POLITECNICO
MILANO 1863

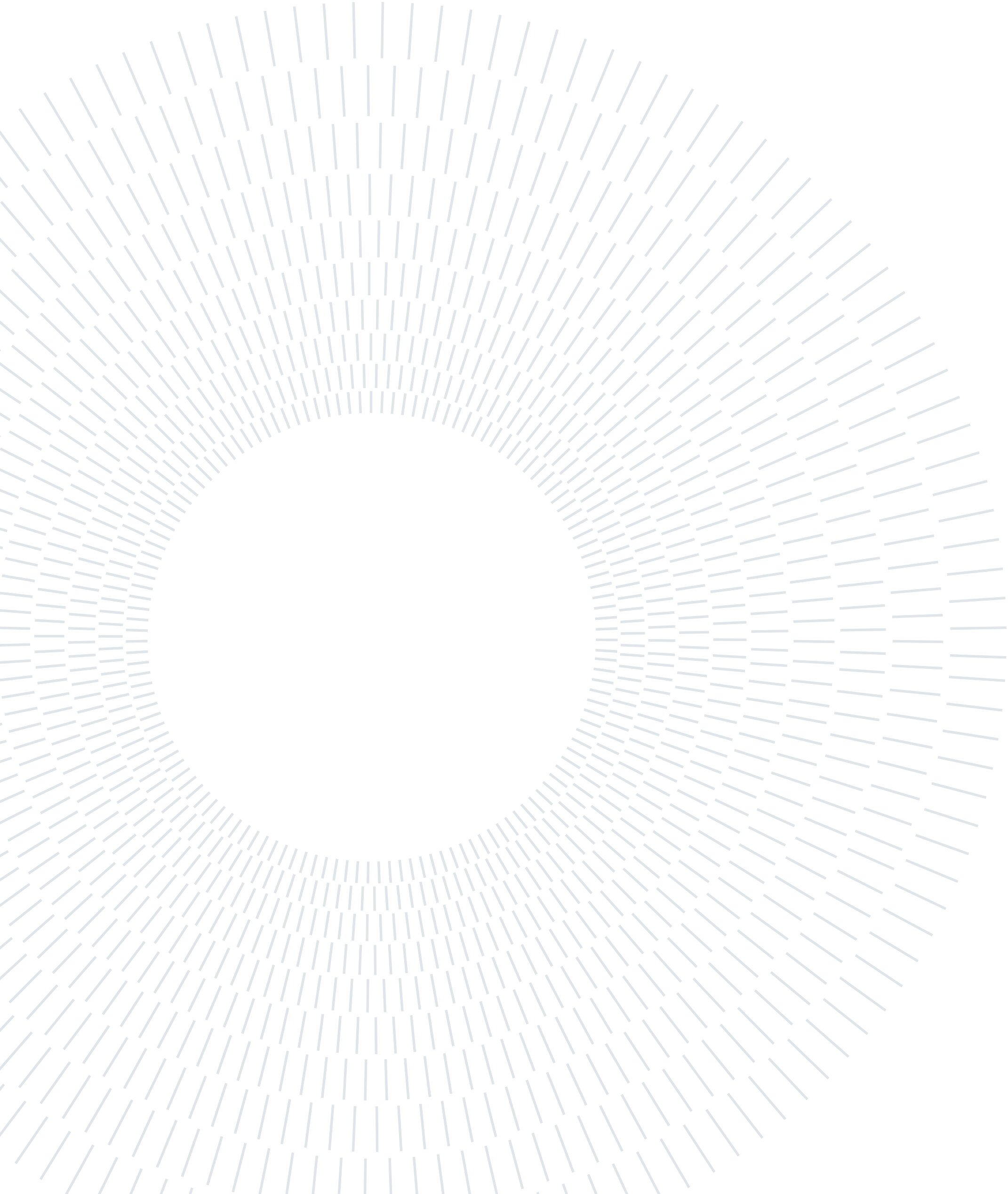
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Reducing read and write latency in a Delta Lake-backed offline feature store

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Giovanni Manfredi**

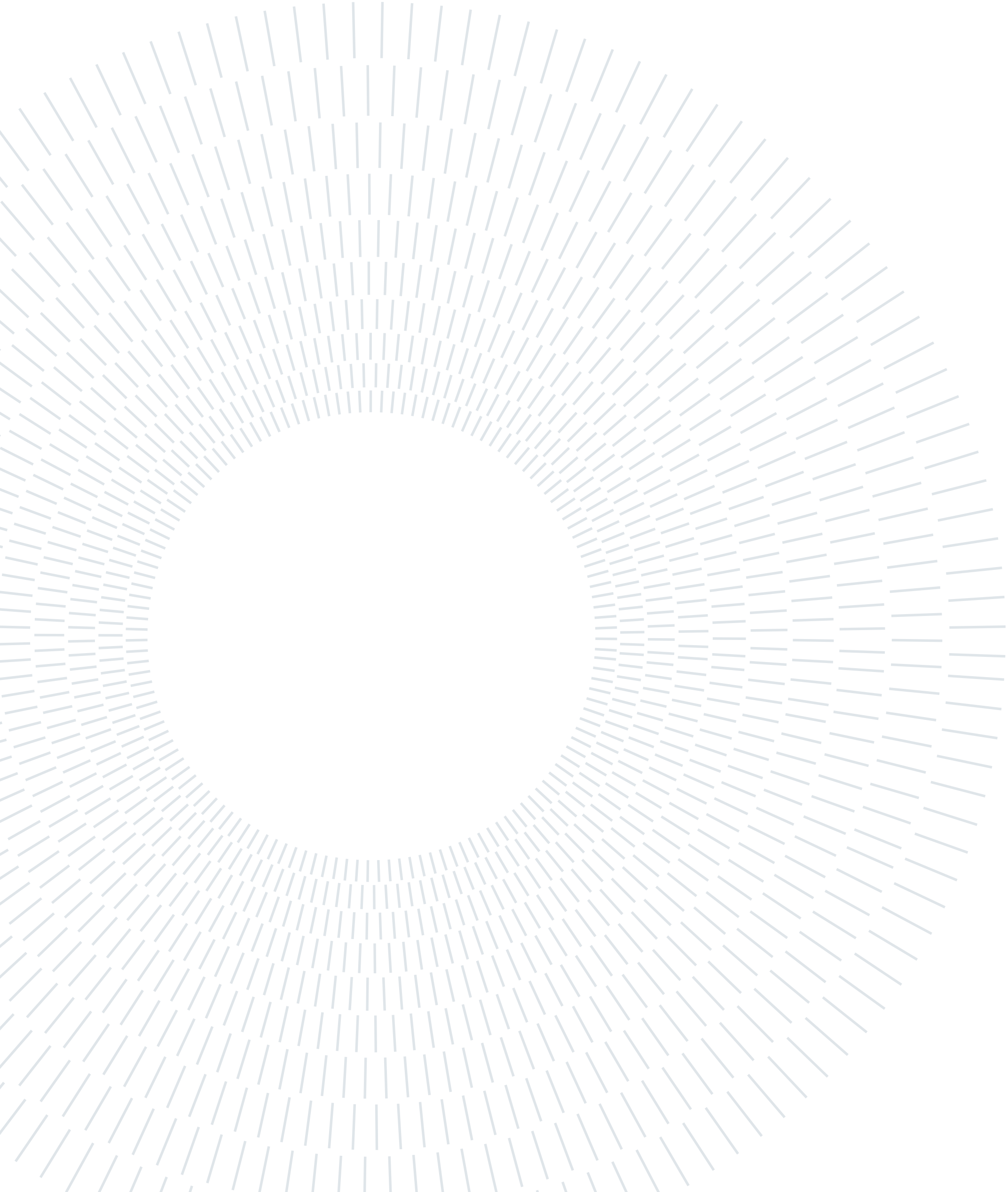
Student ID: 222696
Advisor: Prof. Davide Martinenghi
Co-advisors: Prof. Vladimir Vlassov (KTH), Sina Sheikholeslami (KTH),
Fabian Schmidt (KTH), Salman Niazi (Hopsworks AB)
Academic Year: 2024-25



Abstract

The need to build Machine Learning (ML) models based on large amounts of data brought new challenges to data management systems. Feature stores have emerged as a centralized data platform enabling feature reuse while ensuring consistency between feature engineering, model training, and inference. Recent publications demonstrate that the Hopsworks feature store outperforms existing cloud-based alternatives in training and online inference query workloads. In its offline feature store, the Hopsworks feature store stores batch or historical data, collecting it into feature groups organized in Apache Hudi tables and stored on HopsFS, Hopsworks HDFS distribution. However, even in this system, the latency to perform a write operation is at least one or more minutes, even for small quantities of data (1 GB or less). The hypothesis of this work is that the limitation is caused by Spark, which the system uses to write data on Apache Hudi tables. A promising approach to avoid using Spark appears to be adopting Delta Lake instead of Apache Hudi and access data using a Rust library called delta-rs. This thesis investigates the possibility of reducing the read and write latency in the offline feature store by expanding the delta-rs library to support HDFS and HopsFS and comparatively evaluating the performance of the legacy and newly implemented system. Two major iterations of storage support in delta-rs for HopsFS were developed to meet the strict production-ready requirements. The system was then evaluated by performing and measuring read and write operations increasing the number of CPU cores up to eight. Results confirmed the superior performance of the delta-rs library over the Spark system in all write operations with a latency reduction from ten up to forty times. Delta-rs also surpassed the Spark alternative in read operations with a latency reduction of forty-seven percent, up to forty times. These findings encourage future research investigating Spark alternative when optimizing performance in small-scale (1 GB - 100 GB) data management systems. The system developed will find application in the Hopsworks feature store production environment.

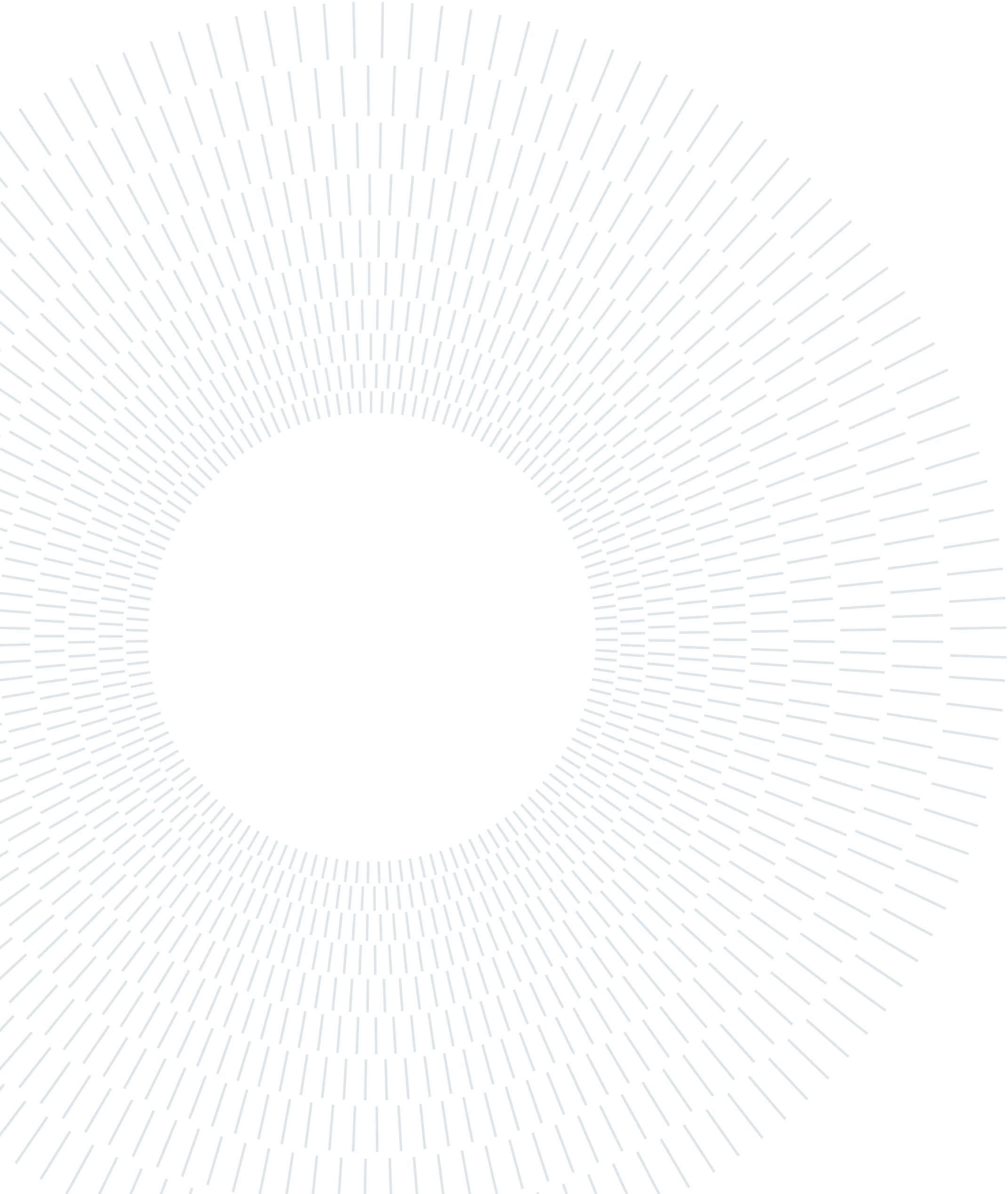
Keywords: Machine Learning, Feature Store, Spark, Delta Lake, delta-rs library, Read-/write latency



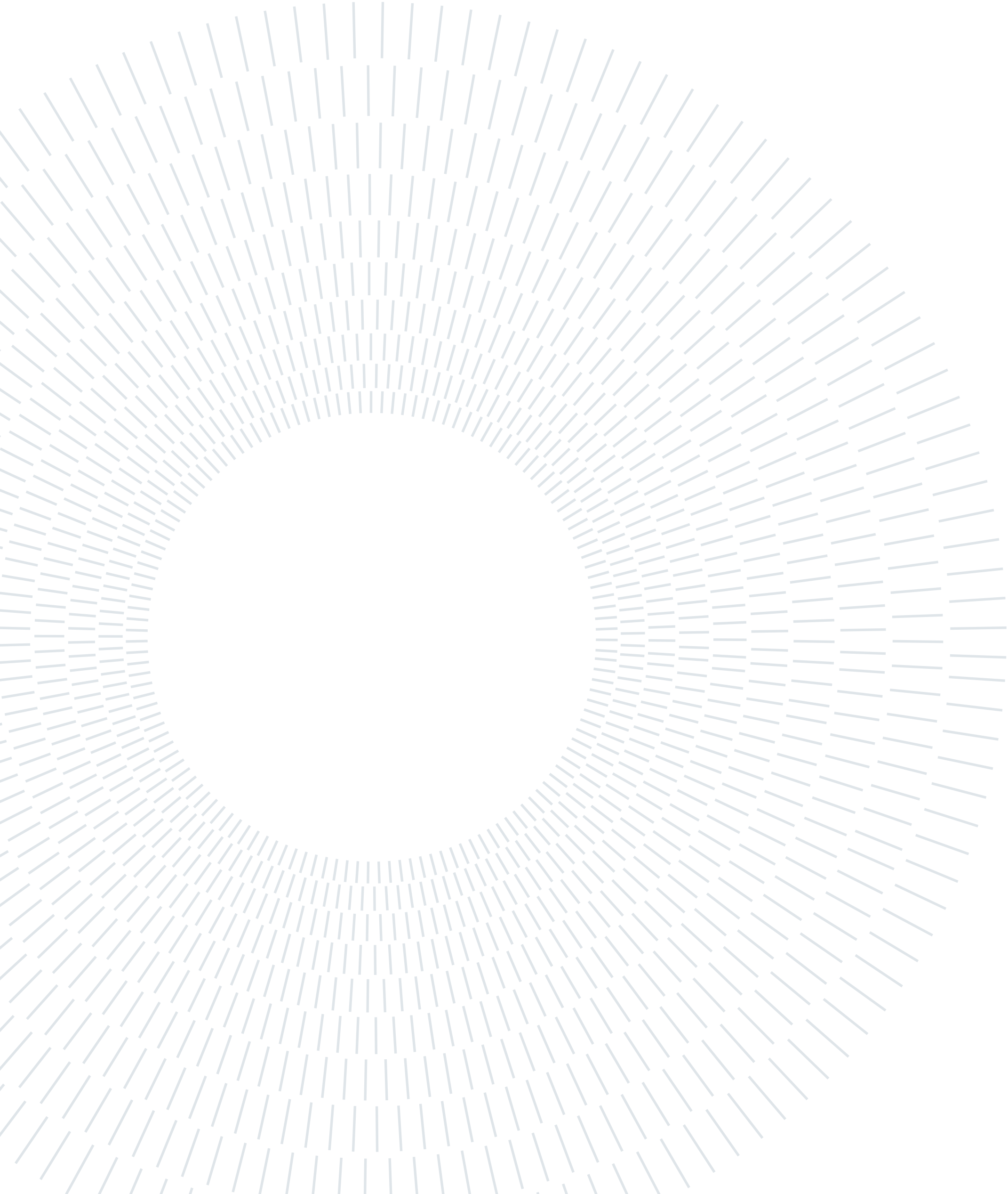
Abstract in lingua italiana

La necessità di costruire modelli di Machine Learning (ML) basati su grandi quantità di dati ha portato nuove sfide ai sistemi di gestione dei dati. I feature stores sono emersi come piattaforma dati centralizzata che consente il riutilizzo delle features. Recenti pubblicazioni dimostrano che il feature store di Hopsworks supera le alternative cloud esistenti. Nel suo offline feature store, Hopsworks memorizza dati batch o storici, raccogliendoli in feature groups organizzati in tabelle Apache Hudi e archiviati su HopsFS, la distribuzione HDFS di Hopsworks. Tuttavia, anche in questo sistema, la latenza per eseguire un'operazione di scrittura è di almeno uno o più minuti, anche per piccole quantità di dati (1 GB o meno). L'ipotesi di questa tesi è che la limitazione sia causata da Spark, che il sistema utilizza per scrivere i dati sulle tabelle di Apache Hudi. Un approccio promettente per evitare l'uso di Spark sembra essere l'adozione di Delta Lake al posto di Apache Hudi e l'accesso ai dati utilizzando una libreria Rust chiamata delta-rs. Questa tesi studia la possibilità di ridurre la latenza di lettura e scrittura nel feature store offline espandendo la libreria delta-rs per supportare HDFS e HopsFS e valutando in modo comparativo le prestazioni del sistema preesistente e di quello appena implementato. Sono state sviluppate due distinte implementazioni per aggiungere supporto in delta-rs per HopsFS, a causa dei severi requisiti di produzione. Il sistema è stato poi valutato eseguendo e misurando le operazioni di lettura e scrittura. I risultati hanno confermato la superiorità delle prestazioni della libreria delta-rs rispetto al sistema Spark in tutte le operazioni di scrittura, con una riduzione della latenza da dieci a quaranta volte. Delta-rs ha superato l'alternativa Spark anche nelle operazioni di lettura, con una riduzione della latenza del 47%, fino a quaranta volte. Questi risultati incoraggiano la ricerca futura sull'alternative a Spark per l'ottimizzazione delle prestazioni nei sistemi di gestione dei dati di piccola dimensione (1 GB - 100 GB). Il sistema sviluppato troverà applicazione nell'ambiente di produzione del feature store di Hopsworks.

Parole chiave: Machine Learning, Feature Store, Spark, Delta Lake, libreria delta-rs, Latenza di lettura/scrittura



Contents



1 | Introduction

Data lakehouse systems are increasingly becoming the primary choice for running analytics in large companies with over 1000 employees [13]. The data lakehouse architecture [3] is preferred over old paradigms, i.e., data warehouses and data lakes, as it builds upon the advantages of both systems, having the scalability properties of data lakes while preserving the Atomicity, Consistency, Isolation and Durability (ACID) properties and computation over the data typical of data warehouses [3]. Additionally, data lakehouse systems include partitioning, which reduces query complexity significantly and provides "time travel" capabilities, enabling users to access different versions of data, versioned over time [9].

Three main implementations of this paradigm emerged over time [47]:

1. **Apache Hudi**: first introduced by Uber [39], and now primarily backed by Uber, Tencent, Alibaba, and Bytedance.
2. **Apache Iceberg**: first introduced by Netflix, and now primarily backed by Netflix, Apple, and Tencent.
3. **Delta Lake**: first introduced by Databricks [2], and now primarily backed by Databricks and Microsoft.

While large communities support all three projects, Delta Lake is acknowledged as the de-facto data lakehouse solution [47]. This recognition is mainly thanks to Databricks, which first promoted this new architecture over data lakes among their clients around 2020 [2].

As a data query and processing engine, Delta Lake is typically used with Apache Spark (from now on, simply Spark) [50]. This approach is practical when processing large quantities of data (1 TB or more) in the cloud, but whether this approach is effective on a small data scale (1 GB - 100 GB) remains to be investigated [24].

DuckDB [38], a Data Base Management System (DBMS) and Polars [46], a DataFrame library, highlighted the limitations of Spark. When processing data locally with smaller volumes, an Spark cluster underperforms compared to other alternatives. This result ultimately increases costs and computation time when using Spark [14, 27]. This approach with small-scale (1 GB - 100 GB) use cases would improve performance significantly.

Another important consideration is that Python, due to its simplicity and high level of abstraction, has emerged as the most widely used programming language in the field of data science [29]. Python is currently the most popular general-purpose programming language [22, 32], and it is by far the most used language for Machine Learning (ML) and Artificial Intelligence (AI) applications [40]; this is mainly thanks to its strong abstraction capabilities and accessibility. This trend can also be observed by looking at the most popular libraries among developers, where two Python libraries make the podium: NumPy and Pandas [32]. In this scenario, using a Python client for Delta Lake would be beneficial as developers would not have to resort to Spark and its Python Application Programming Interface (API) (PySpark).

This native Python access for Delta Lake directly benefits Hopsworks *Aktiebolag*, tr. Limited company (AB), the host company of this master thesis. Hopsworks AB develops a homonymous feature store for ML. This centralized, collaborative data platform enables the storage and access of reusable features [1]. This architecture also supports point-in-time correct datasets from historical feature data [36].

This presented project aims to reduce the latency (seconds) and thus increase the data throughput (rows/second) for reading and writing on Delta Lake tables that act as an offline feature store in Hopsworks. Currently, the writing pipeline is Spark-based, and the fundamental hypothesis of the project is that a faster non-Spark alternative is possible. If successful, Hopsworks AB will consider incorporating this system into the open-source Hopsworks feature store, significantly enhancing the experience for Python users working with smaller datasets (1 GB - 100 GB). More generally, this work will outline the possibility of Spark alternatives in small-scale use cases.

This thesis's main contributions are the following:

- Two code implementations adding support for Hadoop Distributed File System (HDFS) and Hopsworks' HDFS distribution (HopsFS) in the delta-rs library. Of these, the first one is incomplete, as the second was preferred according to the consistency and maintainability requirements defined in Section 3.1.3. These code contributions are more than two thousand Lines Of Code (LOC) for the first implementation and eight hundred for the second. Note that while these metrics might provide some insight into the contribution's value, this work's true value is in cre-

---

[1] Definition from the company's website at `https://www.hopsworks.ai/`

ating a production-ready solution that correctly navigates a complex data stack of technologies with intricate dependencies, answering all requirements. The most relevant recognition of this contribution is the inclusion of this code implementation in a production environment in the Hopsworks feature store shortly after the thesis publication.

- The experiments' results detail the difference in performance between the newly implemented system and the legacy system in read and write operations expressed as latency and throughput. Experiments were also performed at different Central Processing Unit (CPU) configurations and table sizes fifty times, enabling a confidence interval estimate. Results report that the new system using the delta-rs library to access data has a latency reduction compared to the legacy system from ten up to forty times in write operations and from forty-seven percent up to forty times in read operations. These results are a solid contribution to the data management field, confirming present research on the limitations of using Spark with small amounts of data. Additionally, the value of this work consists of the large number of experiments conducted in a well-defined environment, which enables reproducible results using the code made available in this thesis.

1.1. Background

Three key aspects of this project are essential to a comprehensive understanding: the development of the data lakehouse architecture, the significance and workflows of Spark, and the emergence of Python as a dominant programming language.

Data lakehouse is a term coined by Databricks in 2020 [26] to define a new design standard emerging in the industry. This new paradigm combined the capability of data lakes in storing and managing unstructured data with the ACID properties typical of data warehouses. Data warehouses became a dominant standard in the '90s, and early 2000s [8], enabling companies to generate Business Intelligence (BI) insights, managing different structured data sources. The problems related to this architecture were highlighted in the 2010s when the need to manage large quantities of unstructured data rose [15]. So data lakes became the pool where all data could be stored, on top of which a more complex architecture could be built, consisting of data warehouses for BI and ML pipelines. This architecture, while more suitable for unstructured data, introduces many complexities and costs related to the need to have replicated data (data lake and data warehouse) and several Extract Load Transform (ELT) and Extract Transform Load (ETL) computations. Data lakehouse systems solved the problems of data lakes by implementing data management and performance features on top of open data formats such as Parquet [44]. Three key technologies enabled this paradigm: (i) a metadata layer for data lakes, tracking which files are part of different tables; (ii) a new query engine design, providing optimizations such as Random Access Memory (RAM)/Solid State Drive (SSD) caching; and (iii) an accessible API access for ML and AI applications. Uber first open-sourced this architecture design with Apache Hudi in 2017 [39], and then Databricks did the same with Delta Lake in 2020 [2].

Spark is a distributed computing framework used to support large-scale data-intensive applications [48]. Developed as an evolution of the MapReduce paradigm, Spark has become the de-facto standard for big data processing due to its superior performance and versatility. Spark significantly improved its performance compared to its predecessor, i.e., Hadoop MapReduce (10 times better in its first iteration) [48] thanks to its use of in-memory processing. This feature means that Spark avoids going back and forth between storage disks to store the computation results. Spark, open-sourced under the Apache foundation as Apache Spark (from now on referred to as Spark), has seen widespread success and adoption in various applications, becoming the de-facto data-intensive computing platform for the distributed computing world. While Spark is often used as a comprehensive solution [50], different solutions might be better suited for a specific scenario. An example of this is the case of Apache Flink [7], designed for real-time data streams, which prevails over Spark where low latency real-time analytics are required. Similarly, Spark might not be the best tool for lower-scale applications where Spark's high-scaling capabilities may not be necessary. This is the case of DuckDB [38] and Polars [46], that by focusing on a small-scale data (1 GB - 100 GB) provide a fast On-Line Analytical Processing (OLAP) embedded database and DataFrame management system respectively offering an overall faster computation compared to starting a Spark cluster for to perform the same operations. These technologies demonstrate that new applications outperforming Spark in specific use cases are possible and already in use. In particular, Apache Flink and DuckDB show that this is possible for real-time data streaming or small-scale computation. In this project, the latter use case is going to be explored.

Python can be considered the primary programming language among data scientists [45]. Many first adopted Python thanks to its focus on ease of use, high abstraction level, and readability. These features helped create a fast-growing community behind the project, which led to the development of many libraries and APIs. So now, more than thirty years after its creation, it has become the de-facto standard for data science thanks to many daily used Python libraries such as TensorFlow, NumPy, SciPy, Pandas, PyTorch, Keras and many others. Python is also considered to be the most popular programming language, according to the number of results by search query (*+"<language> programming"*) in 25 different search engines [2]. This ranking is computed yearly in the TIOBE Index [22]. The April 2024 rankings reveal that Python holds a rating of 16.41%, followed by C at 10.21%. The index also highlights trends from recent years, clearly illustrating Python's rise over traditionally popular languages like C and Java, both of which Python surpassed between 2021 and 2022. These scores underline the importance of providing Python APIs, particularly for programmers and data scientists, to enhance engagement and expand the capabilities of a framework.

1.2. Problem

The Hopsworks feature store [10] first used Apache Hudi for their offline feature store, as it was the first open-sourced data lakehouse in 2017. Recently, Hopsworks AB added support for using Delta Lake as an offline feature store, following its clients' requests. Spark is the system's query engine, i.e., it executes the query (read, write, or delete) on the offline feature store. Running the system showed that even a write operation on a small dataset, consisting of 1 GB of data or less, takes one or more minutes to complete.

This hurts Hopsworks' typical use case, which sits between tests on small quantities of data (1 GB - 10 GB) and production scenarios on a larger scale but still relatively small (10 GB - 100 GB).

This research's underlying hypothesis is that this slow transaction time is a Spark-specific issue. This has led Hopsworks to adopt Spark alternatives [24] for reading in their Apache Hudi system. Delta Lake supports Spark alternatives for accessing and querying the data, and of particular interest is the delta-rs library [3] that enables Python access to Delta Lake tables without using Spark. However, the delta-rs library does not support HDFS, and consequently HopsFS [30].

1.2.1. Research Questions

This research project has the ultimate objective to evaluate and compare the performance of the current Spark system that operates on Apache Hudi to a Rust system that uses delta-rs library [3] operates on Delta Lake, using HopsFS [30]. To achieve this, support for HDFS (and thus also HopsFS) must be added to the delta-rs library so that it can be compatible with the Hopsworks system. Therefore, the project addresses the following

---

[2] Evaluation methodology defined at `https://www.tiobe.com/tiobe-index/programminglanguages_definition/`

[3] Project repository available at `https://github.com/delta-io/delta-rs`

two Research Questions (RQs):

RQ1: How can we add support for HDFS and HopsFS to the delta-rs library to enable reading and writing on Delta Lake tables on the Hopsworks offline feature store?

RQ2: What is the difference in read and write latency and throughput between the current legacy system operating on the Hopsworks offline feature store and the delta-rs library operating on HopsFS?

Note that RQ2 was formulated to reflect the experiments that will be performed. This is because, even though measured performance should be similar if delta-rs will be included in the Hopsworks client in the future, it is formally not operating on the offline feature store but only using the same file system, HopsFS.

1.3. Purpose

This thesis project aims to reduce the read and write latency (seconds) and thus increase the data throughput (rows/second) for operations on the Hopsworks offline feature store. This study will compare the performance of the current legacy pipeline, Spark-based for writing, with the delta-rs pipeline on a small-scale data by evaluating differences in terms of latency and throughput in read and write operations. As a prospect for future work, if delta-rs is proven to be a more performant alternative, Hopsworks AB will consider integrating this pipeline into their application.

Overall implications for this thesis work are much broader considering Spark's popularity in the open-source community (more than 2800 contributors during its lifetime [31]). Choosing delta-rs over Spark gives developers a broader range of alternatives when working on a small-scale data (1 GB - 100 GB).

1.4. Goals

This project aims to reduce the latency and thus increase the data throughput for reading and writing on Delta Lake tables on HopsFS. The accomplishment of this purpose is bound to a list of Goals (Gs), here set. These are also related to the set of RQs, outlining a clear structure of the various project milestones.

1. Gs aimed to answer RQ1:

   G1: Understand delta-rs library architecture and dependencies.

   G2: Identify what needs to be implemented to add HDFS support to the delta-rs library.

G3: Implement HDFS support in the delta-rs library.

G4: Verify that HDFS support is also compatible with HopsFS.

2. Gs aimed to answer RQ2:

G5: Design the experiments to evaluate the performance difference between the current legacy access to Apache Hudi and the delta-rs library-based access to Delta Lake, in HopsFS.

G6: Perform the designed experiments.

G7: Visualize the experiments' results, focusing on allowing an effective comparison of performances.

G8: Analyze and interpret the results in a dedicated thesis report section.

The completion of these Gs will create several Deliverables (Ds) listed below:

D1: Code implementation adding support to HDFS and HopsFS in the delta-rs library. This D is related to completing goals G1–G4. This deliverable also represents the project's system implementation contribution.

D2: Experiment results on the difference in performance between current legacy access to Apache Hudi and the delta-rs library-based access to Delta Lake, in HopsFS. This D is related to completing goals G5–G7.

D3: This thesis document. It provides more detail on the implementation, design decisions, expected performance, and analysis of the results. This D is a comprehensive report of all the thesis work, including the analysis of results defined in G8.

1.5. Ethics and sustainability

As a systems research project, the focus of this study revolves around software and, in particular, developing more efficient data-intensive computing pipelines that find wide applications in ML and the training of neural networks. Software, according to the Green Software Foundation [4], can be "part of the climate problem or part of the climate solution" [20]. Green Software can be defined as software that reduces its environmental impact by using less physical resources and less energy and optimizing energy use to use lower-carbon sources [20]. In the context of ML and training of neural networks, reducing training time

---

[4] Foundation's website available at `https://greensoftware.foundation/`

7
AFFORDABLE AND CLEAN ENERGY
9
INDUSTRY, INNOVATION AND INFRASTRUCTURE

Figure 1.1: Illustrations of the SDG supported by this thesis.

(and so also the read and write latency operation on the dataset) has been proven to positively impact the reduction in carbon emissions [33, 34].

This project contributes to the Sustainable Development Goals (SDGs) [5] 7 – Affordable and Clean Energy and 9 – Industry Innovation and Infrastructure, more specifically the targets 7.3 – Double the improvement in energy efficiency and 9.4 – Upgrade all industries and infrastructures for sustainability. This thesis contributes to these goals by reducing latency and thus increasing the data throughput for reading and writing on Delta Lake tables on HopsFS. This purpose also follows the fundamental green software principles reducing CPU time use compared to the previous system. Reducing CPU usage time reduces energy consumption, leading to a lower carbon footprint.

Ultimately, this will lead to improved energy efficiency and reduced carbon footprint of data-intensive computing pipelines, which find wide applications in ML and neural network training.

1.6. Research methodology

This work starts from a few Industrial Needs (INs), provided by Hopsworks, and a few Project Assumptions (PAs) validated through a literature study.
Hopsworks's INs are:

IN1 : the Hopsworks feature store using the legacy pipeline has high latency (one or more minutes) and low throughput in writing operations on a small-scale data (1 GB - 100 GB). These performances highlight the potential for using Spark alternatives in a small-scale data use case.

IN2 : Hopsworks, adapting to their customer needs, supports the Delta Lake table format. Improving the read and write operations speed on this table format would improve

---

[5] SDGs website available at `https://sdgs.un.org/`

a typical use case for Hopsworks feature store users.

The PAs are listed below. These assumptions will be validated in Chapter 2.

PA1 : Python is the most popular programming language and the most used in data science workflows. ML and AI developers prefer Python tools to work. This popularity means that high-performance Python libraries will typically be preferred over alternatives (even more efficient) that are Java Virtual Machine (JVM) or other environments based.

PA2 : Rust libraries have proven to have the chance to improve performance over C/C++ counterparts (Polars over Pandas). A Rust implementation could strongly improve reading and writing operations on the Hopsworks feature store.

