Degree Project in Computer Science

Second cycle, 30 credits

Reducing read and write latency in an Iceberg-backed offline feature store

Integrating HopsFS and PyIceberg Python library to reduce read and write latency on the Iceberg-backed Hopsworks offline feature store, with a comparative analysis of alternative solutions

SEBASTIANO MENEGHIN

Reducing read and write latency in an Iceberg-backed offline feature store

Integrating HopsFS and PyIceberg Python library to reduce read and write latency on the Iceberg-backed Hopsworks offline feature store, with a comparative analysis of alternative solutions

SEBASTIANO MENEGHIN

Master's Programme, ICT Innovation, 120 credits
Date: February 18, 2025

Supervisors: Sina Sheikholeslami, Fabian Schmidt, Davit Bzhalava
Examiner: Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: Minska läs- och skrivlatens i en Iceberg-stödd offlinebutik för funktioner
Swedish subtitle: Integrering av HopsFS och PyIceberg Python-bibliotek för att minska läs- och skrivfördröjningen i den Iceberg-stödda offline-lagringen Hopsworks, med en jämförande analys av alternativa lösningar.

Abstract

The growing need for efficient data management in Machine Learning (ML) workflows has led to the widespread adoption of feature stores, centralized data platforms that supports feature engineering, model training and prediction inference. The Hopsworks' feature store has demonstrated outperformance compared to its alternatives, leveraging Apache Hudi and Spark for offline data storage, but suffers from high write and read latency, even for small quantities of data (1GB or less). This thesis explores the potential of Apache Iceberg as an alternative table format, integrating it with HopsFS (Hopsworks HDFS distribution) and PyIceberg Python library to reduce latencies.

The research begin with an evaluation of potential system integration alternatives, documenting the advantages and limitations of each approach. Then, a PyIceberg-based architecture is implemented and evaluated, benchmarking it against the existing Spark-based solution and an alternative Delta Lake implementation (delta-rs). Extensive experiments were conducted across varying table sizes and CPU configurations to assess write and read performance. Results show that PyIceberg significantly reduces write latency – from 40 to 140 times lower than the legacy system – and read latency – from 55% to 60 times lower than the legacy system. Compared to delta-rs, PyIceberg demonstrates reduced write latency for large tables – up to 7 times lower – and equal read latency, but exhibits lower scaling benefits with additional CPU cores – 20% less than delta-rs.

These findings confirm that alternatives to Spark-based pipelines in small-scale scenarios are possible and are worth of further investigations, and the system implemented will be included in the Hopsworks feature store. Furthermore, this thesis work and results finally provides a baseline for future work about additional open table formats, alternative languages to mitigate Python's performance overhead, and strategies to improve resource utilization in data management platforms.

Keywords

Machine Learning, Feature Store, Spark, Apache Iceberg, Python, Read/Write Latency, Open Table Formats

Sammanfattning

Det växande behovet av effektiv datahantering i arbetsflöden för maskininlärning (ML) har lett till en utbredd användning av feature stores – centraliserade dataplattformar som stöder feature engineering, modellträning och inferens. Hopsworks feature store har visat bättre prestanda jämfört med sina alternativ och använder Apache Hudi och Spark för offline-datalagring. Dock lider systemet av hög skriv- och läslatens, även för små datamängder (1 GB eller mindre). Denna avhandling undersöker potentialen hos Apache Iceberg som ett alternativt tabellformat och integrerar det med HopsFS (Hopsworks HDFS-distribution) samt PyIceberg Python-biblioteket för att minska latensen.

Forskningen inleds med en utvärdering av potentiella systemintegrationsalternativ, där fördelar och begränsningar med varje metod dokumenteras. Därefter implementeras och utvärderas en PyIceberg-baserad arkitektur, vilken jämförs med den befintliga Spark-baserade lösningen samt en alternativ Delta Lake-implementering (delta-rs). Omfattande experiment genomfördes med varierande tabellstorlekar och CPU-konfigurationer för att bedöma skriv- och läsprestanda. Resultaten visar att PyIceberg avsevärt minskar skrivfördröjningen – från 40 till 140 gånger lägre än det äldre systemet – och läsfördröjningen – från 55% till 60 gånger lägre än det äldre systemet. Jämfört med delta-rs uppvisar PyIceberg minskad skrivfördröjning för stora tabeller – upp till sju gånger lägre – och liknande läsfördröjning, men har sämre skalningsfördelar vid ökning av CPU-kärnor (20% mindre än delta-rs).

Dessa resultat bekräftar att alternativ till Spark-baserade pipelines i småskaliga scenarier är möjliga och värda ytterligare undersökningar. Det implementerade systemet kommer att integreras i Hopsworks feature store. Dessutom utgör denna avhandling en baslinje för framtida forskning kring ytterligare öppna tabellformat, alternativa programmeringsspråk för att hantera Pythons prestandabegränsningar samt strategier för att förbättra resursutnyttjandet i datahanteringsplattformar.

Nyckelord

Maskininlärning, Feature Store, Spark, Apache Iceberg, Python, Läs- och skrivlatens, Open Table Formats

Sommario

La crescente necessità di piattaforme per una efficience gestione dei dati per applicazioni di *Machine Learning (ML)* ha portato a un'ampia diffusione dei *feature store*, piattaforme dati centralizzate che supportano *feature engineering*, addestramento di modelli e inferenza. Il *feature store* di Hopsworks ha dimostrato prestazioni superiori rispetto alle sue alternative, sfruttando Apache Hudi e Spark per il *suo offline feature store*. Tuttavia, questo sisteme soffre di un'elevata latenza di scrittura e lettura, anche per piccole quantità di dati (1GB o inferiori). Questa tesi esplora l'uso di Apache Iceberg come *table format* alternativo, integrandolo con HopsFS (distribuzione Hopsworks di HDFS) e la libreria Python PyIceberg per ridurre tali latenze.

Questa ricerca inizia con un'analisi delle possibili strategie di integrazione, documentando i vantaggi e i limiti di ciascun approccio. Successivamente, viene implementata e valutata un'architettura basata su PyIceberg, confrontata con la soluzione esistente basata su Spark e con un'alternative basata su Delta Lake (delta-rs). Esperimenti approfonditi sono stati condotti su tabelle di diverse dimensioni e diverse configurazioni CPU per misurare le prestazioni di scrittura e lettura. I risultati mostrano che PyIceberg riduce significativamente la latenza di scrittura – da 40 a 140 volte inferiore rispetto al sistema legacy – e la latenza di lettura – dal 55% a 60 volte inferiore. Rispetto a delta-rs, PyIceberg offre una latenza di scrittura inferiore per le tabelle più grandi – fino a 7 volte minore – e prestazioni di lettura equivalenti, ma mostra minori vantaggi di scalabilità con l'aumento dei CPU cores (20% ridotti rispetto a delta-rs).

Questi risultati confermano che alternative ad architetture basate su Spark, per gestire dati su piccola scala, esistono e sono più efficienti, e dunque il sistema sviluppato verrà integrato nel *feature store* di Hopsworks. Inoltre, questa tesi fornisce una base per futuri studi su nuovi *open table format* e strategie da adottare per ottimizzare l'uso delle risorse nelle piattaforme di gestione dei dati.

Parole Chiave

Machine Learning, Feature Store, Apache Iceberg, PyIceberg, Latenza in Lettura/Scrittura, Open Table Formats.

Acknowledgments

> "The only true voyage of discovery would be not to visit strange lands but to possess other eyes.,, – Marcel Proust

The completion of this master thesis was only possible thanks to the invaluable support of many, to whom I am deeply grateful.

Many thanks to Jim Dowling, how offered me the opportunity to work on this project and Davit Bzhalava, who supported me along the way. I also had the great pleasure of collaborating with the employees of Hopsworks AB, who generously shared their patience and expertise.

Many thanks to my KTH examiner, Vladimir Vlassov, and my supervisors, Fabian Schmidt and Sina Sheikholeslami, for their guidance in conducting thorough research and for their invaluable feedback.

Heartfelt thanks to my parents and relatives, who have supported me, who have inspired me, who have believed in me, giving me a blank canvas with colors they had never used before. Your wisdom, guidance, and the values you instilled in me have shaped who I am today – and the person I strive to become in the future.

