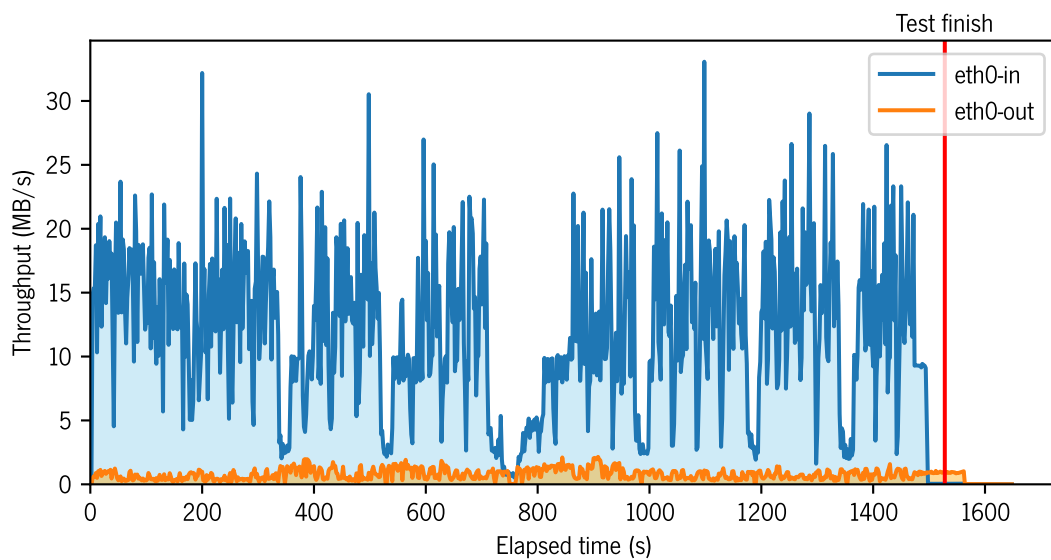


Figure 37: Edge node network usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

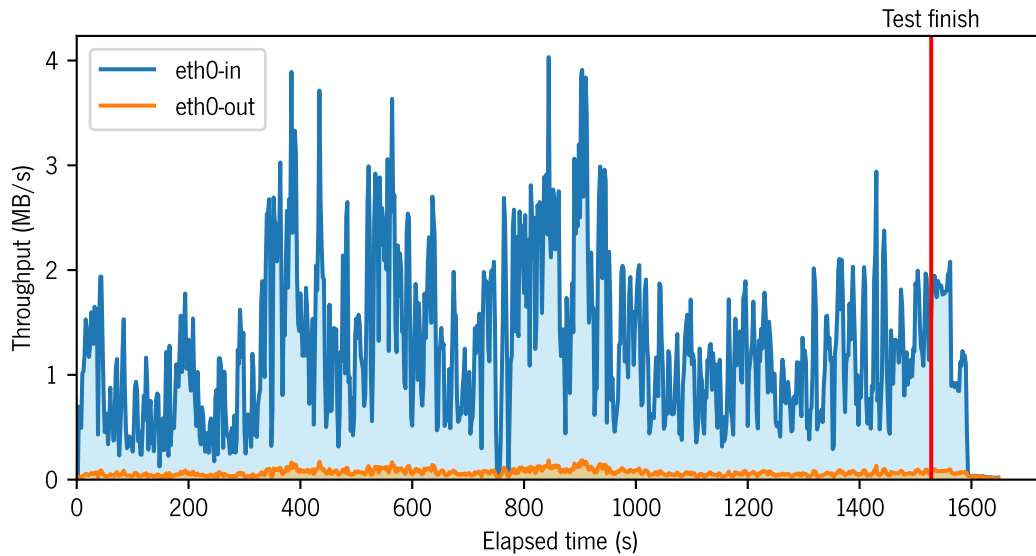


Figure 38: Cloud node memory usage for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

This is due to InfluxDB using a replication queue, where ingested data is placed to later be replicated. A further analysis of InfluxDB's logs revealed this queue is filling up during the test and not all data inserted at the edge layer is being replicated to the cloud node. This proves to be an issue as from the client's side all insertions are confirmed by the database, even though the data has not been replicated as expected, leading to data loss if the edge nodes' bucket is configured to only retain data for a certain period of time through the setting *bucket-retention-period*. This behaviour has been reported to the developers through a GitHub issue¹.

Using downsampling may mitigate the replication queue issue as the volume of replicated data decreases significantly. Even so, there is no guarantee of replication if the volume of downsampled data still exceeds what the replication queue can handle.

2 second downsampling Using downsampling, the same patterns in disk IO usage in Figure 39 and memory usage in Figure 40 can be observed on the edge nodes, indicating the same compaction limitations. However, the observed throughput is significantly higher, reaching 33,921.17 lines per second, a 68.6% increase over the no downsampling scenario, indicating that the replication of high volumes of data has a significant performance impact.

¹ InfluxDB replication issue: <https://github.com/influxdata/influxdb/issues/24434>

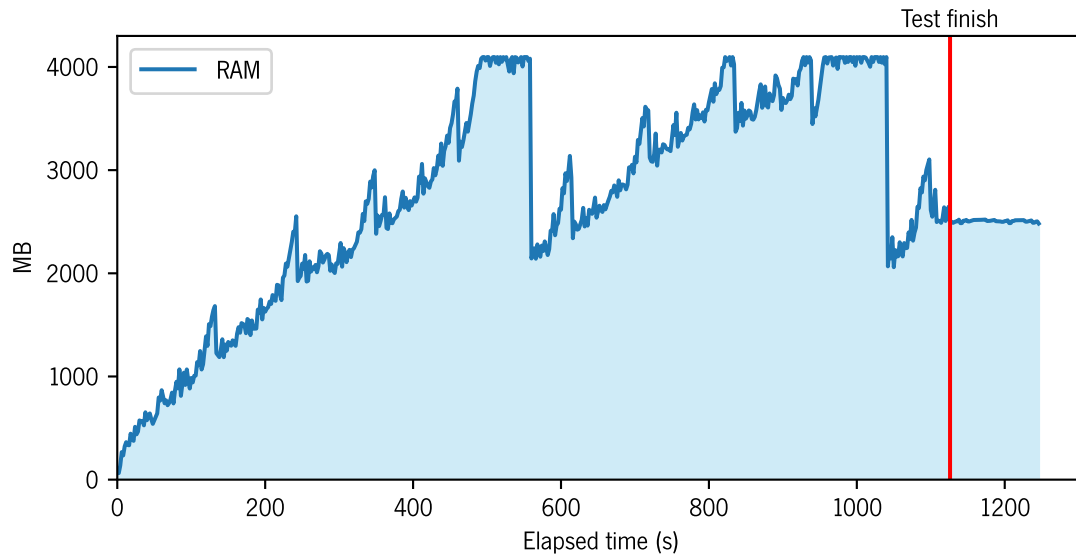


Figure 39: Edge node memory usage for insertion workload targeting InfluxDB with 2 second downsampling in 2 Edge + Cloud scenario

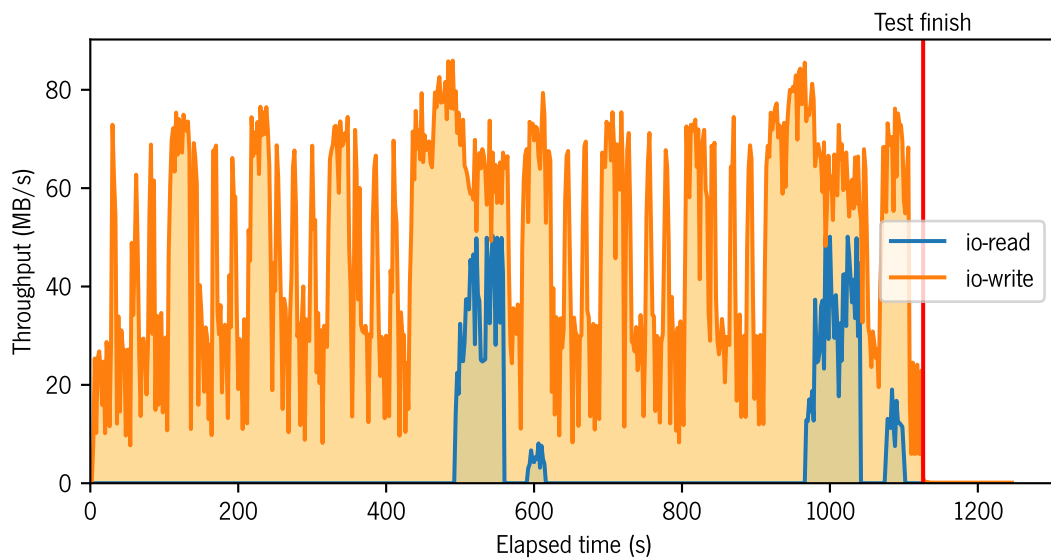


Figure 40: Edge node disk IO usage for insertion workload targeting InfluxDB with 2 second downsampling in 2 Edge + Cloud scenario

IoTDB

For IoTDB the test configuration contained 4 Clients with configurations as listed in Table 33 and totalling a target throughput of 280,000 lines per second.