The project aims at fulfilling the INs with a system implementation approach. First, a HDFS storage support will be written for the delta-rs library to extend the Rust library support to HopsFS [30]. Then, an evaluation structure will be designed and used to compare the performances of the current legacy system and the new Rust-based pipeline. The two approaches will be tested with datasets of different sizes (between 1 GB and 100 GB). This approach is critical to identify if the same system should be used for all scenarios or if they perform differently. The metrics that will be used to evaluate the system are read and write operations latency measured in seconds and data throughout measured in rows per second. Note that latency will be the only metric to be measured, while throughput will be computed from the latency. These metrics were chosen as they most affect the computation time of pipelines accessing Delta Lake tables.

1.6.1. Delimitations

The project is conducted in collaboration with Hopsworks AB, and as such the implementation will focus on working with their system using HopsFS. While the consideration drawn from these results cannot be generalized and be true for any system, they can still provide an insight into Spark limitations, and on which tools perform better in different data scales.

1.7. Thesis structure

Chapter 2 equips readers with the necessary foundational knowledge to navigate the layered data stack that this research operates with. Furthermore, it also introduces the legacy and new system architectures that will be employed during the experiments. Chapter 3 defines the methodology for the two key components of this iterative thesis work, i.e., the system implementation and system evaluation. Chapter 4 details the design choices taken during the system development and describes how to deploy use and the system. Chapter 5 presents the experiments' results, outlining the differences between the defined pipelines and the different CPU configurations. The chapter is also complemented with a discussion section that enables readers to appreciate this thesis's findings and implications. Finally, Chapter 6 summarizes the thesis contribution and findings while discussing the limitations and future work.

2 | Background

This project works on a layered data stack that handles big data, i.e., large volumes of structured and unstructured data types at a high velocity. The data stack handles how data is stored, managed, and retrieved to enable applications built on top of it. No single data stack applies to all cases, i.e., different approaches use different architectures [16, 41]. Therefore, this project defines a data stack and then focuses on the mainly two layers, i.e. the query engine and data management layer. The data stack of the project is illustrated in Figure 2.1.

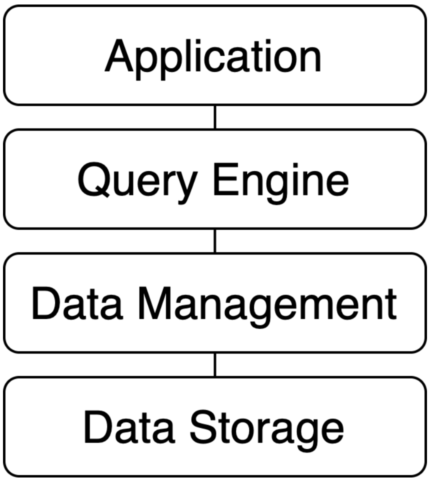


Figure 2.1: Data stack abstraction for this project.

The data stack and this chapter are divided into four sections:

1. **Data Storage**: handles how the data is stored. The data storage layer might be centralized or distributed, on-premise or in the cloud, and store data in files, objects, or blocks.
2. **Data Management** : handles how the data is managed. The data management layer might offer ACID properties, data versioning, support open data formats, and support structured and unstructured data.
3. **Query Engine**: handles how data is queried, i.e., accessed, retrieved and written.

The query engine might offer caching, highly scalable architectures, and API support for multiple programming languages.

4. **Application**: a system that will take advantage of the data stack. In the case of this project, only the software the Hopsworks feature store software will be described.

After explaining the data stack, a section on the legacy and new system architectures complements the Background, showing how the technologies explained function within the pipelines measured during the experiments.

2.1. Data storage

This section describes the data storage layer of this project, namely HopsFS. HopsFS is Hopsworks's evolution of HDFS, a distributed file system. HopsFS complexity will be broken down into parts, providing a great understanding of the tool and a comparison with common alternatives, namely cloud object stores.

2.1.1. File storage vs. Object storage vs. Block storage

Data can be stored and organized in physical storages, such as Hard Disks Drives (HDDs) or SSDs, in three major ways: (1) Files, (2) Objects, and (3) Blocks. Each technique is briefly described, and then a comparative table is shown (Table 2.1). This subsection is a re-elaboration of three articles from major cloud providers (Amazon, Google, and IBM) [1, 17, 21] according to the author's understanding.

File storage

File storage is a hierarchical data storage technique that stores data into files. A file is a collection of data characterized by a file extension (e.g. ".txt", ".png", ".csv", ".parquet") that indicates how the data contained is organized. Every file is contained within a directory that can contain other files or directories (called "subdirectories"). In many file storage systems, directories are called "folders".

This type of structure, prevalent in modern Personal Computers (PCs), simplifies locating and retrieving a single file, and its flexibility allows it to store any kind of data. However, its hierarchical structure requires that to access a file, its exact location should be known. This restriction decreases the scaling possibilities of the system, where a large amount of data must be retrieved simultaneously.

Overall, this solution is still vastly popular in user-facing storage applications (e.g., Dropbox, Google Drive, One Drive) and PCs thanks to its intuitive structure and ease of use. On the other hand, other options are preferred for managing large quantities of data due to its lack of scalability.

Advantages

- Ideal for small-scale operations (low latency, efficient folderization).
- User familiarity and ease of management.
- File-level access permissions and locking capabilities.

Disadvantages

- Difficult to scale due to deep folderization.
- Inefficient in storing unstructured data.
- Limitations in scalability due to reaching device or network capacity.

Object storage

Object storage is a flat data storage technique that stores data into objects. An object is an isolated container associated with metadata, i.e., a set of attributes that describe the data, e.g., a unique identifier, object name, size, and creation date. Metadata is used to retrieve the data more quickly, allowing for queries that retrieve large quantities of data simultaneously, e.g., all data created on a specific date.

The flat structure of an object storage, where all objects are in the same container, called a bucket, is ideal for managing large quantities of unstructured data (e.g., videos, images). This structure is also easier to scale as it can be replicated across multiple regions, allowing faster access in different areas of the world and fault tolerance to hardware failure.

On the other hand, objects cannot be altered once created, and in case of a change, they must recreated. Also, object stores are not ideal for transactional operations, as objects cannot have a locking mechanism. Lastly, object stores have slower writing performance than file or block storage solutions.

Overall, this solution is widely used when high scalability is required (e.g., social networks and video streaming apps) thanks to its flat structure and use of metadata. On the other hand, other options are preferred when transactional operations are required or when high performance on a small number of files that change frequently is necessary.

Advantages

- Potential unlimited scalability.
- Effective use of metadata enabling advanced queries.
- Cost-efficient storage for all types of data (also unstructured).

Disadvantages

- Absence of file locking mechanisms.
- Low performance (increased latency and processing overhead).
- Lacks data update capabilities (only recreation).

Block storage

Block storage is a data storage technique that divides data into blocks of fixed size that can be read or written individually. Each block is associated with a unique identifier and then stored on a physical server (note that a block can be stored in different Operating Systems (OSes)). When the user requests the data saved, the block storage retrieves the data from the associated blocks and then re-assembles the data of the blocks into a single unit. The block storage also manages the physical location of the block, saving a block where is more efficient.

Block storage is very effective for systems needing fast access and low latency. This architecture is compatible with frequent changes, unlike object storage.

On the other hand, block storage achieves its speed by operating at a low level on physical systems, so the cost of the architecture is strictly bound to the storage and servers used, not allowing the architecture to scale according to its demands.

Advantages

- High performance (low latency).
- Reliable, self-contained storage units.
- Data stored can be modified easily.

Disadvantages

- Lack of metadata brings limitations in data searchability.
- High cost to scale the infrastructure.

Table 2.1: Data storage features comparison. Table inspired by major cloud providers articles [1, 17].

Characteristics	File Storage	Object Storage	Block Storage
Performance	High	Low	High
Scalability	Low	High	Low
Cost	High	Low	High

2.1.2. Hadoop Distributed File System

HDFS is a Distributed File System (DFS), i.e., a file system (synonym of file storage) that uses distributed storage resources while providing a single namespace as a traditional file system. HDFS has significant differences compared with other DFSes. HDFS is highly fault-tolerant, i.e., it is resistant to hardware failures of part of its infrastructure, and can be deployed on commodity hardware. HDFS also provides high throughput access to application data, and it is designed to be highly compatible with applications with large datasets (more than 100 GB) [6].

HDFS architecture consists of a single primary node called Namenode and multiple secondary nodes called Datanodes. The Namenode manages the file system namespace and regulates clients' access to files. On the other hand, Datanodes manage the storage attached to the nodes they run on, and they are responsible for performing replication requests when prompted by the Namenode. HDFS exposes to users a file system namespace where data can be stored in files. Internally, a file is divided into one or multiple blocks stored in a set of Datanodes. The blocks are also replicated upon the first write operation, up to a certain number of times (by default, three times, with at least one copy on a different physical infrastructure). The Namenode keeps track of the data location, matching it with the file system namespace. It is also responsible for managing Datanode reachability (through periodical state messages sent by Datanodes) and providing clients with the Datanodes' locations containing the blocks composing the requested file. If a new write request is received, it is still the Namenode that needs to provide the locations of available storage for the file blocks.

Figure 2.2 presents a simplified visual representation of the Namenode and Datanodes basic operations in HDFS.

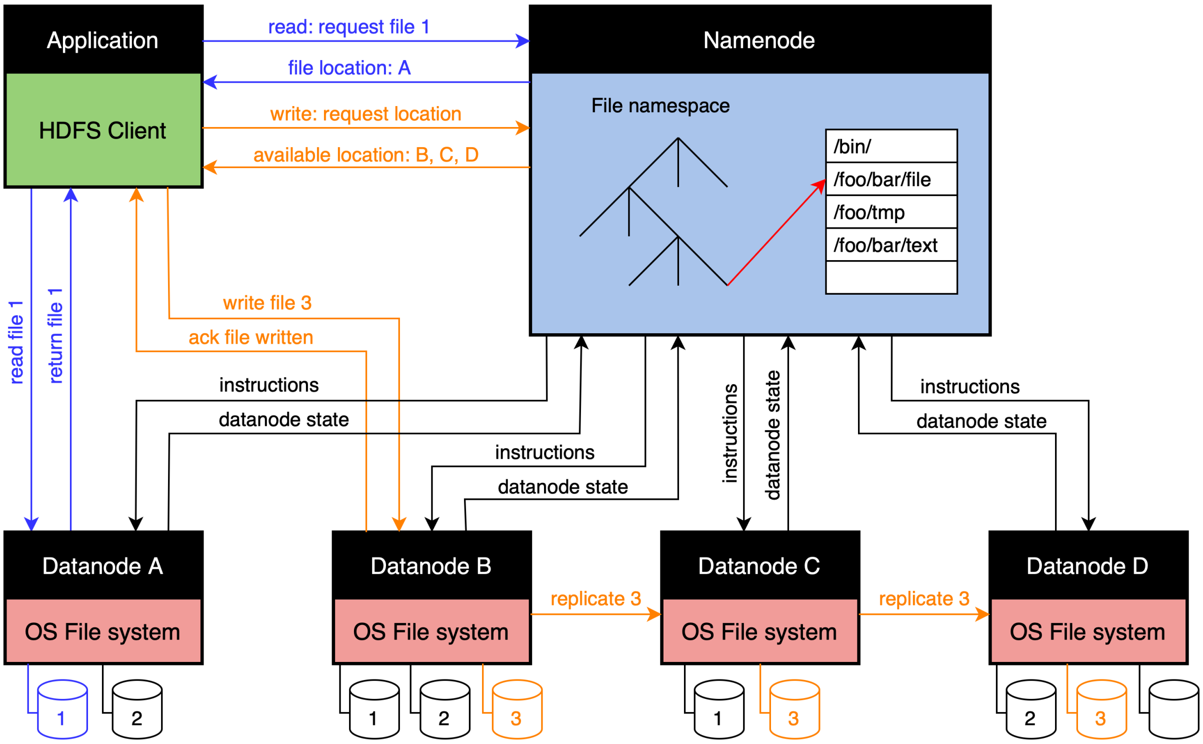


Figure 2.2: Hadoop Distributed File System architecture displaying in different colors basic operations: read (blue), write (orange) and Namenode-Datanodes management messages (black). Note: for representation simplicity, files are not segmented into blocks. Diagram inspired by the Data-intensive Computing lectures at KTH by Prof. A. H. Payberah. Course website available at `https://www.kth.se/student/kurser/kurs/ID2221?l=en`.

2.1.3. HopsFS as an HDFS evolution

HopsFS is HDFS improved release first presented at the 15th USENIX conference in 2017 [30]. HopsFS stores the metadata in an external NewSQL database with high throughput and low operation latency called RonDB [1]. This approach enables HopsFS to avoid having all metadata in a single Namende, allowing for Namenode replication and having the same metadata on RonDB. This solution proved to have from sixteen up to thirty-seven times the performance on HDFS thanks to its ability to scale according to metadata being stored and on similar setups where the same resources are being utilized.

HopsFS is one of the core technologies on which the Hopsworks feature store is built. While the system has not seen more widespread use, it is highly impactful in its current applications, contributing to making Hopsworks "outperform existing cloud feature stores for training and online inference query workloads" [10].

---

[1] Project's website available at `https://www.rondb.com`

2.1.4. HDFS alternatives: Cloud object stores

Thanks to its high scalability and low cost (Section 2.1.1), object storage has been widely adopted to store large quantities of unstructured data. Starting with Amazon Web Services (AWS) in 2006 with its S3 service, many other vendors started offering cloud object storage services. The main advantages of using cloud resources are related to their elasticity, which, combined with object storage capability, enables users to use as much storage as they need and be billed for what was used.

HDFS was first released in 2006, and so it evolved in parallel with cloud objects storage solutions. While HDFS was widely adopted for on-premise solutions, more and more businesses migrated their operations to cloud services. Nowadays, cloud object storage like AWS's S3, Google's Google Cloud Storage (GCS), and Microsoft's Azure are widely used, and libraries, e.g., delta-rs, prioritize support for these platforms as they are widely adopted. HDFS and its evolutions, e.g., HopsFS, still see use, but it fits more specific use cases, as the convenience of a cloud service is highly valued by the market.

2.2. Data management

This section describes the data management layer for this project, i.e., how data is managed, versioned, etc. The main DBMS technology used in this is Delta Lake, and to understand it, Section 2.2.1 revises the problems that technologies aimed to solve and the limitations of these systems. The chapter is complemented with Section 2.2.2, which explains how to access Delta Lake and introduces delta-rs.

2.2.1. Brief history of Data Base Management Systems

In recent years, the rise of big data, large volumes of various structured and unstructured data types at a high velocity, has shown incredible potential. Still, it has also posed several challenges [35]. These primarily impact the software architecture that needs to deal with these issues, which led to an evolution of these technologies [18]. Delta Lake [2] is one of the most recent iterations of this evolution process. To understand this tool well, it is necessary to understand its challenges, starting from the beginning of the data management evolution.

Before big data, companies wanted to gain insights from their data sources using an automated workflow. Here is where ETL and relational databases first came into use. An ETL pipeline, as the name suggests:

1. **Extracts** data from APIs or other company's data sources.
2. **Transforms** data by removing errors or absent fields, standardizes the format to match the database, and validates the data to verify its correctness.
3. **Loads** it into a relational database (e.g., MySQL).

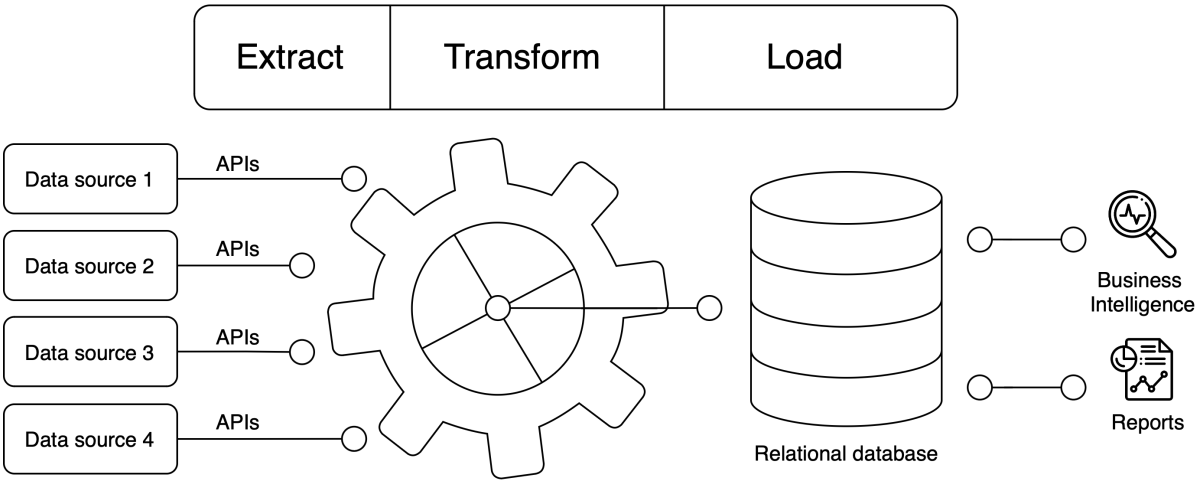


Figure 2.3: ETL system with a relational database. Diagram inspired by AltexSoft educational video available at `https://www.youtube.com/watch?v=qWru-b6m030`.

This type of workflow, represented in Figure 2.3, enabled companies to obtain BI insights and data reports on the company's data. The main limitation of this system sat in its limited capability of creating reports or BI insights based on data sitting on multiple tables. These types of requests are called analytical queries. While they might run less often than simpler queries, they are still crucial for making data-driven decisions (e.g., determining the region that sold more product units in the last year).

When the need to compute analytical queries rose, more complex DBMS substituted the simple relational databases, optimizing for running business-centric complex analytical queries. These systems are called OLAP; its prime example is the data warehouse.

A data warehouse workflow, visualized in Figure 2.4, enables larger quantities of data to be computed and analyzed. Data warehouses enable BI and Reports that consider all data sources and can join multiple tables efficiently. This type of DBMS still keeps relational databases' key features, such as ACID transactions and data versioning.

Over time, the rapidly growing amount of unstructured data (also called big data, e.g., images and videos) created new needs within companies that wanted to take advantage of this new data. Data warehouses were unfit to solve this problem as they only supported

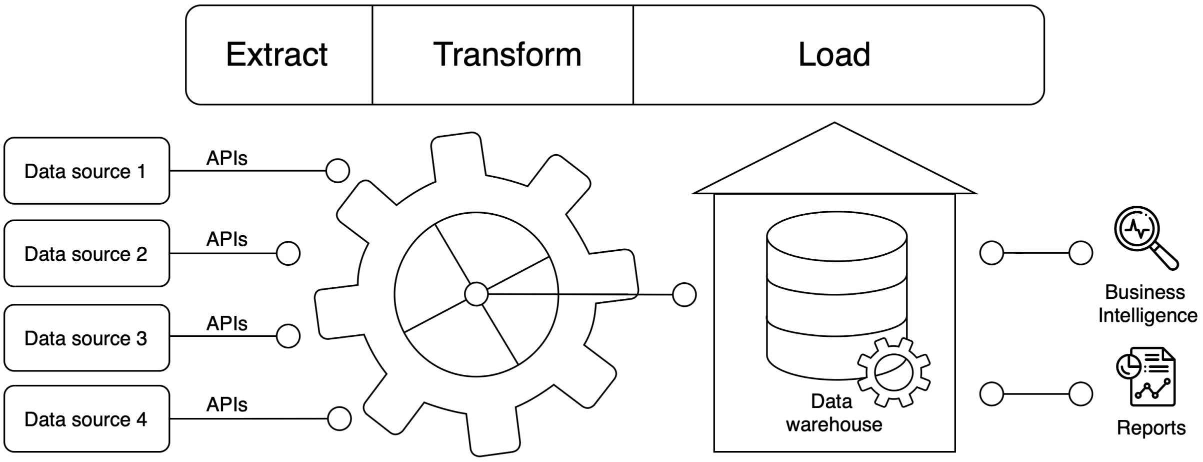


Figure 2.4: ETL system with a data warehouse. Diagram inspired by AltexSoft educational video available at `https://www.youtube.com/watch?v=qWru-b6m030`.

structured data. Furthermore, storing large quantities of data in data warehouses is expensive and does not support any AI/ML workflow.

These issues were tackled by a new paradigm called data lake (Figure 2.5). Data lakes are based on a low-cost object storage system (Section 2.1.1) that consists of a flat structure where all data is loaded after extraction. The architecture structure changes in data lakes as data is first loaded into the data lake and only after it is transformed. This paradigm is called ELT. Transformations are customizable for specific applications, e.g., BI and reports using a data warehouse, an AI/ML analysis.

This architecture reduces storage costs but also increases the system complexity. Higher complexity is typically related to higher costs, as system maintenance is more costly and can lead to more issues. Additionally, since a data lake cannot be queried directly with BI queries or requesting business reports, this leads to the need to maintain a data warehouse still (as in Figure 2.5). This limitation ultimately leads to higher costs in maintaining the multiple storages for the same data. The system also suffers from timeliness due to this lengthy pipeline, as the data needs to go through many steps before it is available in the data warehouse.

These issues outlined that data lakes were not a drop-in replacement for data warehouses, as they served a different purpose and suffered from other problems. These new issues generated the need for a system with data warehouses ACID and data management capabilities while supporting unstructured data. The solution was a new architecture, the data lakehouse.

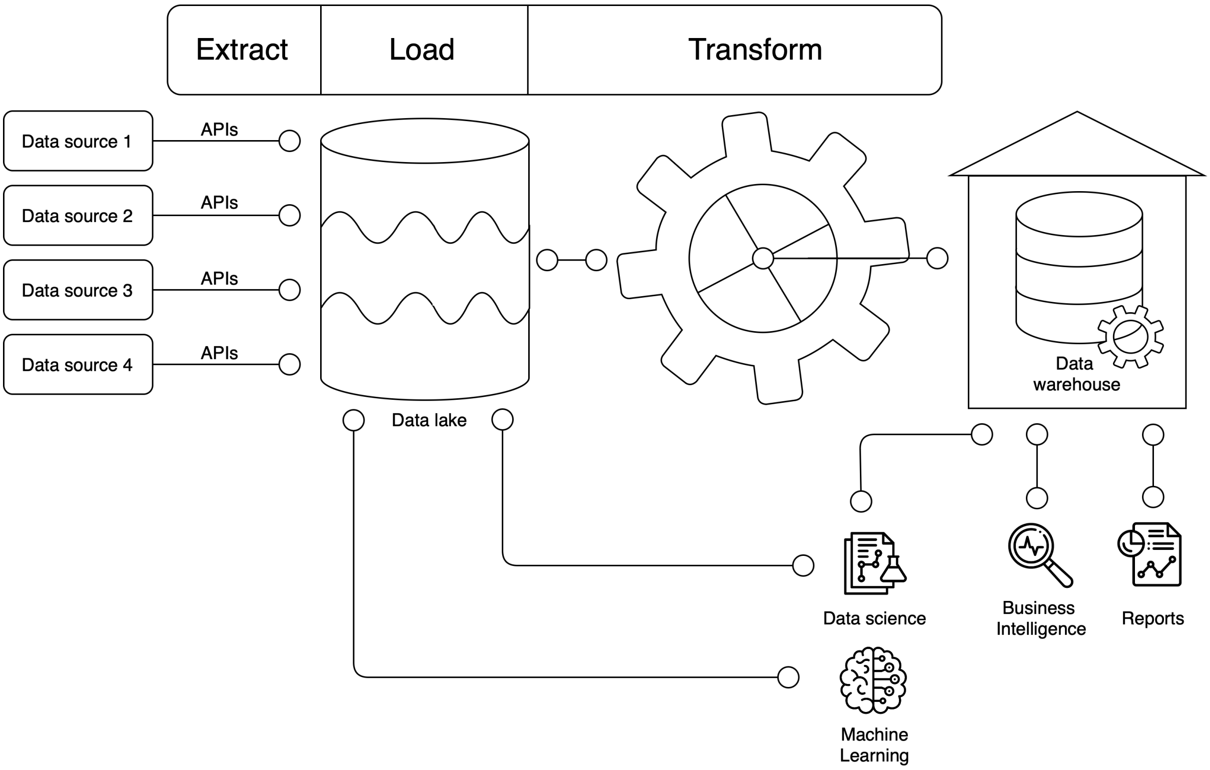


Figure 2.5: ELT system with a data lake. Diagram inspired by AltexSoft educational video available at `https://www.youtube.com/watch?v=qWru-b6m030`.

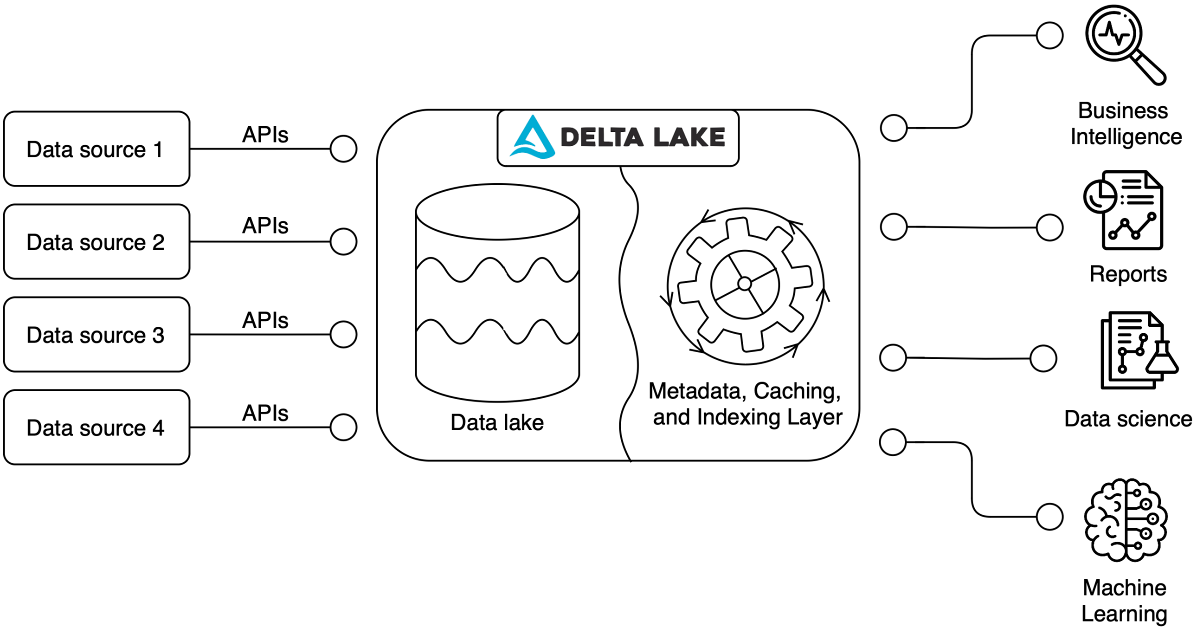


Figure 2.6: Delta lake architecture. Diagram inspired by Delta Lake paper [2].

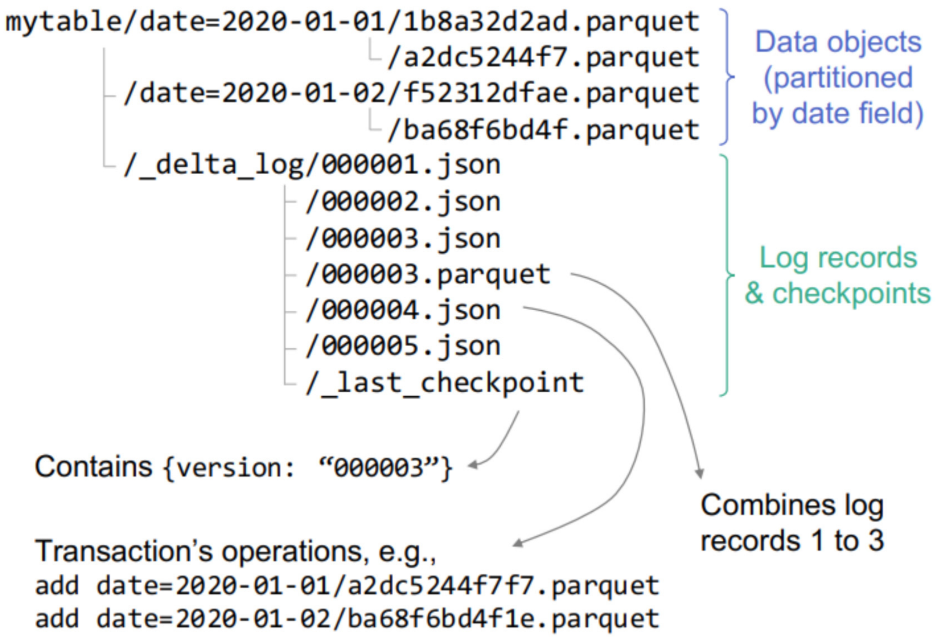

mytable/date=2020-01-01/1b8a32d2ad.parquet
/a2dc5244f7.parquet
/date=2020-01-02/f52312dfae.parquet
/ba68f6bd4f.parquet
/_delta_log/000001.json
/000002.json
/000003.json
/000003.parquet
/000004.json
/000005.json
/_last_checkpoint
Data objects (partitioned by date field)
Log records & checkpoints
Contains {version: "000003"}
Combines log records 1 to 3
Transaction's operations, e.g.,
add date=2020-01-01/a2dc5244f7f7.parquet
add date=2020-01-02/ba68f6bd4f1e.parquet

Figure 2.7: Delta lake table partitioned according to date field. Figure from the slides of the Data-intensive Computing lectures at KTH by Prof. A. H. Payberah. Course website available at `https://www.kth.se/student/kurser/kurs/ID2221?l=en`.

Figure 2.6 shows a Delta Lake system [2]. The data lakehouse term was first introduced by Databricks in 2021 with their paper presented at Conference on Innovative Data Systems Research (CIDR) [3], while data lakehouse solutions already existed on the market, namely Apache Hudi [39], Apache Iceberg and Delta Lake [2]. Data lakehouses combine the benefits of data warehouses and data lakes while simplifying the complexity of storing and accessing data in enterprise data architectures. This new architecture is based on an open-data format called Apache Parquet (from now on, simply parquet), a column-based data file format. Data is saved in a data lake in the form of parquet files. Then, a management layer enables managing transactions, versioning, indexing, and other data management features. These features allow data to lakehouses to offer ACID properties, similarly to data warehouses.

Delta Lake is the technology that will be most used in this project. A Delta Lake Table (i.e., an instance of Delta Lake) operates on a data lake containing parquet files and a transaction log. The transaction log records each operation, enabling versioning and recovery of previous versions (also called time travel). The data lake can be partitioned, e.g., using a date field, making the Delta Lake table look like Figure 2.7.

2.2.2. Accessing Delta Lake

The Delta Lake project is strictly related to Spark, as Databricks (the company that developed Delta Lake) was built by the Spark developers [48] to offer big data management services around Spark. As of version 3.0 of Delta Lake, Delta Kernel was announced [23], a Java library providing low-level access to Delta Lake without needing to write the Delta Lake logic. While this move tried to standardize all accesses to Delta Lake under the Spark/Java environment, new ways to access the library had already been written since Delta Lake was already open-sourced before the Delta Kernel release.

As early as April 2020, the Rust community started implementing a new interface to access Delta Lake written in Rust without JVM dependencies. This library expanded the Delta Lake ecosystem even more, breaking the dependency between Delta Lake and Spark to perform operations on the data lakehouse. This worked particularly well considering that the data science community heavily uses Python and would typically avoid having JVM dependencies. Being delta-rs written in Rust makes the library highly compatible with Python since it can be easily wrapped and deployed as a Python library. This is perhaps why delta-rs can be installed in Python with the name "deltalake".

2.3. Query engine

This section describes the technologies used to query, cache, and process data in this project. The query engine refers mainly to Apache Spark and DuckDB, but the other technologies presented in this chapter operate at the same abstraction level while having different functions.

2.3.1. Apache Spark

Apache Spark (from now on simply Spark) is an open-source distributed computing framework designed to handle large-scale data-intensive applications [50]. Spark builds from the roots of MapReduce and its variants. MapReduce is a distributed programming model first designed by Google that enables the management of large datasets [11]. The paradigm was later implemented as an open-source project by Yahoo! engineers under the name of Hadoop MapReduce [6]. Spark improved this approach using Resilient Distributed Datasets (RDDs) [49]. RDDs are a distributed memory abstraction that enables a lazy in-memory computation tracked through lineage graphs, ultimately increasing fault tolerance [49]. This difference between Hadoop MapReduce and Spark is represented in Figure 2.8.

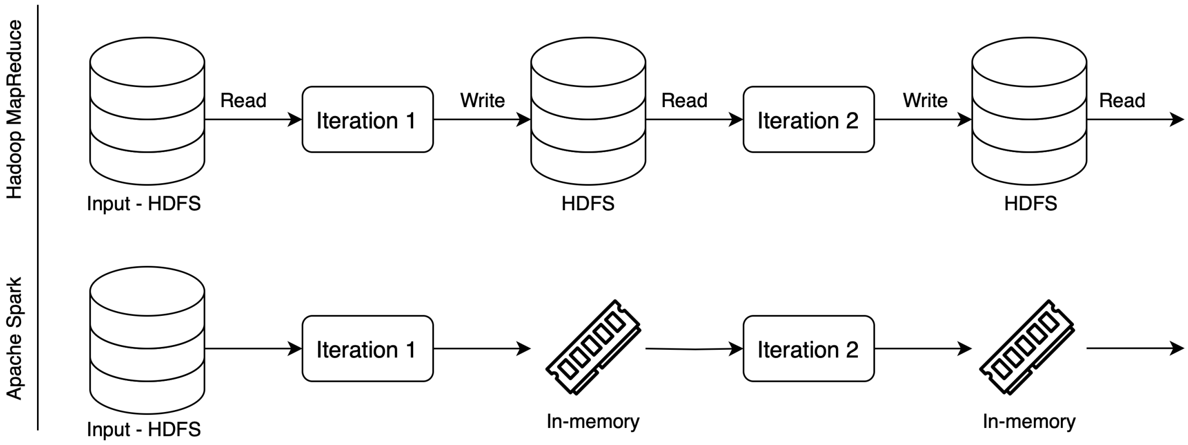


Figure 2.8: Hadoop MapReduce and Apache Spark execution differences. Apache Spark enables fast in-memory computation instead of continuously saving and loading data from disks as Hadoop MapReduce. Diagram inspired by the Data-intensive Computing lectures at KTH by Prof. A. H. Payberah. Course website available at `https://www.kth.se/student/kurser/kurs/ID2221?l=en`.

2.3.2. Apache Kafka

Apache Kafka (from now on, Kafka) is an open-source distributed data streaming platform designed for high-throughput and scalable data processing [25]. Kafka is most typically used for real-time streaming applications thanks to low latency.