Lastly, I cannot begin to express my gratitude to the friends who have stood by my side through adventures and challenges. Michele, Daniele, and Davide, for countless years, you offered me advice and challenged my ideas, guiding me towards a clearer understanding of myself. Alfonso, Giacomo, Luca, and Virginia, you made feel like home, at every step of this journey, no matter where we were. Giovanni, you have filled my mind with 'whys' and 'hows' – questions that helped me grow at every answer, though some still await resolution. Giulia, and all the incredible people from MESA, you have shown me how essential it is to have a cause to believe in and care about. May the future bring us even more time to learn, discover, and laugh – together.

Stockholm, February 2025
Sebastiano Meneghin

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

This document is incomplete. The external file associated with the glossary 'acronym' (which should be called `thesis.acr`) hasn't been created.

Check the contents of the file `thesis.acn`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "thesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "thesis"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.

Chapter 1

Introduction

Data lakehouses' adoption reached in 2024 a critical mass, with more than 50% organizations running their analytics on these platform, and projection up to 67% within 2027 [**?**]. A data lakehouse is a modern data architecture that creates a single platform by combining the key benefits of data lakes (large repositories of raw data in its original form) and data warehouses (organized sets of structured data). Thus, data lakehouse is a cost-effective architecture [**?**] that bridges those architectures by providing the scalability of the data lakes together with analytical computations and Atomicity, Consistency, Isolation and Durability (ACID) properties of data warehouses [**?**]. This surge is strongly related to the acceleration of AI adoption [**?**], which put pressure on storage solutions and data processing capabilities [**?**]. In support to this, data lakehouse systems include data partitioning, reducing the query computational needs, support schemas evolution and provide "time travel" capabilities, enabling data versioning over time [**?**].

From the first appearance of data lakehouses [**?**], three primary applications arose [**?**]:

1. **Apache Hudi**: introduced by Uber in 2017, then supported by Alibaba, Bytedance, Uber and Tencent [**?**].

2. **Apache Iceberg**: open sourced by Netflix * in 2018, now used by Airbnb, Apple, Expedia, LenkedIn, Lyft [**?**].

3. **Delta Lake**: open sourced by Databricks in 2019 [**?**], supported now by Databricks and Microsoft.

*Netflix first implementation at `https://github.com/Netflix/iceberg`

Apache Iceberg (from now on, Iceberg) was originally developed as an alternative to Apache Hive (from now on, Hive), a data warehouse system that enables querying of large datasets stored in Hadoop using a Structured Query Language (SQL)-like language. Iceberg is designed to manage bigger datasets with frequent updates and deletions, and to support schema evolution, going thus beyond Hive's capabilities [**?**]. This is possible thanks to the Iceberg metadata management layer, which enables data warehouse-like capabilities via cloud object storage. In 2024, Iceberg has consolidated its position in 2024 [**?**], and has been recently included in arguably all the major big data vendors' solutions [**?**].

Apache Hudi (from now on, Hudi), Iceberg and Delta Lake were initially designed to fit a single engine, respectively Hive, Trino and Apache Spark (from now on, Spark) [**?**]. These three solutions have been improved to better scalability and flexibility needs, each integrating new engners and features [**?**]. Despite being beneficial, this evolution highlights a potential limitation: systems developed in differnt ways performe differently, and depending on the use case, even widespread solutions might not always be the optimal solution. For example, integrating the system with Spark, a query and compute engine [**?**], is a good solution when processing massive datasets (1 TB+) on the cloud. However, it is yet unclear how good this solution could perform when processing smaller datasets (1 GB to 100 GB) [**?**].

Such Spark limitations of Spark are also brought to light by technologies such as the Python library Polars [**?**] and the Data Base Management System (DBMS) DuckDB [**?**], that combined to process data locally, in small quantities, showed much better performance than a Spark Cluster. This solution further avoids the high expenses and computation time lying behind Spark usage in the same scenario. Thus, similar solutions employed in small-scale scanarios (from 1GB to 100GB) could greatly increase performance and reduce costs [**?**] [**?**].

Remaining in the data science domain, Python has perhaps become the de facto standard programming language. Python is arguably the most popular general-purpose programming language [**?**, **?**, **?**], and the go-to option for Machine Learning (ML) and Artificial Intelligence (AI) applications [**?**]. Among the most commonly used libraries by ML application developers, we find NumPy and Pandas [**?**], while PyTorch and TensorFlow are the first two for Neural Network (NN) libraries. Accessing data lakehouse solutions via a Python client would be the favorite option for most of ML and AI developers, so they do not have to resort to alternatives, such as Spark and its Application Programming Interfaces (APIs) (e.g., PySpark).

Therefore, supporting Iceberg with a native Python client would be directly beneficial for Hopsworks AB, the host company of this master thesis. Hopsworks AB develops a homonymous AI Lakehouse for ML. This software includes a multi-user data platform, called feature store, that enables storage and access of reusable features *. It also implements resource-efficient and point-in-time joins between datasets, adding historical retrieval features to saved data [**?**].

This project seeks to reduce latency (in seconds), thus increasing throughput (in rows/second), of reading and writing data on Iceberg tables, as offline feature store in Hopsworks. At time of writing, the writing pipeline is Spark-based and uses Hudi as Open Table Format (OTF), while this thesis seeks to employ Iceberg without Spark-dependencies. If the hypothesis that a faster non-Spark alternative is possible is proved, Hopsworks AB will consider this alternative as an integration or replacement of their current open-source feature store implementation. This could considerably improve the experience of Python users working with small-scale datasets (1 GB to 100 GB). More broadly, this thesis will discuss the feasibility of Spark alternatives in small-scale use scenarios [**?**].

Revise the following part of the introduction once drafted implementation and conclusion chapters

The primary contributions of this work are:

- The evaluation of the possible technologies for the integrations between Iceberg and Hopsworks feature store, and the selection of the best one according to defined requirements, described in Section 3.1.2. The solution most fitting the requirements consists of SQLCatalog as catalog, DuckDB as query engine, and PyArrowFileIO as FileIO.

- The results of the experiments conducted on the newly integrated PyIceberg system and the Hudi legacy system, showing differences in latency and throughput for read and write operations. These experiments were conducted fifty times, with different dataset sizes and Central Processing Unit (CPU) settings. The new system, accessing Iceberg from a Python client, reduced write latency by 40 to 140 times, and read latency by 15 to 60 times in the smaller tables, compared to what was experienced using the old system.

- The results of the comparison between the newly integrated PyIceberg system and the recently implemented delta system [**?**], on which the

*Definition from the company's website at https://www.hopsworks.ai/

same experiments above were conducted in a parallel thesis work. The PyIceberg system has write latency from 5 to 7 times lower compared to what was experienced using the delta-rs system, and comparable read latency. The delta-rs system showed better support for multiple CPU cores setting.

These findings provide a significant contribution to the data management industry, supporting existing studies on the constraints of utilising Spark with small-scale volumes of data. Furthermore, the reproducibility of the experiments conducted adds great value to this work. Thanks to the well-defined environment and the available code, these experiments can be used as starting or final point for further exploration and testing in this field.

1.1 Background

There are three crucial components for a comprehensive understanding of this work: the evolution of data infrastructure to data lakehouses, the role of Spark as a data management tool, and the rise of Python as the most used programming language in the data science field.

The term "data lakehouse" was used by Databricks in 2020 [**?**] to characterize a new architectural standard developing across the industry. This novel paradigm integrates the data lake's capacity to store and manage unstructured data with the ACID features characteristic of data warehouses. Data warehouses became the standard in the 1990s and early 2000s [**?**], facilitating firms in deriving business intelligence insights from data coming from various structured data sources. The architectural issues of this technology became evident at the end of the 2010s, with the rise of Big Data, characterized by increasing volumes, variety, and velocity of data, including significant amounts of unstructured information [**?**]. Data lakes were thus introduced, as a more flexible and scalable solution, serving as a central repository for all data. They enable the development of more intricate built-upon architectures, including data warehouses for Business Intelligence (BI) and ML pipelines. While being more appropriate for unstructured data, this architecture entails several complications and expenses associated with the need for duplicated data (data lake and data warehouse) and complex Extract Load Transform (ELT)/Extract Transform Load (ETL) pipelines. Data lakehouse solutions addressed the challenges of data lakes by combining the best of both worlds: the flexibility and scalability of data lakes with the data governance, reliability, and performance of data warehouses. This is achieved

by integrating data management and performance capabilities directly into open data formats like Parquet [?]. Three pivotal technologies facilitated this paradigm:

- A metadata layer, providing data lineage and facilitating efficient data discovery and access.
- An optimized query engine, leveraging techniques like Random Access Memory (RAM)/Solid State Drive (SSD) caching and advanced query optimization.
- A user-friendly API, simplifying data access and integration with ML and AI applications.