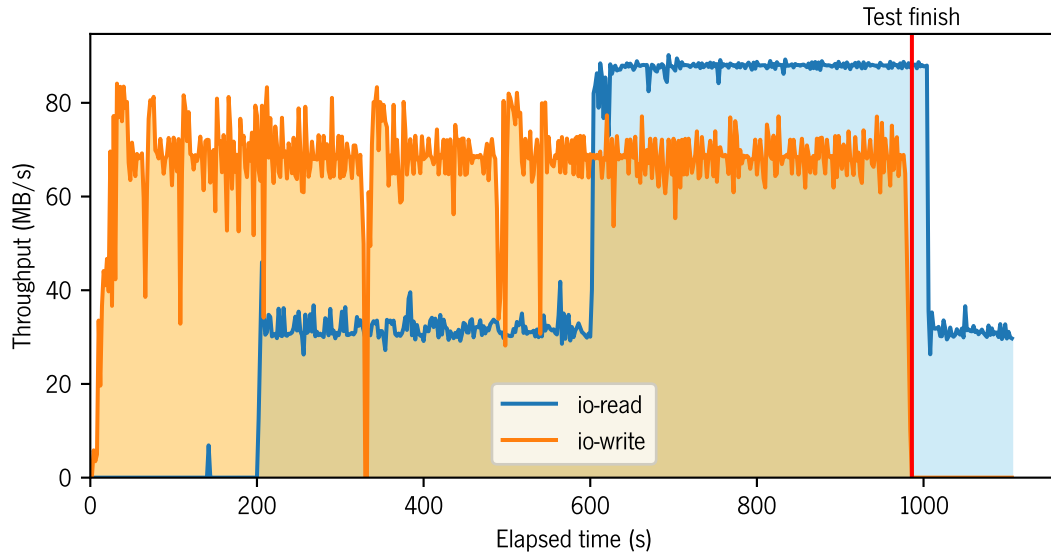


Figure 41: Edge node disk IO usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

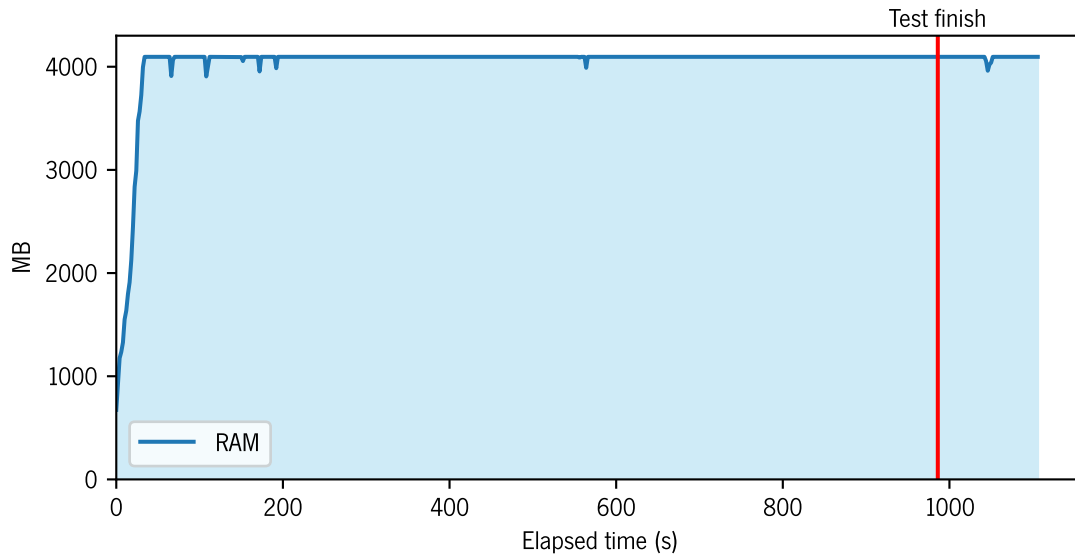


Figure 42: Edge node memory usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

This configuration seems to be limited by the edge nodes' disk IO and memory limits, as shown in Figures 41 and 42, respectively, with both resources being fully utilized for the duration of the test. As such, a deployment with more edge nodes should be able to yield improved results.

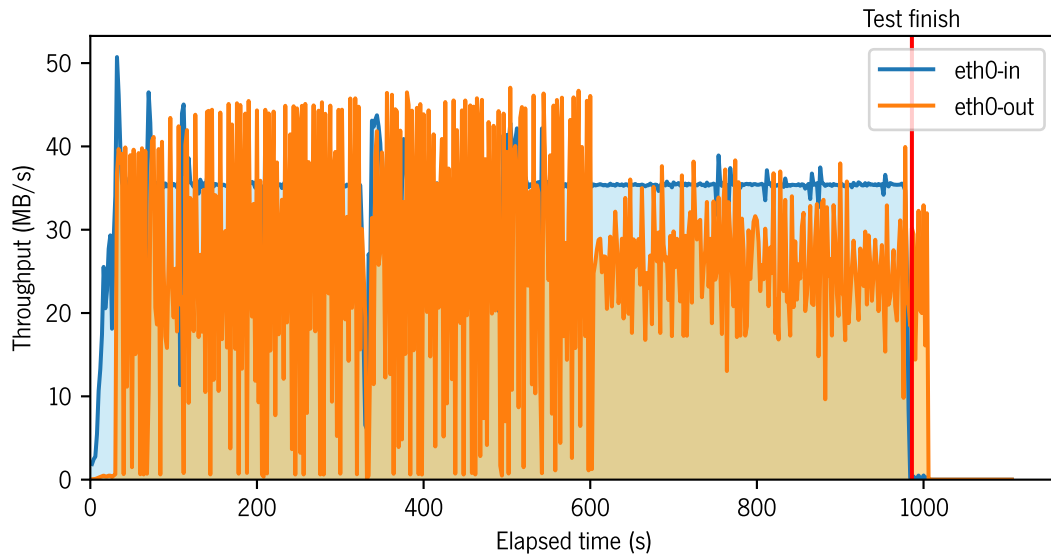


Figure 43: Edge node network usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

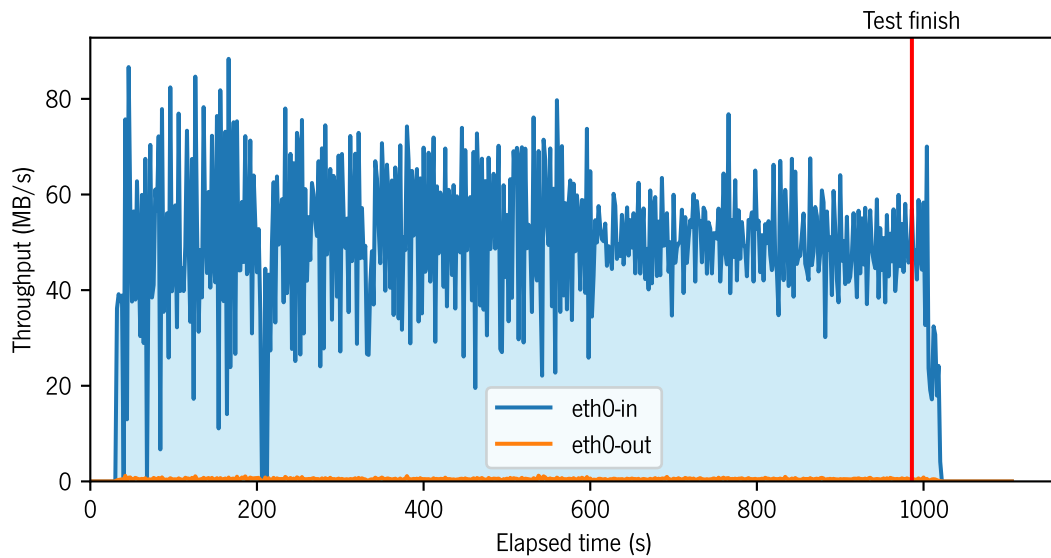


Figure 44: Cloud node network usage for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

Network usage in an edge node and the cloud node, shown in Figure 43 and 44 respectively, shows that after the test execution is finished there is still activity between the edge and cloud nodes. This indicates that replication is still ongoing. Looking at the data present in the cloud node reveals that not all data is successfully replicated, even though all insertion operations are confirmed by the database. This results in some data loss at the cloud node.

When compared to the cloud scenario, this deployment does not prove to be beneficial for insertion

performance, resulting in a 17.5% and a 12.0% drop in average throughput for InfluxDB and IoTDB, respectively. However, this workload shows how downsampling can be used to increase throughput by reducing the amount of replicated data, with InfluxDB showing a 53% increase in throughput when using 2 second downsampling.

4.4.2 Query Workload

For the query workload, the database nodes are initially loaded with data, over which queries performed. The amount of data used for pre-population is the same between the tested databases, as to allow comparisons between them, but also the same as in the previously conducted Cloud scenario tests. This achieves a scenario with the same amount of sensors and queries but a different deployment strategy, enabling their comparison.

To enable both of these comparison vectors, the queries performed and their rate of submission needs to be the same, so that the load induced is the same and the resulting query latencies can be compared. However, since this scenario contains heterogeneous nodes, this does not translate directly to the same amount of queries.

As each edge node contains half the ingested data, each query performed in an edge node only comprises half as much data as if it is performed in the cloud node. To generate an equivalent query workload, the query load is split equally between the edge and cloud layers, with all nodes on both layers being submitted the same amount of queries. The configuration used by each Client is listed in Table 34. The end result is that 50% more queries are performed when compared to the Cloud scenario, but of which 66% only comprise half the data inserted, essentially distributing the same load over more nodes.