Figure 2.9 shows the components and messages exchanged in a Kafka cluster. The key components of Kafka are:

1. **Producer**: an application that produces and labels data with specific topics (shapes in the figure).
2. **Zookeeper**: responsible for keeping track of brokers and the topic's current offset.
3. **Broker**: a computational node (or server) that handles data on a specific topic. It is responsible for receiving messages from producers. Once messages are received, the broker forwards them upon request by the consumers. This mechanism enables the asynchronous protocol.
4. **Consumer**: an application interested in topic-specific data. To access data, it is subscribed to a topic and receives new messages when published. More consumers can subscribe to a topic.

The protocol allows applications acting as producers and consumers to avoid having a synchronous protocol. This feature enables producers to achieve high throughput since

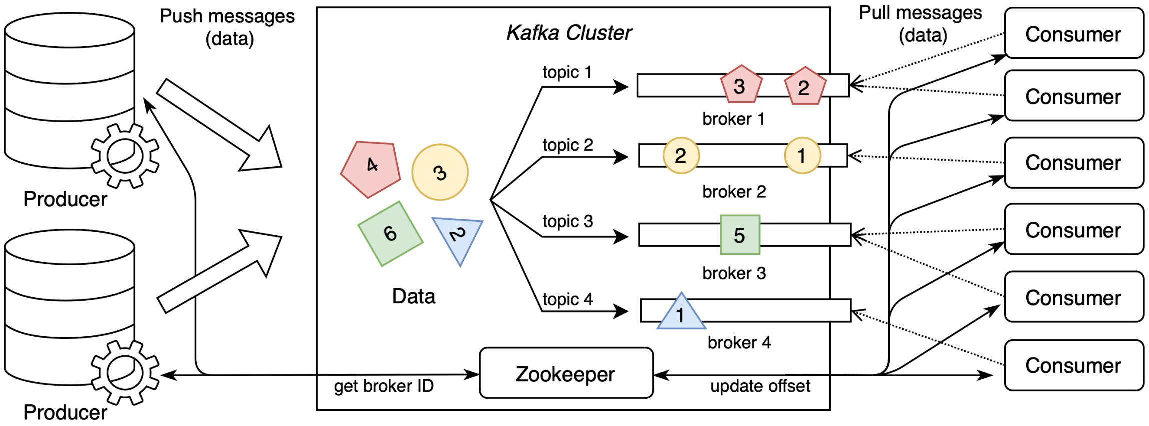


Figure 2.9: Kafka architecture. A shape and a color characterize messages associated with a topic. Note: in Kafka, messages on different topics are replicated across different brokers and are not represented in the image for simplicity. It is to be noted that for each topic, a broker is elected as the leader while the other brokers replicate its contents. Diagram inspired by DoubleCloud article available at `https://double.cloud/blog/posts/2023/03/the-many-use-cases-of-apache-kafka/`.

they can send messages without waiting for consumers to process them. Additionally, this allows consumers to be flexible with workload size.

Given its distributed nature, Kafka allows several producers, consumers, and brokers to exist simultaneously. This feature enables the system to be tuned according to the needs of a specific application.

2.3.3. DuckDB

DuckDB [38] is an open-source, embedded, OLAP DBMS. DuckDB was designed to process small quantities of data (1 - 100 GBs) within the same process or application that runs it instead of a different process/application. These features create an efficient OLAP database that can be used for data analysis and data processing on a small-scale data, without the complexity of a more complex DBMS, e.g., Teradata [42].

The light structure that characterizes DuckDB enables this system to be highly responsive with low latency. The system's limitations regard data size. As DuckDB uses in-memory processing, it cannot handle big workloads (1TB or more) requiring disk loads.

2.3.4. Arrow Flight

Arrow Flight is a high-performance framework for data transfer over a network, typically Arrow tables [28]. This protocol enables the transfer of large quantities of data stored in a format, e.g., Arrow tables, without having to serialize or deserialize it for transfer. This greatly speeds up the data transfer, making Arrow Flight extremely efficient. Arrow Flight is designed to be cross-platform and has support for multiple programming languages (C++, Python, Java). The protocol also supports parallelism, speeding up transfers using multiple nodes on parallel systems. Arrow Flight protocol is built on top of gRPC, enabling standardization and easier development of connectors.

2.4. Application - Hopsworks feature store

This section describes the application layer, which takes advantage of the data stack described. The software described is the Hopsworks Feature Store, which this project contributes to.

2.4.1. MLOps fundamental concepts

Machine Learning Operations (MLOps) are a set of practices to automate and simplify ML workflows from the data collection to the model deployment. MLOps considers the problem of developing and deploying a ML system from a code, data and model point of view. While for a typical software application, only code needs versioning, for a ML application, data and models also needs versioning, as training on different data versions might affect performance. The need for data versioning saw feature stores emerge as a solution for the problem [19]. Feature stores are central to the MLOps process, serving as fast-access storage.

Figure 2.10 shows a simple MLOps architecture using the Hopsworks' AI data platform. After data is gathered from various sources, a feature pipeline processes the data, performing model-independent transformations and saving the resulting features in the feature store. The training pipeline then runs model-dependent transformation (based on the specific model that will be trained) on the features retrieved from the feature store and saves the output, i.e., the model in a model registry. Then, a last process performs inference, typically embedded in deployed applications. For this process, it will be enough to take the new features gathered on the platform and perform an inference on the specific model saved in the model registry.

This type of architecture allows an asynchronous decoupled pipeline that enables the system to be maintainable, scalable, and highly effective for production scenarios.

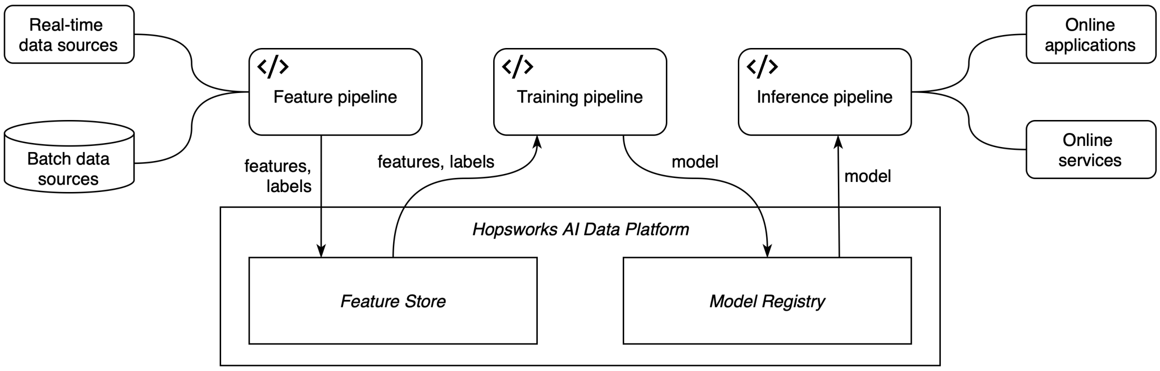


Figure 2.10: MLOps pipeline using a feature store and a model registry. Diagram inspired by Hopsworks documentation available at `https://www.hopsworks.ai/dictionary/feature-store`.

2.4.2. Hopsworks feature store

As first introduced in the previous section, the feature Store is a key data layer in an MLOps workflow. The feature store enables feature reusability and centralized and more accessible collaboration on model training and deployment. The Hopsworks feature store organizes features in feature groups, i.e., a mutable collection of features. Feature groups can be queried via the Hopsworks API, allowing developers to perform Create Read Update Delete (CRUD) operations.

The Hopsworks feature store, in addition to supporting batch data sources, also supports real-time data streaming. To support both systems (or hybrids), the Hopsworks feature store saves features in two storages: the offline feature store and the online feature store. The offline feature store is a column-based storage suited for batch data that is updated with a low frequency (every few hours at maximum frequency). The online feature store, on the other hand, is a row-based, key-value data storage based on RonDB. These characteristics enable low latency and real-time data processing. To keep this dual system consistent, the Hopsworks feature store has a unique point of entry for data, which is Kafka, that guarantees at least one message delivery to both storages. This architecture enables the system to support both workflows while keeping consistent data storage.

2.5. System architectures

This section describes the architectures of the legacy and new systems that will be run and measured in the experimental part of this thesis. It is divided into four subsections, and for each, a schema presented shows the protocol step by step.

2.5.1. Legacy system - Writing

Figure 2.11 [2] shows the legacy Hopsworks feature store write process from the client onto the offline feature store. The process is mainly split into two synchronous parts: upload and materialization. In the upload step, the Pandas DataFrame given as input is converted into rows and sent to Kafka one row at a time. Then, when the upload is completed, the client will be notified. Asynchronously, a Spark job, the Hudi Delta Streamer, has been running in the cluster since the Hopsworks cluster was started. This job periodically retrieves messages from Kafka, and then once it retrieves a full table, it writes the table in a column-oriented format to Apache Hudi, which sits on top of a HopsFS system. Once the materialization is completed, the Python client will be notified of completion.

As in the pipeline, the upload and the materialization are two parts of the process that do not act synchronously. During the experimental part of the thesis, the materialize function was called to measure the latency of the whole process without having to account for the Hudi Delta Streamer data retrieval period. This allows the system to perform the materialization on call instead of waiting for the period. This enabled the experiments to retrieve accurate data on the total latency of the process.

2.5.2. Legacy system - Reading

Figure 2.12 [3] shows the legacy Hopsworks feature store read process from the client onto the offline feature store. Unlike the writing process, the process is not Spark-based and uses a Spark alternative: a combination of an Arrow Flight server and a DuckDB instance. This avoids the conversion into row-based tables for sending the data, keeping the unified standard Arrow Table, which is a column-oriented format.

---

[2] For enhanced visualization, refer Figure A.1

[3] For enhanced visualization, refer Figure A.2

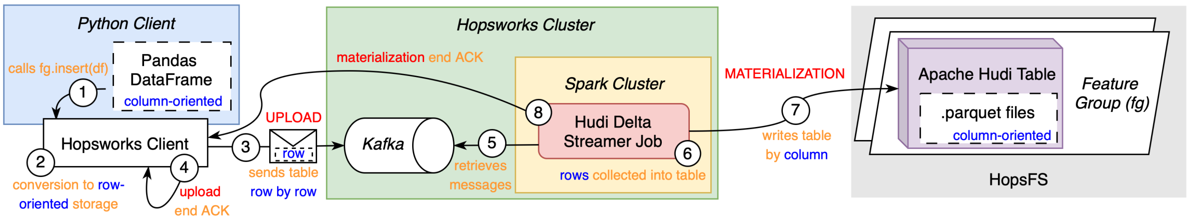


Figure 2.11: Legacy system writing a Pandas DataFrame from a Python client to the Hopsworks offline feature store. Each step is represented with a number. The table format conversion is outlined in blue, i.e., from columns to rows and then from row to columns. Steps from one to four represent the upload process, while the materialization process is complete at step eight. The diagram was realized based on one-to-one interviews with Hopsworks AB employees developing the Hopsworks feature store.

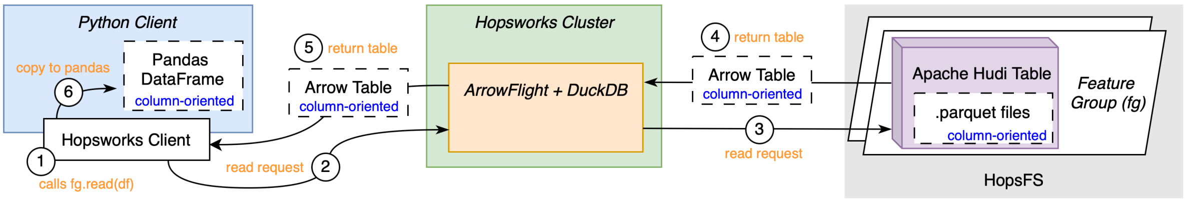


Figure 2.12: Legacy system reading a table from the Hopsworks offline feature store and loading it into the Python client's local memory. The process is streamlined using Arrow Tables that avoid table conversion. Diagram inspired by the Hopsworks feature store paper [10].

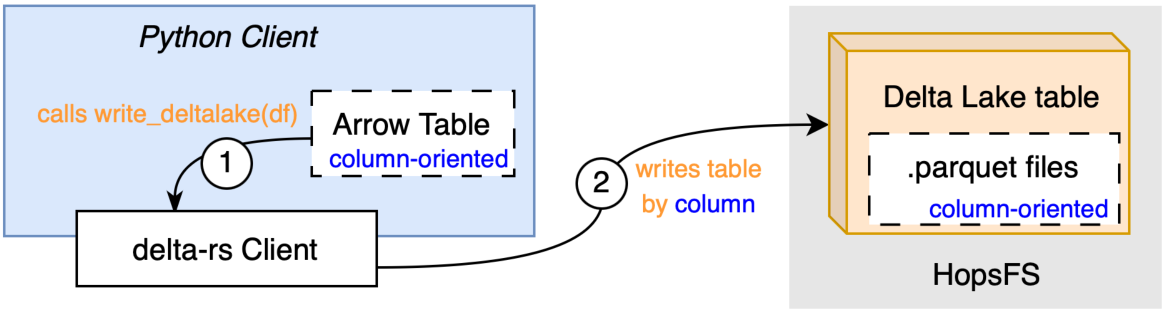


Figure 2.13: Delta-rs library writing an Arrow Table from a Python client to a Delta Lake table store on HopsFS.

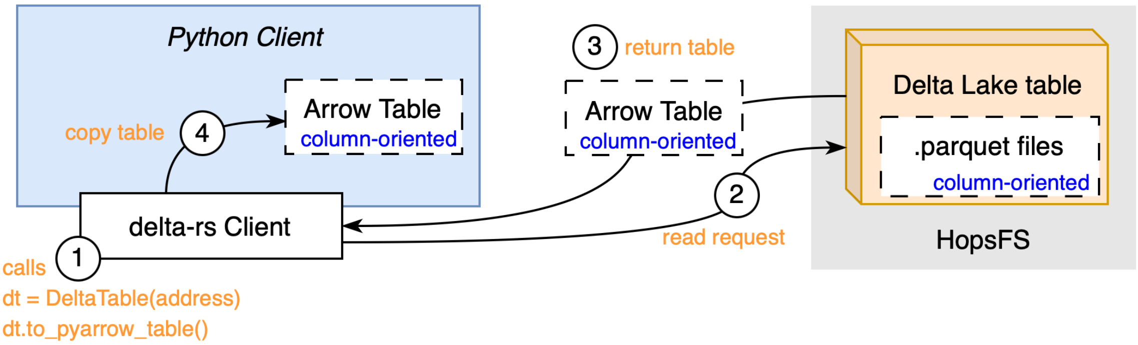


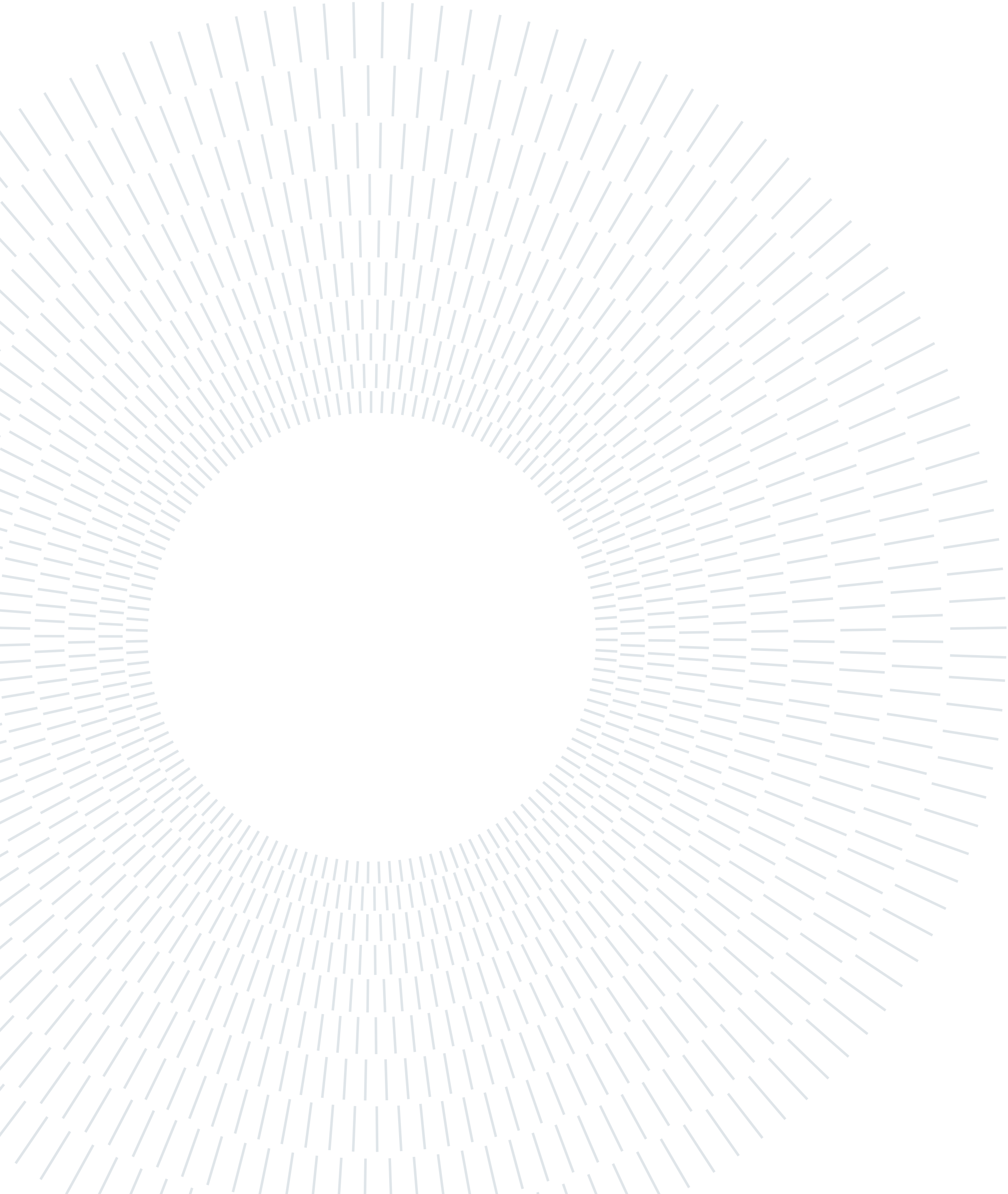
Figure 2.14: Delta-rs library reading a Delta Table stored in HopsFS and loading it into memory.

2.5.3. New system - Writing

Figure 2.13 shows how the delta-rs library writes on a Delta Lake table instanced on top of HopsFS. The delta-rs library streamlines the process without passing from a server instance (Spark).

2.5.4. New system - Reading

Figure 2.14 shows how the delta-rs library reads on a Delta Lake table instanced on top of HopsFS. The delta-rs library streamlines the process without passing from a server instance (Arrow Flight).



3 | Method

This chapter following the two RQs defined in Section 1.2.1 defines two methodologies that will be applied sequentially in this project. Section 3.1 defines the system implementation process that outputs the D1, i.e., the code implementation. D1 will enable the system evaluation defined in Section 3.2, which will output D2, i.e., the results of the experiment which analysis will be delivered in D3.

3.1. System implementation – RQ1

The core of this thesis resides in the system implementation section, which answers RQ1.

This section explains the method and the principles used to carry out the software development process of adding support for HDFS and HopsFS to the delta-rs library. This section is divided into four sub-sections: Research paradigm, explaining the research framework that will be used to implement the system; Development process, describing the activities that will be carried out to implement the system; Requirements, defining the functional and non-functional requirements of the system and Development environment, detailing the tools and resources that will be used during the development process.

3.1.1. Research paradigm

The research paradigm of the system implementation section of this thesis is positivist, believing that reality is certain and can be measured and understood. In the context of the development process, this research paradigm means that the new system requirements can be defined, and implementation errors can be outlined, understood, and, if possible, fixed. This approach leads to a strict definition of the development process, which depends on functional requirements that must be fulfilled.

3.1.2. Development process

The development process will follow an iterative and incremental approach described in Figure 3.1. This methodology will be applied as it allows flexibility while creating incrementally a working system [12]. This project will require numerous interactions with HopsFS maintainers (i.e., the industrial supervisor) due to the need for the delta-rs library to interact with HopsFS. This review process creates the need for a feedback loop, allowing the system to fit all the requirements according to all stakeholder's expectations.

As it can be noted from Figure 3.1, each step of this process is related to one of the goals (G1–G4) associated with RQ1 in Section 1.4. The activities and the relationship between each activity and the associated goal(s) are explained here below:

1. **Identify problems collaboratively**: this activity partially solves G1–G2, as it is an initial system analysis, performed together with the industrial supervisor, who is knowledgeable on Hopsworks' infrastructure (in particular HopsFS). This task fixes the project's initial requirements and investigates what needs to be implemented at a high-level abstraction.

2. **Analyse system**: this activity partially solves G1–G2 each time it is re-iterated, as it performs low-level code-based analysis of how the system works and what needs to be implemented to support HDFS in delta-rs. This activity also starts an iterative loop that will end once the system fulfills the requirements described in Section 3.1.3.

3. **Design software**: this activity solves partially G3, as the first part of the software development. In this activity, a solution is designed from the information gathered in the system analysis.

4. **Code software**: this activity solves partially G3, as the second part of the software development. In this activity, the solution designed is coded.

5. **Test system**: this activity partially solves G4, as the first part of the tests performed to verify the solution validity via unit tests. Failed unit tests will trigger a new development loop iteration, where this failure will be considered as the starting point in the system analysis.

6. **Verify system integration**: this activity partially solves G4, as the second part of the tests performed to verify the solution validity via integration tests. Failed integration tests will trigger a new development loop iteration, where this failure will be considered as the first starting point in the system analysis. On the other

hand, if the integration test succeeds, the loop will be restarted if the system does not yet fit a requirement. Otherwise, the development process finishes if the system fulfills all the functional requirements described in Section 3.1.3.

This process will produce a final deliverable (D1), which is a Python wheel of the delta-rs library containing the support for HDFS and HopsFS.

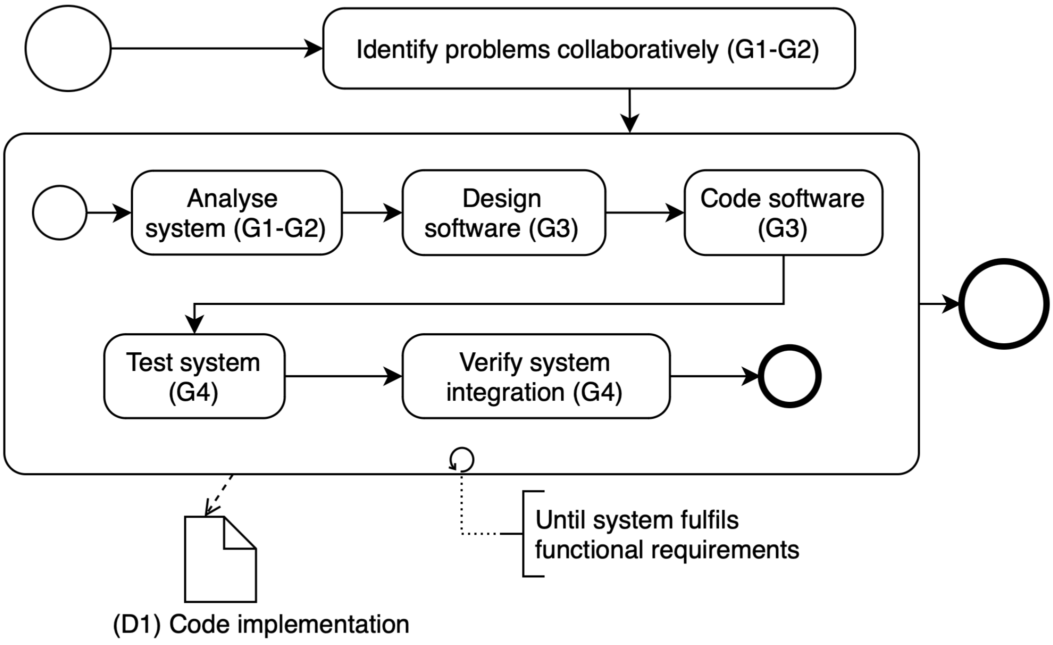

Identify problems collaboratively (G1-G2)
Analyse system (G1-G2)
Design software (G3)
Code software (G3)
Test system (G4)
Verify system integration (G4)
Until system fulfils functional requirements
(D1) Code implementation

Figure 3.1: Business Process Model and Notation (BPMN) diagram of the System implementation process answering to RQ1. Each activity is associated with a specific goal (G). The process produces a deliverable (D), in this case, a code implementation. The development loop iterates until the functional requirements (defined in Section 3.1.3) are fulfilled.

3.1.3. Requirements

In the first steps of the system analysis, a series of requirements are defined in agreement with the industrial partner Hopsworks AB to favor the creation of a solution that could be later used within the company in a production environment. These are divided into two categories: functional and non-functional requirements.

The **functional requirements** are:

1. **Write Delta Tables**: the solution should allow to write Delta Lake tables on HopsFS via the delta-rs library.

2. **Read Delta Tables**: the solution should allow to read Delta Lake tables on HopsFS via the delta-rs library.
3. **Communicate via TLS**: the solution should interact with HopsFS via Transport Layer Security (TLS) protocol version 1.2.

The **non-functional requirements** are:

1. **Consistent**: the solution should be consistent with the current open-source codebase used when appropriate.
2. **Maintainable**: the solution should minimize the need for maintenance and support of the codebase in the future, minimizing changes to open-source code. When appropriate, the solution's changes should be compatible with a future upstream merge to the modified open-source project.
3. **Scalable**: the solution should be able to handle larger quantities of data (up to 100 GB) read or written on Delta Tables.

3.1.4. Development environment

The system implementation will be developed by making use of several technologies, here categorized:

- **Computing resources**: the system implementation will be developed in a remote environment accessed via Secure Shell protocol (SSH) from a computer terminal. This remote Virtual Machine (VM) is selected as mounting HopsFS on a local machine is non-trivial, and developing locally could result in inconsistencies when the solution is reproduced in a virtual environment.
- **Writing code**: the Vim text editor will be the development tool of choice in combination with Conquer of Completion (CoC) [1] providing language-aware autocompletion and rust-analyzer [2] access for on-code compiler errors.
- **Libraries and dependencies**: the environment will be set in a Docker container for more straightforward development and test reproducibility.
- **Code versioning and shared development**: GitHub will be used for versioning, collaborating with open-source projects (e.g., delta-rs), and sharing the developed solution.

---

[1] Project's repository available at `https://github.com/neoclide/coc.nvim`

[2] Project's repository available at `https://github.com/fannheyward/coc-rust-analyzer`

3.2. System evaluation – RQ2

The system evaluation complements the system implementation by measuring the performance of the developed solution, answering RQ2. This evaluation process will be carried out following a sequential approach.

This section details the research paradigm, the method, and the principles that will be used to carry out the system evaluation process, measuring the performance (latency, measured in seconds, and throughput, measured in rows/second) of reading and writing on Delta Lake or Apache Hudi while on HopsFS of the current legacy pipeline and Rust-based pipelines.

3.2.1. Research paradigm

The research paradigm for the system evaluation section of this thesis is more hybrid between positivism and post-positivism. While still performing a confirmatory research approach based on defined objectives, it considers the limitations and biases of this approach, not seeking to generalize the results obtained to other cases. This approach is motivated by the limitations and biases given by the industrial context of this thesis while performing a confirmatory analysis based on a newly implemented system.

3.2.2. Evaluation process

The evaluation process will follow a sequential approach described in Figure 3.2. Each step of this process is related to one of the goals (G5-G8) associated with the RQ2 in Section 1.4. The relationship between each activity and the associated goal(s) is here explained:

1. **Design experiments**: this activity maps perfectly to G5, designing the experiments that will be conducted to evaluate the performance difference in performance between the current legacy access to Apache Hudi compared to the delta-rs library-based access to Delta Lake in HopsFS.
2. **Perform experiments**: this activity maps perfectly to G6, using the code implementation (D1) to conduct the designed experiments on the analyzed systems. Here, data is collected as latency, which is expressed in seconds.
3. **Transform data according to metrics**: this activity is requisite to fulfill G7, as throughput is computed from latency and not measured. The relationship that relates throughput (rows/second), latency(seconds), and size of table (rows) is the

following:

$$throughput\ (rows/second) = \frac{number\ of\ rows\ (rows)}{latency\ (seconds)}$$

4. **Visualize results**: this activity maps perfectly to G7, visualizing the experiments' result according to two metrics, i.e., latency measured in seconds and throughput measured in rows/second. This activity also generates a deliverable (D2) of the experiment results complete with tables and histograms, i.e., Chapter 5.
5. **Analyze results**: this activity maps perfectly to G8, analyzing and interpreting the results delivered in D2. This deliverable contributes to D3, generating the analysis of results, i.e., Chapter 5.

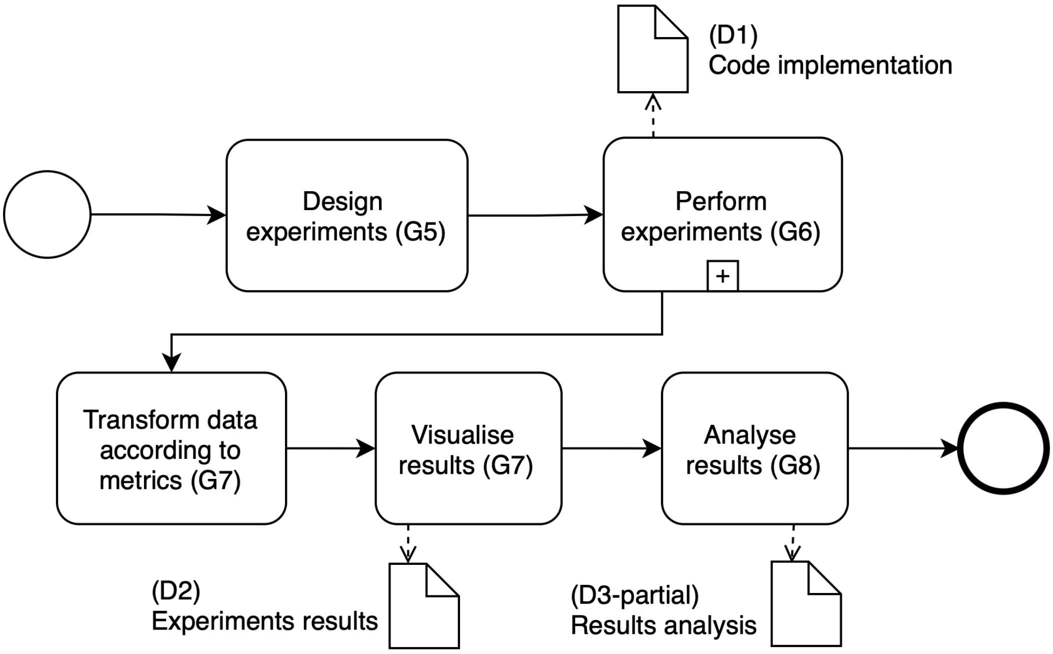


Figure 3.2: BPMN diagram of the System evaluation process answering to RQ2. Each activity is associated with a specific goal (G). The process produces two deliverables (D), the experiments results (D2) and a results analysis (D3-partial).

3.2.3. Industrial use case

Several choices must be made for the system evaluation: which data will be used, which environment will run the experiments, and which metrics will be used to evaluate the system. While the other sections of this chapter detail which decisions were made, this section aims to outline a typical use case of the system implementation that can motivate the choices about how the system is going to be evaluated.

During the research work in Hopsworks AB, the author talked to various employees and formed an idea of a typical client use case for the Hopsworks feature store. While these parameters are qualitative, they depict a specific use case around which this thesis work was built. Here below are these use case characteristics:

- **Usage of rows over storage size**: contrary to Chapter 1, where the size of workload is mainly referred to by storage size (bytes), in the experimental part of the thesis, only the number of rows will be used to refer to size. This choice is motivated by the need to find a reliable unit that measures a table size. Storage (bytes) is not strictly linked to a table structure (a table could have a lot of columns and a small number of rows and occupy the same memory as a table with a lot of rows but few columns), and it varies across storage structures (row-oriented format vs. column-oriented format, i.e., csv vs. parquet). Thus, storage size (bytes) alone is unreliable and cannot be used to measure data pipeline performance. This thesis kept the number of rows as the main unit to measure data size. Still, this metric was supplemented by specifying the number of columns and the data types in each row in Section 3.2.4.
- **Table size**: within Hopsworks, the author had the chance to see that most of the company's clients' workloads were limited (from 1M to 100M rows), while few clients had massive workloads (more than 1 BN rows). Given this outlook, this project opted to improve performance for the smaller workloads, setting the table sizes between 100K and 60M rows. The data selected is further detailed in Section 3.2.4.
- **Type of data**: the Hopsworks feature store works only with structured data (e.g., numbers, strings), and not with unstructured data (e.g., images, videos, audio) so, also the selected dataset in Section 3.2.4 will reflect this scenario.
- **Client configuration**: the performance of the implemented system also depends on the computational and storage resources available on the client configuration running the system. The client configuration was modeled to reflect a limited number of setups. Four CPU configurations were selected: one core, two cores, four cores, and eight cores. As for RAM availability, this will be adapted to the system's needs (as it is unknown before running the experiments). Additionally, to avoid storage I/O bottlenecks and reflect a modern system, a system equipped with a SSD storage will be used. The experimental environment is further detailed in Section 3.2.6.

3.2.4. Data

The data that will be used to perform the read and write operations comes from TPC-H benchmark suite [3]. TPC-H is a decision support benchmark by Transaction Processing Performance Council (TPC). It consists of a series of business-oriented ad-hoc queries designed to be industry-relevant [43]. The data coming from this benchmark suite was used as it provides a recognized standard for data storage systems [37], and it has already been used in similar related work [5, 38]. Any part of the data can be generated using the TPC-H data generation tool [4].

The TPC-H benchmark contains eight tables. Of these two, SUPPLIER and LINEITEM were selected for the following reasons. The two tables are, respectively, the smallest (10000 rows) and largest (6000000 rows) tables whose size (number of rows) depends on the Scale Factor (SF). The SF can be varied to obtain tables of different sizes (number of rows), allowing a progressive change in the table size (number of rows).

The SUPPLIER table has seven columns, while the LINEITEM table has sixteen. This difference influences the average size of memory each row occupies. Below for each table their columns are listed, specifying which data type they store.