Uber first open-sourced this architectural design with Hudi in 2017 [?], followed by Netflix in 2018, with Iceberg [?], and ultimately by Databricks, with Delta Lake in 2020 [?].

Spark is a distributed computing platform designed to facilitate large-scale, data-intensive applications [?]. Developed as an improvement over Hadoop MapReduce (from now on just MapReduce), Spark addresses its limitations, such as high latency due to disk I/O and limited support for iterative algorithms. Spark achieves significantly higher performance by leveraging in-memory computing, eliminating the need for frequent disk access to store intermediate results. This, coupled with its DAG execution model, Resilient Distributed Datasets (RDDs), and support for multiple processing models (batch, stream, ML, graph), has established Spark as the de facto standard for many data-intensive workloads. While Spark offers a powerful and versatile platform, it may not always be the most suitable choice for all scenarios. For instance, Apache Flink [?], specifically designed for stream and real-time processing, excels in low-latency applications where Spark may exhibit higher latency. Similarly, for small-scale datasets (from 1 GB to 100 GB), the overhead associated with launching and managing a Spark cluster can be substantial. In such cases, in-memory DBMS like DuckDB [?] and Polars [?] offer compelling alternatives. These solutions, optimized for smaller datasets, provide high-performance On-Line Analytical Processing (OLAP) capabilities and DataFrame operations within an embedded environment, delivering significantly faster results compared to initiating a Spark cluster for equivalent tasks. This project investigates the feasibility of employing alternatives to Spark for efficient data processing on small-scale datasets, exploring their potential advantages in terms of performance.

Python reigns supreme as the programming language of choice for data scientists [**?**]. Its user-friendliness, high-level abstraction, and emphasis on clear code expression initially attracted a large and enthusiastic community [**?**]. This supportive community, in turn, fueled the development of a vast ecosystem of libraries and APIs specifically designed for data science tasks. Over three decades, Python has become the de facto standard in this domain, with popular libraries like NumPy, Pandas, PyTorch, and TensorFlow, forming the backbone of countless data science projects. Python is regarded as the most popular programming language based on the volume of search results for the query (+*"<language> programming"*) across 25 distinct search engines *. The TIOBE index [**?**] and 2024 GitHub report [**?**] underscore the current trends, clearly demonstrating Python's ascendance, also evident from the following milestones:

- Python surpasses C in 2021.
- Python surpasses Java in 2022.
- Python surpasses JavaScript in 2024.

These results emphasize the need for offering Python APIs, especially for programmers and data scientists, to augment interaction and broaden the framework's possibilities.

1.2 Problem

The Hopsworks feature store [**?**] first used Hudi for its offline feature store, since in 2017 it was the first data lakehouse to be open-sourced. Hopsworks AB thus actively implemented new technologies for their software, to cater growing customer's needs. In the legacy system, Spark serves as the query engine, executing queries (read, write, or delete) on the offline feature store. This system demonstrated that a write operation, even on a tiny dataset (1 GB or less), takes one or more minutes to finalise.

This adversely affects Hopsworks' standard use case, which operates between testing scanarios on tiny data volumes (from 1 GB to 10 GB) and production scenario on higher volumes, despite still with relative small volumes (from 10 GB to 100 GB).

*Evaluation methodology defined at https://www.tiobe.com/tiobe-index/programminglanguages_definition/

The fundamental hypothesis of this study is that the prolonged transaction time is a problem determined by Spark. This has prompted Hopsworks to implement Spark alternatives [**?**] for data ingestion in their Hudi system, and to search for alternatives to the Hudi-Spark tandem [**?**]. Iceberg provides support for several alternative to Spark [**?**], and Iceberg tables can be accessed directly from Python, via the PyIceberg library *. However, the Hopsworks feature store has not been yet integrated with Iceberg, and its underlying file system, Hopsworks' HDFS distribution (HopsFS) [**?**], and several implementations are possible [**?**].

1.2.1 Research Questions

This research project aims to assess and compare the performance of the legacy system, relying on Hudi and Spark, against newly developed systems providing an alternative to Spark, and a comprehensive comparison of the latters. Those systems features the PyIceberg library *, which operates on Iceberg tables, and the Rust delta-rs library †, which operates on Delta Lake tables, in both cases hosted on HopsFS. To do this, an implementation for the interaction of Iceberg with the Hopsworks feature store must be designed. Accordingly, this study tackles the two following Research Questions (RQs):

RQ1: What are the differences in read and write latency and throughput on the Hopsworks offline feature store, between the existing legacy system and the PyIceberg alternative?

RQ2: What are the differences in read and write latency and throughput on the Hopsworks offline feature store, between the new PyIceberg system and the delta-rs alternative?

It is fundamental to notice that, while the measured performance is expected to be comparable if PyIceberg and/or delta-rs are included into the Hopsworks client in the future, they are officially not functioning via the offline feature store API but only using the same file system, HopsFS.

1.3 Purpose

This thesis project aims to diminish read and write latency (seconds) and thus enhance data throughput (rows/second) for operations on the Hopsworks

*PyIceberg repository accessible at https://github.com/apache/iceberg-python

†Project repository available at https://github.com/delta-io/delta-rs

offline feature store. This research will evaluate the performance of the existing legacy pipeline, which utilises Spark for writing, against PyIceberg pipeline on a small-scale dataset by assessing variations in latency and throughput during read and write operations. If the PyIceberg alternative is shown to be a more efficient option, Hopsworks AB will contemplate including this pipeline into their application. The same assessment will be conducted, within the same data scale domain, between PyIceberg and delta-rs pipelines.

The overall ramifications of this thesis are far larger, given Spark's prominence within the open-source community, which has had almost 3000 contributors during its existence [**?**]. Opting using PyIceberg or delta-rs instead of Spark provides developers with a wider array of options for managing small-scale data (1 GB - 100 GB). This thesis work also provides an industrial use case, that can be used to understand which alternative to Spark could fit the best a specific developer need.

1.4 Goals

This project aims to reduce latency and hence enhance data throughput for reading and writing on Iceberg tables inside HopsFS. The achievement of this objective is tight to a specified list of Goals (Gs), here described. These are connected to the collection of RQs, delineating a distinct framework entailing each project milestone.

1. Gs aimed to answer RQ1:

   G1: Understand PyIceberg library tools, and identify which are needed to interact with HopsFS.

   G2: Implement interaction between PyIceberg and HopsFS.

   G3: Design the experiments to evaluate the performance difference between the legacy access to Apache Hudi and the PyIceberg library-based access to Apache Iceberg, on HopsFS.

   G4: Perform the designed experiments.

   G5: Visualize the experiments' results, focusing on an effective comparison of performances.

   G6: Examine and describe the findings in a dedicated Section of the thesis report.

2. Gs aimed to answer RQ2:

G7: Investigate related work on delta-rs library-based access to Delta Lake, on HopsFS.

G8: Visualize experiments' results of PyIceberg and delta-rs alternatives to the legacy system, focusing on an effective comparison of performances.

G9: Examine and describe the findings in a dedicated Section of the thesis report.

To reach these Gs, several Deliverables (Ds) will be created:

D1: Experiment results on the difference in performance between the legacy access to Apache Hudi and the PyIceberg library-based access to Apache Iceberg, on HopsFS This D is related to completing Gs3–5.

D2: Experiment results on the difference in performance between PyIceberg library-based access to Apache Iceberg and the delta-rs library-based access to Delta Lake, on HopsFS. This D is related to completing G7 and G8.

D3: This thesis report. It offers additional detail on the implementation, design choices, performance, and evaluation of the results. This D a comprehensive report of all thesis work, incorporating the analysis specified in G6 and G9.