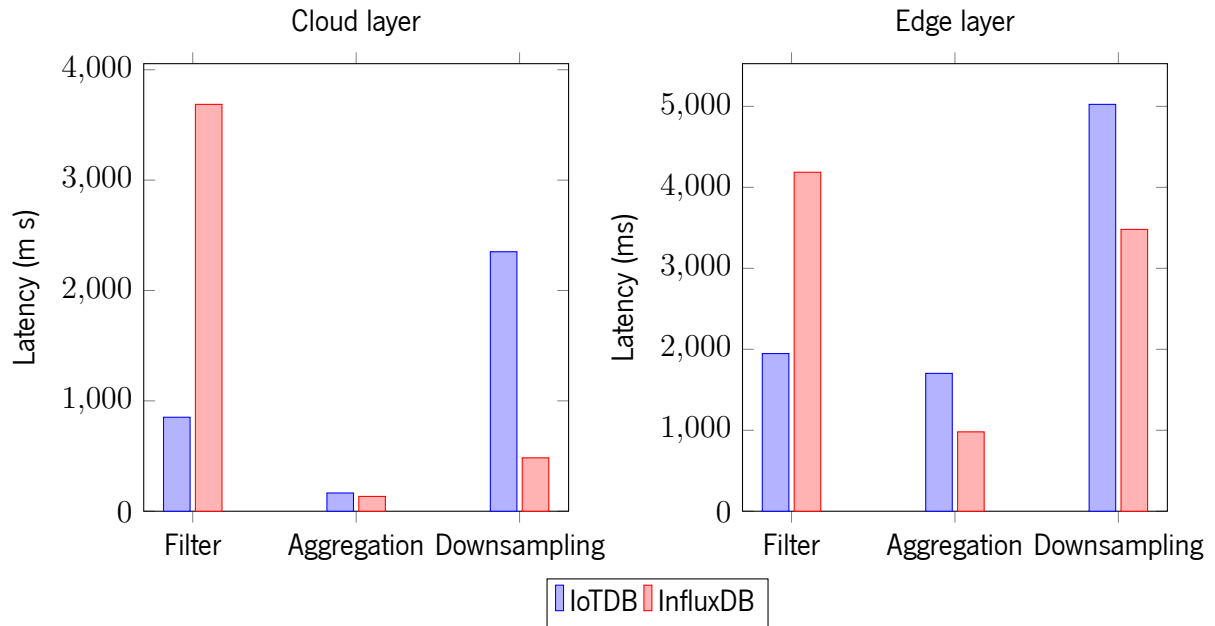


Figure 45: Query latency for query workload in 2 Edge + Cloud scenario

Looking at the edge layer average latencies in Figure 45, InfluxDB's average filter latency is 115.5% higher than IoTDB's, while its aggregation and downsampling latencies are 42.4% and 30.7% lower, respectively.

InfluxDB

Since in this workload the dataset is initially loaded with its original timestamps, testing with downsampling is not possible as the downsampling tasks are applied to data which have timestamps in the past x units of time, and not data which were inserted in the past x units of time. As such, the only possible test was performed without downsampling.

Due to the issue with the replication queue previously identified in Section 4.4.1, the initial data population does not result in all the data reaching the cloud node. This can also be confirmed by looking at the average volume returned by each filter query at the cloud layer, which dropped 48.8% from 18,944.4 entries per query to 9,689.9 entries per query, indicating this database node is not completely populated. The cloud layer's query latency performance should then not be considered valid for comparison.

At the edge layer, the average latency for filter queries decreased 47.9% when compared to the cloud-only deployment, while for aggregation and downsampling queries average latency increased 160.78% and 141.37%, respectively.

IoTDB

When analysing the metrics corresponding to the cloud layer, even though the average aggregation latency is lower when compared to the cloud scenario, at 165.24 ms versus 234.28 ms, the same cannot be said for the filter and downsampling queries. This is unexpected since in this scenario the cloud layer is subjected to half the load when compared to the cloud scenario. Further investigation on how the population data is stored on all nodes, shows that on the cloud node, all data received from the edge nodes is stored on delayed or unsequenced TsFiles. Delayed TsFiles are used when IoTDB identifies received data as being out of order compared to what is stored on the normal TsFiles, using these secondary files to avoid a more expensive write operation at the cost of more expensive query operations and the need for compactions. This mechanism can be beneficial when a small portion of data is received out-of-order. It is currently unclear as to why data received from the edge nodes is identified by the cloud node as being delayed. This behaviour has been reported to the developers in a GitHub issue².

At the edge layer, query performance is worse for all types when compared to the cloud-only deployment, with 150.4%, 626.8%, and 135.8% increase in outlier value filters, aggregations, and downsampling queries, respectively.

This workload shows that InfluxDB achieves lower average latencies than IoTDB for aggregations and downsampling queries, while IoTDB has lower latency for filter operations, when subjected to a query load.

4.4.3 Mixed Workload

As with the query workload, the mixed workload is configured such that its results could be used for comparison not only between databases but also between deployment configurations, i.e., between cloud and hybrid edge-cloud deployments. To achieve this, the insertion load is simply being split between both edge nodes, while the query load is split between the edge and cloud nodes, in the same way as in the previously presented query workload. Once again, this results in a 50% increase in overall submitted queries, but with two thirds of them comprising half the data achieving essentially the same overall load. The goal is that the same overall load is distributed over a higher number of nodes.

The resulting workload is therefore composed of 4 identical insertion Clients, 2 for each edge node, with configurations as listed in Table 36, and 3 identical query Clients, 1 for each node, configured as listed in Table 37.

As mentioned in Section 4.1.2, when using downsampling, outlier filter queries are not be submitted

² IoTDB unsequenced files issue: <https://github.com/apache/iotdb/issues/11441>

to the cloud node.

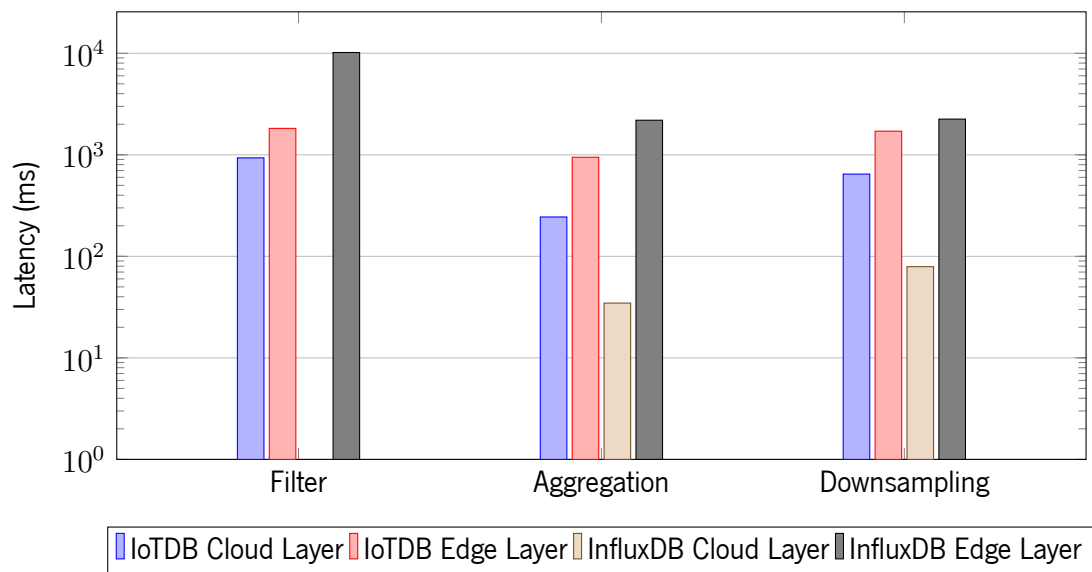


Figure 46: Query latency bar chart for mixed workload in 2 Edge + Cloud scenario

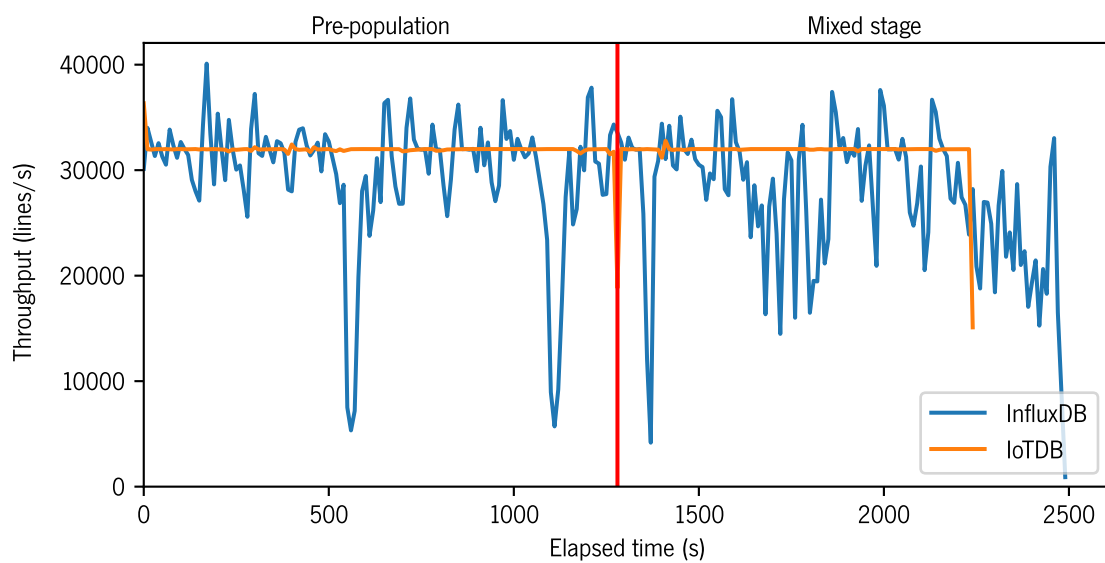


Figure 47: Throughput comparison for mixed workload in 2 Edge + Cloud scenario

Figure 46 shows the query latency for the best scenario of each database. For InfluxDB this corresponds to using 2 second downsampling. Note that InfluxDB's cloud layer does not perform outlier filter queries since the results of filter queries are affected by the downsampling. This graph uses a logarithmic scale since there is a wide range of latency values.

IoTDB is able to pretty much match the targeted throughput, with an average throughput of 27,931.63 lines per second for the mixed stage, while using 2 second downsampling InfluxDB achieves 26,915.31

lines per second during the mixed stage, corresponding to 96.1% of the target throughput. As the throughput comparison in Figure 47 shows, InfluxDB is able to mostly match the targeted throughput with multiple clear dips due to compactions during the population stage.