- SUPPLIER
    - S_SUPPKEY : identifier
    - S_NAME : fixed text, size 25
    - S_ADDRESS : variable text, size 40
    - S_NATIONKEY : identifier
    - S_PHONE : fixed text, size 15
    - S_ACCTBAL : decimal
    - S_COMMENT : variable text, size 101
- LINEITEM
    - L_ORDERKEY : identifier
    - L_PARTKEY : identifier
    - L_SUPPKEY : identifier

---

[3] Benchmark suite website available at `https://www.tpc.org/tpch/`

[4] Available at `https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp`

  - L_LINENUMBER : integer
  - L_QUANTITY : decimal
  - L_EXTENDEDPRICE : decimal
  - L_DISCOUNT : decimal
  - L_TAX : decimal
  - L_RETURNFLAG : fixed text, size 1
  - L_LINESTATUS : fixed text, size 1
  - L_SHIPDATE : date
  - L_COMMITDATE : date
  - L_RECEIPTDATE : date
  - L_SHIPINSTRUCT : fixed text, size 25
  - L_SHIPMODE : fixed text, size 10
  - L_COMMENT : variable text, size 44

As mentioned in Section 3.2.3, measuring the memory size in terms of bytes that a row of a table occupies has no single approach, as it depends on how the row is stored (e.g., a DBMS, a row or column-oriented format). This limitation is why no specific ratio between a row and a data size in terms of bytes is given. Nonetheless, this section provides all information on the data, from how to retrieve it and how it is composed. These details enable the reader to calculate the storage occupancy of each table depending on its storage of choice.

Considering the different number of columns of the two tables used, the selected metrics, i.e., latency (seconds) and throughput(rows/second), cannot be used to compare results across different tables. This is why the comparative evaluation only considers different configurations on the same table.

This project used five table variations to benchmark the code solution as D1. SF was varied to obtain a table at each significant figure, from 10000 rows to 60000000 rows. These are the tables:

1. *supplier_sf1*: size = 10000 rows
2. *supplier_sf10*: size = 100000 rows
3. *supplier_sf100*: size = 1000000 rows

4. *lineitem_sf1*: size = 60000000 rows
5. *lineitem_sf10*: size = 60000000 rows

3.2.5. Experimental design

The experiments aim to highlight the differences between the newly implemented system based on the delta-rs library and the current legacy system. Three different experimental pipelines were designed to isolate the benefit of using delta-rs over Spark and provide a baseline:

1. **delta-rs - HopsFS**: the system implemented in Chapter 4. It comprises a Rust pipeline with Python bindings, enabling performing operations (i.e., reading, writing) on Delta Lake tables. This pipeline writes on HopsFS.
2. **delta-rs - LocalFS**: this pipeline uses the same library as the system implemented but saves data on the Local File System (LocalFS). This pipeline provides a comparison within the delta-rs library, isolating the impact on performance caused by writing on HopsFS, a distributed file system.
3. **Legacy pipeline**: is the Hopsworks feature store which writes data on Hudi tables. This system uses a pipeline based on Kafka and Spark to write data on the Hudi tables, saved on HopsFS. The pipeline uses a Spark alternative, DuckDB, and Arrow Flight to read data as explained in Section 2.5.2.

Furthermore, the experiments will verify how the performances of the three systems will change based on the CPU resources provided: one core, two cores, four cores, or eight cores. Each time, the experimental environment will be modified, creating a new Jupyter server where the experiments will run with increasingly more resources. These CPU configurations were chosen together with the industrial supervisor, according to the typical Hopsworks use case (Section 3.2.3). The data used for experiments, as described in Section 3.2.4, will come from two different tables. These are modified according to a SF, for a total of five times.

Additionally, during the writing experiments performed using the legacy pipeline, the contribution of different parts of the process will be measured, namely the upload time and materialization time, dichotomy explained in Section 2.5.1. This measurement will verify how different parts of the legacy pipeline scaled with table sizes and if Spark was the bottleneck of the architecture.

The Apache Hudi pipelines are preferred over the new Spark-based pipeline reading and writing on Delta Lake because they were released and tested extensively on the Hopsworks platform, so they provide more guarantees of obtaining relevant results. Additionally, at the time of experiment design, these two variations, Spark writing on Delta Lake and Spark writing on Apache Hudi, were considered similarly in read and write performance.

In conclusion, the experiments conducted will be a total of three (pipelines) times four (CPU configurations) times five (tables) times two (read and write operations), i.e., one hundred and twenty experiments, performed fifty times each to create statistically significant results.

3.2.6. Experimental environment

The experimental environment consists of a physical machine in Hopsworks' offices, which is virtualized to enable remote shared development. The CPU details of the machine are present in Listing 3.1, noting that only eight cores at maximum were accessed during the experiments. It should be observed that this experimental environment, while virtualized in isolation, runs on shared computing resources, so experiment results might vary depending on the machine's load. Considering this, all experiments will be run when the machine load is low (less than 50% of CPU and RAM usage) to limit having results depending on external workloads running.

The machine mounts about 5.4 TBs of SSD memory. This storage allows the machine to have fast read and write speed, 2.7 GB/s, and 1.9 GB/s, respectively (measured with a simple *dd* bash command).

The experimental environment will be set up with a Jupyter Server of different CPU cores, depending on the experiment. The Jupyter server is allocated by default with 2048 MB of RAM out of the 192 GB available on the experimental machine. This amount will be adjusted during the experiments according to the needs of the experiments (see Section 5.1.4).

Listing 3.1: Output of a *lscpu* bash command in the experimental environment.

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical,
                          48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   32
  On-line CPU(s) list:    0-31
Vendor ID:                AuthenticAMD
```

```
  Model name:                  AMD Ryzen Threadripper
                               PRO 5955WX 16-Cores
    CPU family:                25
    Model:                     8
    Thread(s) per core:        2
    Core(s) per socket:        16
    Socket(s):                 1
    Stepping:                  2
    Frequency boost:           enabled
    CPU max MHz:               7031.2500
    CPU min MHz:               1800.0000
    BogoMIPS:                  7985.56
Virtualization features:
  Virtualization:              AMD-V
Caches (sum of all):
  L1d:                         512 KiB (16 instances)
  L1i:                         512 KiB (16 instances)
  L2:                          8 MiB (16 instances)
  L3:                          64 MiB (2 instances)
```

3.2.7. Evaluation framework

The system evaluation framework is designed to evaluate three key aspects of the system using different metrics:

1. **Functional requirements**: defined in Section 3.1.3, functional requirements will be measured by verifying the success or failure of running an experiment. This will not happen by design, as the system implementation phase continues until all functional requirements are met.

2. **Non-functional requirements**: defined in Section 3.1.3, non-functional requirements are: consistency, maintainability and scalability. The first two requirements are mainly addressed during implementation, while scalability is measured during the system evaluation experiments. The metric used for measuring this requirement is the throughput measured in rows/second, as defined in RQ2.

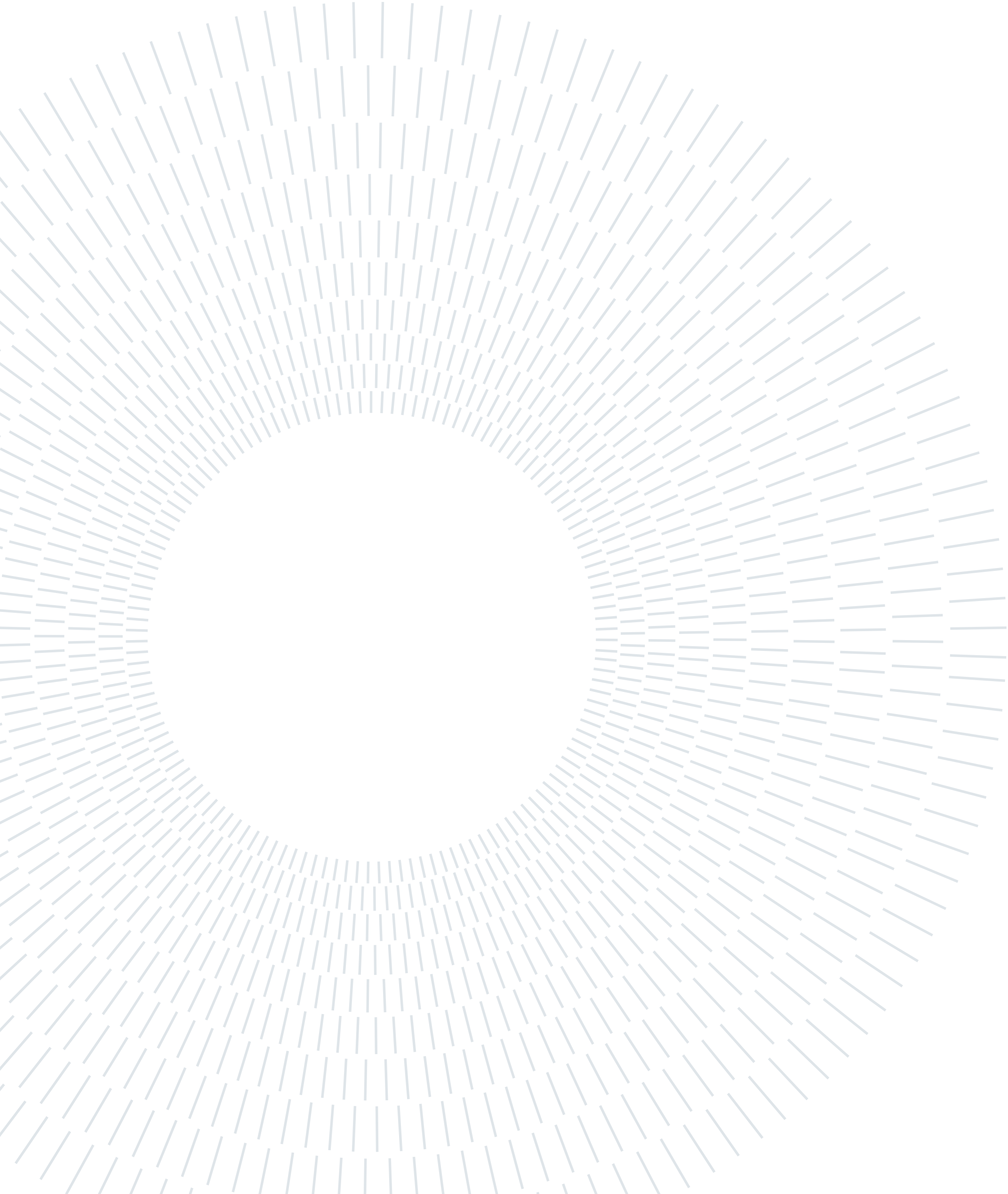
3. **How does the legacy system compare to other pipelines?**: this question answers directly RQ2, measuring the throughput measured in rows/second of the

different pipelines, defined in Section 3.2.5. Results are then compared using a visual approach.

3.2.8. Assessing reliability and validity

Results are significant according to their reliability and validity. In this project work, each experiment will be performed fifty times to ensure the reliability of the experiment results on the system performance. This number was agreed to balance the results' consistency and the limited resources available (in terms of time and computing resources).

Due to the complex nature of the pipelines used, the data distribution of results could vary from one experiment to the other. This hampers the possibility of comparing results, significantly impacting the relevance of the results analysis. A bootstrapping technique will be used to restore the validity of the data collected. Data will be resampled with substitution a thousand times, then an average with a confidence interval for each experiment will be calculated. This will benefit the accuracy of the results presented.



4 | Implementation

This chapter follows the system implementation process defined in Section 3.1 detailing and explaining how the system was developed, how to deploy it, use it, and how the code implemented was run during the experimental part of this thesis work.

4.1. Software design and development

The first step of the development process, defined in Section 3.1.2, consists of identifying what the delta-rs library (version 0.15.x) lacks to satisfy the requirements, more specifically, how to support HDFS and HopsFS in the delta-rs library. This step outlines the library structure (colloquially defined as "crate") divided into sublibraries (colloquially defined as "subcrates"), which is illustrated in Figure 4.1. As the figure shows, the delta-rs crate has a subcrate for each storage connector, so adding support for different storage is a matter of implementing a new subcrate to the delta-rs crate.

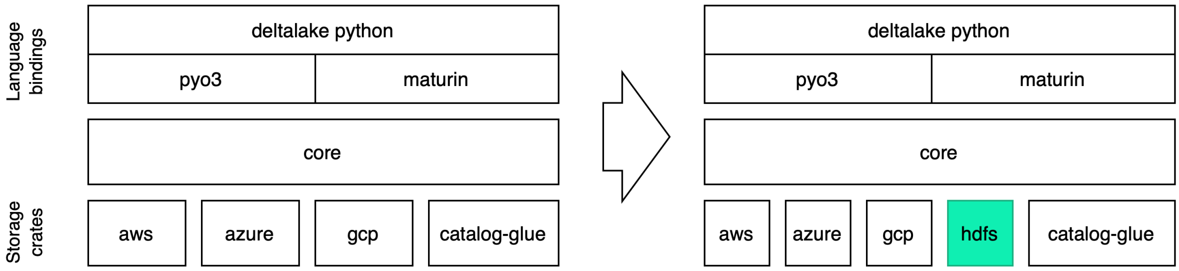


Figure 4.1: Delta-rs library (v. 0.15.x) before and after adding HDFS support. Diagram inspired by delta-rs architecture presented by R. Tyler Croy at the Data+AI Summit 2021. Recording available at `https://www.youtube.com/watch?v=scYz12UK-OY&t=585s`.

The delta-rs library uses an external interface called object-store defined in the Arrow-rs library [1]. Every storage connector implements this interface, and the other parts of the library interact with the storage layer. Thus, the new HDFS storage must also implement

---

[1] Project's repository available at `https://github.com/apache/arrow-rs/tree/master/object_store`

this interface.

The development process was divided into two subsections: a first approach and a final approach. The first approach was carried out by developing a HDFS subcrate for the delta-rs crate using the libhdfs library, i.e., a HDFS client in C++, while implementing the object-store interface. This contribution is of more than two thousand LOC. While it was being developed to fit all requirements, this first solution was rolled out, favoring a new approach that would consider the support for HDFS added in delta-rs with version 0.18.2. This decision was made to fit the defined non-functional requirements of consistency with current code and maintainability over time. The knowledge acquired during the first iteration development enabled the second approach to apply specific and limited changes to the hdfs-native library, i.e., the Rust library serving as HDFS client, with a contribution of eight hundred LOC.

The following two sections detail the significant problems and issues solved by the two code contributions, explaining the library's architecture and dependencies in more depth.

4.1.1. First approach

At the beginning of the project, a first approach was taken to provide support for HDFS to the delta-rs library version 0.15.x. Analyzing past contributions to the library revealed that HDFS was supported in version 0.9.0 of the delta-rs library, but support was removed due to three main reasons:

1. The HDFS support caused the testing pipeline to fail, and no trivial solution was found.
2. The HDFS support had JVM dependencies. This was considered a strong limitation for Python users, as having Java installed is high overhead (in terms of storage and computation) for performing an operation in Python. Having these Java dependencies would have meant that many Python users would not have used the library altogether.
3. The community around the library did not have contributors or many users, which were using HDFS as storage layer.

Starting from the HDFS support in the delta-rs library in version 0.9.0, a solution was designed to fix the testing issues and provide a working storage support for HDFS. The architecture is described in Figure 4.2.

This implementation uses a C++ library called libhdfs, which contains all the methods

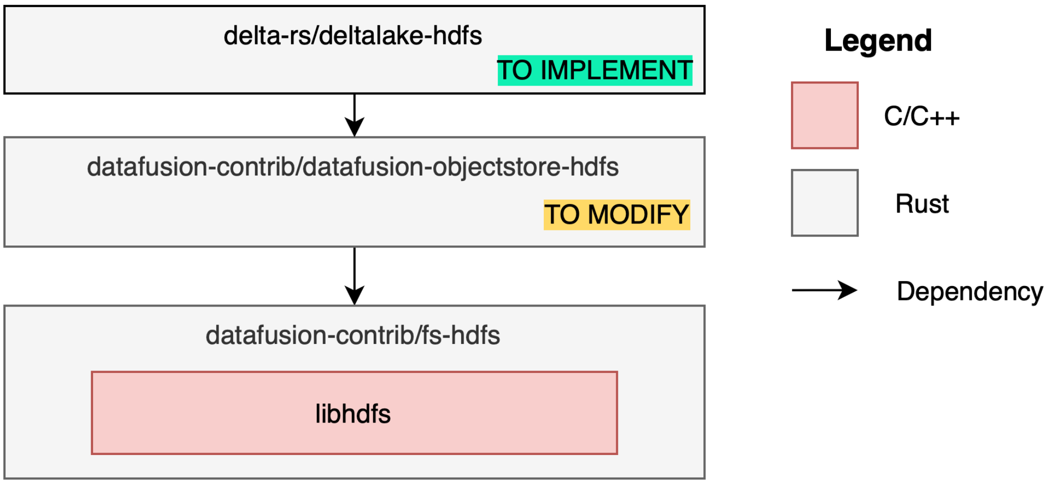


Figure 4.2: Architecture of the first implementation approach. The repositories that need to be implemented or modified are highlighted in green and yellow, respectively.

required to work as an HDFS client. This library is contained in the Rust library fs-hdfs. The datafusion-objectstore-hdfs uses the libhdfs library to provide an interface for HDFS that implements object-store, the interface used in the delta-rs library to interact with storage connectors.

This approach first required the rewrite of the datafusion-objectstore-hdfs [2] as it used an older object-store interface version (0.8.0 vs. 0.10.0), and it was no longer possible to upgrade it easily. Secondly, the JVM dependencies, while this project did not have a strict requirement on not having them, being able to have the dependencies consistently work on different development environments proved to be a challenge. This approach's contribution can be estimated considering the two thousands LOC written to implement the object-store interface for HDFS in the object_store_hdfs library [3].

Ultimately, this approach was abandoned following the release of version 0.18.2 of the delta-rs library. This decision was taken to comply with the maintainability requirement defined in Section 3.1.3.

4.1.2. Final solution

Version 0.18.2 of the delta-rs introduced support for HDFS via the hdfs-native library [4]. This is a Rust library that re-implements the HDFS client, avoiding the use of libhdfs,

---

[2] Code available at `https://github.com/datafusion-contrib/datafusion-objectstore-hdfs`

[3] Code available at `https://github.com/Silemo/object_store_hdfs`

[4] Project's repository available at `https://github.com/Kimahriman/hdfs-native`

and thus has no JVM dependencies. This architecture is illustrated in Figure 4.3.

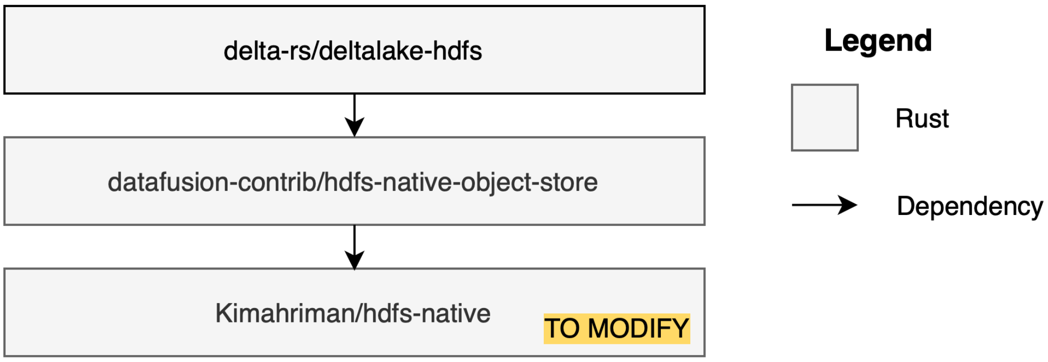


Figure 4.3: Architecture of the final implementation approach. The repositories that need to be modified are highlighted in yellow.

While being used by the delta-rs library, this approach had some incompatibilities with HopsFS. Here below is a list of them:

1. **Different HDFS protocol version**: HDFS makes use of a Remote Procedural Call (RPC) protocol to interact with a HDFS client. Hdfs-native is based on protocol version 3.2, while HopsFS was compatible with version 2.7. This was caused by the hdfs-native library implementing all the most recent HDFS functionalities, while HopsFS did not.
2. **No support for TLS in hdfs-native**: hdfs-native security is based on Kerberos, but it does not secure packet transfers using TLS. Using TLS is a functional requirement defined in Section 3.1.3, necessary for a production environment that can also be deployed over the internet.

These incompatibilities were solved one by one during development in the following way:

1. **Upgrading HopsFS protocol version**: together with the industrial supervisor responsible for the maintenance of HopsFS, the numerous differences (around forty instances between classes and entities) between the two protocol versions (2.7 vs. 3.2) were highlighted, and one by one resolved. This way, version 3.2.0.14 of HopsFS was released, adding support to the HDFS RPC protocol version 3.2, making HopsFS compatible with the hdfs-native library.
2. **Adding support for TLS in the hdfs-native library**: TLS support to hdfs-native was added via the use of an external Rust library called tokio-rustls version 0.26 [5]. This was achieved by modifying the hdfs-native library to work with TLS

---

[5] Library available at `https://github.com/rustls/tokio-rustls`

streams and initiate communication via a TLS handshake. The major issue encountered regarded loading the certificates on the offline feature store, which are typically dynamically loaded and whose path is specified via an environment variable. The certificate's names and locations also changed with the version change from Hopsworks offline feature store 3.7 to 4.0, which required the code to fit this update.

The work done in the first solution iteration, while the code itself was not used for the final solution, provided the author with in-depth knowledge of the delta-rs and object-store libraries, dependencies, and interactions that enabled the second solution to be achieved in a reduced time. The overall contribution of the final solution is more limited in the LOC, about eight hundred. Still, it consists of a more elegant approach, which aligns with the consistency and maintainability requirements defined in Section 3.1.3.

All changes were applied to copies (colloquially known as "forks") of the open-source repositories related to this project, namely delta-rs [6], hdfs-native-object-store [7], and hdfs-native [8].

4.2. Software deployment and usage

Once HDFS and HopsFS support has been added to the delta-rs library, it is sufficient to build a Python wheel, i.e., a pre-built binary package format for Python modules and libraries, for delta-rs. The Python wheel for the library can be built by following the instructions already present in the delta-rs library in the README.md file present in the python folder [9].

The usage of the delta-rs library is explained in detail in the delta-rs documentation [10], so in this section, only the method used for the experiments will be shown and explained. Listing 4.1 shows a simple example of writing to HDFS or HopsFS (formally in a Python script, only the address changes, as HopsFS is based on HDFS). As the listing shows, the script writes a small table of two columns and three rows.

---

[6] Code available at `https://github.com/Silemo/delta-rs`

[7] Code available at `https://github.com/Silemo/hdfs-native-object-store`

[8] Code available at `https://github.com/Silemo/hdfs-native`

[9] Instructions available at `https://github.com/Silemo/delta-rs/blob/main/python/README.md`

[10] Documentation available at `https://delta-io.github.io/delta-rs/`

Listing 4.1: Writing a DataFrame on a Delta Table with delta-rs on HDFS or HopsFS.

```
from deltalake import write_deltalake
import pandas as pd

df = pd.DataFrame({"num": [1, 2, 3],
                   "letter": ["a", "b", "c"]})
write_deltalake("hdfs://rpc.sys:8020/tmp/test", df)
```

An example of a read operation is shown in Listing 4.2. After being read, the Delta Table is converted to a pyarrow table to ensure an explicit in-memory operation (only calling the DeltaTable object is a lazy operation that does not load the data into memory).

Listing 4.2: Reading a DataFrame on a Delta Table with delta-rs on HDFS or HopsFS. Note: without the last line, the Delta Table is not loaded into memory, as delta-rs has a lazy evaluation approach.

```
from deltalake import DeltaTable

dt = DeltaTable("hdfs://rpc.sys:8020/tmp/test")
dt.to_pyarrow_table()
```

4.3. Experiments set-up

As defined in Section 3.2.5, experiments run different system configurations with five tables fifty times per experiment. Two main approaches were selected to measure the run time of the experiments, i.e., the read and write latency. This decision is motivated by the need to measure accurate results with the impossibility of using the most precise measurement approach in all systems.

The first approach uses the Python timeit function, which isolates the process running the code, proving a reasonable estimate of the operation latency. As illustrated in Listing 4.3, timeit can be used by defining a SETUP_CODE that runs before the experiment and a TEST_CODE that when running is measured and the time (expressed in seconds) is the return value of the timeit function. This approach was selected as the timeit function provides a clear interface to run and measure a small code script. The script here does not run a repeated number of times as the Delta Table must be deleted before re-running the experiment, and this requires time that shouldn't be included in the experiment time (nor in the setup, as the first time the table is not there). The limitation of this method is that it was not possible to use it to measure the total write latency on the legacy system while making a breakdown of the two steps performed by the system, i.e., upload and materialization (see Section 2.5.1). Therefore, the timeit function was used to measure delta-rs operations accurately, while the second approach was used to measure legacy systems. This was not considered problematic during the experiments as some trials revealed that the latency measured by the two methods was equal within a 95% confidence interval.

Listing 4.3: Timeit usage to measure the time required to write a Delta Lake table to HopsFS.

```
import timeit
SETUP_CODE='''import pyarrow as pa
from deltalake import write_deltalake'''

TEST_CODE='''
HDFS_DATA_PATH = "hdfs://rpc.sys:8020/exp"
LOCAL_PATH = "/abs/path/table.parquet"
pa_table = pa.parquet.read_table(LOCAL_PATH)
write_deltalake(HDFS_DATA_PATH, pa_table)'''

# Measure the execution runtime
write_result = timeit.timeit(setup  = SETUP_CODE,
                             stmt   = TEST_CODE,
                             number = 1          )
```

The second measuring approach was to record the time before and after the script run and then calculate the difference. This made it possible to calculate multiple differences without recreating the experiment numerous times. Listing 4.4 shows an example of this approach. This approach enabled experiments to measure both the write latency of the legacy system and the upload and materialization latency.

Listing 4.4: A simple time difference approach to measure the time required to write a Delta Lake table to HopsFS.

```python
import time
import pyarrow as pa
from deltalake import write_deltalake
HDFS_DATA_PATH = "hdfs://rpc.sys:8020/exp"
LOCAL_PATH = "/abs/path/table.parquet"
pa_table = pa.parquet.read_table(LOCAL_PATH)

before_writing = time.time()
write_deltalake(HDFS_DATA_PATH, pa_table)
after_writing = time.time()