1.5 Ethics and Sustainability

This research project focusses on software, namely the development of more efficient data-intensive processing pipelines applicable in ML and NN training. The Green Software Foundation * states that software may be "part of the climate problem or part of the climate solution." Green software is described as software that minimises its environmental effect by using fewer physical resources and less energy, while optimising energy use to employ lower-carbon sources [**?**]. In the realm of ML and NN training, minimising training duration–and therefore the read and write latencies associated to the dataset–has been proved to significantly decrease carbon emissions [**?**, **?**]. This thesis minime latency and thus enhance data throughput for reading and writing on Iceberg tables on HopsFS. This objective adheres to the essential principles of green software by reducing CPU utilisation time relative to the prior system.

*Foundation's website available at https://greensoftware.foundation/

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

Minimising CPU utilisation time lowers energy consumption, resulting in a reduced carbon impact. This as a concequence, enhance energy efficiency of data-intensive computer pipelines, which are extensively used in ML and NN training.

For the same reason, this project supports two Sustainable Development Goals (SDGs) *, namely Affordable and Clean Energy (Goal 7), and Industry Innovation and Infrastructure (Goal 9). In particular, it tackles objectives 7.3, "Double the improvement in energy efficiency", and 9.4, "Upgrade all industries and infrastructures for sustainability".

Furthermore, the experiments created to satisfy G2–3 has been carefully designed to minimize the number of trails necessary for statistical relevance, pursuing resource effiency. All the data used for this thesis are thoroughly explained, and all the scripts made available, to best address reproducibility of this project.

1.6 Research Methodology

This thesis is built from a few Industrial Needs (INs), provided by Hopsworks AB, and a few Project Assumptions (PAs), validated through a literature study. The Hopsworks's INs are:

IN1 : The Hopsworks offline feature store, supported by the legacy pipeline, exhibits high latency (over one minute) and low throughput in write operations for small-scale data (from 1 GB to 100 GB). These performances indicate the potential of employing Spark alternatives in a small-scale data scenario.

IN2 : Hopsworks actively looks to customer needs and software's integration

*SDGs website available at https://sdgs.un.org/

capabilities. Improving the read and write operations performance on their feature store and/or integrate their software with additional table formats, like Iceberg, is benefitted by all Hopsworks feature store users.

The PAs will be validated in Chapter 2. Those PAs are:

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. Thus high-performance Python libraries will typically be preferred over Java Virtual Machine (JVM) or other alternatives.

PA2 : Spark has been proved to perform worse than other options when processing data in small-scale scenarios (from 1 GB to 100 GB). Alternatives to the Spark-based system could strongly improve reading and writing operations on the Hopsworks feature store, in different ways.

This thesis work fulfill the INs following a system implementation and evaluation guided by its Gs. Initally, an integration between PyIceberg and HopsFSs will be implemented [**?**], followed by test to validate the approach. Then, an evaluation structure will be designed and used to compare the performances of the current legacy system, the newly integrated PyIceberg pipeline and the delta-rs pipeline developed in the related work. These experiments will involve datasets of varying sizes and evaluate critical performance metrics such as read and write latency (in seconds), and throughput (in rows/second). Those two metrics were chosen as they most affect the computational time of accessing OTFs, thus being fair comparators between pipelines.

1.7 Research Limitation

The project is conducted in collaboration with Hopsworks AB, and as such the implementation will focus on working with their feature store and related system, supported by HopsFS. The considerations drawn in this thesis provide an insight into Spark limitations, and on which tools perform better in different data scales. However, those results cannot be generalized for any system.

1.8 Structure of the thesis

Chapter 2 provides to readers the basic information needed to understand the layered data stack analyzed in this study. Additionally, it presents the

legacy and novel system architectures that will be used in the experiments. Chapter 3 delineates the methodologies for the system integration and the system evaluations. Chapter 4 illustrates the decisions made during the system selection phase and describes the design of the experiments. Chapter 5 presents the results of the experiments, focussing on the differences between the described pipelines, with various CPU settings. The chapter includes a discussion Section describing the major findings and implications of this thesis. Chapter 6 eventually summarises the contributions and discoveries of this thesis, highlights its limitations, and sets the discussion on prospective future research.

Chapter 2

Background and Related work

This project aims to improve performances of systems consisting of a layered data stack, able to handle large volumes of structured and unstructured data, at a high velocity, also defined as Big Data [**?**]. A generic data stack handles storing, management and retrieval of the data, offering access to those to the applications on top. However, there is no a single data stack for all cases, but different architectures used on different needs [**?**, **?**, **?**]. The following Sections of this report defines the data stack for this use case, which is is illustrated in Figure 2.1.

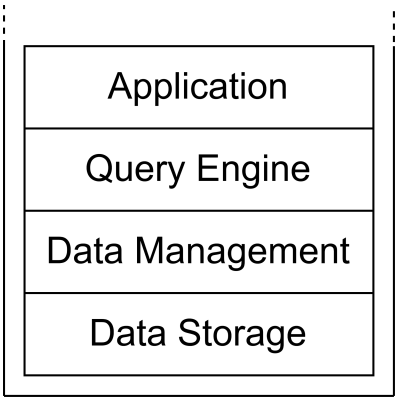

Figure 2.1: Data stack abstraction for this project.

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

1. **Data Storage**: handles the physical storage of data. This layer determines how data is physically stored, including aspects like centralization or distribution, on-premise or cloud deployment, and storage formats such as files, objects, or blocks.

2. **Data Management**: handles the organization, governance, and lifecycle of data. The data management layer may provide features such as ACID properties, data versioning, support for open data formats, and the ability to store and manage both structured and unstructured data effectively.

3. **Query Engine**: handles the execution of data queries. This layer is responsible for efficiently accessing, retrieving, and writing data based on user requests. Key features of a query engine may include caching mechanisms, highly scalable architectures, and support for diverse programming languages through APIs.

4. **Application**: a system that utilizes the capabilities of the underlying data stack to achieve specific objectives. In this project, the focus will be on the Hopsworks feature store software.

Following the data stack section, the legacy and new system architectures are explained in Section 2.5, showing how the technologies are reflected within the pipelines measured during this thesis' experiments.

2.1 Data storage

This Section describes first what data storage is and which typologies of storage exist. Thus, this project's data storage layer, HopsFS, is presented, starting from its evolution from Hadoop Distributed File System (HDFS), going into the details about its tools, and ending with possible alternatives, the cloud object storages.

2.1.1 Block storage vs. File storage vs. Object storage

Data can be stored and organized in physical storages, such as Hard Disk Drives (HDDs) or SSDs. The three main type of data storage are (1) Block storage, (2) File storage, and (3) Object storage briefly compared in Table 2.1. Each typology is describe in the following paragraphs, and their pros and cons are summarized in Table 2.2. This Subsection is a re-elaboration of three articles from major cloud providers (Amazon, Google, and IBM) [**?**, **?**, **?**] according to the author's understanding.

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

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

Block Storage

Block storage is a data storage method that divides data into discrete blocks of fixed size, each assigned a unique identifier. These blocks are stored independently on a storage system, such as a Storage Area Network (SAN) or within a cloud environment.

This decentralized approach offers several key advantages. Firstly, it enables high performance with fast read/write speeds and low latency, crucial for demanding applications like databases and virtual machine environments. Secondly, block storage provides direct, low-level access to storage volumes, similar to physical disks, granting users and applications granular control over data organization and management. This flexibility allows for a wide range of use cases, including powering virtual machine environments, supporting high-performance databases, and enabling efficient file sharing.

While offering significant benefits, block storage also presents certain limitations. It typically requires specialized hardware and infrastructure, potentially leading to higher costs compared to other storage options. Furthermore, while it offers a degree of scalability, expanding beyond certain limits can become complex and costly. Despite these considerations, block storage remains a vital technology for modern IT environments, enabling high performance, flexibility, and agility in data management.

File Storage

File storage is a hierarchical data organization method that stores data in files, which are organized into folders within a structure of directories and subdirectories. Files are characterized by extensions (e.g., ".txt", ".png", ".csv"), defining how the data is organized and accessed. This system simplifies locating and retrieving individual files when their exact paths are known, making it intuitive and user-friendly.

This structure is particularly beneficial for managing structured data and is widely used in Personal Computer (PC) and Network Attached Storage (NAS) devices. It enables centralized file sharing on Local Area Network (LAN) and supports common file-level protocols, ensuring compatibility across Windows and Linux systems. Storing data on a separate NAS device or in the cloud also enhances data protection and disaster recovery, with options to replicate data across multiple geographic locations for added security.