InfluxDB

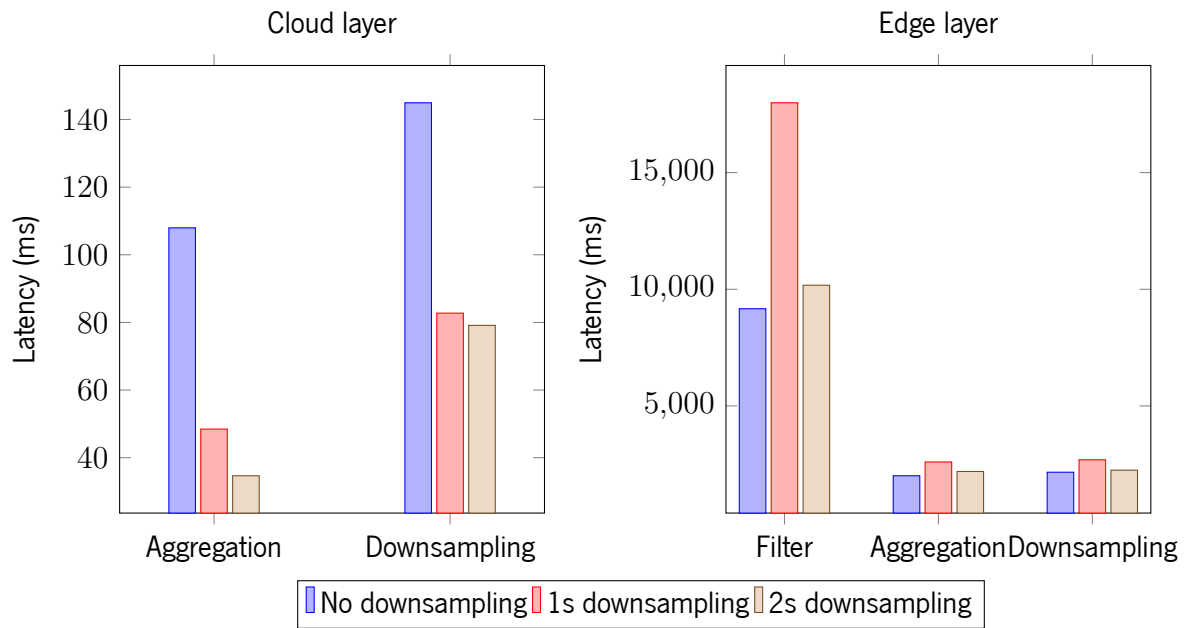


Figure 48: Query latency bar chart for mixed workload targeting InfluxDB in 2 Edge + Cloud scenario

Figure 48 shows the query latency achieved by InfluxDB for each layer and downsampling configuration. At the cloud layer, there are significant latency improvements when using downsampling at the edge, as this pre-processing reduces the work needed for aggregation and downsampling queries at the cloud layer. Meanwhile, at the edge layer, downsampling seems to affect query latency negatively since it demands the continuous processing of all data. Interestingly, 2 second downsampling achieves improved query performance when compared to 1 second downsampling. This may be due to the fact that the 2 second downsampling task is performed less frequently and over a higher volume of data resulting in more efficient operations.

The improved throughput when downsampling the replicated data indicates the replication mechanism has a significant performance cost for high data volumes.

IoTDB

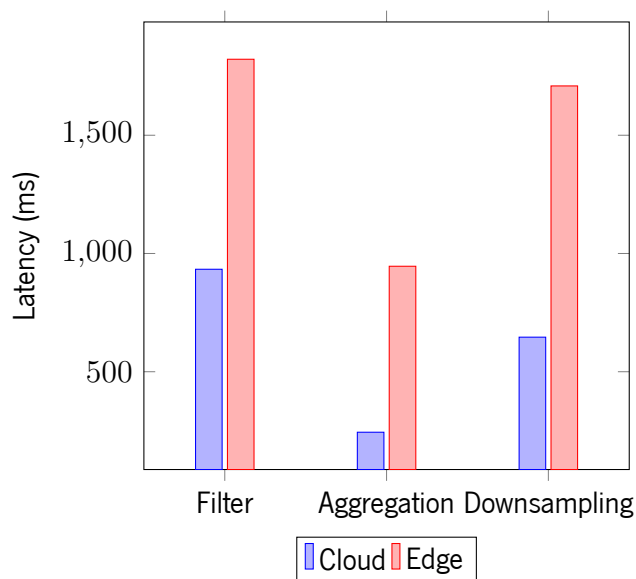


Figure 49: Layer query latency bar chart for mixed workload targeting IoTDB in 2 Edge + Cloud scenario

Figure 49 shows the query latency achieved by IoTDB for each layer. For IoTDB, the edge layer query latency performance for all query types is worse when compared to the cloud-only deployment. This may be unexpected since this layer is subjected to only part of the load in this test. However, the resource limitations of the edge nodes mean that even this decreased load is not enough to provide better average latencies.

Interestingly, the cloud layer shows better performance for all query types when compared to the cloud-only deployment, even though there occurred the same phenomenon as in the query workload in Section 4.4.2, where the cloud node stores data on unsequenced TsFiles. This is probably due to the decreased load on this layer.

This workload shows that when deployed in this Edge + Cloud configuration, InfluxDB still struggles to handle the targeted throughput, even when using downsampling, while IoTDB matches the targeted throughput. IoTDB also shows lower latency for all query types at the edge layer. At the cloud layer, InfluxDB can achieve lower average latencies for aggregations and downsampling queries when using downsampling. An analysis of per-layer query latencies was not previously possible.

4.5 4 Edge + Cloud Scenario

This scenario attempts to study the scalability of an edge-cloud hybrid deployment by increasing the number of edge nodes, which should improve performance in an edge-limited deployment. Four edge nodes are deployed and connected to a central cloud node, to which data is replicated. It uses the resource limiting profile defined to emulate a Raspberry Pi 4, as listed in Section 6. This resource limiting is applied to the 4 used edge nodes, with the cloud node having no resource limiting.

4.5.1 Insertion Workload

For this workload, in the same way as in the previous section, multiple concurrent Clients connect to and submit data into each of the edge nodes, achieving a load equally distributed between them.

Table 9: Throughput table for insertion workload in 4 Edge + Cloud scenario

Downsampling Option	InfluxDB	Apache IoTDB
No Downsampling	35081.71 lines/s	538293.9 lines/s
1s Downsampling	54007.03 lines/s	
2s Downsampling	54524.36 lines/s	

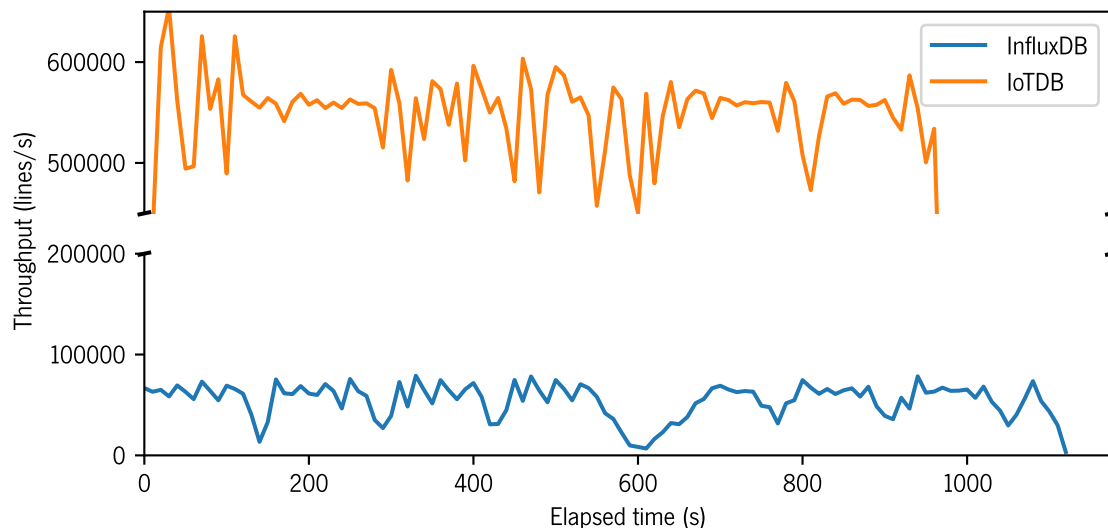


Figure 50: Throughput comparison for insertion workload in 4 Edge + Cloud scenario

For the throughput comparison in Figure 50, the downsampling configuration that results in the highest throughput for each database is used. Just like in the 2 Edge + Cloud deployment in Section 4.4.1, for

InfluxDB, this corresponds to 2 second downsampling. As listed in Table 9, InfluxDB achieves an average throughput of 54,524.36 lines per second, while IoTDB manages to reach 538,293.9 lines per second without downsampling.

These results translate to a 32.7% increase in throughput for InfluxDB and a 67.9% increase in throughput for IoTDB, when compared to the cloud-only deployment.

InfluxDB

For this scenario, the test configuration for InfluxDB uses 8 clients with configurations as listed in Table 39, totalling a target throughput of 64,000 lines per second.

With this deployment, using downsampling results in a high number of failed insert operations, up to 180 failed operations in a single test, corresponding to approximately 0.03% of all operations. Analysing the latency scatter plot in Figure 51 reveals these failed inserts occurred during large latency spikes, causing some operations to exceed the 10 seconds timeout. As with previous tests, these latency spikes can be correlated to increases in disk IO throughput, shown in Figure 52, which happen due to InfluxDB's compaction mechanism. Looking at the data present in the database after the tests revealed all data is replicated successfully in spite of the failed inserts.