write_result = after_writing - before_writing
```

5 | Results and Analysis

This chapter is the output of the system evaluation process defined in Section 3.2. It starts with Section 5.1, which presents the experiments' results in tables, histograms, and written descriptions. Then, Section 5.2 complements the chapter by analyzing and discussing the results' findings.

5.1. Major results

This section presents the main results of the one hundred and twenty experiments performed as defined in Section 3.2.5. The experiments are grouped into subsections according to the measured operation, i.e., read or write. Each subsection presents histograms and tables to visualize the results using both metrics, i.e., latency and throughput. Latency is expressed in seconds, and throughput is represented in rows/second.

Results are reported using the log scale for clarity, as results differing from more than one significant figure are not clearly visible using a linear scale in a histogram representation. Additionally, for each measurement, a 95% confidence interval was calculated using the bootstrapping technique mentioned in Section 3.2.8. Nonetheless, this interval was not reported in the histograms in this section, as it would be hardly readable. This is because all results are out of each other's 95% confidence interval. This data is reported in the Appendix B and C with a histogram and a table for each experiment expressed in both metrics.

Of the two metrics defined in RQ2, only latency was measured during the experiments, while throughput was calculated with the formula present in Section 3.2.2. Latency and throughput are inversely correlated by a constant factor since all experiments were performed with fixed-size tables. This relationship means that if latency is halved, throughput doubles, if latency quarters, throughput quadruples. Since results are reported according to both metrics, this creates an information redundancy. Trends will be described using latency to avoid repetition in this section. Throughput trends will be described if they reflect a significant behavior different from the latency trends.

5.1.1. Write experiments

Figures 5.1 and 5.2, and Tables 5.1 and 5.2 report the results of the write experiments performed. The results are expressed in latency in Figure 5.1 and Table 5.1, and expressed in throughput in Figure 5.2 and Table 5.2. The experiments were performed on the three systems defined in Section 3.2.5. The five tables of different sizes being written were defined in Section 3.2.4.

Both histograms, i.e., Figures 5.1 and 5.2, report the results of the experiment performed with one CPU core. On the other hand, the Tables 5.1 and 5.2 on top of reporting the results of the one CPU core experiment also present a calculated percentage of improvement (decrease in the case of latency, increase in the case of throughput) of the metric as the CPU cores increase.

delta-rs on HopsFS vs. delta-rs on LocalFS

The latency measured using the delta-rs on LocalFS pipeline is around ten times lower than the latency measured using the delta-rs on HopsFS pipeline for small tables, i.e., 10K and 100K rows. On the other hand, the latency in the two pipelines has the same significant figure in experiments performed with larger tables, i.e., 1M, 10M, 6M, and 60M rows. Overall, the latency measured in the delta-rs on LocalFS pipeline remains lower in absolute terms than the latency measured using the delta-rs on HopsFS pipeline in all experiments.

delta-rs on HopsFS vs. Legacy pipeline

The latency measured using the delta-rs on HopsFS pipeline results more than ten times lower than the latency measured using the Legacy pipeline in all experiments. This trend is more prominent for smaller tables (10K and 100K rows) where latency measured using the delta-rs on HopsFS pipeline is forty times lower than the latency measured using the Legacy pipeline. The tendency is less marked for larger tables (6M and 60M rows) where latency measured using the delta-rs on HopsFS pipeline is around twenty times lower than the latency measured using the Legacy pipeline.

Change of performance as the CPU cores increase

During experiments with more CPU cores, in delta-rs based pipelines (writing on HopsFS or LocalFS) the latency during write operation decreases by a considerable amount: 20-30%, only when writing larger tables (6M and 60M rows), while it decreases by a lower

Table 5.1: Write experiment results expressed as latency. Experiments performed with more than one CPU core are expressed as latency percentage decrease compared to the one CPU core experiment.

Pipeline	Number of rows	1 CPU core latency (seconds)	2 CPU cores (% decrease)	4 CPU cores (% decrease)	8 CPU cores (% decrease)
delta-rs HopsFS	10K	1.250 88	−0.92	2.75	−9.33
	100K	1.368 28	4.40	2.34	5.54
	1M	9.381 52	9.23	10.32	11.52
	6M	19.754 69	17.54	17.87	20.33
	60M	177.307 07	24.39	30.01	31.22
delta-rs LocalFS	10K	0.039 57	−21.88	−15.53	−11.25
	100K	0.152 40	10.01	13.54	10.45
	1M	8.422 52	14.69	14.68	14.17
	6M	17.906 34	14.74	18.71	20.24
	60M	172.345 52	24.67	29.57	30.38
Legacy	10K	50.227 67	−0.99	−2.10	−1.99
	100K	59.561 87	−0.38	0.06	−1.20
	1M	112.190 48	3.23	3.01	2.50
	6M	511.816 93	7.51	5.83	7.01
	60M	2715.772 85	13.81	13.61	14.39

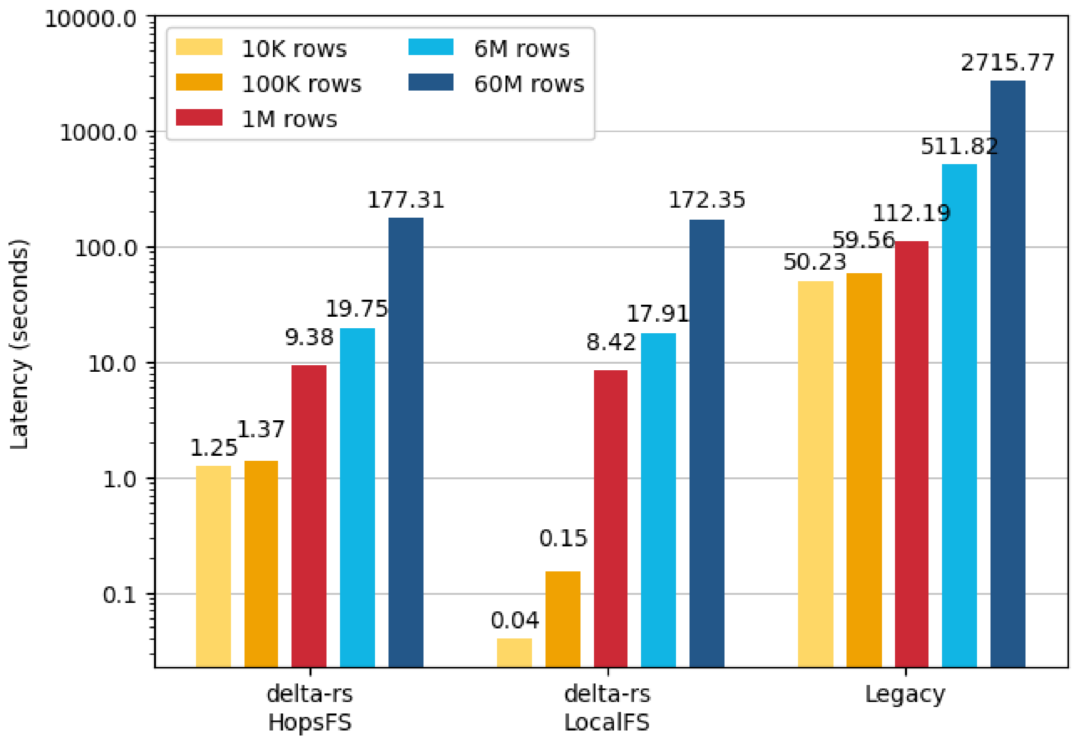


Figure 5.1: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with one CPU core.

Table 5.2: Write experiment results expressed as throughput. Experiments performed with more than one CPU core are expressed as throughput percentage increase compared to the one CPU core experiment.

Pipeline	Number of rows	1 CPU core throughput (k rows/second)	2 CPU cores (% increase)	4 CPU cores (% increase)	8 CPU cores (% increase)
delta-rs HopsFS	10K	7.994 36	−0.91	2.83	−8.53
	100K	73.084 38	4.60	2.40	5.87
	1M	106.592 42	10.17	11.51	13.01
	6M	303.725 33	21.27	21.76	25.52
	60M	338.395 98	32.26	42.87	45.39
delta-rs LocalFS	10K	252.682 38	−17.95	−13.44	−10.12
	100K	656.157 39	11.13	15.66	11.67
	1M	118.729 19	17.22	17.21	16.51
	6M	335.076 75	17.29	23.02	25.38
	60M	348.137 84	32.76	41.99	43.65
Legacy	10K	0.199 09	−0.98	−2.06	−1.95
	100K	1.678 92	−0.38	0.06	−1.19
	1M	8.913 41	3.34	3.10	2.57
	6M	11.722 94	8.12	6.19	7.54
	60M	22.093 15	16.02	15.76	16.81

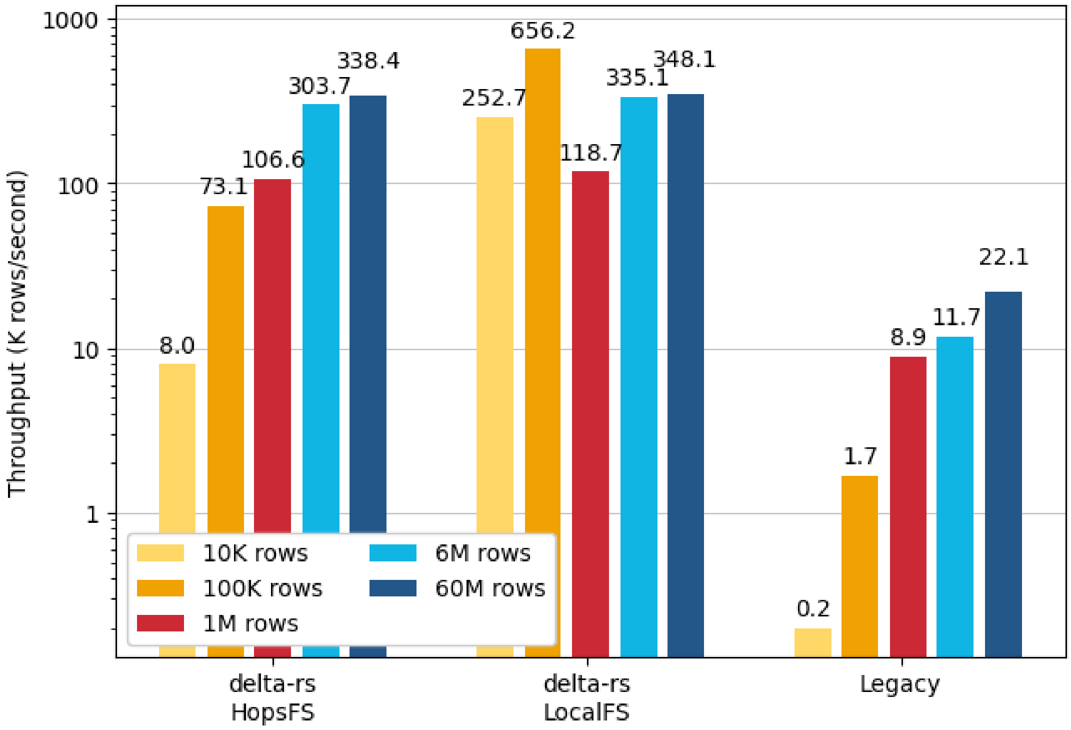


Figure 5.2: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with one CPU core.

margin: 5-10% on smaller tables (100K and 1M rows), even slightly increasing on the smallest table (10K rows). It should be noted that latency decreases as described in the two CPU cores experiments, remaining on similar improvements even with more CPU cores.

Considering the Legacy pipeline, experiments with more CPU cores did not decrease the latency by more than 7% except for the largest table (60M rows). This table benefitted from a latency decrease of around 14%. The smallest tables (10K and 100K) reported slight increases in the measured latency.

5.1.2. Read experiments

Figures 5.3 and 5.4, and Tables 5.1 and 5.4 report the results of the read experiments performed. The results are expressed in latency in Figure 5.3 and Table 5.3, and expressed in throughput in Figure 5.4 and Table 5.4. The experiments were performed on the three systems defined in Section 3.2.5. The five tables of different sizes being written were defined in Section 3.2.4.

Both histograms, i.e., Figures 5.3 and 5.4 report the results of the experiment performed with one CPU core. On the other hand, the Tables 5.3 and 5.4 on top of reporting the results of the one CPU core experiment also present a calculated percentage of improvement (decrease in the case of latency, increase in the case of throughput) of the metric as the CPU cores increase.

delta-rs on HopsFS vs. delta-rs on LocalFS

The latency measured using the delta-rs on LocalFS pipeline is around ten times lower than the latency measured using the delta-rs on HopsFS pipeline in the experiment with the smallest table, i.e., 10K rows. On the other hand, the latency in the two pipelines has the same significant figure on experiments performed with larger tables, i.e., 100K, 1M, 10M, 6M, and 60M rows. Overall, the latency measured in the delta-rs on LocalFS pipeline remains lower in absolute terms than the latency measured using the delta-rs on HopsFS pipeline in all experiments.

delta-rs on HopsFS vs. Legacy pipeline

The latency measured using the delta-rs on HopsFS pipeline results from 47% up to forty times lower than the latency measured using the Legacy pipeline. The largest latency reduction compared to the legacy system (forty times) is obtained with the 100K rows

Table 5.3: Read experiment results expressed as latency. Experiments performed with more than one CPU core are expressed as latency percentage decrease compared to the one CPU core experiment.

Pipeline	Number of rows	1 CPU core latency (seconds)	2 CPU cores (% decrease)	4 CPU cores (% decrease)	8 CPU cores (% decrease)
delta-rs HopsFS	10K	0.053 42	22.65	18.84	18.95
	100K	0.057 57	1.15	3.76	5.19
	1M	0.538 55	56.53	65.00	67.71
	6M	1.948 99	53.40	72.74	74.48
	60M	22.980 65	50.34	75.72	87.20
delta-rs LocalFS	10K	0.004 19	31.48	35.91	29.66
	100K	0.026 96	51.54	65.76	64.84
	1M	0.420 09	52.45	78.64	89.75
	6M	1.682 23	55.56	77.99	89.57
	60M	19.565 47	51.72	75.41	88.32
Legacy	10K	0.631 59	1.06	−0.67	0.67
	100K	2.650 10	−0.50	0.39	−0.46
	1M	8.596 36	−0.24	−1.81	2.89
	6M	33.529 64	0.46	0.23	0.30
	60M	33.697 72	0.16	0.13	1.64

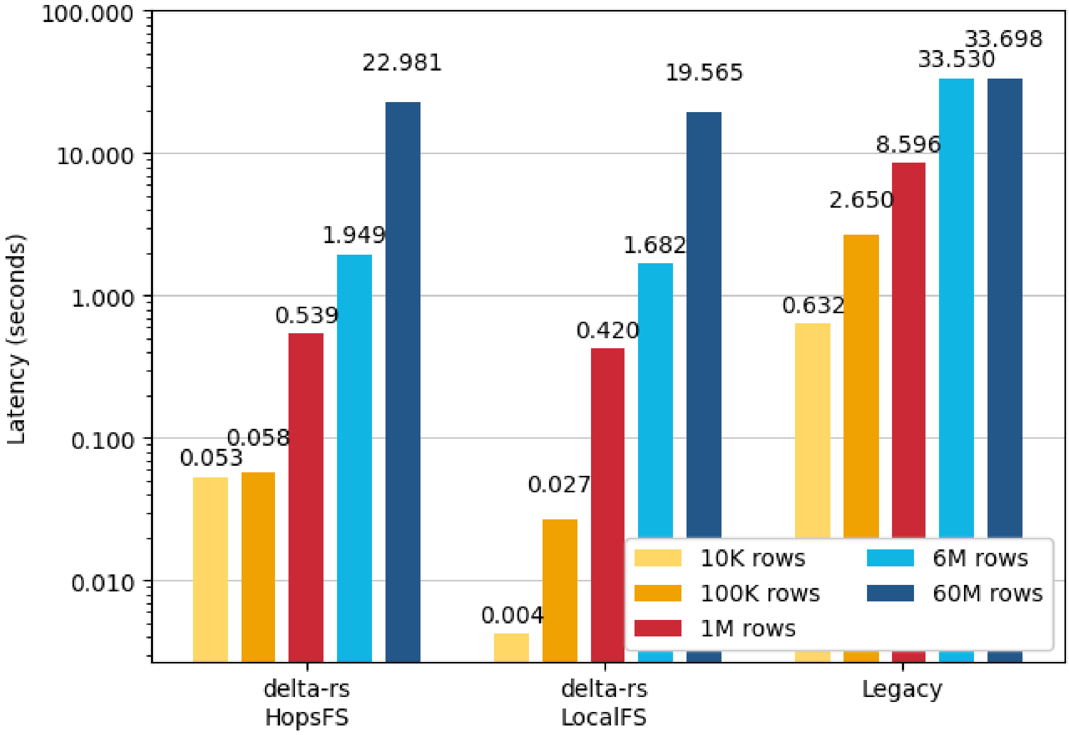


Figure 5.3: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with one CPU core.

Table 5.4: Read experiment results expressed as throughput. Experiments performed with more than one CPU core are expressed as throughput percentage increase compared to the one CPU core experiment.

Pipeline	Number of rows	1 CPU core throughput (k rows/second)	2 CPU cores (% increase)	4 CPU cores (% increase)	8 CPU cores (% increase)
delta-rs HopsFS	10K	187.168 53	29.28	23.21	23.38
	100K	1736.907 99	1.17	3.90	5.48
	1M	1856.831 67	130.02	185.74	209.69
	6M	3078.512 99	114.57	266.87	291.92
	60M	2610.891 46	101.35	311.83	681.03
delta-rs LocalFS	10K	2384.586 99	45.94	56.04	42.18
	100K	3708.257 87	106.37	192.07	184.38
	1M	2380.403 81	110.28	368.24	875.15
	6M	3566.674 54	125.01	354.40	858.64
	60M	3066.626 44	107.11	306.75	756.07
Legacy	10K	15.832 85	1.07	−0.67	0.67
	100K	37.734 32	−0.50	0.39	−0.45
	1M	116.328 20	−0.24	−1.78	2.98
	6M	178.946 12	0.46	0.23	0.30
	60M	1780.535 63	0.16	0.13	1.67

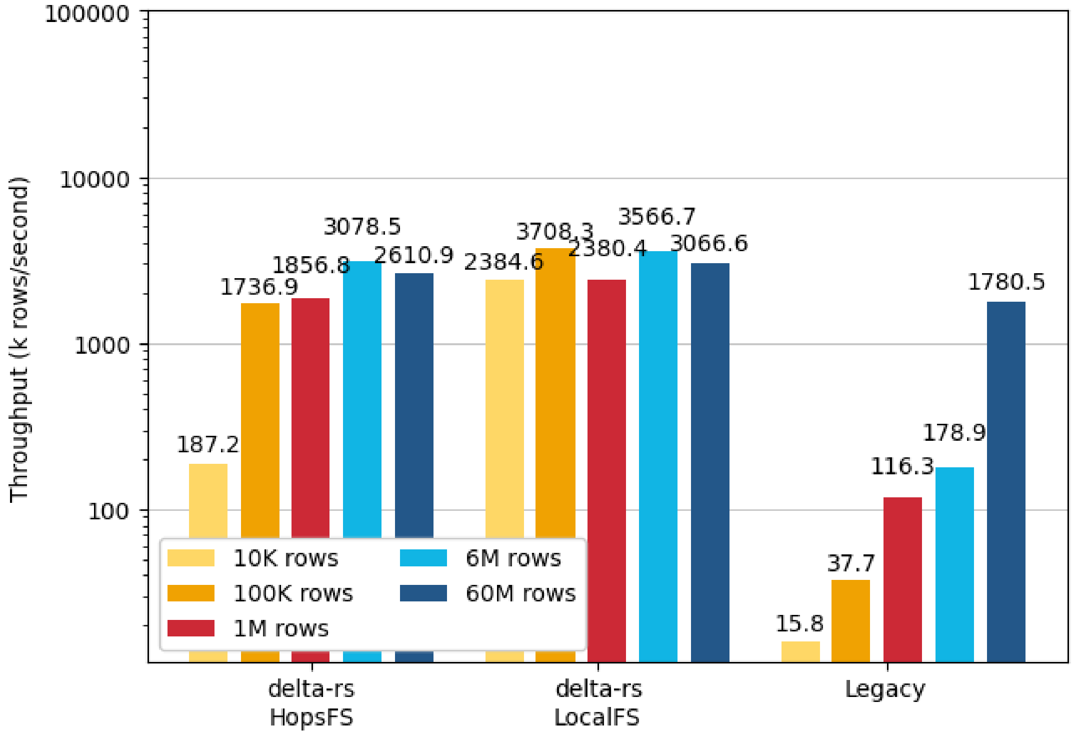


Figure 5.4: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with one CPU core.

table. In contrast, the smallest decrease in latency between newer and older systems is obtained with the largest table, i.e., 60M rows. For all other tables, the delta-rs on HopsFS pipeline has around ten times lower latency than the legacy system.

Change of performance as the CPU cores increase

During experiments with more CPU cores, in delta-rs based pipelines (reading on HopsFS or LocalFS), the latency during read operation decreases by a considerable amount: 50%, when reading larger tables (1M, 6M and 60M rows), while it decreases by a lower margin: 20-30% on smaller tables, i.e., 10K and 100K rows. It should be noted that latency decreases in reads with larger tables (1M, 6M, and 60M rows) following an inverse linear relationship with the increase of CPU cores: two CPU cores, latency halves, four CPU cores, latency quarters, eight CPU cores, latency is decreased to an eighth. On the other hand, throughput follows a linear relationship with the increase of CPU cores.

Considering the Legacy pipeline, experiments with more CPU cores did not decrease the latency by more than 2%. Histograms comparing the three pipelines look radically different in experiments with more CPU cores due to how delta-rs scales with the increase of CPU cores. They can be accessed in the Appendix B.

5.1.3. Legacy pipeline write latency breakdown

Table 5.5 and Figure 5.5 show the write latency breakdown of the Legacy pipeline into upload time and materialization time, the different steps of the Legacy pipeline as explained in Section 2.5.1. The breakdown is proposed for all five tables defined in Section 3.2.4. Figure 5.5 reports the data from the one CPU core experiment. In contrast, Table 5.5 reports both the one CPU core experiment data and also a calculated percentage of improvement, i.e., decrease, of the latency as the CPU cores increase.

Considering the upload time contribution to the write latency, this represents a small percentage (around 5%) when writing smaller tables, i.e., 10K and 100K rows. Nonetheless, the upload contribution grows following a similarly linear pattern in larger tables, i.e., between 100K and 60M rows. This radically changes the proportion between the upload and materialized contribution to the write latency, making the upload time 90% of the total write latency for the 60M rows table. On the other hand, the materialization time contribution, while starting with high latency contribution (95% of the total), its absolute value does not increase by more than a significant figure even if the table size increased by three significant figures.

Table 5.5: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. Experiments performed with more than one CPU core are expressed as latency percentage decrease compared to the one CPU core experiment.

Number of rows	1 CPU core latency (seconds)		2 CPU cores (% decrease)		4 CPU cores (% decrease)		8 CPU cores (% decrease)	
	upl.	mat.	upl.	mat.	upl.	mat.	upl.	mat.
10K	2.48	47.72	3.99	−1.27	4.10	−2.47	4.22	−2.35
100K	3.66	55.90	6.37	−0.78	5.55	−0.27	6.25	−1.69
1M	22.59	89.57	17.52	−0.39	14.89	−0.01	16.51	−1.05
6M	244.61	267.24	15.83	−0.10	13.46	−1.15	15.10	−0.38
60M	2437.78	278.05	15.33	0.39	15.15	0.16	15.94	0.82

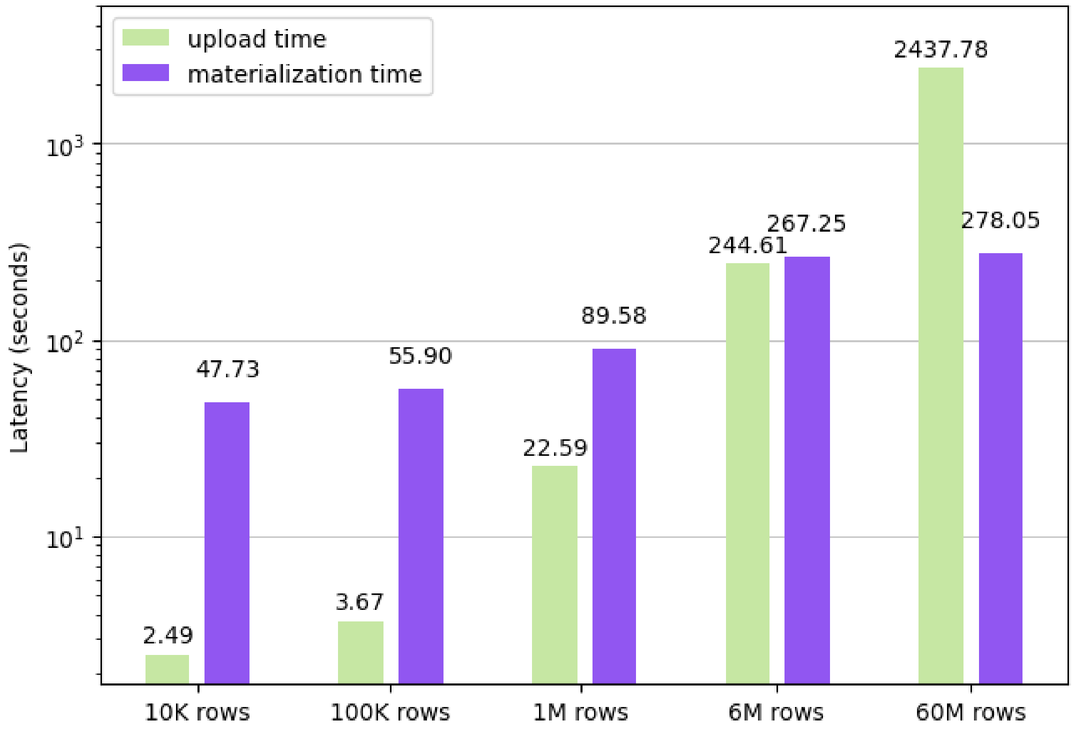


Figure 5.5: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with one CPU core.

Observing the results of experiments using more CPU cores, the upload time benefits from a higher number of CPU cores, in particular with larger tables (15% latency decrease) and less with smaller tables (4% latency decrease). On the other hand, the materialize time does not improve performance, having either small decreases in latency or small increases (both around 1-2%).

5.1.4. In-memory resources usage

The experimental environment resources defined in Section 3.2.6 were adjusted according to the computational needs. In particular, write operations were demanding on the available RAM resources, requiring up to 24 GBs to operate with the larger tables (6M and 60M rows). The system was adjusted to allocate 32768 MBs of RAM to avoid slowing down operations.

5.2. Results analysis and discussion