However, as the volume of files grows, scaling becomes challenging. Locating files in a large hierarchy can be time-consuming, and scaling often requires investing in additional or higher-capacity hardware. Cloud-based file storage services mitigate these challenges by offering scalable, off-site storage managed by service providers. These services eliminate hardware maintenance costs and provide flexible, subscription-based models that adapt to varying storage and performance needs.

File storage remains popular for applications requiring simplicity and centralized access, such as file sharing, personal storage, and cloud-based platforms like Dropbox and Google Drive. While other storage solutions may be better suited for managing massive datasets or unstructured data, file storage's accessibility, affordability, and ease of use ensure its ongoing relevance.

Object Storage

Object storage is a flat data storage method that organizes data into self-contained objects, each containing metadata that describes attributes like size, creation date, and unique identifiers. This metadata not only defines the data but also enables efficient querying and retrieval of large datasets. This makes object storage particularly well-suited for managing unstructured data, such as videos, images, and other media files that do not fit neatly into traditional hierarchical systems.

The flat structure of object storage eliminates complex hierarchies like folders and directories, simplifying organization and improving scalability. This structure allows object storage systems to replicate data across multiple regions, enhancing accessibility and fault tolerance in case of hardware failures. As a result, users benefit from faster data access in different parts of the world and robust disaster recovery options.

However, object storage has limitations. Objects are immutable, meaning they cannot be directly altered once created. Any changes require the creation of a new object. Additionally, object storage does not support transactional

operations, as it lacks mechanisms like file locking, making it unsuitable for applications requiring frequent updates or real-time data changes. It also has slower writing performance compared to file or block storage solutions.

Overall, object storage is an excellent choice for use cases requiring high scalability, such as social networks, video streaming platforms, and cloud-based services. Its flat structure and metadata-driven design are ideal for managing large, static datasets. However, other storage options are preferred when high performance is required for frequently changing files or when transactional consistency is critical.

Table 2.2: Data storage pros and cons comparison. Table inspired by major cloud providers articles [**?**, **?**, **?**].

Storage Typology	**Pros**	**Cons**
Block	High performance High reliability Easy updates	Lacks metadata Not easily searchable High cost
File	Easy on small-scale User-friendly User-manageable File-level locking	Inefficient on unstructured data Limited scalability
Object	Ideal on unstructured data Cost-effective Highly scalable Efficient advanced retrieval	No file-level locking Low performance No data updates

2.1.2 Hadoop Distributed File System

HDFS, a distributed file system *, is designed to store and process massive datasets efficiently. It leverages a cluster of commodity hardware, allowing for cost-effective scalability and high availability. Unlike traditional file systems, HDFS prioritizes high-throughput data access, making it well-suited for applications that process large volumes of data (higher than 100 GB), such as log analysis, data warehousing, and machine learning [**?**].

At the core of HDFS lies a master-slave architecture. A single Namenode acts as the central control point, managing the file system namespace, tracking

*HDFS official guide available at https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html

file metadata, and controlling client access to files. Multiple Datanodes serve as worker nodes, each responsible for storing and managing a portion of the data within the cluster. HDFS divides files into large blocks, which are then replicated across multiple Datanodes to ensure data redundancy and fault tolerance. This distributed storage approach enhances data availability and minimizes the risk of data loss due to hardware failures. Figure 2.2 presents a simplified visual representation of the Namenode read/write operations and Datanode orchestrating operations in HDFS.

The Namenode maintains a comprehensive record of the file system namespace, including file locations, block mappings, and access permissions. This metadata is crucial for efficient data retrieval and processing. By strategically placing data replicas across different nodes, HDFS minimizes data movement, optimizing performance and reducing network congestion. This aligns with the core principle that "moving computation is cheaper than moving data". HDFS is designed for applications that primarily require write-once-read-many access patterns. This design choice simplifies data management and optimizes performance for batch processing tasks, where data is typically written once and then read multiple times for analysis or processing.

Furthermore, HDFS relaxes some of the strict requirements defined by POSIX, such as low-latency access, to prioritize high-throughput data transfer. This allows HDFS to efficiently handle large-scale data processing tasks, making it a cornerstone of many big data applications.

2.1.3 HopsFS

HopsFS, introduced at the 15th USENIX Conference in 2017 [**?**], represents a significant evolution of HDFS. HopsFS addresses the critical scalability and metadata management limitations inherent to its predecessor, decentralizing metadata management by leveraging RonDB *, a distributed NewSQL database, which allows metadata to be stored and managed across multiple Namenodes. This architecture supports Namenode replication and dynamic scaling, significantly enhancing throughput and operational efficiency.

HopsFS encapsulates file system operations as distributed transactions, using advanced NewSQL features such as partition pruning and write-ahead caching. These techniques enable faster metadata retrieval and scalable operations, as seen in experiments where HopsFS outperformed HDFS by 16 to 37 times in throughput for real-world workloads [**?**]. Despite its scalability

*Details available at https://www.rondb.com

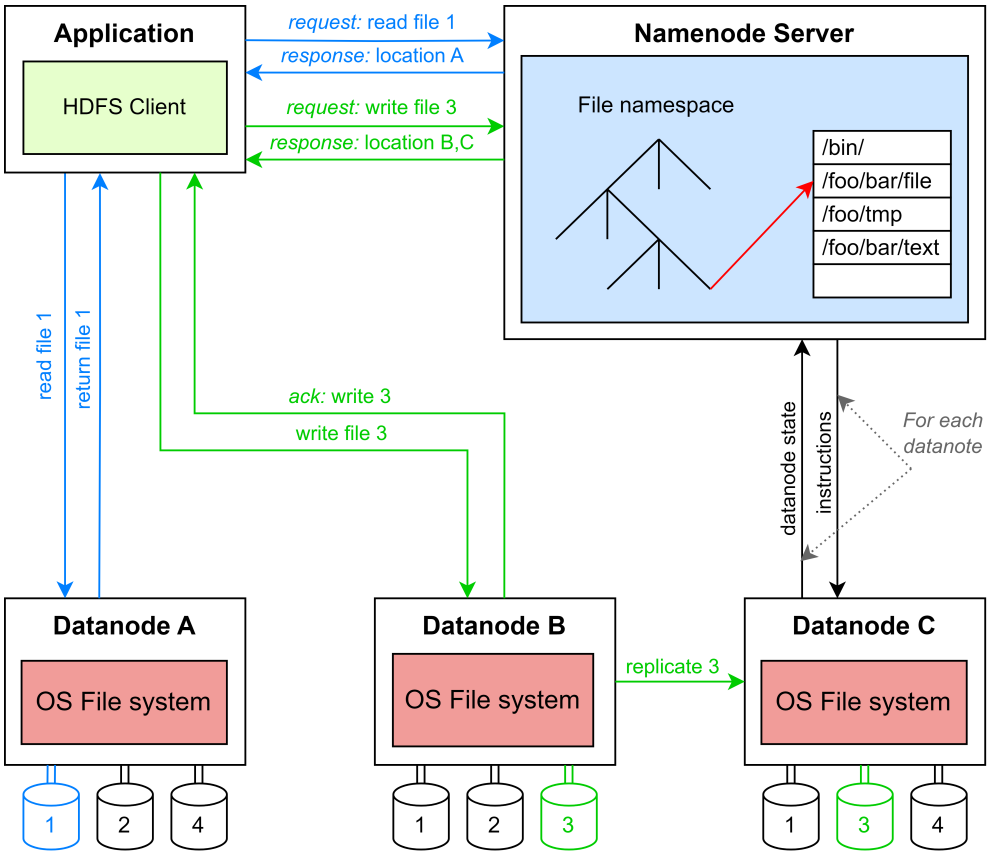

Figure 2.2: Hadoop Distributed File System (HDFS) architecture displaying in different colors basic operations: read (blue), write (green) and Namenode-Datanodes management messages (black). Note: for representation simplicity, files are not segmented into blocks and a single Namenode-Datanode message exchange is pictured. 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.

and performance advantages, HopsFS remains backward-compatible with HDFS, serving as a drop-in replacement. While HDFS is widely adopted as a cornerstone of big data applications, HopsFS provides a forward-looking solution for environments where metadata scalability and performance are critical. Both systems demonstrate the strengths of distributed file systems in managing massive datasets, but HopsFS builds on the foundation of HDFS by addressing its limitations and pushing the boundaries of what distributed file systems can achieve.