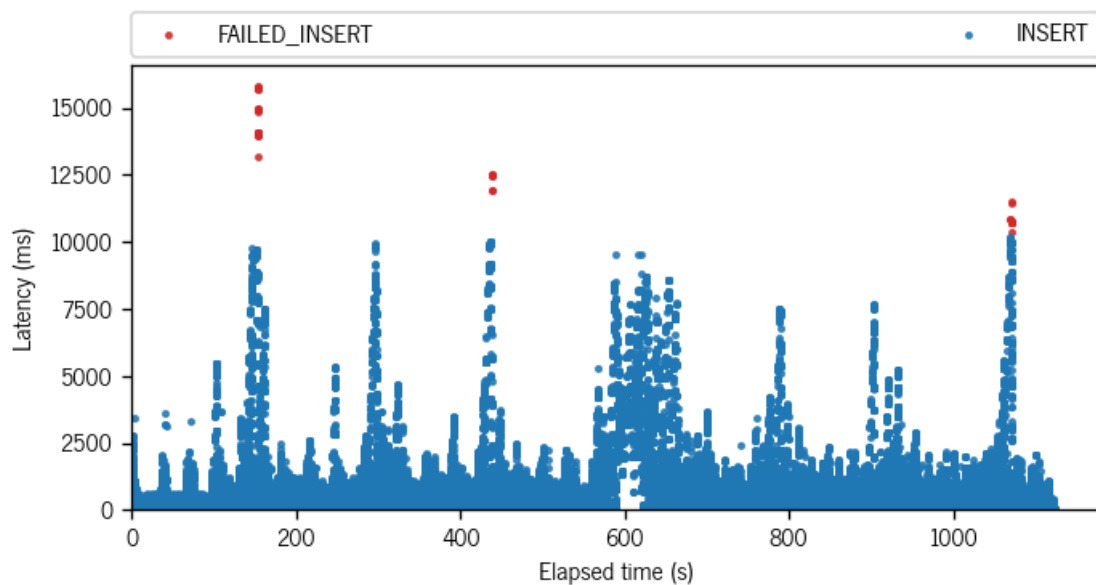


Figure 51: Insertion latency scatter plot for insertion workload targeting InfluxDB with 2 second downsampling in 4 Edge + Cloud scenario

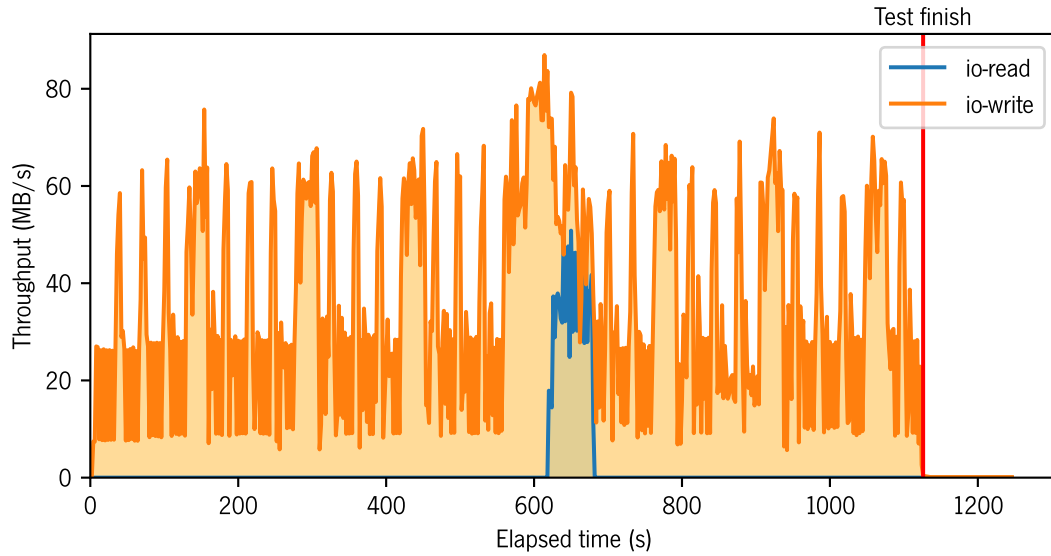


Figure 52: Edge node disk IO usage for insertion workload targeting InfluxDB with 2 second downsampling in 4 Edge + Cloud scenario

When compared to the 2 Edge + Cloud scenario, InfluxDB achieves increases in throughput of 74.1% when using no downsampling, 79.2% using 1 second downsampling, and 79.7% when using 2 second downsampling. This shows how the edge layer can be scaled independently of the cloud layer and that doing so can increase the overall system performance.

IoTDB

For IoTDB, the used test configuration uses 8 clients configured as listed in Table 40.

With this deployment, IoTDB achieves an increase in throughput of 96.7% when compared to the 2 Edge + Cloud scenario, an almost two-fold performance boost from doubling the number of edge nodes, representing almost perfect scalability.

This also represents a 67.2% increase in insertion performance when compared to the Cloud-only deployment. The cloud node can therefore achieve a higher throughput when aided by an edge layer and the database's internal replication mechanism.

Attempts at achieving a higher throughput result in Docker killing the edge node containers for running out of memory. After tests with a setting not allowing Docker to kill the node containers for running out of memory resulted in the IoTDB instances returning errors and going into a read-only state. Testing with each node running on a separate machine yielded no improvements in throughput.

Analysing the data present in the cloud node after a successful test run reveals that not all data was

replicated successfully, with a loss of up to 1.6% of data in the case of some devices. There is no indication of this behaviour returned to the client, and it can only be observed either by looking at the data present in the cloud node or by looking at the status of the replication pipe on the edge node, which was *WARN*, indicating ‘data loss or other error has occurred’ (iot [2023]). Looking at the node’s logs also shows multiple errors. This behaviour has been documented and reported to the developers in a GitHub issue ³.

This workload shows significant throughput increases when compared to the Cloud-only scenario, showing how the edge layer can be scaled independently from the cloud layer in these deployments.

4.5.2 Query Workload

As with the previous query workloads, data is pre-populated through each of the edge nodes, which replicate it to the cloud node. The volume of data loaded is the same as in the previous tests, as to allow comparisons between them. When compared to the 2 edge deployment, the cloud load is kept the same, while the edge load is now distributed over the 4 edge nodes. All nodes are subjected to the same amount and rate of queries, with Clients configured as listed in Table 41, although for the cloud node this translates to a more demanding load as it contains more data.

This test is composed of five Clients, one for each edge node and one for the cloud node, with the following configuration.

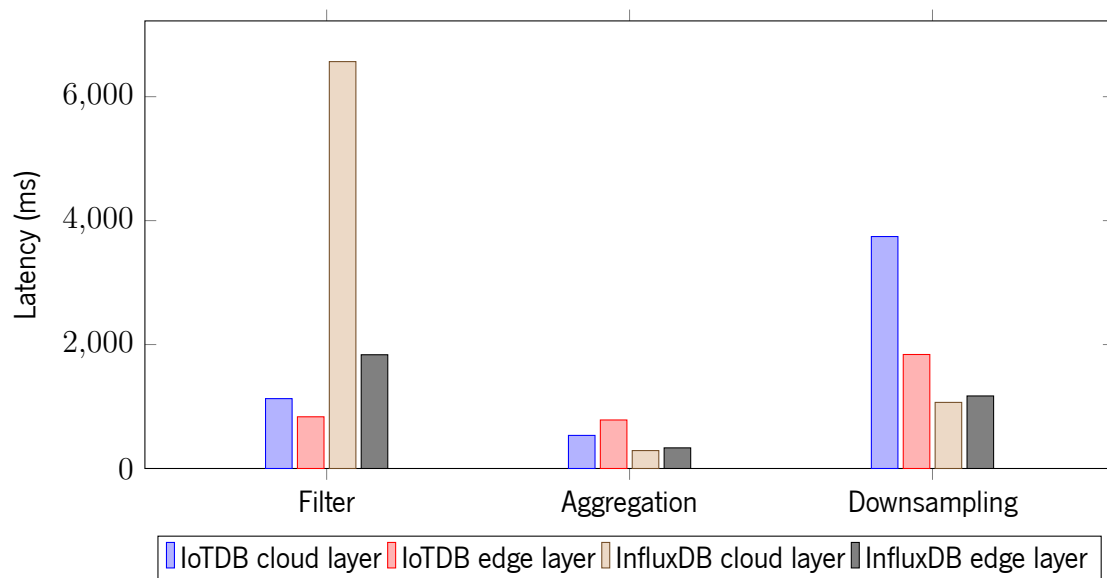


Figure 53: Query latency bar chart for query workload in 4 Edge + Cloud scenario

Figure 53 shows the average latency for each query type at the cloud and edge layers for both

³ IoTDB replication issue: <https://github.com/apache/iotdb/issues/11428>

databases. In it can be seen that while IoTDB presents significantly lower average latencies for filter queries on both layers, InfluxDB achieves lower latencies for both aggregation and downsampling queries.

InfluxDB

Distributing the load during the population phase over a higher number of edge nodes mitigates the identified replication queue issue, resulting in more data reaching the cloud layer. This, in addition to performing the loading phase at a slower rate allowed for all records to be replicated correctly.

With this deployment we can observe query latency improvements in both layers when compared to the original cloud-only deployment. At the cloud layer there is an 18.32%, 23.07% and 26.02% decrease in average latency, for outlier filters, aggregations and downsampling respectively, when compared to the cloud scenario, while the edge layer sees a 77.17%, 11.48% and an 18.83% decrease, respectively.

IoTDB

For IoTDB, only downsampling operations at the edge layer show lower latency than the original Cloud-only scenario, with all other query types for both the edge and cloud layer having lower latencies in the original deployment.

When compared to the 2 edge deployment, there are considerable improvements in query latency at the edge layer, with 57.2%, 54.0%, 64.4% reductions for filters, aggregation and downsampling queries, respectively. Just like for the 2 Edge + Cloud scenario in Section 4.4.2, looking at the results for the cloud layer reveals some unexpected values. Once again, although the cloud layer is subjected to the same load, the average latency observed has increased significantly for all query types when compared to the 2 edge deployment.

This workload shows how InfluxDB can benefit from additional edge nodes when subjected to a query load, showing lower latencies at both layers than in the original Cloud-only scenario. IoTDB, however, failed to show any significant improvements when compared to the Cloud-only scenario.

4.5.3 Mixed Workload

The mixed workload is configured to perform the same throughput and queries as in the mixed workload tests, but now distributing the load over 4 edge nodes. The resulting workload is composed of 4 identical insertion Clients, 1 for each edge node, with configurations as listed in Table 43, and 5 identical query Clients, configured in the same way as in Table 44, 1 for each edge node.