This section discusses the main results presented in Section 5.1, explaining their meaning, their implication for the company, and more generally for the research area. Additionally, this section provides a collection of project considerations outlined either during development or after the experiments conducted.

5.2.1. Discussion on major results

The results presented in Section 5.1 reveal the differences in latency (and thus also data throughput) between the newly implemented system using the delta-rs library and the legacy system. While the experiments present task-specific differences in performance between the two systems, overall, the system developed in this thesis has from ten up to forty times lower latency than the legacy system in all experiments when using the tables from 10K to 6M rows. These findings not only confirm the hypothesis that a Rust-based system would be faster than a Spark-based system when operating on small tables (from 10K to 6M rows) but also show that delta-rs is a preferable alternative to how read operations are currently performed using a combination of Arrow Flight and DuckDB (Section 2.5.2). Said outcomes further solidify the idea of the pivotal role that Rust will play in optimizing computer systems [4].

Something more can be said by taking a more in-depth look at the results of the writing experiments. In writing experiments, even with the largest table (60M rows), the newly implemented system has a latency from ten up to forty times lower than the legacy system.

When considering the smallest tables (10K and 100K rows), the improvement of the new system over the legacy pipeline is up to forty times. This confirms the need for Spark alternatives when elaborating on small-size data. For the use case defined in Section 3.2.3, it is clear that the new system is a better alternative in terms of performance but also costs, as maintaining a Spark cluster for this small amount of data makes little sense.

Moving on to reading experiments, as explained in Section 2.5.2, the legacy system, when reading, is already using a Spark alternative, i.e., Arrow Flight and DuckDB. The results show that there is a smaller difference between the two systems when reading the largest table (60M rows) with only 46% improvement. Nonetheless, the new system scales much better as the CPU cores increase, up to 89.75% with eight cores, while performance in the legacy system remains more or less the same. This has to do with how resources are allocated in the system. The user can dynamically modify his local resources, and since delta-rs uses local resources to operate, the user can tune the system as needed. On the other hand, the Arrow Flight server using DuckDB has a predefined amount of resources that cannot be modified as easily (the Hopsworks cluster needs to be restarted). Overall, the flexibility that the new system offers, at the expense of delegating computation on the client side, is remarkable and might benefit hybrid workflows.

These findings impact Hopsworks AB considerably, as it affects their main product, i.e., the Hopsworks feature store. The results recommend adopting the developed system over the current system using Apache Hudi as an offline feature store. Considering the Hopsworks AB approach of supporting multiple ways to load and save data, an alternative to the total substitution of the system would be leaving the option to the user to choose which data lakehouse format the offline feature store should save the data. It should be noted that the experiments and system evaluation were performed on the use case defined in Section 3.2.3. For different use cases, more experiments should be performed. The author expects that there will be a data size threshold where using a Spark-based will make more sense in terms of performance (lower latency).

More generally speaking, these findings cannot be generalized due to the intrinsic bias of conducting an industrial master thesis within the company developing the product to evaluate. Furthermore, the defined use case (Section 3.2.3) restricts the results on specific table sizes and computational resources. Results could vary if reading or writing more data with a different amount of resources. Nonetheless, the results confirm the research expectations, supporting the idea that Spark alternatives should be considered (and sometimes preferred) when working with small-sized tables (10K to 60M rows). This outcome is particularly relevant for Spark's numerous open-source community (more than 2800 contributors during its lifetime [31]). These findings also encourage more research and experiments on Spark alternatives and Rust applications in data management system implementations.

5.2.2. Considerations on the legacy system

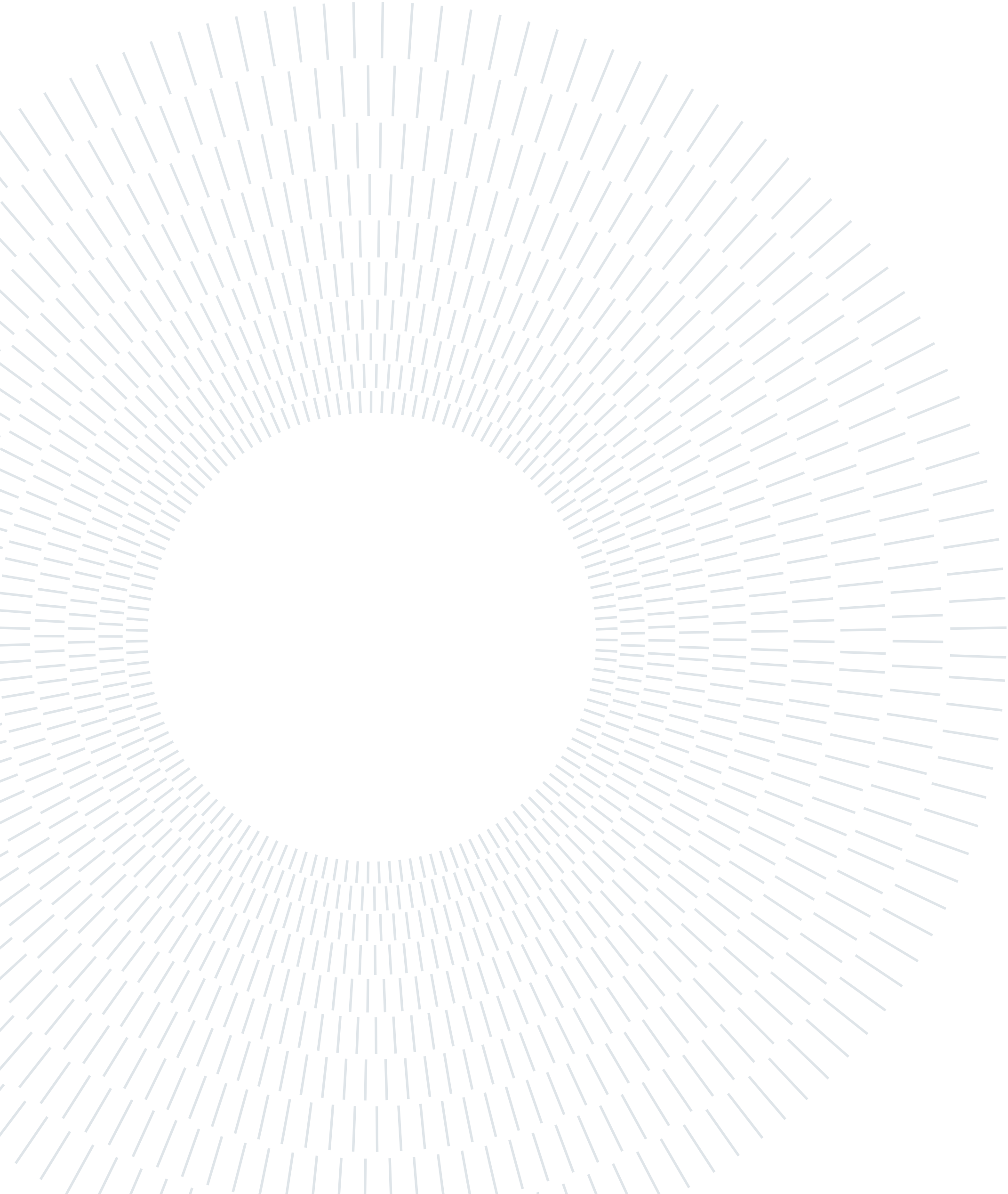
The legacy system presented in Sections 2.5.1 and 2.5.2 has a layered architecture design to fit multiple needs. For example, Kafka is used as a single point of upload of data both to the offline feature store analyzed in this thesis work and the online feature store designed for streaming pipelines. Using Kafka ensures that data is consistent between the two feature stores (online and offline) but also represents a system performance limitation. As results in Figure 5.5 and Table 5.5 show, Kafka, as data increases, represents the bottleneck of the architecture, accounting for 95% of the write latency when writing a 60M rows table. This mainly concerns how Kafka is used in the architecture and how data is sent, i.e., row by row. This work helped highlight this issue, but more research and experiments are required to find an effective solution. Enabling multiple uploads via concurrency mechanisms might speed up the upload process. Alternatively, sending columns instead of rows and keeping a column structure along the pipeline might also help.

Another aspect that limited the capability of the legacy architecture is how resources are allocated for the Spark cluster. These can be modified more easily than the Arrow Flight server's computation resources. However, balancing two systems (the client's and the Spark client's) creates an added complexity that is unnecessary when elaborating on small quantities of data.

5.2.3. Considerations on the delta-rs library

The system implementation focus of this project was adding support for HopsFS to the delta-rs library. This was also made possible thanks to the library's built-in modularity, which has a different sub-library for each storage connector. The community around the library is very active, and each question made on their communication channels, namely GitHub and Slack, would always receive an answer within a few hours or a day. The only recommendation the author would give to the library's maintainers would be to document further, perhaps with the help of architectural diagrams, the inner workings of the library's processes, specifying how and when data gets uploaded into memory. The first iteration of the read experiments had to be scratched due to calling a lazy function that would not load data into memory.

Considering the results presented in Section 5.1, the similarities in performance (latency and throughput) of the two delta-rs pipelines, operating on the LocalFS and HopsFS respectively, suggest that the library is operating at its full potential also in the newly implemented system. Delta-rs on the LocalFS still performs faster, but this might have to do with the different nature of the two storage systems, i.e., local and distributed.



6 Conclusions and Future work

This chapter presents the conclusions of the thesis work. It starts with Section 6.1 that compares the experimental findings with the Research Question to outline this thesis contribution. Section 6.2 then explains the limitations of the project in terms of resources and scope. Finally, Section 6.3 complements the chapter by considering possible future work stemming from this thesis.

6.1. Conclusions

This thesis work posed two RQs defined in Section 1.2.1. These were:

RQ1: How can we add support for HDFS and HopsFS to the delta-rs library to enable reading and writing on Delta Lake tables on the Hopsworks offline feature store?

RQ2: What is the difference in read and write latency and throughput between the current legacy system operating on the Hopsworks offline feature store and the delta-rs library operating on HopsFS?

The work conducted in this thesis answered these two questions by performing a system implementation and then evaluating the newly implemented system.

The first step was adding support for HopsFS in the delta-rs library. This was achieved by performing two code implementations of which the last one was used in the experiments. These two iterations proved necessary as HDFS support was added to the delta-rs library during development, so to fit the consistency and maintainability non-functional requirements, modifying that approach to support HopsFS was preferred. These code contributions are of more than two thousand LOC for the first implementation and eight hundred for the second. While these metrics presented may offer some insight into the value of the contribution, on the other hand, the true worth of this work lies in the development of a production-ready solution that successfully navigates a complex data stack with intricate interdependencies while meeting all specified requirements. The most significant acknowledgment of this contribution is the incorporation of this code implementation into the production environment of the Hopsworks feature store shortly after the publication of the thesis.

The second step was measuring and comparing the newly implemented system's performance with the legacy system. The metrics used were the latency (seconds) of the read and write operations and the throughput (rows/second), which was calculated by dividing the table size (in rows) by the latency of the operation. The results presented in Section 5.1 revealed that the delta-rs-based access to Delta Lake has a latency at least ten times lower in both read and write operations for tables from 10K to 6M rows in size. The write experiments show that the delta-rs library performs up to forty times better with smaller tables (10K and 100K) while still outperforming by ten times the legacy pipeline on the largest table (60M rows). This difference suggests that larger tables might have a threshold where a Spark-based system would perform better, but more experiments with larger tables are needed to verify the trend. Similarly, in the reading experiments, delta-rs outperforms the legacy pipeline more with smaller tables (10K and 100K rows), even if only by a factor of fifteen instead of forty, as seen for the writing experiments. This is probably caused by using a Spark alternative in the legacy system's reading process (Arrow Flight and DuckDB), which already improves Spark performances on smaller tables. One last notable finding on the difference between the newly implemented system and the legacy pipeline is the difference in scalability as resources increase. Experiments were conducted with an increasing number of CPU cores (from one to eight CPU cores), and the results showed that delta-rs, being a local process, is much more suited for making use of those resources with an up to 31% reduction in latency during the writing and an 87% reduction during reading.

Overall, the experiments' results recommend adopting the newly implemented system in the defined use case (Section 3.2.3), either as a replacement or an alternative for users who wish to store data in a Delta Table within the offline feature store.

6.2. Limitations

The limitations of this study mostly derive from the constraints of resources, in terms of time and computational resources, and scope, which is mainly linked to the defined use case (Section 3.2.3).

This project's scope is to improve the latency of the Hopsworks offline feature store, and this is the technology on which the implementation and the experiments were based. This outlines the limited generalization of the results obtained, which are biased from the use of technology in collaboration with the company developing it. Additionally, a specific use case was defined (Section 3.2.3) to choose the CPU loads and data loads on which the experiments would be conducted. This helps to define a perimeter of the thesis contribution but also limits the thesis impact on this specific use case, requiring more research to verify the same hypothesis in a different scenario.

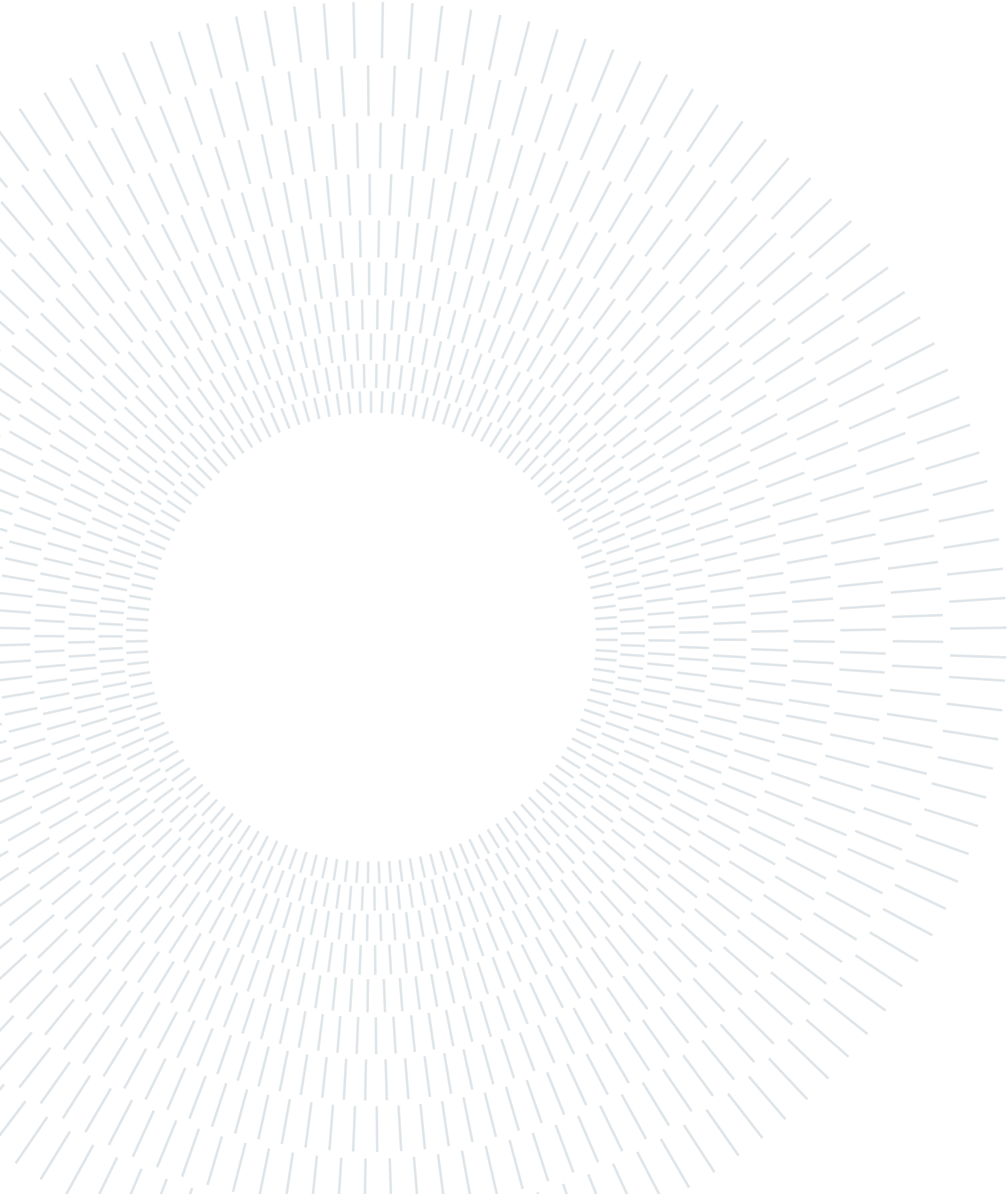
While great in size, the computational resources provided were used in a shared environment that could only be used for a limited amount of time and only if it was not operating other, more critical workloads. Time also played a role in limiting the number of experiments to calculate a 95% confidence interval. All experiments were run fifty times, which significantly increased the time required to perform all experiments.

6.3. Future work

The results and limitations of this thesis offer a good starting point for future work. As outlined in the limitations section, this thesis's scope and resources were limited. Conducting new experiments on the system performance by relaxing one or more constraints could bring new results that can be more general.

Considering the system research contribution of this thesis, this could be expanded mainly for Hopsworks AB needs. The code allowing delta-rs to read and write Delta Tables on HopsFS will probably see use mainly in Hopsworks AB, as even if HopsFS is an open-source technology, it does not see much use outside the Hopsworks tools. Nonetheless, future system contributions could still use the code developed in this thesis as a baseline to be compared with new delta-rs implementations or other ways to read and write on an offline feature store.

Future work on the system evaluation could expand on one or more of these aspects: data, pipelines, and experimental environment. Expanding on data would mean running the experiments with larger tables (600M rows, 6BN rows, etc.) to explore and verify if it exists a threshold where a Spark-based system performs better than delta-rs and at which table size. Also, making variations on the data sources would be a valid approach, although this would make this thesis an invalid baseline. The defined pipelines are specific to delta-rs or the Hopsworks architecture. Verifying the speed of other offline feature stores as Databricks would help generalize the results in a broader area. This would help determine which approach—Spark or delta-rs—performs best across different systems. As mentioned in the limitations, the experimental environment was a shared environment with extensive resources, but the current machine usage by other company employees varied. Using clean, isolated hardware could help future research verify this thesis's findings, isolating the noise of a shared environment.