2.1.4 Cloud object stores, an alternative

The advent of cloud computing has revolutionized data storage, with object storage emerging as a dominant paradigm [**?**]. Pioneered by Amazon Web Services (AWS) with its S3 service in 2006, cloud object storage services have rapidly proliferated, offered by major providers like Google Cloud Storage (GCS) and Microsoft Azure. This widespread adoption stems from the inherent advantages of cloud-based solutions, particularly their scalability and cost-effiency, as explained in 2.1.1. Cloud object storage empowers users to dynamically scale their storage capacity on-demand, paying only for the resources consumed. This pay-as-you-go model eliminates the need for significant upfront investments in hardware and infrastructure, significantly reducing operational costs. Furthermore, cloud providers leverage economies of scale to unparalleled levels of availability and scalability that are difficult to achieve for most organizations.

HDFS, initially released in 2006, evolved alongside the rise of cloud object storage. While HDFS has been widely adopted for on-premise deployments, the increasing popularity of cloud services has shifted the landscape. Cloud object storage solutions like AWS S3, GCS, and Azure Blob Storage have gained significant traction, with applications prioritizing support for these platforms due to their widespread adoption. While HDFS and its advancements, such as HopsFS, still have their place in specific use cases, the convenience and scalability offered by cloud object storage services have made them the preferred choice for many organizations. However, in the latest year, hybrid solution have been created, to extend cloud object storage with typical file-based system advantages, such has HopsFS–S3 [**?**].

2.2 Data management

This Section introduces the concept of data mananagement layer, starting from its origin and evolution in to today's technologies. This dissertation includes advantages and limitations of each evolutive step, and describe the functions of DBMS. what a data storage is and which typologies of storages exist. The Subsections 2.2.2 focuses on the architecture of data lakehouses, explaing in details each component involed. Then, in Subsections 2.2.3, three data lakehouse frameworks are compared, namely Hudi, employed in the legacy version of Hopsworks feature store, Iceberg and Delta Lake, the two alternatives evaluated.

2.2.1 Brief history of Data Base Management Systems

Since the 2010s, with the advent of Big Data, the data volume, variety, and production velocity have increased exponentially [**?**, **?**]. While on one side, this proved to be of enormous value, on the other this posed several challenges [**?**] on data architectures, which had to evolve to cope with these new needs. Data lakehouse frameworks, like Hudi, Iceberg and Delta Lake [**?**, **?**, **?**] are the last step of this evolution. However, to truly understand this tools, it is needed to start from the beginning of DBMS evolution.

Before big data, companies already wanted to get insights from their data, automating the workflow from the data sources to point of access to this dataa. Here is where ETL and relational databases first came into use. An ETL pipeline consists of three steps:

1. **Extracts** data from APIs or other various data sources.
2. **Transforms** data according to one or more goal. Often this means removing absent fields, standardizes to a specific format to match the database schema, and validates the data.
3. **Loads** it into a relational database (e.g., MySQL).

This workflow structure enabled companies to generate BI insights and data reports based on organizational data. However, a key limitation of this approach is its restricted ability to perform analytical queries that require joining multiple tables and working on several data dimensions. These types of queries, while executed less frequently than simpler queries, are essential for strategic decision-making (e.g., identifying the customer segment with the highest profitability over the past year, with the capabilities of drilling down on products, marketing and sales information).

To address the increasing demand for analytical queries, more advanced DBMs replaced traditional relational databases, optimizing performance for business-oriented analytical workloads. These systems, known as OLAPs, introduced specialized storage and query execution strategies tailored for large-scale analytical processing. The most notable example of an OLAP system is the data warehouse, which revolutionized how organizations handle structured data analysis. A data warehouse workflow, enables companies to process and analyze significantly larger datasets than traditional databases. Unlike transactional databases, which optimize for rapid insert/update operations, data warehouses focus on read-optimized queries by structuring data into columnar formats, reducing scan times for analytical queries. They

maintain core relational database properties such as ACID transactions and data versioning, ensuring data integrity and consistency for complex business intelligence applications.

Over time, the exponential growth of unstructured data (e.g., images, videos, logs, and sensor data), posed new challenges for companies aiming to leverage such information. Traditional data warehouses struggled to accommodate this data due to their rigid schema requirements and high storage costs. Moreover, they were not designed to support AI/ML workflows that rely on diverse, unstructured datasets. To overcome these limitations, organizations adopted a new paradigm known as data lakes. Data lakes leverage cost-efficient object storage (Section 2.1.1) and a schema-on-read approach, where raw data is loaded first and transformed only when needed. This shift from the traditional ETL model to ELT offers greater flexibility in handling diverse data types. For example, businesses can use data lakes to store raw IoT sensor readings and later refine them for predictive maintenance models, once the needed data transformation will be designed. Despite reducing storage costs and improving flexibility, data lakes introduced new complexities. Unlike structured databases, querying data lakes directly for BI reports is impractical, as they lack indexing and transactional consistency. Additionally, maintaining separate storage solutions for structured (data warehouses) and unstructured (data lakes) data increases operational complexity and costs. A major drawback of data lakes is the risk of becoming "data swamps", repositories filled with ungoverned, low-quality data that provide little business value. Recognizing these challenges, organizations sought a hybrid solution combining the best features of data warehouses and data lakes. This led to the emergence of the data lakehouse architecture. First described by Databricks in 2020 at Conference on Innovative Data Systems Research (CIDR) [**?**], data lakehouses integrate structured and unstructured data storage capabilities of data lakes, while maintaining the governance, ACID compliance, and indexing capabilities of data warehouses. The data lakehouse approach resolves many pain points associated with separate data warehouses and data lakes. It enables organizations to use a single storage for all types of data, while supporting high-performance SQL queries and ML workloads. Furthermore, by leveraging open file formats like Apache Parquet, ORC and Avro, data lakehouses ensure interoperability with various analytics engines, avoiding the risk of getting data locked into a proprietary format [**?**]. All these capabilities and features make lakehouses an ideal solution for enterprise-scale data processing. Table A.1 provides a comparison of data warehouses, data

lakes, and data lakehouses, highlighting key takeaways for each technology and summarizing what described so far.

2.2.2 Data lakehouse architecture

Data lakehouse architectures combines the desirable attributes of data warehouses and data lakes, mitigating the challenges encountered by both these technologies, and eliminate the need of a two-tier data stacks to run varying analytical workloads. The key components of data lakehouses are at the bases of any other architecture seen so far, but their design focus on reducing components coupling, providing more agility and choice when architecting such platform [**?**]. The data lakehouse components, graphically presented in Figure 2.3, are:

- **Data storage**: is where data files land after ingestion from various systems, usually a Cloud object store, as described in Section 2.1.
- **File formats**: hold the actual raw data and are phisically stored in the data storage. They are open-source file formats, such Apache Parquet or JSON, and are typically column-oriented.
- **Table format**: also referred as OTF, acts as metadata layer on the top of the file formats, that abstracts the underlying physical data structure. They offers API access to the query and storage engines.
- **Storage engine**: handles data management tasks, aimed at optimizing efficiency of queries over the data. It performes tasks as data compation, indexing, and partitioning.
- **Catalog**: sometimes described as metastore, it enables efficient search and discovery. The catalog keeps track of information about each table (name, column names, data types) and a reference to metadata for each table (table format).
- **Query engine**: is responsible for processing data, performing read and write operarations leveraging the table format API. They are further explained in Section 2.3.

OTF are open standards, and by design support interoperability throughout the stack, as visible in Figure 2.3. Thus, depending on the integration capabilities of the table format, there are several implementation options for data storage, file formats, storage engine, catalog and query engine.