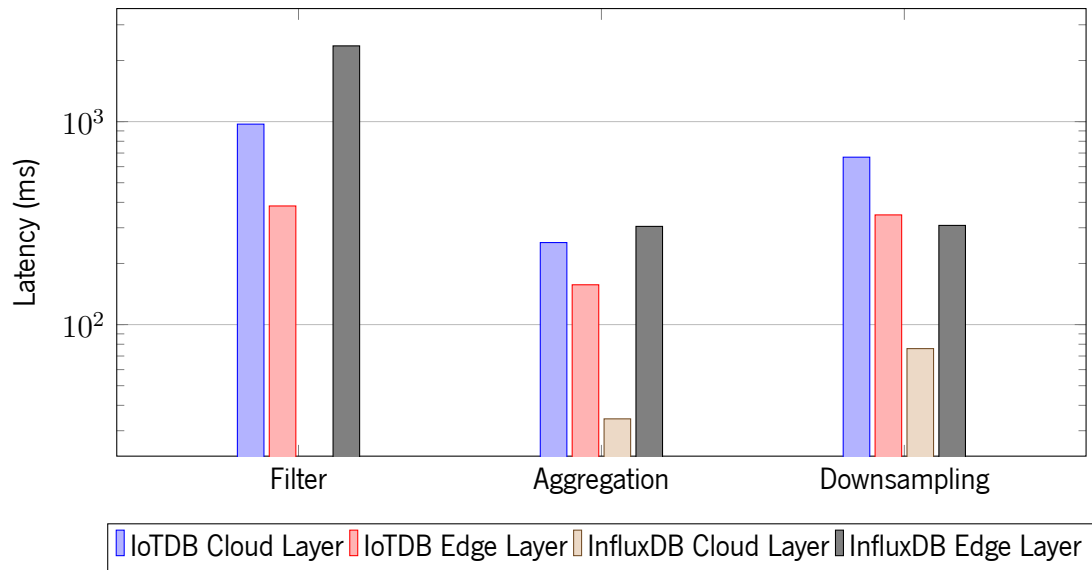


Figure 54: Query latency bar chart for mixed workload in 4 Edge + Cloud scenario

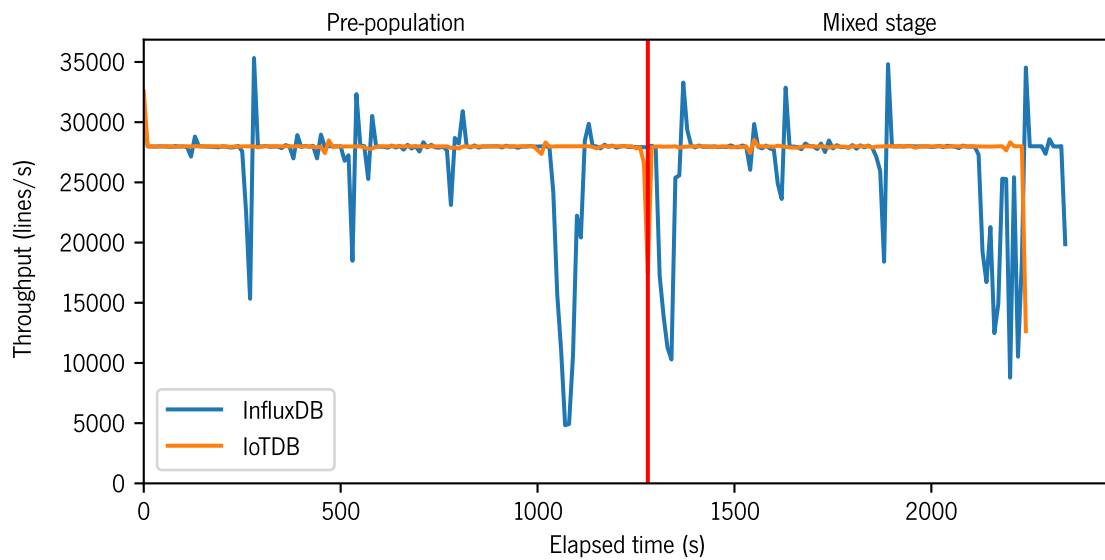


Figure 55: Throughput comparison for mixed workload in 4 Edge + Cloud scenario

The throughput comparison in Figure 55 shows that IoTDB is able to match the targeted throughput, achieving an average of 27,902.4 lines per second, while InfluxDB achieves 26,792.61467 lines per second, 4.3% below the target of 28,000.

By looking at the query latencies for each database's optimal configuration, InfluxDB using 2 second downsampling, and IoTDB using no downsampling, in Figure 54, we can see how beneficial having the edge layer downsampling data is, as query latency on the cloud layer is significantly lower for InfluxDB, having 86.5% and 88.6% lower average query latencies for aggregations and downsampling queries, respectively.

On the other hand, edge layer query performance favours IoTDB, with 83.7% lower average outlier filter latency, 48.4% lower aggregation latency, but 12.6% higher downsampling latency.

InfluxDB

This scenario results in significantly improved edge layer query performance when compared to the 2 edge deployment, with 76.8%, 86.1%, and 86.3% lower average latencies for filters, aggregations, and downsampling queries, respectively. When comparing to the original cloud deployment, the edge layer also achieves 86.5%, 67.2%, and 67.9% lower average latencies for filters, aggregations, and downsampling queries, respectively, while the cloud layer lower achieves 96.3%, and 92.1% lower average latencies for aggregations, and downsampling queries, respectively.

This deployment therefore shows the benefits not only of performing downsampling at the edge layer, resulting in lower latency at the cloud layer, but also how distributing the load over a higher number of nodes can improve performance at the edge layer.

IoTDB

At the cloud layer, this deployment yields very similar results to the 2 Edge + Cloud scenario, with 4.1%, 3.8%, and 3.4% higher average latencies for filter queries, aggregations and downsampling queries, respectively. This is expected since this node is subjected to the same load and contains the same data, simply being replicated from more edge nodes.

At the edge layer, there is a significant improvement in average query latencies when comparing to the deployment with 2 edge nodes, dropping by 78.9%, 83.4%, and 79.7% for filters, aggregations, and downsampling queries, respectively. This is also expected behaviour, since each edge node now receives and queries over a smaller volume of data.

Query latencies for both layers represent significant improvements over the cloud-only deployment. Overall, this workload shows how this scenario can enable greatly reduced query latencies for mixed workloads when compared to both the Cloud-only and the 2 Edge + Cloud scenarios, while maintaining a similar throughput.

4.6 Summary

The testing of MulletBench's rate limiting shows that it is able to successfully limit its Clients' rate of operations, with the achieved throughput being very close to the targeted throughput. Its resource limiting

has also been validated, showing clear bottlenecks in resource usage and subsequent performance degradation on the limited nodes. Limiting network conditions by introducing latency between the Clients and database nodes shows an observable increase in latency of operations for both databases.

When deployed in a Cloud-only configuration, with a single database node, IoTDB shows significantly better performance than InfluxDB with an insertion workload, achieving a 6.8x throughput increase. The query workload tests show IoTDB with a significant lower average latency for filter queries and a slight advantage for aggregations, while InfluxDB has a slight advantage in downsampling queries. When subjected to a mixed workload, InfluxDB struggles to handle the targeted throughput and performs worse than IoTDB for all query types, including 1413.6% higher average filter latency, while IoTDB is able to keep a constant throughput.

Tests in a 2 Edge + Cloud scenario were composed of 2 resource limited nodes simulating the edge layer, with replication to a single cloud node. In this scenario, IoTDB achieves 8.0x the throughput of InfluxDB's with the best tested downsampling configuration, when subjected to an insertion workload. When subjected to a query workload, at the edge nodes, InfluxDB achieves lower average latencies than IoTDB for aggregations and downsampling queries, while IoTDB has lower latency for filter operations. An issue with InfluxDB's replication system has been identified, which prevented a correct comparison of the cloud layer query performance. This workload also allowed for the identification of a potential issue in IoTDB, in which replicated data is stored in unsequenced TsFiles on the Cloud node, resulting in degraded querying performance. Under a mixed workload, InfluxDB still struggles to handle the targeted throughput, even when using downsampling, while IoTDB matches the targeted throughput while also having lower latency for all query types at the edge layer. At the cloud layer, InfluxDB can achieve lower average latencies for aggregations and downsampling queries when using downsampling.

Tests using downsampling at the edge layer with InfluxDB show it can achieve a 53% increase in throughput under an insertion workload, and a 40% increase in throughput, while increasing query latency at the edge up to 11% and decreasing latencies at the cloud, when under a mixed workload.

Increasing the number of edge nodes to 4 results in significant performance improvements for both databases for all workloads. When using 2 second downsampling at the edge layer, InfluxDB achieves a 32.7% increase in throughput, while IoTDB achieves a 67.9% increase in throughput, when compared to the Cloud-only deployment, and even greater increases when compared to the 2 Edge + Cloud deployment. When subjected to a query workload, InfluxDB achieves lower latencies for all queries and at both layers, when compared to the Cloud-only deployment. Overall, IoTDB is unable to show any performance improvements for query workloads when compared to the Cloud-only deployment, apart from slightly lower

latencies for downsampling queries at the edge layer. Under a mixed workload, InfluxDB achieves similar throughput to the Cloud-only deployment, when using 2 second downsampling, and shows significant latency improvements for all query types on both layers. IoTDB achieves similar throughput to the Cloud-only deployment, while showing an up to 66.9% reduction in average latency at the edge layer, and up to 30.7% lower average latency at the cloud layer. Comparing both databases using this scenario does not yield significant differences, when compared to the 2 Edge + Cloud scenario

These results provide insight into the performance of both databases under different workloads, deployment configurations, and downsampling options, through metrics which were not previously obtainable. In sum, they show how the edge layer can be used to improve performance, both by distributing the load over a higher number of nodes, and by performing downsampling, resulting in lower latencies at the cloud layer and higher throughput.

Chapter 5

Conclusions and future work

In this thesis, a benchmarking system capable of providing metrics that accurately measure the performance of a **TSDBMS** deployed in an Edge-Cloud distributed architecture was developed.

An in-depth study and survey of currently existing **TSDBMS** benchmarks builds a new taxonomy of the existing benchmarks, analysing their workloads, data, metrics and orchestration features, showing that existing **TSDBMS** benchmarking solutions lack metrics and orchestration features suitable for the evaluation of Edge-Cloud hybrid **TSDBMSs**.