Bibliography

[1] Amazon-Web-Services. Block vs File vs Object Storage - Difference Between Data Storage Services - AWS. URL `https://aws.amazon.com/compare/the-difference-between-block-file-object-storage/`.

[2] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, Aug. 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415560. URL `https://dl.acm.org/doi/10.14778/3415478.3415560`.

[3] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8, 2021. URL `https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf`.

[4] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 156–161, New York, NY, USA, May 2017. Association for Computing Machinery. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103006. URL `https://dl.acm.org/doi/10.1145/3102980.3103006`.

[5] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. Van Bussel, H. Van Hovell, M. Xue, R. Xin, and M. Zaharia. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2326–2339, Philadelphia PA USA, June 2022. ACM. ISBN 978-1-4503-9249-5. doi: 10.1145/3514221.3526054. URL `https://dl.acm.org/doi/10.1145/3514221.3526054`.

[6] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. 2005. URL `http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.10.0/docs/hdfs_design.pdf`.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. Jan. 2015. URL `https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf`.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, Mar. 1997. ISSN 0163-5808. doi: 10.1145/248603.248616. URL `https://dl.acm.org/doi/10.1145/248603.248616`.

[9] D. Croci. Data Lakehouse, beyond the hype, Dec. 2022. URL `https://bitrock.it/blog/data-lakehouse.html`.

[10] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, A. A. Ormenisan, S. Niazi, D. Bzhalava, K. Mak, V. Jouffrey, M. Ronström, R. Cunningham, R. Zangis, D. Mukhedkar, A. Khazanchi, V. Vlassov, and J. Dowling. The hopsworks feature store for machine learning. In *Companion of the 2024 International Conference on Management of Data*, Sigmod/Pods '24, pages 135–147, New York, NY, USA, June 2024. Association for Computing Machinery. ISBN 9798400704222. doi: 10.1145/3626246.3653389. URL `https://dl.acm.org/doi/10.1145/3626246.3653389`.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, Mar. 2004. URL `https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters`.

[12] M. L. Despa. Comparative study on software development methodologies. *Database Systems Journal*, 5(3), 2014. URL `https://dbjournal.ro/archive/17/17_4.pdf`.

[13] Dremio. State of the Data Lakehouse. Technical report, 2024. URL `https://www.dremio.com/wp-content/uploads/2023/11/whitepaper-2024-state-of-the-data-lakehouse_report.pdf`.

[14] T. Ebergen. Updates to the H2O.ai db-benchmark!, Nov. 2023. URL `https://duckdb.org/2023/11/03/db-benchmark-update.html`.

[15] L. J. Eder. Unstructured Data and the 80 Percent Rule, Aug. 2008. URL `https://web.archive.org/web/20240302060635/http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/`.

[16] M. Frampton. *Complete Guide to Open Source Big Data Stack.* Jan. 2018. ISBN 978-1-4842-2149-5. URL `https://link.springer.com/book/10.1007/978-1-4842-2149-5`.

[17] Google-Cloud. How Object vs Block vs File Storage differ. URL `https://cloud.google.com/discover/object-vs-block-vs-file-storage`.

[18] I. Gorton and J. Klein. Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems. *IEEE Software*, 32(3):78–85, May 2015. ISSN 1937-4194. doi: 10.1109/MS.2014.51. URL `https://ieeexplore.ieee.org/document/6774768`.

[19] J. Hermann and M. Del Balso. Meet Michelangelo: Uber's Machine Learning Platform, Sept. 2017. URL `https://www.uber.com/en-RO/blog/michelangelo-machine-learning-platform/`.

[20] A. Hussain. What is Green Software?, Aug. 2021. URL `https://greensoftware.foundation/articles/what-is-green-software`.

[21] IBM-Cloud-Education. Object vs. File vs. Block Storage: What's the Difference?, Oct. 2021. URL `https://www.ibm.com/blog/object-vs-file-vs-block-storage/`.

[22] P. Jansen. TIOBE Index, Apr. 2024. URL `https://www.tiobe.com/tiobe-index/`.

[23] R. Johnson, M. Armbrust, R. Xin, D. Lee, T. Das, B. Samwel, T. Kim, S. Sun, H. Raja, R. Potharaju, J. Yu, and S. Pierce. Announcing Delta Lake 3.0 with New Universal Format and Liquid Clustering, June 2023. URL `https://www.databricks.com/blog/announcing-delta-lake-30-new-universal-format-and-liquid-clustering`.

[24] A. Khazanchi. Faster reading with DuckDB and arrow flight on hopsworks : Benchmark and performance evaluation of offline feature stores. Master's thesis, KTH Royal Institute of Technology / KTH, School of Electrical Engineering and Computer Science (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2023. URL `https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-337297`.

[25] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. *Proceedings of the NetDB*, 2011. URL `https://notes.stephenholiday.com/Kafka.pdf`.

[26] B. Lorica, M. Armbrust, R. Xin, M. Zaharia, and A. Ghodsi. What Is a

Lakehouse?, Jan. 2020. URL `https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html`.

[27] H. Makait, S. Johnson, and M. Rocklin. Benchmark Results for Spark, Dask, DuckDB, and Polars — TPC-H Benchmarks at Scale, May 2024. URL `https://tpch.coiled.io/`.

[28] W. McKinney. Introducing Apache Arrow Flight: A Framework for Fast Data Transport, Oct. 2019. URL `https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/`.

[29] A. Nagpal and G. Gabrani. Python for Data Analytics, Scientific and Technical Applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 140–145, Feb. 2019. doi: 10.1109/AICAI.2019.8701341. URL `https://ieeexplore.ieee.org/document/8701341`.

[30] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Feb. 2017. ISBN 978-1-931971-36-2. URL `https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi`.

[31] OpenHub. The Apache Spark Open Source Project on Open Hub. URL `https://openhub.net/p/apache-spark`.

[32] S. Overflow. Stack Overflow Developer Survey 2023. URL `https://survey.stackoverflow.co/2023/`.

[33] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean. Carbon Emissions and Large Neural Network Training, Apr. 2021. URL `https://arxiv.org/abs/2104.10350`.

[34] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink, Apr. 2022. URL `https://arxiv.org/abs/2204.05149`.

[35] H. E. Pence. What is Big Data and Why is it Important? *Journal of Educational Technology Systems*, 43(2):159–171, Dec. 2014. ISSN 0047-2395, 1541-3810. doi: 10.2190/ET.43.2.d. URL `https://journals.sagepub.com/doi/10.2190/ET.43.2.d`.

[36] A. Pettersson. Resource-efficient and fast Point-in-Time joins for Apache Spark : Optimization of time travel operations for the creation of machine learning training datasets. Master's thesis, KTH, School of Electrical Engineering and Computer Sci-

ence (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2022. URL `https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-317443`.

[37] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *Sigmod Record*, 29(4):64–71, Dec. 2000. ISSN 0163-5808. doi: 10.1145/369275.369291. URL `https://dl.acm.org/doi/10.1145/369275.369291`.

[38] M. Raasveldt and H. Mühleisen. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam Netherlands, June 2019. ACM. ISBN 978-1-4503-5643-5. doi: 10.1145/3299869.3320212. URL `https://dl.acm.org/doi/10.1145/3299869.3320212`.

[39] P. Rajaperumal. Uber Engineering's Incremental Processing Framework on Hadoop, Mar. 2017. URL `https://www.uber.com/blog/hoodie/`.

[40] M. Raschka and S. Vahid. *Python Machine Learning (3rd Edition)*. Packt Publishing, Dec. 2019. ISBN 978-1-78995-575-0. URL `https://www.packtpub.com/en-us/product/python-machine-learning-9781789955750?srsltid=AfmBOorAl09qiabPkZthSSkbdsaVCIyJHB-Nm8cTNaBRqAE-T-rzcCZ6`.

[41] S. Sakr. Big Data Processing Stacks. *IT Professional*, 19(1):34–41, Jan. 2017. ISSN 1941-045X. doi: 10.1109/MITP.2017.6. URL `https://ieeexplore.ieee.org/document/7839846`.

[42] R. Shah and R. Tkachuk. Improve Your OLAP Environment with Microsoft and Teradata. 2007. URL `https://assets.teradata.com/resourceCenter/downloads/Brochures/eb5289.pdf`.

[43] Transaction-Processing-Performance-Council-(TPC). TPC benchmark H: Decision support, standard verification, revision v.3.0.1, 2022. URL `https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf`.

[44] Twitter. Dremel made simple with Parquet, Sept. 2013. URL `https://blog.x.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet`.

[45] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995. URL `https://ir.cwi.nl/pub/5007/05007D.pdf`.

[46] R. Vink. I wrote one of the fastest DataFrame libraries, Feb. 2021. URL `https://www.ritchievink.com/blog/2021/02/28/i-wrote-one-of-the-fastest-dataframe-libraries/`.

[47] K. Weller. Apache Hudi vs Delta Lake vs Apache Iceberg - Data Lakehouse Feature Comparison, Jan. 2024. URL `https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison`.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association. doi: 10.5555/1863103.1863113. URL `https://dl.acm.org/doi/10.5555/1863103.1863113`.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, July 2011. URL `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf`.

[50] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, Oct. 2016. ISSN 0001-0782, 1557-7317. doi: 10.1145/2934664. URL `https://dl.acm.org/doi/10.1145/2934664`.

A | System architectures

This appendix reports the legacy architecture diagrams shown in Section 2.5 increased in size to improve readability.

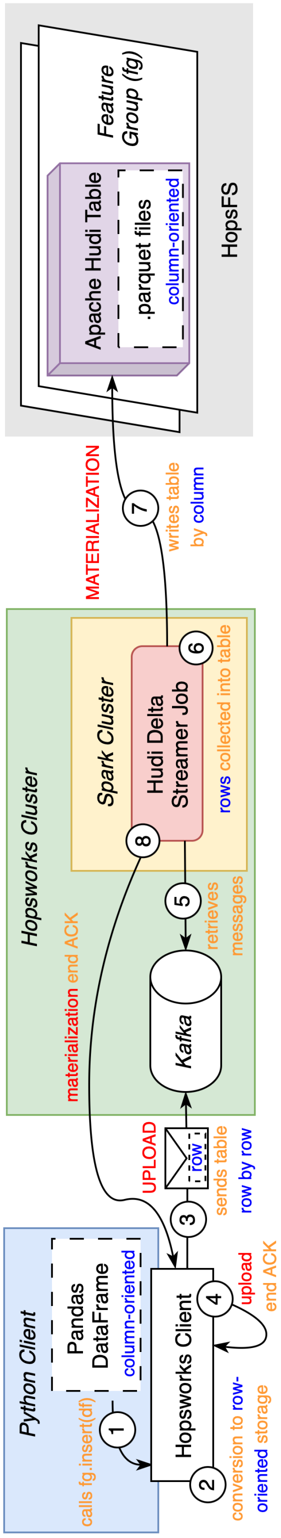


Figure A.1: Legacy system writing a Pandas DataFrame from a Python client to the Hopsworks offline Feature Store. This image was magnified to enhance visualization.

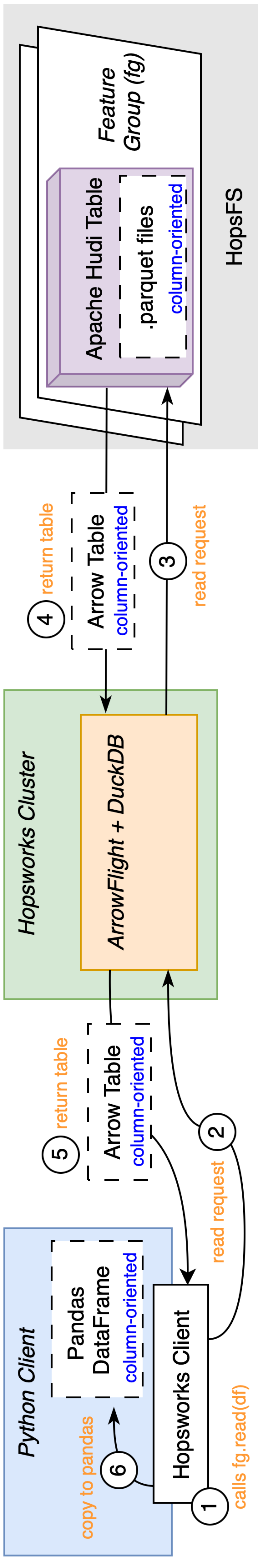
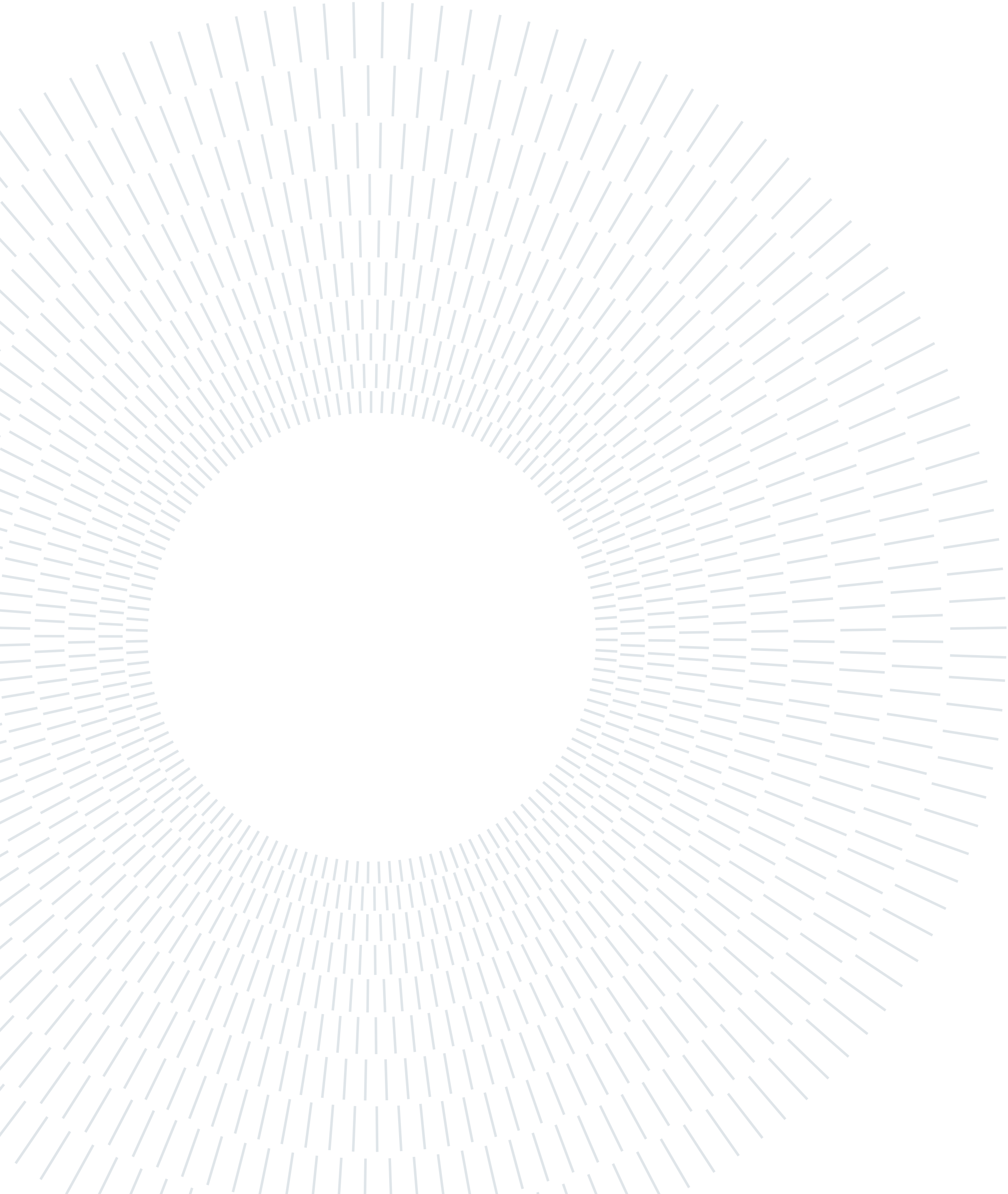


Figure A.2: Legacy system reading a table from the Hopsworks offline feature store and loading it into the Python client's local memory. This image was magnified to enhance visualization.



B | Write experiments results

This appendix reports all graphs and tables related to the write experiments conducted. Results are reported first expressed as latency (measured during the experiments) and then as throughput (computed from the latency and table size).

Table B.1: Write experiment results expressed as latency. The experiment was performed with one CPU core.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	Latency (seconds) 95% Confidence Interval high
delta-rs HopsFS	10K	1.250 88	1.238 07	1.265 45
	100K	1.368 28	1.337 57	1.389 82
	1M	9.381 52	9.239 71	9.529 04
	6M	19.754 69	19.332 70	20.117 85
	60M	177.307 07	174.628 71	180.017 32
delta-rs LocalFS	10K	0.039 57	0.037 70	0.041 53
	100K	0.152 40	0.145 98	0.158 88
	1M	8.422 52	8.283 96	8.563 76
	6M	17.906 34	17.480 40	18.335 85
	60M	172.345 52	169.748 08	174.731 38
Legacy	10K	50.227 67	49.535 01	50.936 64
	100K	59.561 87	58.894 66	60.184 96
	1M	112.190 48	111.371 62	113.009 15
	6M	511.816 93	510.751 13	512.836 72
	60M	2715.772 85	2699.880 61	2731.952 25

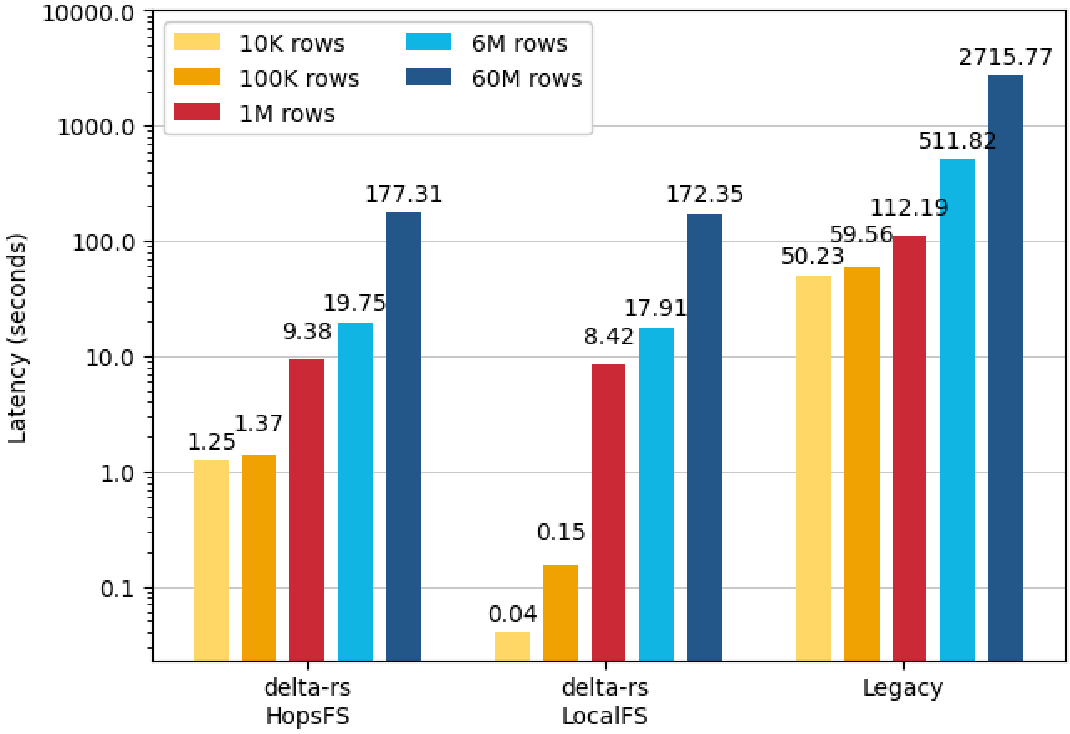


Figure B.1: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with one CPU core.

Table B.2: Write experiment results expressed as latency. The experiment was performed with two CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	1.262 39	1.250 79	1.276 39
	100K	1.308 12	1.280 50	1.332 17
	1M	8.515 36	8.343 33	8.700 77
	6M	16.290 42	15.906 59	16.673 62
	60M	134.060 89	131.650 31	136.397 61
delta-rs LocalFS	10K	0.048 23	0.046 40	0.049 97
	100K	0.137 14	0.134 02	0.140 50
	1M	7.185 30	7.037 47	7.351 28
	6M	15.266 32	14.851 72	15.651 67
	60M	129.820 07	127.600 20	132.046 89
Legacy	10K	50.724 05	50.107 69	51.306 86
	100K	59.788 10	58.979 97	60.474 27
	1M	108.564 99	108.011 24	109.081 28
	6M	473.379 54	472.345 34	474.437 40
	60M	2340.770 13	2333.994 43	2347.971 27

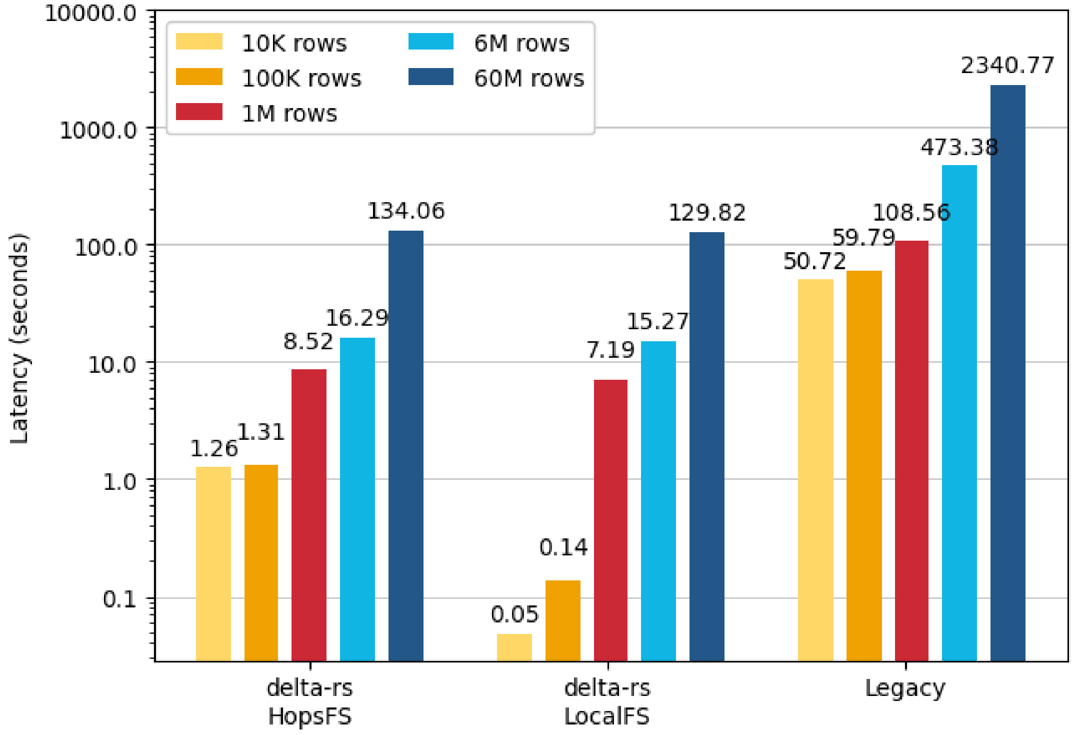


Figure B.2: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with two CPU cores.

Table B.3: Write experiment results expressed as latency. The experiment was performed with four CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	1.216 42	1.202 32	1.232 31
	100K	1.336 22	1.322 94	1.349 42
	1M	8.413 25	8.247 70	8.582 72
	6M	16.224 02	15.879 46	16.595 86
	60M	124.102 42	121.577 23	126.815 30
delta-rs LocalFS	10K	0.045 72	0.043 41	0.048 07
	100K	0.131 76	0.128 80	0.134 99
	1M	7.185 74	7.006 79	7.363 43
	6M	14.555 78	14.176 79	14.941 92
	60M	121.376 23	119.172 56	123.698 90
Legacy	10K	51.284 65	50.622 82	51.903 67
	100K	59.526 55	58.905 37	60.153 22
	1M	108.816 74	108.252 17	109.342 34
	6M	481.983 53	481.044 35	482.929 92
	60M	2346.046 87	2336.993 96	2355.198 97

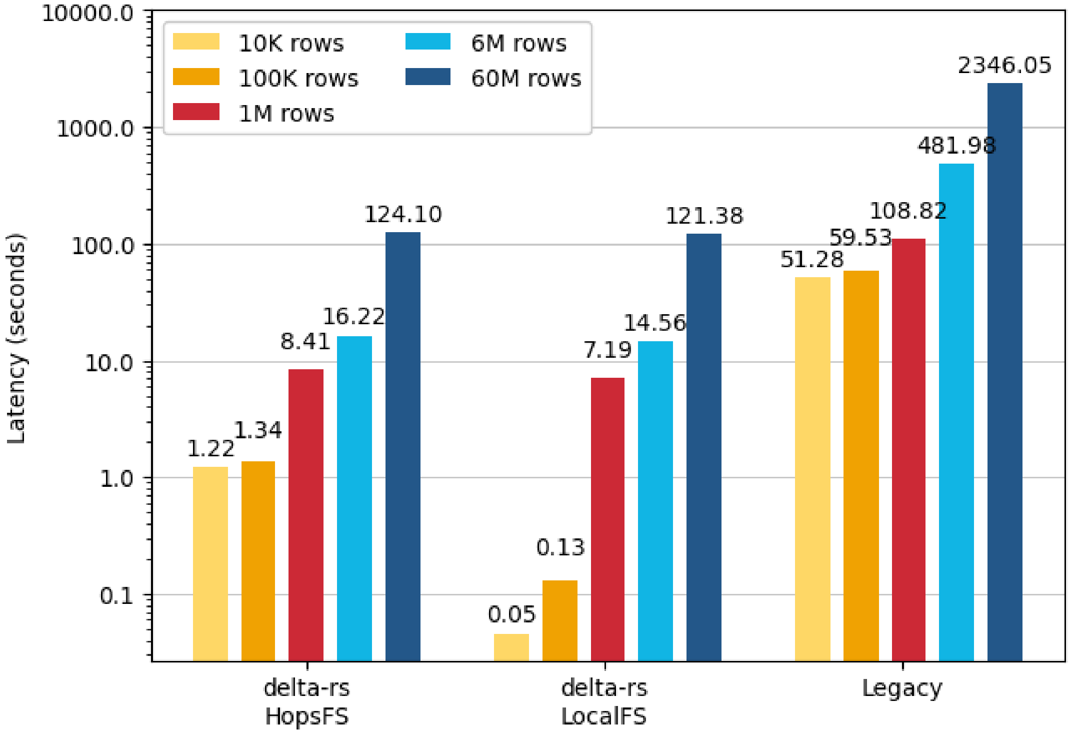


Figure B.3: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with four CPU cores.

Table B.4: Write experiment results expressed as latency. The experiment was performed with eight CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	Latency (seconds) 95% Confidence Interval high
delta-rs HopsFS	10K	1.367 56	1.249 34	1.572 24
	100K	1.292 43	1.265 48	1.310 99
	1M	8.301 20	8.149 18	8.470 40
	6M	15.738 47	15.289 74	16.160 84
	60M	121.950 14	119.593 76	124.180 97
delta-rs LocalFS	10K	0.044 02	0.041 74	0.046 40
	100K	0.136 48	0.132 81	0.140 61
	1M	7.228 72	7.075 11	7.398 93
	6M	14.281 57	13.905 08	14.661 26
	60M	119.979 15	117.764 16	122.208 82
Legacy	10K	51.228 59	50.594 78	51.864 76
	100K	60.277 51	59.729 07	60.771 30
	1M	109.381 89	108.868 30	109.882 63
	6M	475.943 45	474.839 93	477.052 74
	60M	2324.979 17	2319.042 03	2331.047 94

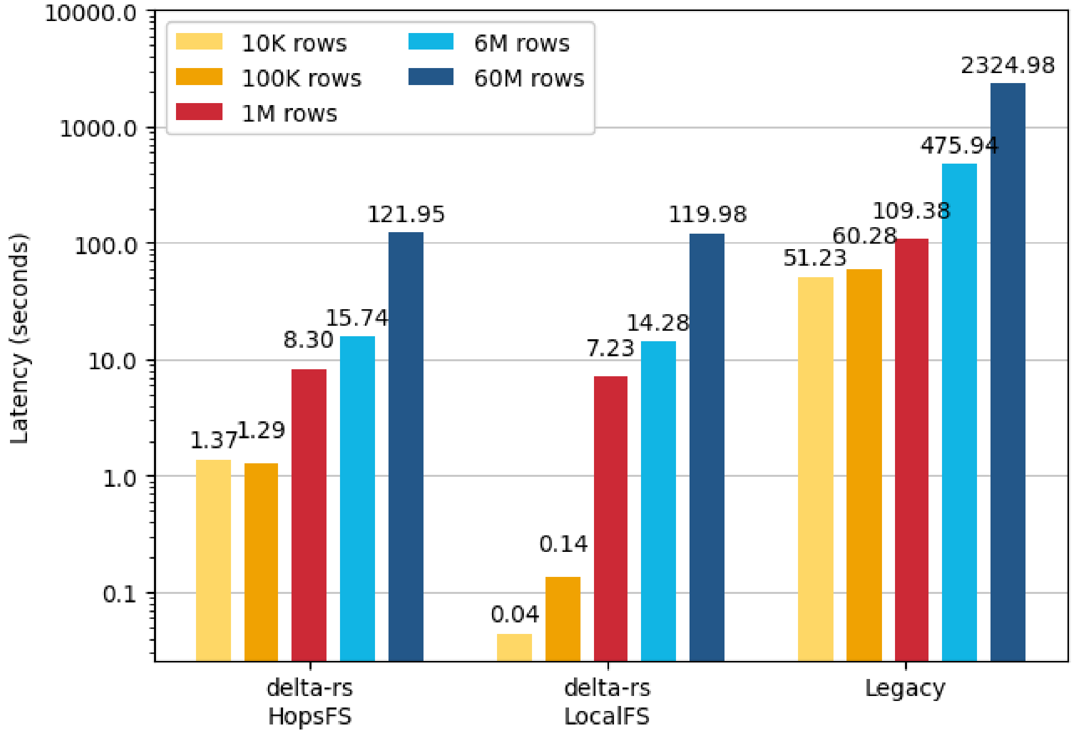


Figure B.4: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with eight CPU cores.

Table B.5: Write experiment results expressed as throughput. The experiment was performed with one CPU core.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval	
			low	high
delta-rs HopsFS	10K	7.994 36	7.902 30	8.077 05
	100K	7.308 43	7.195 14	7.476 21
	1M	106.592 42	104.942 26	108.228 50
	6M	303.725 33	298.242 52	310.354 91
	60M	338.395 98	333.301 26	343.586 10
delta-rs LocalFS	10K	252.682 38	240.734 05	265.186 32
	100K	656.157 39	629.367 77	684.985 18
	1M	118.729 19	116.771 10	120.715 14
	6M	335.076 75	327.227 70	343.241 43
	60M	348.137 84	343.384 22	353.464 96
Legacy	10K	0.199 09	0.196 32	0.201 87
	100K	1.678 92	1.661 54	1.697 94
	1M	8.913 41	8.848 84	8.978 94
	6M	11.722 94	11.699 63	11.747 40
	60M	22.093 15	21.962 31	22.223 20

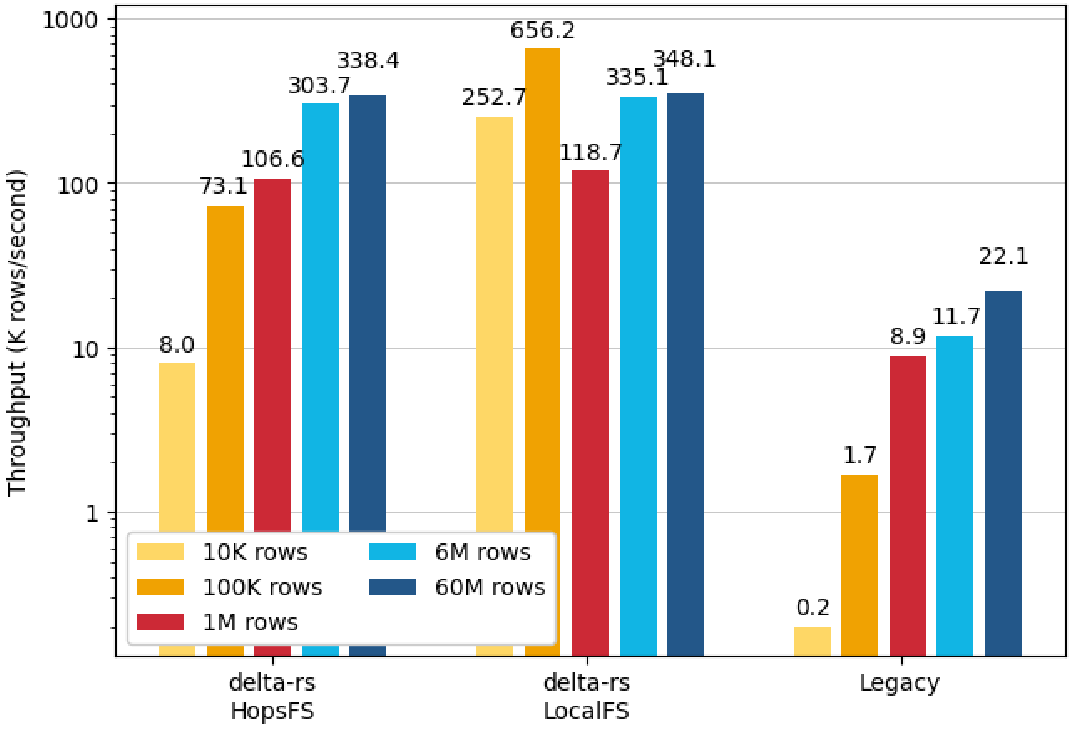


Figure B.5: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with one CPU core.

Table B.6: Write experiment results expressed as throughput. The experiment was performed with two CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	Throughput (k rows/second) 95% Confidence Interval high