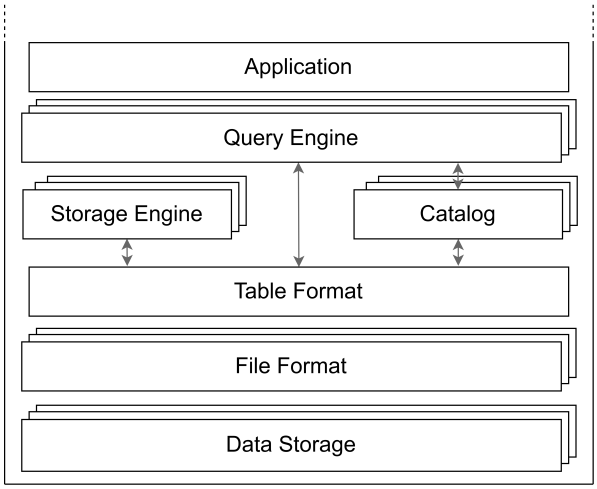

Figure 2.3: Abstraction of architecture of a data lakehouse. Inspired by "Open table formats in perspective" article on OneHouse Website, available at https://www.onehouse.ai/blog/open-table-formats-and-the-open-data-lakehouse-in-perspective

Additionally, recent introduction of tools like Apache XTable * demonstrates the trend towards a universal compatibility between OTFs.

Open Table Formats

OTFs provide an abstraction layer on top of data lakes, enabling database-like functionalities, enhancing data management capabilities significantly [**?**, **?**]. One of the cornerstone features of OTFs is support for full Create Read Update Delete (CRUD) operations. The ability to perform updates and deletes sets datalake house data storage apart from traditional file-based storages, where such operations are cumbersome and inefficient. Performance and scalability are other notable features that OTFs bring to the table. These formats are designed to excel in Big Data environments, where data volumes are massive and continue to grow. OTFs could support various optimization techniques, such as indexing, partitioning, and caching, to expedite data retrieval and processing. This not only improves query performance but also ensures that the system can scale horizontally to accommodate increasing data loads without a significant degradation in performance. As a result, organizations can manage their data ecosystems more effectively, making data-driven insights more accessible and actionable. Transactional support

*XTable repository available at https://xtable.apache.org/

with ACID compliance is another key feature of OTFs. This ensures that all data transactions are processed reliably, maintaining data integrity and consistency across the board. This is particularly important in scenarios where multiple transactions occur simultaneously or when the system needs to recover from partial failures. OTFs guarantee that each transaction is completed successfully or fully rolled back, providing an essential level of data reliability and trustworthiness for critical business operations. This comprehensive functionality allows for flexible and complex data workflows, and ensures that data lakes and warehouses can be updated in real time, reflecting the most current state of information.

2.2.3 Data lakehouse comparison

The three main data lakehouse frameworks investigated in this project are Hudi [**?**], Iceberg [**?**] and Delta Lake [**?**]. Their popularity has increased proportionally with the popularity of the data lakehouse architecture, as visible by the growing community of each of these technologies in Figure 2.4, becoming the de-facto standards for data lakehouse implementations [**?**]. Some alternatives are newly being developed, such Apache Paimon *, but are still at early stages of their development and will not investigated in this thesis project. The following subsections present each of these technologies, specifying their development history, key features, integration capabilities, API offered and in which are the use cases they are most suitable for.

Apache Hudi

Apache Hudi, open-sourced by Uber in 2017 [**?**], is an open-source framework that addresses the challenges of low-latency data ingestion and incremental processing. Hudi enables efficient record updates and deletions in data lakes, thus eliminating the need to rewrite entire datasets [**?**]. The framework supports Change Data Capture (CDC) for efficient updates and deletes. Hudi offers two primary write optimization modes: Copy on Write (CoW) for high read performance and Merge on Read (MoR) for balancing both write and read performance.

Hudi's metadata layer is structured around a timeline-based architecture, maintaining a history of all table operations, thus providing version control. The commit timeline records actions such as inserts, updates, deletes, and compactions, enabling time travel by referencing specific commit instants.

*GitHub repository available at https://github.com/apache/paimon/.

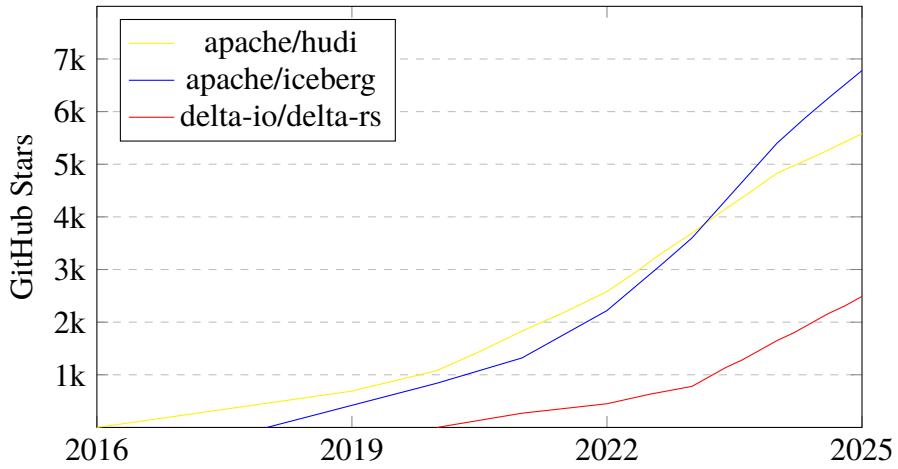

Figure 2.4: Trend of GitHub stars of following repositories https://github.com/apache/iceberg, https://github.com/apache/hudi, https://github.com/delta-io/delta-rs.

The metadata table optimizes file listing by indexing data files, partitions, and record locations, improving query performance. Hudi also maintains delta logs (MoR) to track incremental changes before they are compacted into columnar storage. The file groups and file slices structure organizes base and log files, ensuring efficient data versioning.

Hudi is optimized for real-time analytics through low-latency streaming ingestion, and integrates seamlessly with Spark and Flink for data ingestion and processing. Hudi also supports reading data from Hive, Impala, and Presto. The framework incorporates multimodal indexing, such as Bloom filters and record-level indexing, and employs both Multi-Version Concurrency Control (CC) and Optimistic CC for transaction management. Typical use cases for Hudi include streaming ingestion with frequent updates and deletes, incremental data processing, and real-time updates in domains like IoT, fintech, and log data processing [**?**]. While Hudi excels in real-time ingestion, its metadata overhead can be significant for large-scale analytical queries, potentially resulting in slower query performance compared to other formats like Iceberg and Delta Lake [**?**].

Hudi's core implementation is in Java, and it integrates deeply with Apache Spark, offering a Spark datasource API for reading and writing Hudi tables. This makes Java and Scala the primary languages for Hudi development and usage. While a standalone Python Software Development Kit (SDK) for Hudi doesn't exist, Python users can still interact with Hudi tables through PySpark,

leveraging the Spark API. Although community-driven efforts like hudi-rs *, a native Rust implementation with Python bindings, are emerging, offering read support for some technologies, Hudi is currently still JVM-centric.

Apache Iceberg

Apache Iceberg, open-sourced by Netflix in 2018 [**?**], is an open-source framework that addresses the limitations of Hive tables by emphasizing scalability and correctness for large-scale analytics. By separating metadata from data, Iceberg supports efficient snapshot-based isolation and query planning [**?**, **?**]. The framework provides full ACID transactions, ensuring atomicity, consistency, isolation, and durability. Iceberg allows for schema evolution, enabling the addition, removal, and renaming of columns, without breaking existing queries. It also supports partition evolution, allowing changes to partitioning schemes without necessitating data rewrites.

Iceberg's metadata layer consists of multiple components that enable efficient time travel and point-in-time query planning. The manifest list acts as an index, pointing to multiple manifest files, each of which tracks a subset of data files. These manifest files store information about data file locations, partitioning, and statistics (e.g., min/max values). The snapshot metadata records changes over time, referencing manifest lists and enabling time travel by tracking table versions. Iceberg also maintains a table metadata file, which holds high-level properties, schema, partitioning details, and references to the latest snapshot.

Iceberg is designed to scale to petabyte-scale datasets with optimized metadata management using manifest files. Its architecture enables data warehouse-like functionality, leveraging cloud object storage for efficient data access. Furthermore, Iceberg supports collaboration across multiple applications with transactionally consistent access. The framework integrates with query engines like Spark, Trino, Presto, and Dremio and supports Flink for both reading and writing. Iceberg excels in read performance, particularly for tables with large numbers of partitions [**?**]. It is ideal for analytical workloads that involve large-scale datasets and frequent schema or partition evolution. However, Iceberg's write performance may not be as optimized for streaming scenarios compared to other formats.

Iceberg's core is written in Java, and it is heavily integrated with Apache Spark [**?**, **?**]. The most common way to interact with Iceberg is through SparkSQL or other query engines like Trino and Presto, which support

*hudi-rs repository available at https://github.com/apache/hudi-rs.