MulletBench addresses this problem by enabling the evaluation of **TSDBMSs** in Edge-Cloud hybrid deployments through the introduction of novel features such as per-node and per-layer metrics, automatic multi-layer database deployment, and orchestration of distributed clients. These features enable a fine-grained analysis of the database's performance in a multi-layer configuration, improve the reproducibility of the benchmark, ease the evaluation of different database configurations and allow for scaling workloads.

The evaluation confirms MulletBench's ability to limit its clients' operation submission rate, the database nodes' resources and the tests' network conditions.

An in-depth evaluation of InfluxDB and Apache IoTDB is presented, where MulletBench allowed for a thorough comparison of the performance of the two, as well as for great insight into the individual contributions of each edge node. This evaluation includes a comparison of the targeted databases in multiple workloads, downsampling configurations and deployment scenarios. Results show that the edge layer can be used to improve performance by distributing the workload over multiple layers and performing downsampling at the edge layer, increasing overall throughput and reducing cloud latency. Results also show that IoTDB generally presents superior performance to InfluxDB for all workloads. This evaluation also allowed for the identification of flaws in both systems' replication mechanisms when these are subjected to high throughput workloads.

The conclusions taken from the results are enabled by MulletBench's novel features, and it would not have been possible to identify them with previously existing solutions, therefore demonstrating its

usefulness.

5.1 Future work

MulletBench can currently be used to evaluate the performance of **TSDBMSs** in multi-layer deployments, but there are still some features that could be added to improve its functionality and usability.

As highlighted by the testing campaign, databases sometimes fail to successfully replicate data, without any indication of this failure. This results in the need to manually check the validity of the results, by querying how much data the nodes contain. A validation phase could be implemented, in which, after the execution phase is completed, MulletBench would check if the data was successfully inserted into all nodes, allowing for the detection of replication failures, and eliminating the need for manual verification.

Additional resource profiles for edge nodes could be added, allowing for the evaluation of databases in different hardware configurations.

Support for additional **TSDBMSs**, including those which do not include explicit Edge-Cloud replication features, could also be added, allowing for the evaluation and comparison of more systems in multi-layer deployments.

Finally, tests at a greater scale could also be conducted, including a greater number of edge nodes as well as multiple cloud nodes, allowing for the evaluation of more complex scenarios and approximating production environments.

Bibliography

Ranking of time series dbms, 2022a. URL <https://db-engines.com/en/ranking/time+series+dbms>. Accessed on December 2022.

Dbms popularity broken down by database model, 2022b. URL https://db-engines.com/en/ranking_categories. Accessed on December 2022.

Edge-cloud collaboration, Oct 2023. URL <https://iotdb.apache.org/UserGuide/V1.1.x/Edge-Cloud-Collaboration/Sync-Tool.html#show-pipe>. Accessed on October 2023.

Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, 2016.

Andreas Bader. Comparison of time series databases. Master’s thesis, University of Stuttgart, 2016.

Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Ilk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

Maxim Buevich, Anne Wright, Randy Sargent, and Anthony Rowe. Respawn: A distributed multi-resolution time-series datastore. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 288–297. IEEE, 2013.

Hongming Cai, Boyi Xu, Lihong Jiang, and Athanasios V Vasilakos. Iot-based big data storage systems in cloud computing: perspectives and challenges. *IEEE Internet of Things Journal*, 4(1):75–87, 2016.

Stephan Cejka, Ralf Mosshammer, and Alfred Einfeldt. Java embedded storage for time series and meta data in smart grids. In *2015 IEEE International Conference on Smart Grid Communications (Smart-GridComm)*, pages 434–439. IEEE, 2015.

Yanpei Chen, Francois Raab, and Randy Katz. From tpc-c to big data benchmarks: A functional workload model. In *Specifying Big Data Benchmarks*, pages 28–43. Springer, 2012.

- Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- Ousmane Diallo, Joel JPC Rodrigues, Mbaye Sene, and Jaime Lloret. Distributed database management techniques for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):604–620, 2013.
- Bernhard Firner, Robert S Moore, Richard Howard, Richard P Martin, and Yanyong Zhang. Poster: Smart buildings, sensor networks, and the internet of things. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 337–338, 2011.
- Peeyush Gupta, Michael J Carey, Sharad Mehrotra, and oberto Yus. Smartbench: a benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment*, 13(12):1807–1820, 2020.
- Susan Hall. Iotdb provides data management for industrial edge it, Oct 2020. URL <https://thenewstack.io/iotdb-provides-data-management-for-industrial-edge-it/>.
- Rui Han, Lizy Kurian John, and Jianfeng Zhan. Benchmarking big data systems: A review. *IEEE Transactions on Services Computing*, 11(3):580–597, 2017.
- 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. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 588–599. IEEE, 2021.
- HashiCorp. Vagrant by hashicorp, 2010. URL <https://www.vagrantup.com/>. Accessed on January 2023.
- InfluxData. Influxdb time series data platform, 2013. URL <https://www.influxdata.com/products/influxdb/>. Accessed on January 2023.
- Carlos Kamienski, Juha-Pekka Soininen, Markus Taumberger, Ramide Dantas, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and André Torre Neto. Smart water management platform: IoT-based precision irrigation for agriculture. *Sensors*, 19(2):276, 2019.
- Roman Karlstetter, Robert Widhopf-Fenk, and Martin Schulz. Querying distributed sensor streams in the edge-to-cloud continuum. In *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 192–197. IEEE, 2022.

- Abdelouahab Khelifati, Mourad Khayati, Anton Dignös, Djellel Difallah, and Philippe Cudré-Mauroux. Tsm-bench: Benchmarking time series database systems for monitoring applications. *Proc. VLDB Endow.*, 16(11):3363–3376, aug 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611532. URL <https://doi.org/10.14778/3611479.3611532>.
- Rui Liu and Jun Yuan. Benchmarking time series databases with iotdb-benchmark for iot scenarios. *arXiv preprint arXiv:1901.08304*, 2019.
- J. Menegazzo and A. von Wangenheim. Multi-contextual and multi-aspect analysis for road surface type classification through inertial sensors and deep learning. In *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8, 2020. doi: 10.1109/SBESC51047.2020.9277846.
- Jalal Mostafa, Sara Wehbi, Suren Chilingaryan, and Andreas Kopmann. Scits: A benchmark for time-series databases in scientific experiments and industrial internet of things. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management, SSDBM '22*, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396677. doi: 10.1145/3538712.3538723. URL <https://doi.org/10.1145/3538712.3538723>.
- John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikradya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. Vergedb: A database for iot analytics on edge devices. In *CIDR*, 2021.
- Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE communications surveys & tutorials*, 16(1):414–454, 2013.
- Meikel Poess, Raghunath Nambiar, Karthik Kulkarni, Chinmayi Narasimhadevara, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of tpcx-iot: The first industry standard benchmark for iot gateway systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1519–1530. IEEE, 2018.
- Andrea Reale. A guide to edge iot analytics, Feb 2019. URL <https://www.ibm.com/blogs/internet-of-things/edge-iot-analytics/>.
- Yuya Sasaki. A survey on iot big data analytic systems: Current and future. *IEEE Internet of Things Journal*, 9(2):1024–1036, 2021.
- Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.
- Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 102–107. IEEE, 1999.

Bonil Shah, PM Jat, and Kalyan Sashidhar. Performance study of time series databases. *arXiv preprint arXiv:2208.13982*, 2022.

Timescale. Time series benchmark suite (tsbs), 2022. URL <https://github.com/timescale/tsbs>.

Transaction Processing Performance Council (TPC). Tpc-c, 2010. URL <https://www.tpc.org/tpcc>. Revision 5.11.

Transaction Processing Performance Council (TPC). Tpc-ds, 2021. URL <https://www.tpc.org/tpcds>. Revision 3.2.0.

Transaction Processing Performance Council (TPC). Tpcx-iot, 2022. URL <https://www.tpc.org/tpcx-iot>. Revision 2.0.1.

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

William Yu Chung Wang and Yichuan Wang. Analytics in the era of big data: The digital transformations and value creation in industrial marketing, 2020.

Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. *IEEE industrial electronics magazine*, 11(1):17–27, 2017.

Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

Yuqing Zhu, Yanzhe An, Yuan Zi, Yu Feng, and Jianmin Wang. Iotdatabench: Extending tpcx-iot for compression and scalability. In *Performance Evaluation and Benchmarking: 13th TPC Technology Conference, TPCTC 2021, Copenhagen, Denmark, August 20, 2021, Revised Selected Papers*, pages 17–32, 2021.

Appendix A

Configuration Options

A.1 Deployment Orchestrator Options

The deployment configuration file is an Ansible inventory file, meaning it must follow its structure conventions. By defining specific variables for hosts in specific groups, one can create a custom test configuration. Hosts defined in the *edgeservers* group refer to the edge database nodes to be used in a test's execution. Similarly, hosts defined in the *cloudservers* group refer to cloud database nodes. Both of these groups differ only in the ability for hosts in the *edgeservers* group to be configured for resource limiting and replication. All other options are shared between both groups.

A.1.1 Shared edgeservers and cloudservers variables

Table 10: Shared edgeservers and cloudservers group variables for Deployment Orchestrator

Option	Description
database	name of tested database (influx or iotdb)
downsampling	whether to perform downsampling on inserted data, boolean
downsampling_interval	time interval to use for aggregation (e.g., 1s)
network_sim	list of entries for network simulation configurations
blkio_path	block device path for IO limiting

Table 11: Network simulation variables for Deployment Orchestrator

Option	Description
target	name of node to target in network configuration
latency	amount for roundtrip network delay (e.g., 50ms)
latency_normal_distribution	variation to be applied over latency with normal distribution (e.g., 10ms)
bandwidth	network bandwidth rate (e.g., 400Kbps)
reordering_rate	percentage of packet reordering rate

A.1.2 Edgeservers variables

Table 12: Edgeservers specific group variables for Deployment Orchestrator

Option	Description
limited_resources	whether node container resources should be limited, boolean
use_resource_profile	whether to use preset resource profile for limiting, boolean
limited_profile	name of preset resource profile
limited_resources_cpu	limit for CPU usage in cores
limited_resources_mem	limit for memory usage in bytes (e.g., 2048M)
limited_resources_read_bps	limit for read bytes per second in bytes (e.g., 15M)
limited_resources_write_bps	limit for written bytes per second in bytes
limited_resources_read_iops	limit for read operations per second in operations
limited_resources_write_iops	limit for write operations per second in operations
replication_targets	list of cloud nodes to replicate data to

Depending on which database should be tested, some additional database specific variables should also be set.