delta-rs HopsFS	10K	7.921 46	7.834 58	7.994 91
	100K	76.445 07	75.064 99	78.094 33
	1M	117.434 78	114.932 31	119.856 14
	6M	368.314 40	359.849 75	377.201 98
	60M	447.557 80	439.890 38	455.752 81
delta-rs LocalFS	10K	207.319 22	200.086 26	215.493 42
	100K	729.159 67	711.739 66	746.138 54
	1M	139.172 97	136.030 55	142.096 47
	6M	393.021 85	383.345 60	403.993 52
	60M	462.178 14	454.384 03	470.218 68
Legacy	10K	0.197 14	0.194 90	0.199 57
	100K	1.672 57	1.653 59	1.695 49
	1M	9.211 07	9.167 47	9.258 29
	6M	12.674 81	12.646 55	12.702 57
	60M	25.632 58	25.553 97	25.707 00

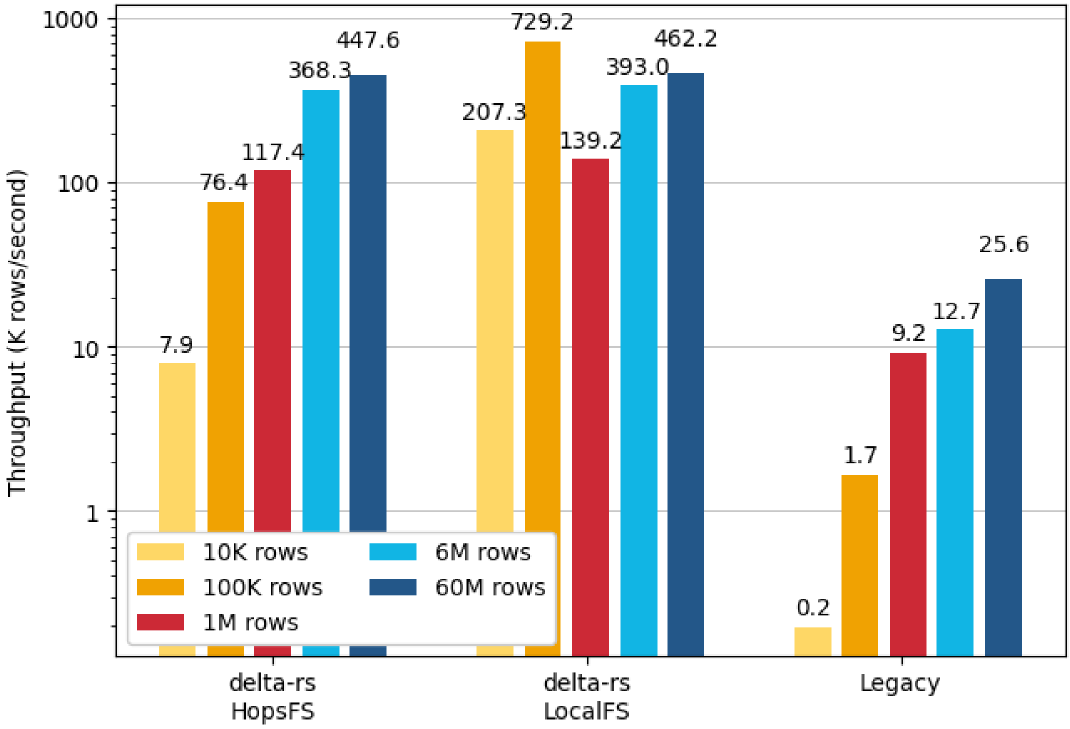


Figure B.6: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with two CPU cores.

Table B.7: Write experiment results expressed as throughput. The experiment was performed with four CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	high
delta-rs HopsFS	10K	8.220 83	8.114 82	8.114 82
	100K	74.837 42	74.105 66	75.588 71
	1M	118.860 08	116.513 10	121.245 82
	6M	369.822 02	361.535 77	377.846 52
	60M	483.471 60	473.128 99	493.513 43
delta-rs LocalFS	10K	218.713 64	208.026 04	230.314 12
	100K	758.924 22	740.794 74	776.391 15
	1M	139.164 32	135.806 13	142.718 64
	6M	412.207 28	401.554 59	423.226 78
	60M	494.330 70	485.048 75	503.471 56
Legacy	10K	0.194 99	0.192 66	0.197 53
	100K	1.679 92	1.662 42	1.697 63
	1M	9.189 76	9.145 58	9.237 68
	6M	12.448 55	12.424 16	12.472 86
	60M	25.574 93	25.475 55	25.674 00

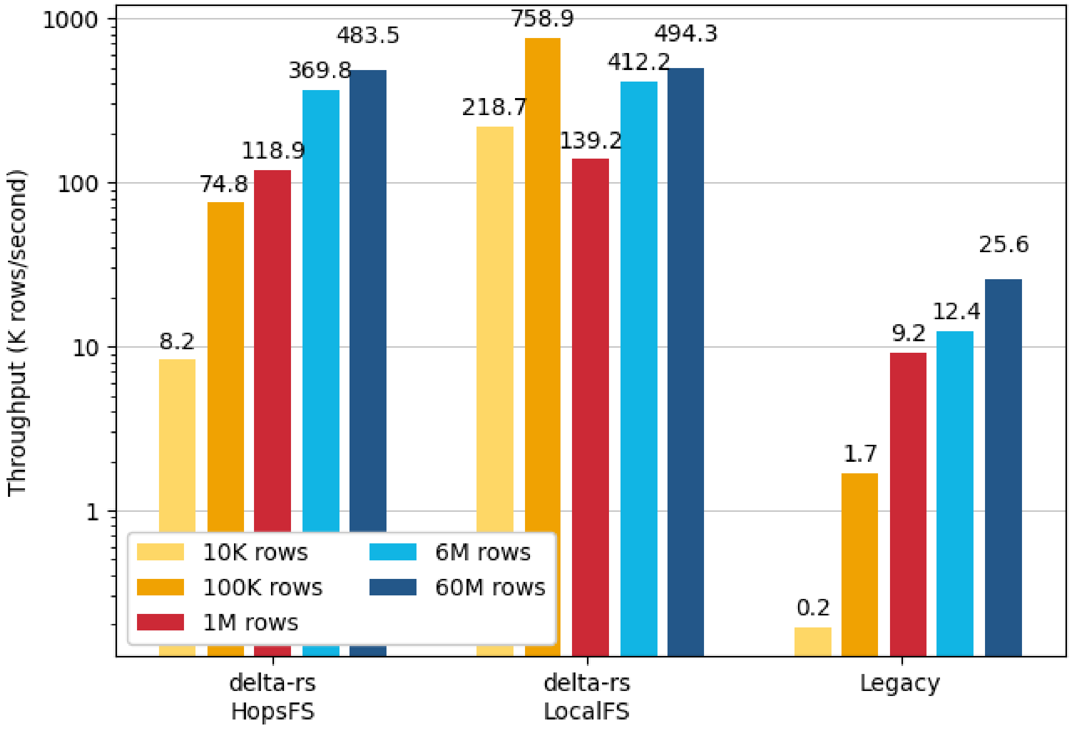


Figure B.7: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with four CPU cores.

Table B.8: Write experiment results expressed as throughput. The experiment was performed with eight CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval	
			low	high
delta-rs HopsFS	10K	7.312 28	6.360 32	8.004 22
	100K	77.373 37	76.277 82	79.021 04
	1M	120.464 39	118.058 14	122.711 60
	6M	381.231 26	371.267 72	392.419 78
	60M	492.004 31	483.165 79	501.698 37
delta-rs LocalFS	10K	227.120 95	215.487 14	239.541 28
	100K	732.701 41	711.160 38	752.932 00
	1M	138.337 01	135.154 66	141.340 41
	6M	420.121 65	409.241 76	431.496 69
	60M	500.086 88	490.962 88	509.492 86
Legacy	10K	0.195 20	0.192 80	0.197 64
	100K	1.658 99	1.645 51	1.674 22
	1M	9.142 28	9.100 61	9.185 40
	6M	12.606 53	12.577 22	12.635 83
	60M	25.806 68	25.739 49	25.872 75

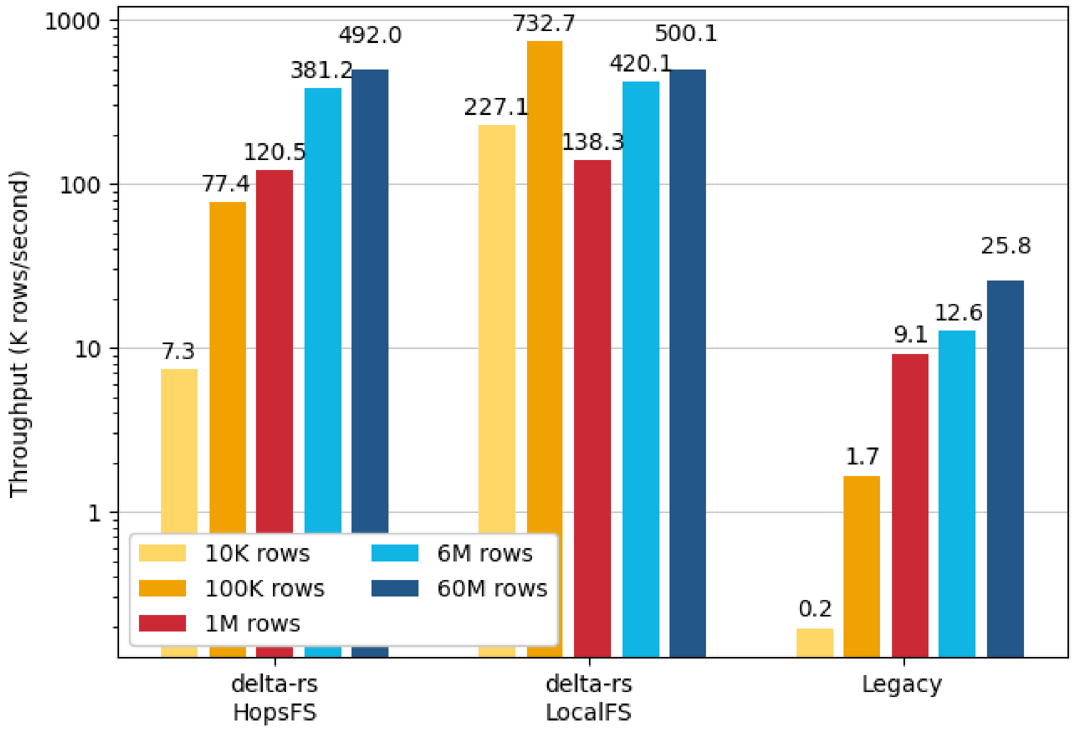
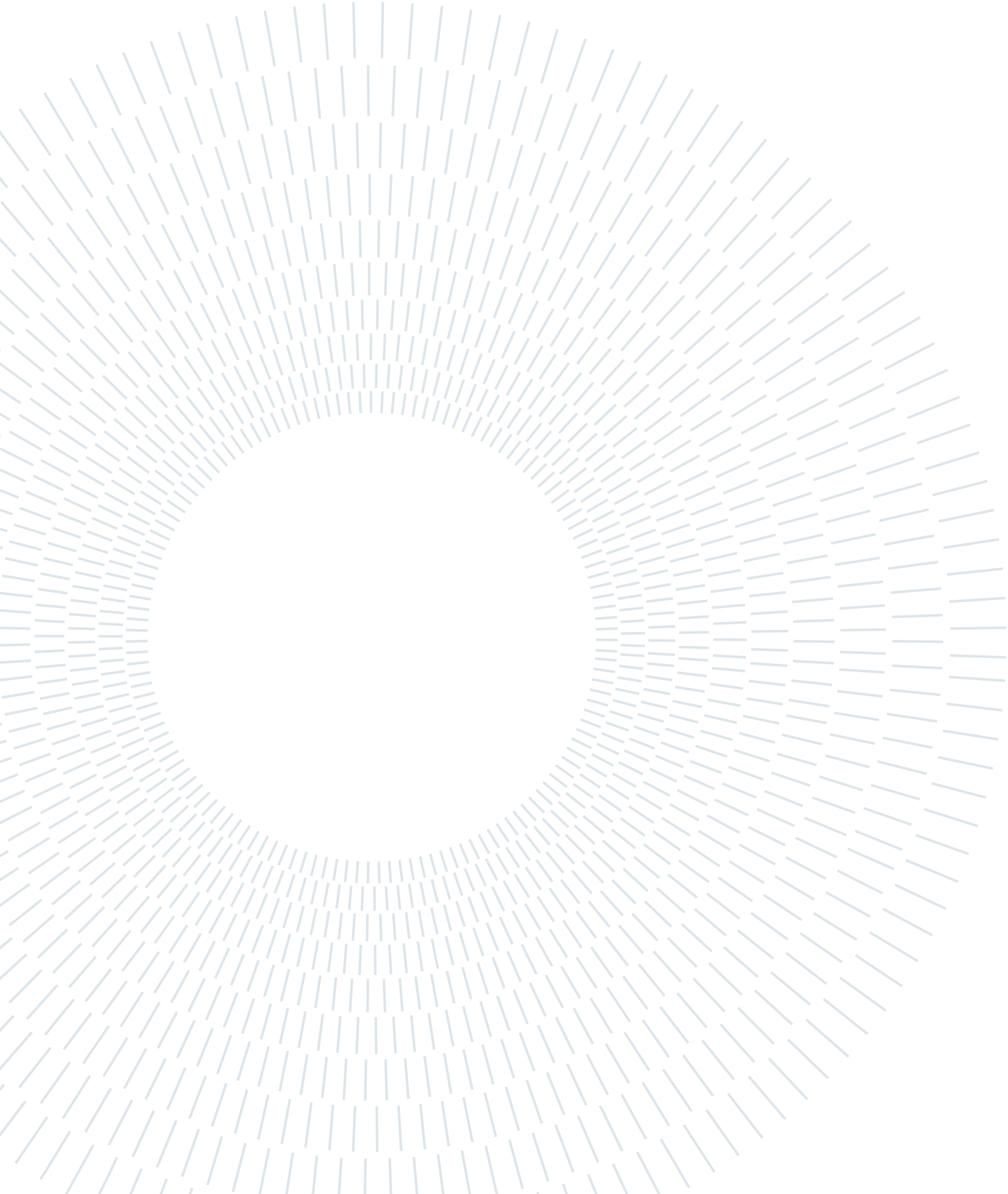


Figure B.8: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with eight CPU cores.



C | Read experiments results

This appendix reports all graphs and tables related to the read experiments conducted. Results are reported first expressed as latency (measured during the experiments) and then as throughput (computed from the latency and table size).

Table C.1: Read experiment results expressed as latency. The experiment was performed with one CPU core.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	0.053 42	0.039 16	0.081 12
	100K	0.057 57	0.055 18	0.060 46
	1M	0.538 55	0.525 58	0.552 29
	6M	1.948 99	1.930 07	1.968 60
	60M	22.980 65	22.840 67	23.142 06
delta-rs LocalFS	10K	0.004 19	0.002 68	0.006 44
	100K	0.026 96	0.019 66	0.034 33
	1M	0.420 09	0.406 13	0.435 63
	6M	1.682 23	1.659 81	1.704 40
	60M	19.565 47	19.346 90	19.777 24
Legacy	10K	0.631 59	0.624 14	0.641 57
	100K	2.650 10	2.642 72	2.658 76
	1M	8.596 36	8.340 94	8.900 47
	6M	33.529 64	33.238 86	33.865 91
	60M	33.697 72	33.362 62	34.086 65

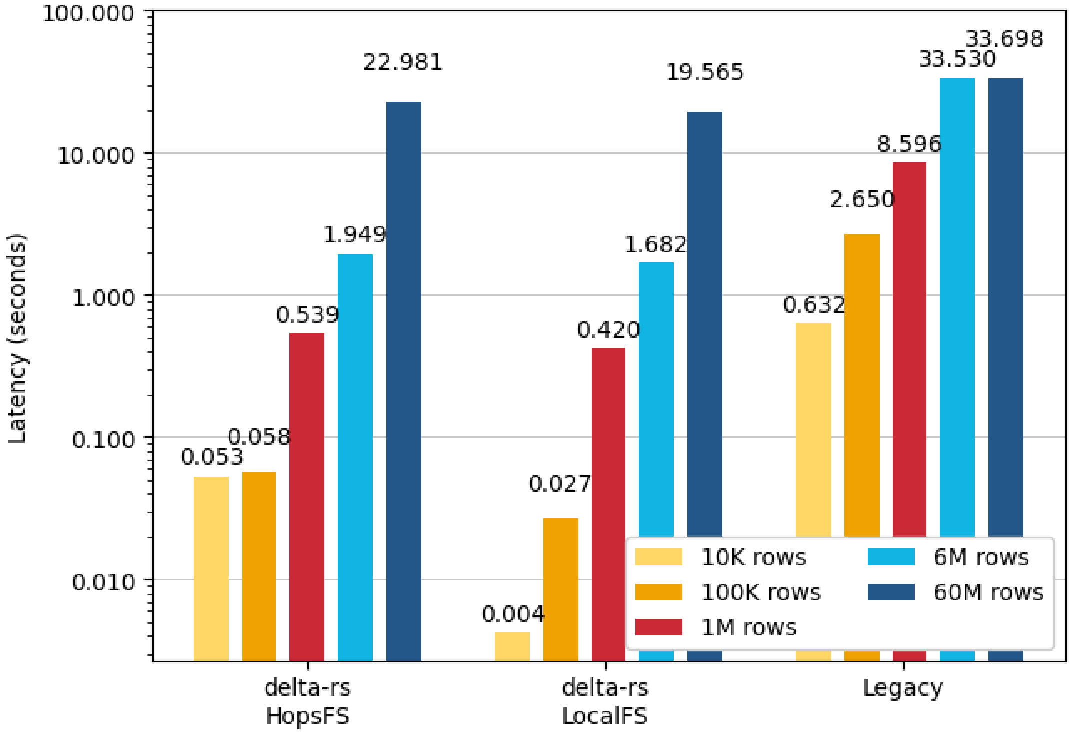


Figure C.1: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with one CPU core.

Table C.2: Read experiment results expressed as latency. The experiment was performed with two CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	0.041 32	0.039 33	0.043 78
	100K	0.056 90	0.051 23	0.066 93
	1M	0.234 13	0.225 28	0.244 26
	6M	0.908 32	0.899 67	0.917 44
	60M	11.413 25	11.276 61	11.588 99
delta-rs LocalFS	10K	0.002 87	0.002 78	0.002 99
	100K	0.013 06	0.010 41	0.016 10
	1M	0.199 77	0.188 58	0.210 56
	6M	0.747 64	0.735 03	0.760 13
	60M	9.446 93	9.372 07	9.517 53
Legacy	10K	0.624 92	0.622 10	0.628 22
	100K	2.663 39	2.656 16	2.671 66
	1M	8.616 67	8.309 89	8.949 38
	6M	33.375 19	33.096 88	33.670 65
	60M	33.642 81	33.301 50	34.063 07

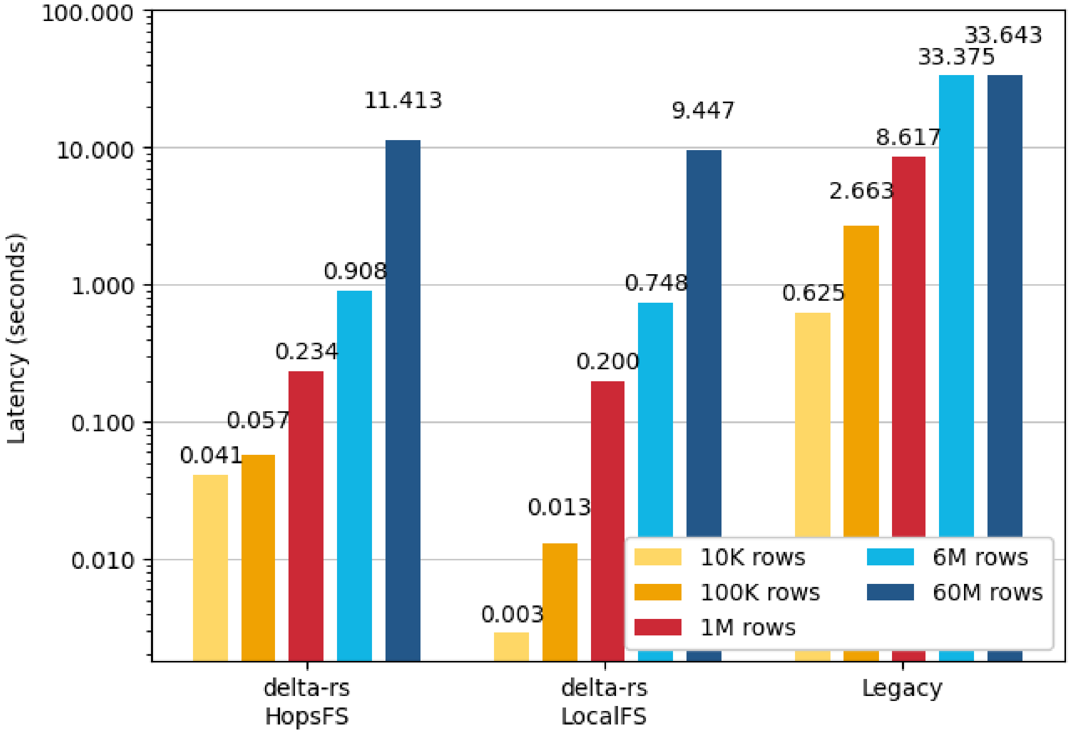


Figure C.2: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with two CPU cores.

Table C.3: Read experiment results expressed as latency. The experiment was performed with four CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	0.043 36	0.039 22	0.050 92
	100K	0.055 40	0.053 78	0.057 89
	1M	0.188 47	0.151 57	0.251 89
	6M	0.531 24	0.507 78	0.571 05
	60M	5.580 11	5.543 97	5.619 36
delta-rs LocalFS	10K	0.002 68	0.002 59	0.002 79
	100K	0.009 23	0.008 52	0.010 20
	1M	0.089 71	0.083 88	0.095 03
	6M	0.370 21	0.360 18	0.380 32
	60M	4.810 23	4.793 38	4.827 89
Legacy	10K	0.635 83	0.623 52	0.659 08
	100K	2.639 85	2.633 49	2.646 23
	1M	8.752 38	8.507 25	9.013 83
	6M	33.452 86	33.194 61	33.756 46
	60M	33.652 45	33.270 16	34.039 00

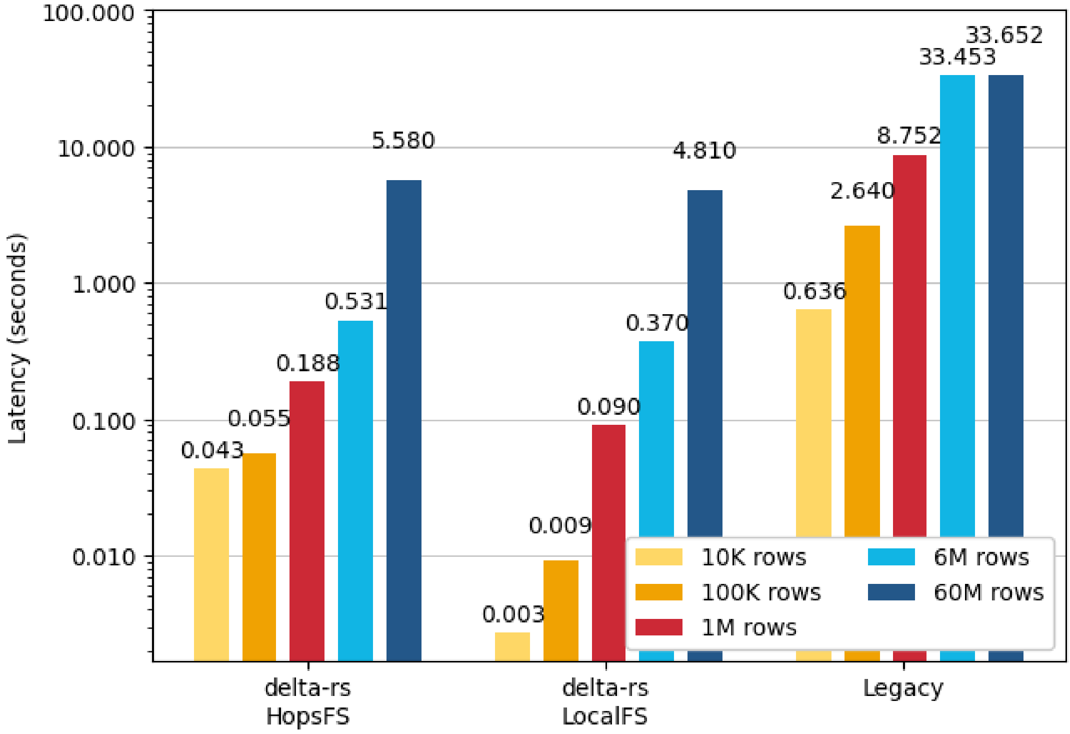


Figure C.3: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with four CPU cores.

Table C.4: Read experiment results expressed as latency. The experiment was performed with eight CPU cores.

Pipeline	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
delta-rs HopsFS	10K	0.043 30	0.038 85	0.051 19
	100K	0.054 58	0.052 86	0.056 63
	1M	0.173 90	0.169 92	0.178 08
	6M	0.497 29	0.486 55	0.510 19
	60M	2.942 36	2.855 54	3.063 60
delta-rs LocalFS	10K	0.002 94	0.002 84	0.003 07
	100K	0.009 48	0.008 72	0.010 56
	1M	0.043 08	0.039 34	0.048 30
	6M	0.175 48	0.170 09	0.180 80
	60M	2.285 50	2.275 01	2.296 07
Legacy	10K	0.627 39	0.622 45	0.632 59
	100K	2.662 17	2.653 09	2.672 18
	1M	8.347 57	8.134 71	8.609 42
	6M	33.428 15	33.153 76	33.749 47
	60M	33.143 41	32.883 03	33.412 99

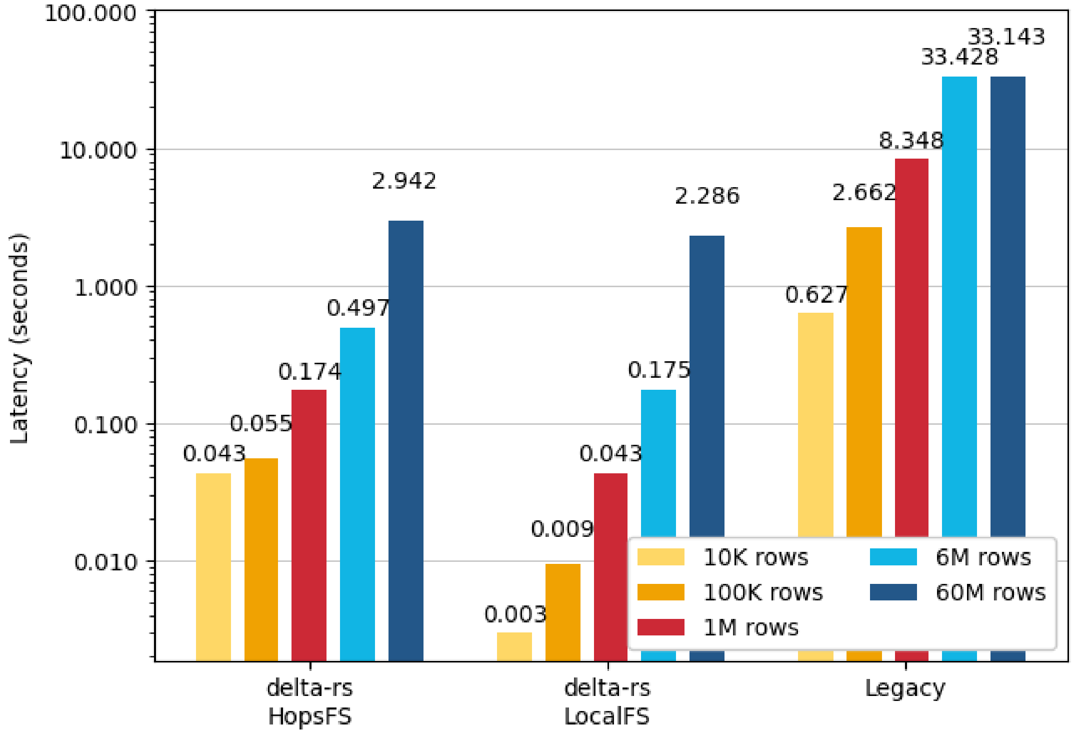


Figure C.4: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with eight CPU cores.

Table C.5: Read experiment results expressed as throughput. The experiment was performed with one CPU core.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	high
delta-rs HopsFS	10K	187.168 53	123.265 55	255.361 73
	100K	1736.907 99	1653.919 40	1811.928 57
	1M	1856.831 67	1810.613 18	1902.653 61
	6M	3078.512 99	3047.839 14	3108.694 31
	60M	2610.891 46	2592.681 85	2626.892 40
delta-rs LocalFS	10K	2384.586 99	1552.085 52	3721.360 68
	100K	3708.257 87	2912.436 00	5084.757 15
	1M	2380.403 81	2295.501 54	2462.258 14
	6M	3566.674 54	3520.297 96	3614.870 06
	60M	3066.626 44	3033.789 96	3101.271 65
Legacy	10K	15.832 85	15.586 54	16.021 96
	100K	37.734 32	37.611 40	37.839 75
	1M	116.328 20	112.353 50	119.890 44
	6M	178.946 12	177.169 28	180.511 59
	60M	1780.535 63	1760.219 74	1798.419 62

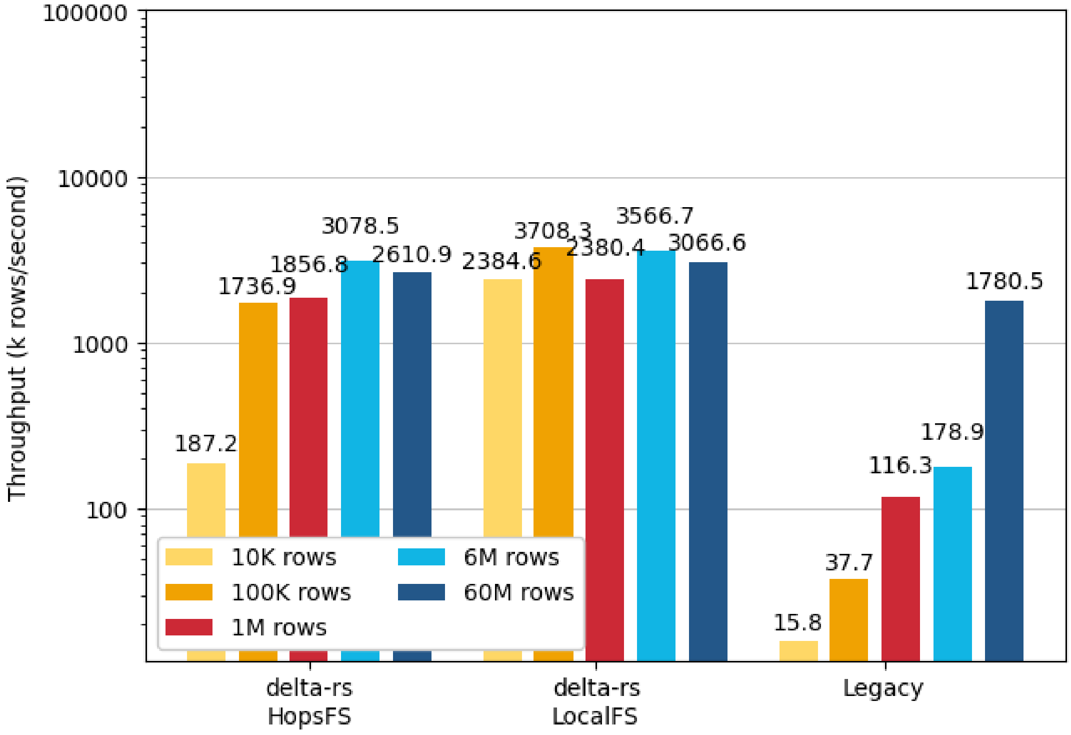


Figure C.5: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with one CPU core.

Table C.6: Read experiment results expressed as throughput. The experiment was performed with two CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	Throughput (k rows/second) 95% Confidence Interval high
delta-rs HopsFS	10K	241.978 33	228.370 18	254.224 19
	100K	1757.179 30	1494.018 76	1951.812 29
	1M	4271.121 39	4093.989 17	4438.846 12
	6M	6605.543 23	6539.936 31	6669.087 56
	60M	5257.046 58	5177.323 95	5320.743 34
delta-rs LocalFS	10K	3479.952 85	3339.119 82	3592.422 55
	100K	7652.747 09	6210.296 38	9604.332 93
	1M	5005.616 42	4749.179 43	5302.536 56
	6M	8025.196 34	7893.332 51	8162.844 93
	60M	6351.267 15	6304.155 38	6401.994 67
Legacy	10K	16.001 83	15.917 73	16.074 53
	100K	37.546 07	37.429 79	37.648 22
	1M	116.054 04	111.739 51	120.338 48
	6M	179.774 24	178.196 71	181.285 93
	60M	1783.441 98	1761.438 30	1801.720 13

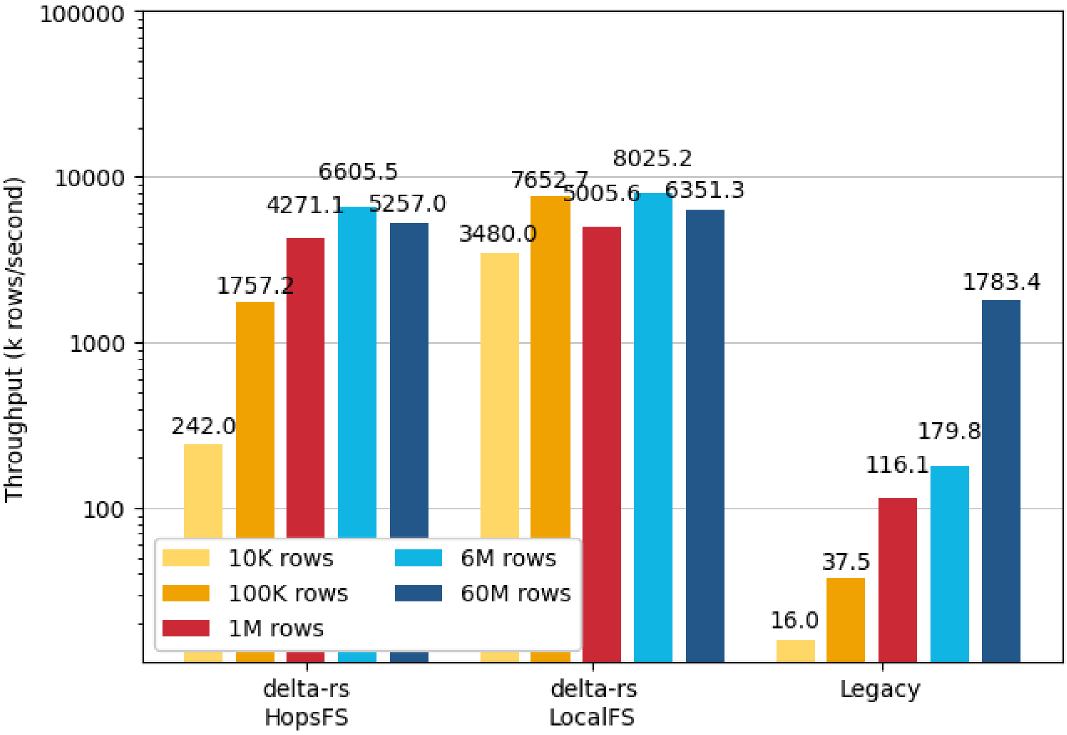


Figure C.6: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with two CPU cores.

Table C.7: Read experiment results expressed as throughput. The experiment was performed with four CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	high
delta-rs HopsFS	10K	230.619 25	196.373 66	254.932 32
	100K	1804.731 06	1727.272 56	1859.407 19
	1M	5305.742 39	3969.956 99	6597.374 04
	6M	11 294.126 19	10 506.919 39	11 816.033 13
	60M	10 752.475 11	10 677.356 47	10 822.556 20
delta-rs LocalFS	10K	3720.941 04	3572.450 85	3854.749 73
	100K	10 830.884 57	9802.960 62	11 735.928 63
	1M	11 146.067 43	10 522.540 71	11 921.622 57
	6M	16 206.973 30	15 775.943 87	16 657.937 25
	60M	12 473.404 92	12 427.787 28	12 517.259 67
Legacy	10K	15.727 24	15.172 59	16.037 90
	100K	37.880 93	37.789 59	37.972 37
	1M	114.254 60	110.940 61	117.546 71
	6M	179.356 80	177.743 72	180.752 22
	60M	1782.930 73	1762.683 96	1803.417 64

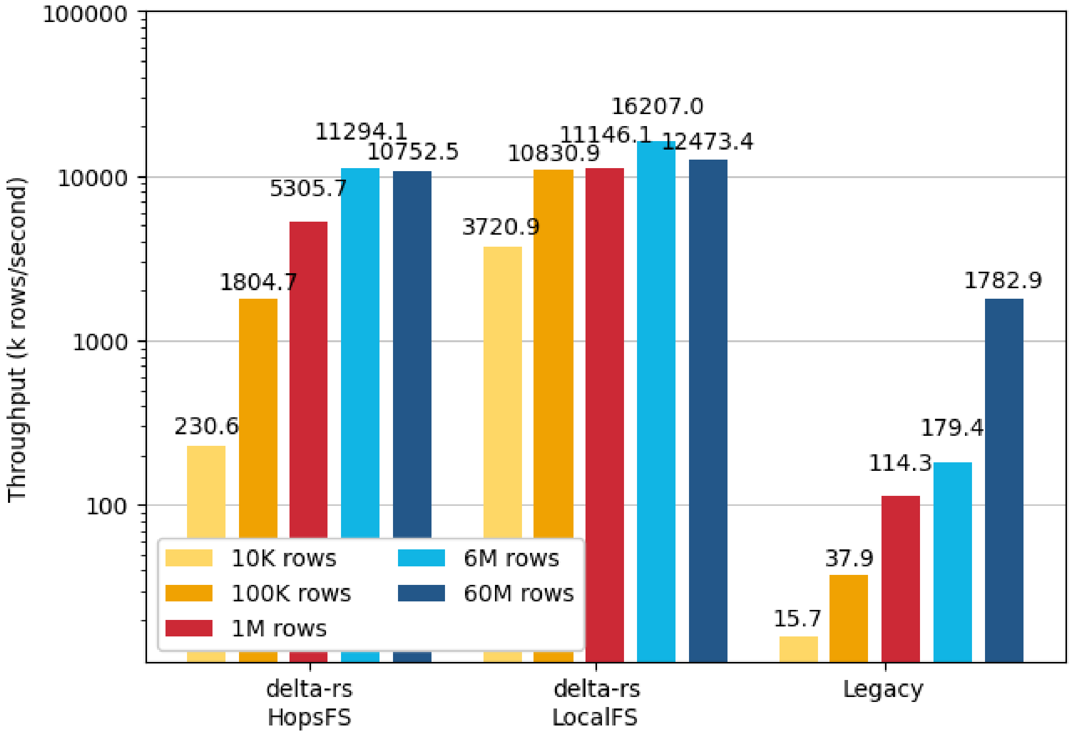


Figure C.7: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with four CPU cores.

Table C.8: Read experiment results expressed as throughput. The experiment was performed with eight CPU cores.

Pipeline	Number of rows	Throughput (k rows/second)	Throughput (k rows/second) 95% Confidence Interval low	high
delta-rs HopsFS	10K	230.925 18	195.345 44	257.360 67
	100K	1832.056 83	1765.756 84	1891.544 56
	1M	5750.343 66	5615.170 71	5885.017 95
	6M	12 065.182 02	11 760.178 93	12 331.709 77
	60M	20 391.779 56	19 584.753 63	21 011.721 36
delta-rs LocalFS	10K	3390.322 42	3256.074 86	3510.692 31
	100K	10 545.410 87	9465.275 36	11 463.060 73
	1M	23 212.466 79	20 701.230 33	25 414.131 22
	6M	34 191.396 37	33 185.181 29	35 275.254 56
	60M	26 252.420 19	26 131.518 36	26 373.488 80
Legacy	10K	15.939 01	15.807 91	16.065 39
	100K	37.563 30	37.422 50	37.691 86
	1M	119.795 20	116.151 77	122.929 95
	6M	179.489 42	177.780 54	180.974 90
	60M	1810.314 36	1795.708 67	1824.648 83

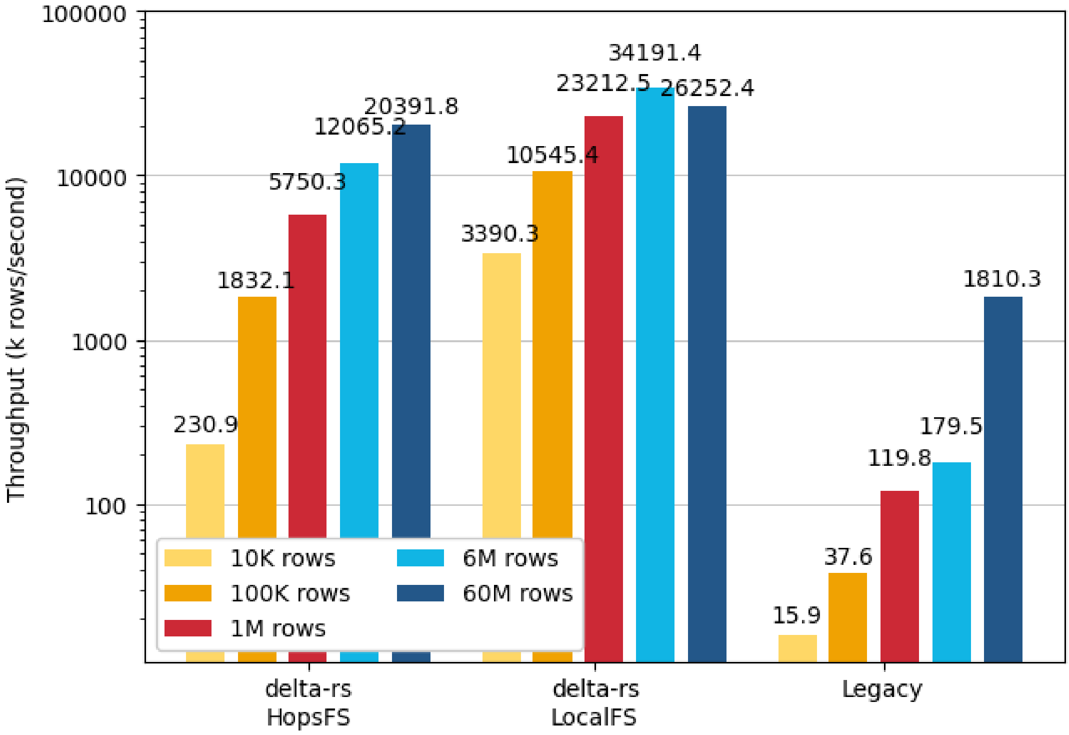
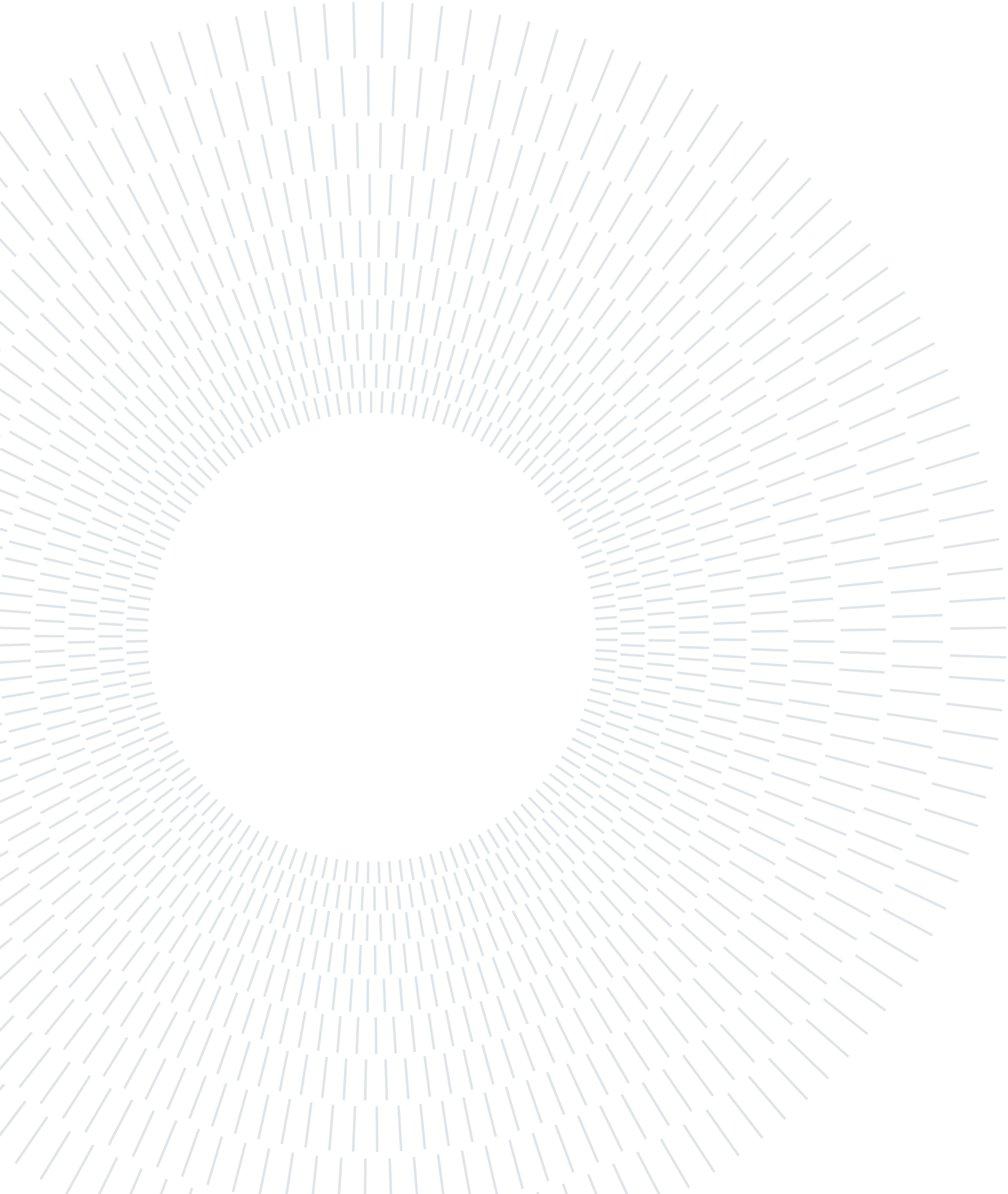


Figure C.8: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with eight CPU cores.



D | Legacy pipeline write latency breakdown results

This appendix reports all graphs and tables related to all write latency breakdown of the upload and materialization steps in the legacy pipeline.

Table D.1: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with one CPU core.

Step	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval	
			low	high
upload	10K	2.4865	2.3896	2.6261
materialize		47.7262	47.0445	48.4031
upload	100K	3.6684	3.6310	3.7098
materialize		55.9005	55.2494	56.5541
upload	1M	22.5934	22.4496	22.7349
materialize		89.5754	88.8286	90.3049
upload	6M	244.6123	244.0368	245.1905
materialize		267.2490	266.4287	268.1549
upload	60M	2437.7840	2422.8704	2453.6746
materialize		278.0504	276.2340	280.0921

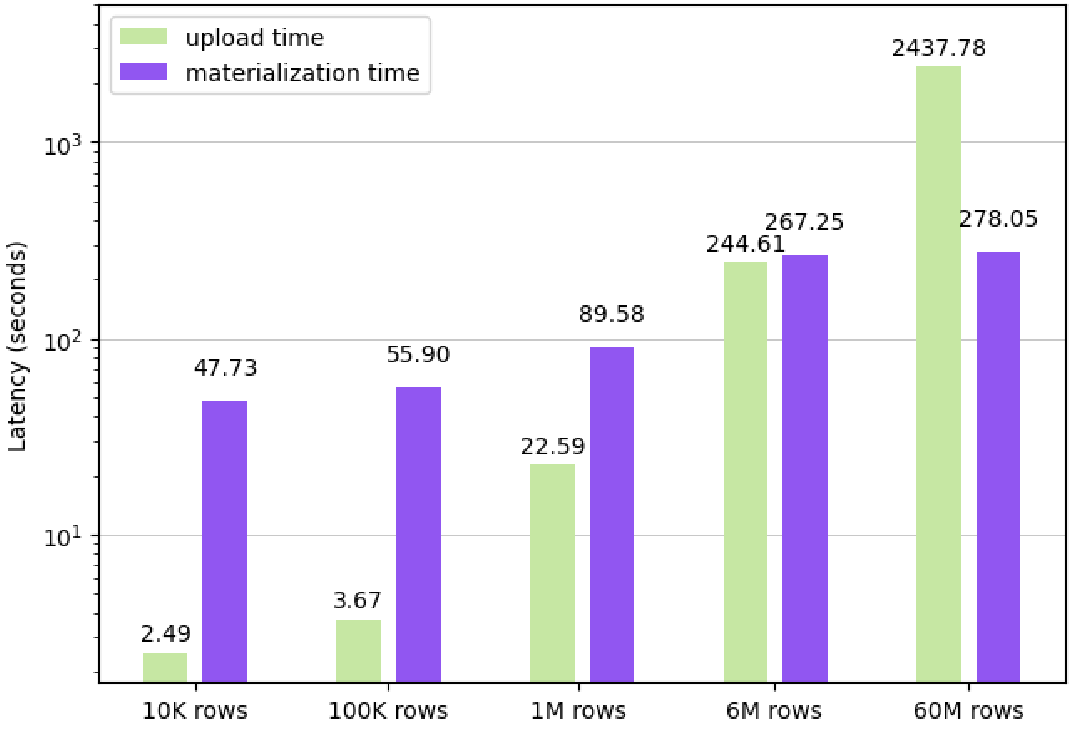


Figure D.1: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with one CPU core.

Table D.2: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with two CPU cores.

Step	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
upload	10K	2.3873	2.3276	2.4466
materialize		48.3305	47.7020	48.9923
upload	100K	3.4348	3.4008	3.4671
materialize		56.3367	55.5626	57.1129
upload	1M	18.6349	18.5673	18.7104
materialize		89.9267	89.4012	90.4514
upload	6M	205.8854	205.2177	206.4984
materialize		267.5079	266.6853	268.3512
upload	60M	2064.1357	2057.6396	2070.4450
materialize		276.9608	275.7156	278.2849

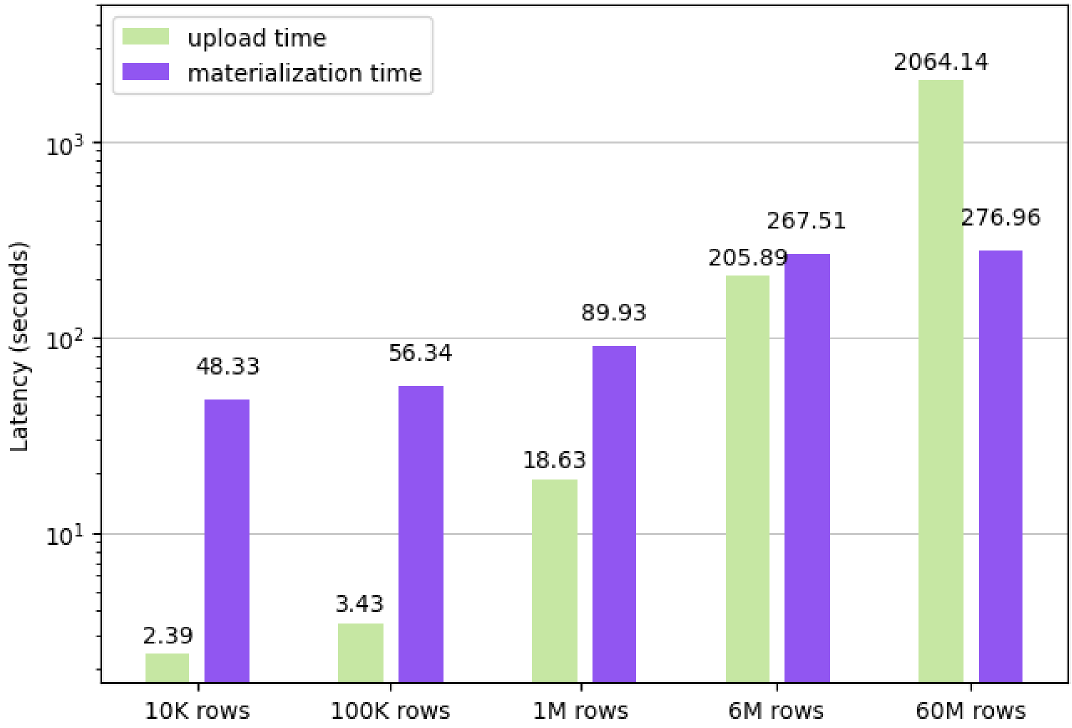


Figure D.2: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with two CPU cores.

Table D.3: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with four CPU cores.

Step	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
upload	10K	2.3846	2.3299	2.4335
materialize		48.9061	48.2436	49.5470
upload	100K	3.4650	3.4245	3.5071
materialize		56.0524	55.3682	56.6822
upload	1M	19.2296	19.1455	19.3161
materialize		89.5864	89.0209	90.1313
upload	6M	211.6758	211.0694	212.2839
materialize		270.3233	269.5967	270.9895
upload	60M	2068.5260	2060.3358	2077.2837
materialize		277.6001	276.0065	279.1456

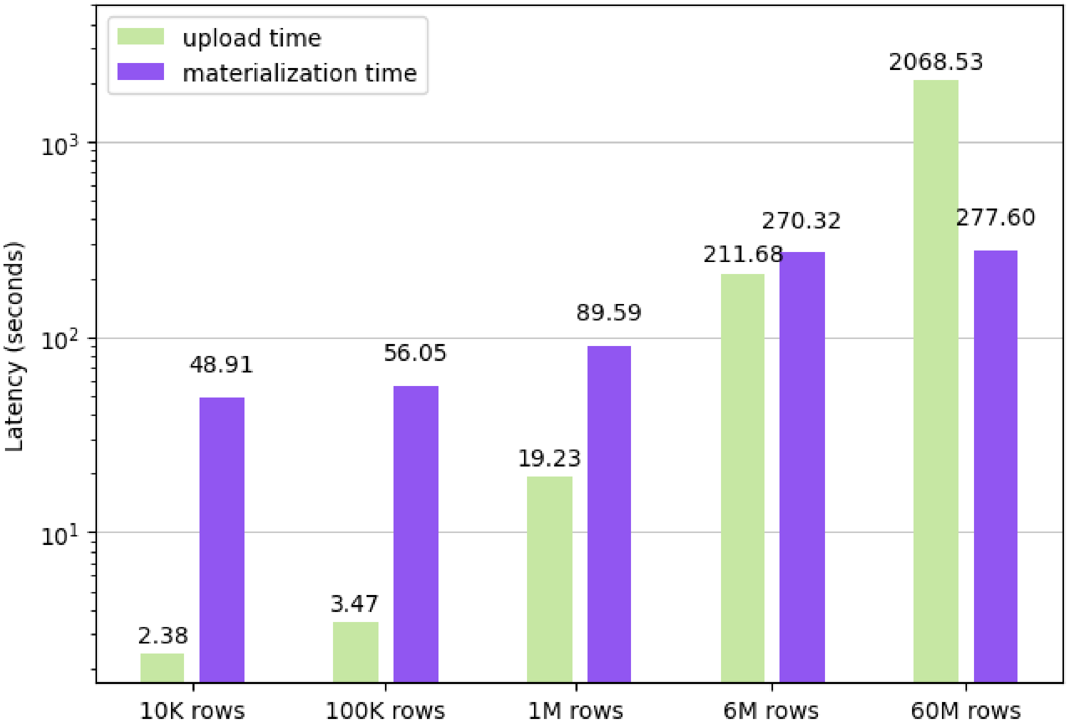


Figure D.3: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with four CPU cores.

Table D.4: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with eight CPU cores.

Step	Number of rows	Latency (seconds)	Latency (seconds) 95% Confidence Interval low	high
upload	10K	2.3815	2.3304	2.4358
materialize		48.8485	48.1979	49.4467
upload	100K	3.4392	3.4081	3.4700
materialize		56.8428	56.3177	57.3685
upload	1M	18.8642	18.7808	18.9532
materialize		90.5153	90.0306	90.9718
upload	6M	207.6646	207.1606	208.2090
materialize		268.2752	267.3569	269.2456
upload	60M	2049.1371	2043.5991	2055.4782
materialize		275.7636	274.2773	274.2773

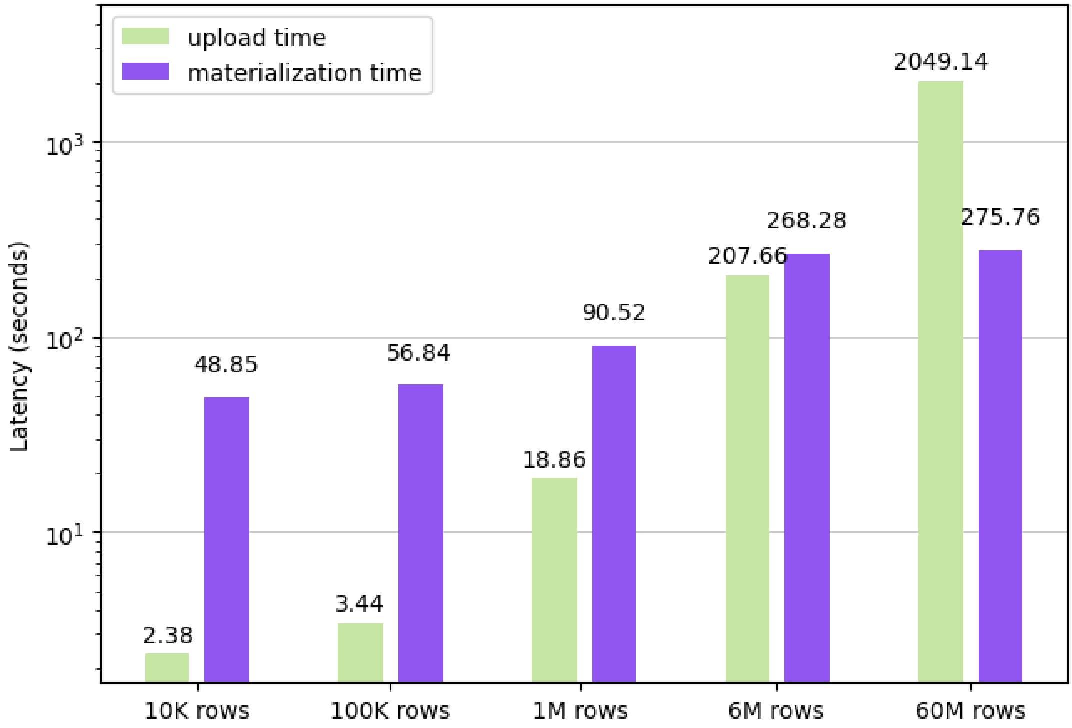
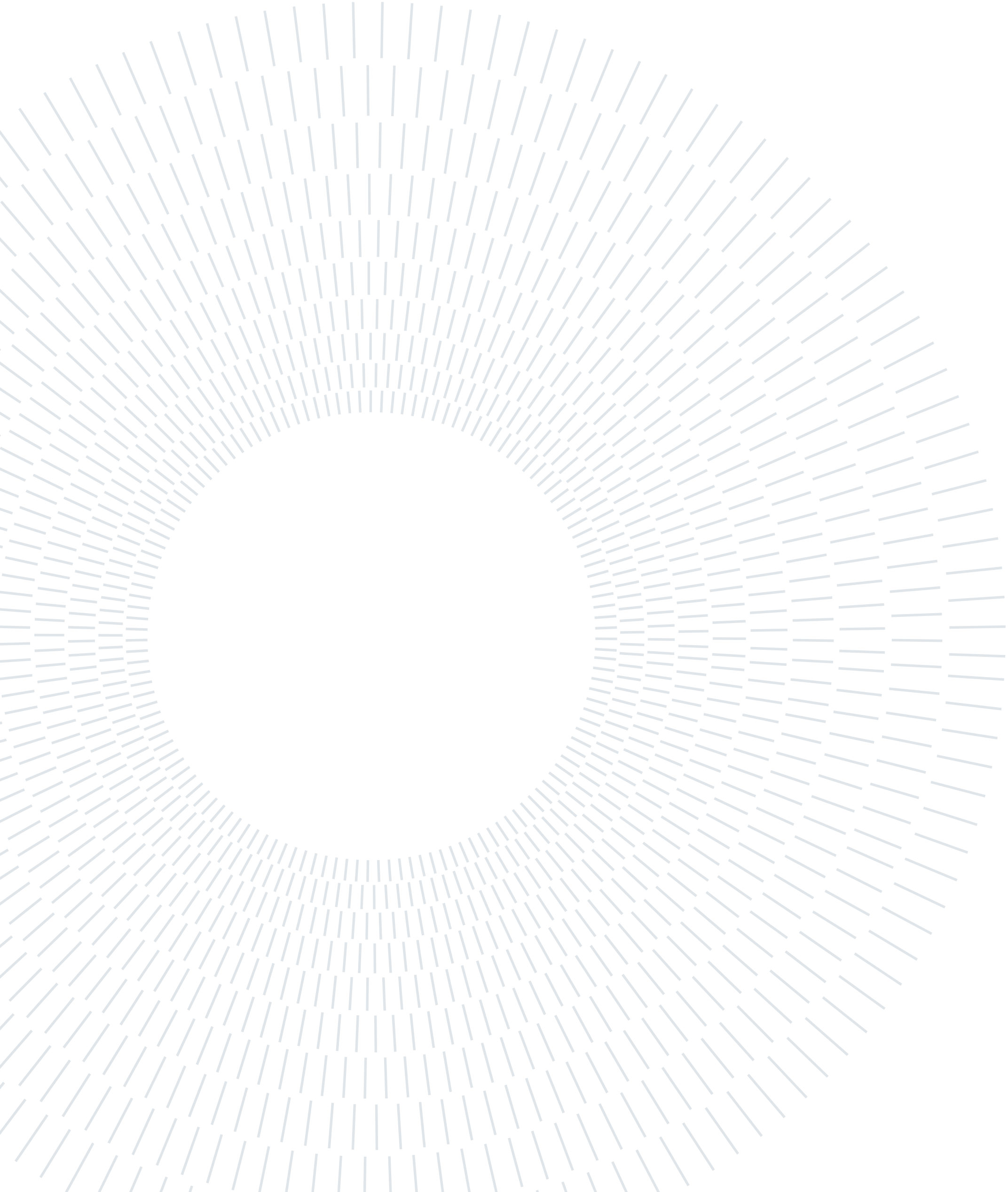
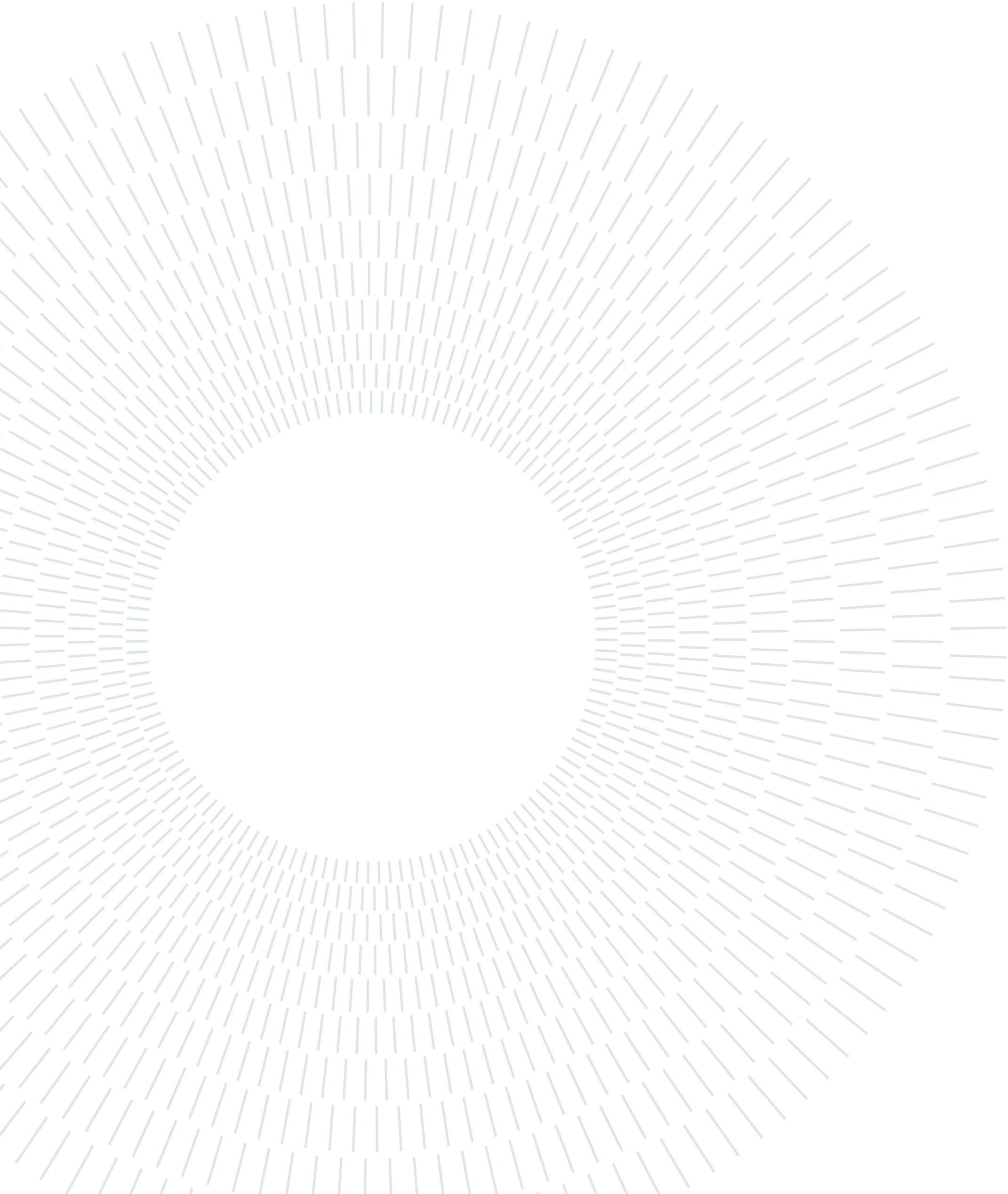


Figure D.4: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with eight CPU cores.



List of Figures

List of Tables



Acknowledgements

> “ Persistence and resilience only come from having been given the chance to work through difficult problems „
>
> – Gever Tulley

The work that brought me to complete this master’s thesis made me grow incredibly, laying a foundation of the person I am today. This would not have been possible without many individuals, whom I would like to thank in this section.

I would like to thank Jim Dowling for giving me the opportunity to work on this project and Salman Niazi for helping me make it happen. Many thanks also to all the Hopsworks AB employees who welcomed me from day one.

I would like to thank my KTH examiner, Vladimir Vlassov, and my two supervisors, Fabian Schmidt and Sina Sheikholeslami, for their many corrections to my numerous drafts.

I would like to thank my supervisor in Politecnico di Milano, Davide Martinenghi, for collaboration and support in this cross-university project.

I would like to thank my EIT Digital project coordinator, Federico Schiepatti, who guided me in navigating a complex cross-university bureaucracy from my application to my graduation.

I would like to thank all my friends who made me feel part of a larger family in the last years. Especially my Milan group, to which I will never be tired of returning: Sebastiano, Virginia, Luca, Alfonso, Giacomo, and Andrea. A special mention goes to Sebastiano, who, in the last year, was always by my side, a fantastic partner in an ever-changing world. Even if we have parted ways for now, I would still love to visit you in any city you might be in.

I would like to thank my parents and my brothers. In my most challenging moments, they helped me get back on track to complete this project while supporting me all the way. You enabled me to perform my best now and in the future.

Finally, I would like to thank my loved one, Elena, who was by my side well before I started my academic path. You were my space of serenity, even in the darkest hour. You brought me joy when I felt I couldn't feel any. I look forward to our present and our future together.