Table 13: InfluxDB variables for Deployment Orchestrator

Option	Description
influx_username	username for super-user
influx_password	password for super-user
influx_org	name of organization
influx_bucket	bucket name for data
influx_token	authentication token
influx_version	version of InfluxBD to use
influx_port	specific port for InfluxBD instance (defaults to 8086)

Table 14: Apache IoTDB variables for Deployment Orchestrator

Option	Description
iotdb_device_path	path for devices in iotdb (e.g. "root.gps.mpu.*")
iotdb_version	version of Apache IoTDB to use
iotdb_port	specific port for IoTDB instance (defaults to 6667)

A.1.3 benchmark_clients

Table 15: benchmark_clients group variables for Deployment Orchestrator

Option	Description
dataset	name of dataset used in test
current_time	whether operations should use current time or original dataset time, boolean
shared_dataset	whether the loaded dataset should be shared by all workers, boolean
shared_connection	whether database connection should be shared by all workers, boolean
stage	stage to which Client belongs, integer
monitor	whether Client container should be monitored, boolean
target	name of host targeted by Client
data_file	name of data file to use
num_workers	number of workers to generate load
type	type of load to be generated, insert or query
insert_volume	total number of entries to be inserted by each worker
batch_size	number of entries in each insert operation
insert_rate	number of insert operations per second per worker
query_count	total number of query operations to be submitted
query_rate	number of query operations per second per worker
agg_weight	weight for probability of aggregation queries
filter_weight	weight for probability of filter queries
downsample_weight	weight for probability of downsampling queries
outlier_filter_weight	weight for probability of outlier filter queries
count_outlier_filter	whether outlier filter queries should use <i>count</i> aggregation, boolean
filter_z_score	target z-score used for outlier filters
fjp_parallelism	pending request ForkJoinPool size
max_memory	maximum to be used by Client container (e.g. 4G)
orchestrator_ip	IP address of orchestrator node
write_timeout	database write operation timeout in seconds
read_timeout	database read operation timeout in seconds

Additional variables should also be set for the localhost, since this will run the Orchestrator execution component.

Table 16: Deployment variables for Deployment Orchestrator

Option	Description
database	name of tested database (to know name of container to monitor with PCP)
monitoring_interval	string to define at which rate monitoring should be performed (as received by PCP)
final_monitoring_period_sec	length of additional database resource monitoring after test is finished in seconds

Both execution Orchestrator options and Client options are not intended to be written manually, as the deployment phase generates configuration files for the Orchestrator execution component and all Clients in a test automatically, from a single test configuration. Even so, both these configuration options will be presented next so that they can be better understood.

A.2 Execution Orchestrator Options

The execution Orchestrator configuration file uses the YAML format and should follow the following hierarchy.

Table 17: Execution Orchestrator options

Option	Description
database	name of tested database (to know name of container to monitor with PCP)
monitoringInterval	string to define at which rate monitoring should be performed (as received by PCP)
resultsFolder	path to directory where results will be saved (inside container)
clients	list of client
stages	list of test stages
nodes	list of node

Table 18: Execution Orchestrator client options

Option	Description
name	identifier for Client instance
address	IP address where Client instance is running (must be reachable)
type	type of Client, either INSERT or QUERY
target	identifier of database node targeted by Client instance
monitor	whether Client instance should be monitored with PCP , boolean

Table 19: Execution Orchestrator stage options

Option	Description
stage	integer to identify stage in order
clients	list of Client instance identifiers

Table 20: Execution Orchestrator node options

Option	Description
name	identifier for database node
layer	which layer node belongs to, EDGE or CLOUD
address	IP address where database node is running
monitor	whether database node should be monitored with PCP , boolean

A.3 Client Options

Table 21: Client options

Option	Description
clientId	identifier for Client instance
dataFile	name of data file to be used in test
dataset	name of dataset to be used (GPSMPU)
target	database targeted by test
numWorkers	number of workers to use
type	type of load to generate (QUERY or INSERT)
currentTime	whether operations should use current time or original dataset time, boolean
sharedDataset	whether the loaded dataset should be shared by all workers, boolean
sharedConnection	whether database connection should be shared by all workers, boolean
query	query options
insertion	insertion options
iotdb	IoTDB options
influx	InfluxDB options
orchestratorAddress	IP address where orchestrator node is running

Table 22: Client query options

Option	Description
count	total number of queries to be submitted (per worker)
rate	number of queries to be submitted per second (per worker)
aggChance	weight for probability of aggregation
filterChance	weight for probability of filter
outlierChance	weight for probability of outlierFilter
downsampleChance	weight for probability of downsample
numberOfLoops	number of loops of data previously inserted (only for currentTime at false)
countOutlierFilter	whether outlier filter queries should use <i>count</i> aggregation
filterZScore	target z-score used for outlier filters

Table 23: Client insertion options

Option	Description
workerSectionSize	total number of entries to be submitted
batchSize	number of batched entries per insertion
rate	number of insertions to be submitted per second (per worker)

Table 24: Client InfluxDB options

Option	Description
serverURL	URL of server in format <code>http://SERVER_IP:SERVER_PORT</code>
token	security token used in instance
orgID	organization id used in instance
bucket	data bucket
writeTimeout	number of seconds before insertions should timeout
readTimeout	number of seconds before queries should timeout

Table 25: Client IoTDB options

Option	Description
host	IP address of IoTDB instance
port	port of IoTDB instance
username	database username
password	database password
devicePath	path for devices in database
queryAlignByDevice	whether to queries should use align by device instead of time, boolean

Appendix B

Test Configurations

B.1 Cloud Scenario

B.1.1 Insertion Workload

InfluxDB

Table 26: Client configuration for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

Variable	Value
num_workers	80
insert_volume	96000
batch_size	100
insert_rate	1

IoTDB

Table 27: Client configuration for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

Variable	Value
num_workers	300
insert_volume	96000
batch_size	100
insert_rate	1

B.1.2 Query Workload

Table 28: Client configuration for query workload Cloud-only scenario

Variable	Value
num_workers	2
query_count	192
query_rate	0.2
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0

B.1.3 Mixed Workload

Table 29: Pre-population Client configuration for mixed workload Cloud-only scenario

Variable	Value
num_workers	70
insert_volume	128000
batch_size	100
insert_rate	1

Table 30: Insert Client configuration for mixed workload Cloud-only scenario

Variable	Value
num_workers	70
insert_volume	96000
batch_size	100
insert_rate	1

Table 31: Query Client configuration for mixed workload Cloud-only scenario

Variable	Value
num_workers	2
query_count	240
query_rate	0.25
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0

B.2 2 Edge + Cloud Scenario

B.2.1 Insertion Workload

InfluxDB

Table 32: Client configuration for insertion workload targeting InfluxDB in 2 Edge + Cloud scenario

Variable	Value
num_workers	80
insert_volume	96000
batch_size	100
insert_rate	1

IoTDB

Table 33: Client configuration for insertion workload targeting IoTDB in 2 Edge + Cloud scenario

Variable	Value
num_workers	700
insert_volume	96000
batch_size	100
insert_rate	1

B.2.2 Query Workload

Table 34: Query Client configuration for query workload in 2 Edge + Cloud scenario

Variable	Value
num_workers	1
query_count	192
query_rate	0.2
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0

B.2.3 Mixed Workload

Table 35: Pre-population Client configuration for mixed workload 2 Edge + Cloud scenario

Variable	Value
num_workers	70
insert_volume	128000
batch_size	100
insert_rate	1

Table 36: Insert Client configuration for mixed workload in 2 Edge + Cloud scenario

Variable	Value
num_workers	70
insert_volume	96000
batch_size	100
insert_rate	1

Table 37: Query Client configuration for mixed workload in 2 Edge + Cloud scenario

Variable	Value
num_workers	1
query_count	240
query_rate	0.25
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0

Table 38: Cloud query Client configuration for mixed workload with downsampling in 2 Edge + Cloud scenario

Variable	Value
num_workers	1
query_count	240
query_rate	0.25
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	0
count_outlier_filter	false
filter_z_score	4.0

B.3 4 Edge + Cloud Scenario

B.3.1 Insertion Workload

InfluxDB

Table 39: Client configuration for insertion workload targeting InfluxDB in 4 Edge + Cloud scenario

Variable	Value
num_workers	80
insert_volume	96000
batch_size	100
insert_rate	1

IoTDB

Table 40: Client configuration for insertion workload targeting IoTDB in 4 Edge + Cloud scenario

Variable	Value
num_workers	700
insert_volume	96000
batch_size	100
insert_rate	1

B.3.2 Query Workload

Table 41: Query Client configuration for query workload in 4 Edge + Cloud scenario

Variable	Value
num_workers	1
query_count	192
query_rate	0.2
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0

B.3.3 Mixed Workload

Table 42: Pre-population Client configuration for mixed workload 4 Edge + Cloud scenario

Variable	Value
num_workers	70
insert_volume	128000
batch_size	100
insert_rate	1

Table 43: Insert Client configuration for mixed workload in 4 Edge + Cloud scenario

Variable	Value
num_workers	70
insert_volume	96000
batch_size	100
insert_rate	1

Table 44: Query Client configuration for mixed workload in 4 Edge + Cloud scenario

Variable	Value
num_workers	1
query_count	240
query_rate	0.25
agg_weight	40
filter_weight	0
downsample_weight	20
outlier_filter_weight	40
count_outlier_filter	false
filter_z_score	4.0