Iceberg natively, making Java and Scala the dominant languages in these environments. However, Iceberg offers more than just Spark bindings. PyIceberg [**?**] provides a direct Python interface for read and – from release 0.6.0 in 2024 – write operations, making Iceberg accessible to Python developers without requiring PySpark. Furthermore, query engines like Trino, accessible from Python via libraries like PyHive, allow Python users to query Iceberg tables through SQL. Iceberg community has recently developed a Rust implementation, iceberg-rust *, which provides read access to Iceberg tables, further expanding its accessibility to a wider range of language ecosystems.

Delta Lake

Delta Lake, open-sourced by Databricks in 2019 [**?**], is an open-source framework that enhances the reliability and performance of data lakes, facilitating a seamless transition between batch and streaming use cases. Often considered the first data lakehouse, Delta Lake employs a transaction log to record all changes to data, ensuring consistent views and write isolation, which supports concurrent data operations [**?**]. The framework offers ACID transactions, as well as features such as unified batch and streaming processing, indexing, and schema enforcement.

Delta Lake's metadata layer is built around the Delta Log, which records every transaction in JSON files, enabling time travel. The transaction log maintains a sequential history of operations, while checkpoint files (stored in Parquet format) periodically summarize log entries for faster metadata access. The protocol metadata defines the table's versioning, schema, and supported features, ensuring compatibility across different readers and writers. It also incorporates metadata-informed data skipping during merge operations and supports streaming through change data feeds.

Delta Lake tighlty integrates with Spark, thus it is particularly well-suited for real-time streaming, batch processing, and ML pipelines requiring data versioning. While its open-source adoption continues to grow, Delta Lake remains deeply integrated with the Databricks ecosystem [**?**, **?**]. Additionally, its focus on schema and partition evolution is less pronounced compared to frameworks like Iceberg, making Delta Lake less suitable for situations where data schemas are frequently changing.

Delta Lake core is written in Java and Scala [**?**], providing native support within the Spark ecosystem through the delta package. This allows Java and Scala developers to interact directly with Delta tables. Python support is

*iceberg-rust available at https://github.com/apache/iceberg-rust.

provided via PySpark, enabling Python developers to leverage the Spark API for Delta Lake operations. However, Delta Lake's reach extends beyond the JVM. With the release of Delta Kernel [**?**], a Java library providing low-level access, and the development of delta-rs *, a Rust-based implementation, Delta Lake has become accessible to a wider audience. The library delta-rs allows interaction with Delta tables without Spark or JVM dependencies, and its Python bindings makes this particularly attractive to the Python data science community.

In Table A.2 is presented a comparation between key features of the three data lakehouse frameworks.

2.3 Query engine

This Section describes the technologies used to query, cache, and process data in this project. The category of query engine includes mainly Apache Spark and DuckDB, but technologies operating at the same abstraction level of those are here presented, namely Apache Kafka, Arrow Flight and Catalogs.

2.3.1 Apache Spark

Apache Spark is an open-source distributed computing framework designed for large-scale data processing [**?**]. It builds upon MapReduce, a distributed programming model developed by Google for handling massive datasets [**?**], later adapted into Hadoop MapReduce by Yahoo! engineers [**?**]. Spark enhances this model using RDDs[**?**], a disitrbuted memory abstraction that enables lazy in-memory computation, diffently by on-disk MapReduce's computation, that is tracked through lineage graphs. This increases fault tolernace [**?**] and allow to manage even bigger scale computations.

Spark supports various workloads beyond batch processing, leveraging an in-memory execution model for efficiency. Its core components include SparkSQL for querying structured data, Spark Streaming for real-time processing, MLlib for scalable ML, and GraphX for large-scale graph processing. With support for iterative computations, Spark is particularly well-suited for ML and graph analytics. It also provides APIs for Scala, Java, Python (via PySpark), and R, making it accessible to a broad range of users. Despite its advantages, Spark has limitations. Its in-memory execution requires significant RAM, and JVM-based execution can lead to

*delta-rs repository available at https://github.com/delta-io/delta-rs.

performance overhead due to garbage collection. Tuning performance involves fine-tuning configurations, which can be complex. Additionally, Spark Streaming's micro-batch processing introduces higher latency compared to other frameworks, making it not the best solution for small-scale datasets processing [**?**].

2.3.2 Apache Kafka

Apache Kafka is a robust, open-source distributed platform for handingle data streaming, that has become a cornerstone of modern data architectures [**?**]. Kafka supports high throughput data ingestion and processing, making it exceptionally well-suited for handling massive volumes of data in real, given its ability to manage and process continuous streams of data with very low latency. Kafka's architecture, which emphasizes fault tolerance, scalability, and durability, allows it to handle the demands of mission-critical systems that require continuous data flow and processing. The key components of Kafka architecture are:

1. **Producer**: an application that publishes data, labeling into specific topic that group similar messages.

2. **ZooKeeper**: the responsible for managing the Kafka cluster, including broker(s) information and topic message tracking, tracked with an offset for each topic.

3. **Broker**: a processing node, or server, that handles data for specific topics. It receives messages from producers and delivers them to consumers upon request, enabling asynchronous communication. When for a single topic there are multiple brokers, one is elected as a leader, and the others replicate its content.

4. **Consumer**: an application that subscribes to specific topics to receive and process data. Multiple consumers can subscribe to the same topic.

Kafka enables applications to behave as producers and consumers, without the need of developing any synchronization protocol. This enables producers to reach high throughput, as they can broadcast messages without waiting for any acknoledgements or availability singal from the consumers. Due to its distributed architecture, a Kafka ecosystem can be optimised according to specific needs, allowing several brokers, producers and consumers to coexist.

2.3.3 Catalogs

The term catalog is used in the data domain in multiple contexts, in each of those with a different definitions. In the data lakehouse context, thus in this project context, a catalog is a technical catalog also called metastore [**?**]. As explained in Section 2.2.2, a catalog plays an important role in tracking tables and their metadata. It is, at a minumum, the source if truth for a table's current metadata location. This is in contrast to a federated catalog, which is a central portal for data discovery and collaboration, since it tracks datasets across multiple data stores. It focuses on business needs such data governance and documentation, and sets standards for authentication and authorisation [**?**]. A federated catalog may point to tables from Cassandra, Postgres, Hiv, and other systems.

Made clarity over the catalog definition regarding this project, there are several technologies providing such tool. When data lakehouses were first created [**?**, **?**], the Hive Metastore (HMS) was the most popular catalog. HMS is the technical catalog for Hive tables, but it can actually belong to both catalog categories, since it may also track Java DataBase Connectivity (JDBC) tables, and other datasets. However, scale challenges led developers and big tech companies to develope their own more scalable tools, which contributed to the development of data lakehouses technology itself, such the case of Iceberg [**?**].

Data lakehouse technologies support different Catalogs, as shown in Table **??**. Depending on the specific business need or on the available cloud resources and ecosystem, possible "pluggable" catalogs are:

- **SQL Catalog**: is a C library, with Python bindings (SQLite), that provides lightweight disk-base database, that allows accessing the database with nonstandard variants of SQL, where metadata can be saved. This does not require a separate server process, thus it is great in embedded applications, but it is not suitable for large-scale applications.
- **AWS Glue**: provides a Data Catalog that serves as a managed metastore within the AWS ecosystem. It integrates well with otherAWS services, thus it is a common choice for data lakehouses built on AWS.
- **DynamoDB**: it is primarily a No SQL databases that due to its scalability and performance, can be used as a metastore. This is less common compared to other options.
- **JDBC**: per sé is not a catalog, but it's the standard Java API for

connecting to relational databases. In this context, JDBC would be used to connect to and interact with database systems that stores metadata.

- **Nessie**: acts as a versioned metastore, providing Git-like capabilities for managing data lake tables
- **Databricks Unity**: is Databricks' unified governance solution, thus it is a federated catalog, which however includes also technical catalog capabilities. It provides a centralized metastore specifically designed for the Databricks Lakehouse Platform, thus is the most common option for Delta Lake-backed lakehouses [**?**].

As data lakehouse frameworks grew to support more and more languages and engines, pluggable catalogs started causing some practical problems, related to compatibility. It proved indeed difficult, for commercial offerings, to support many different clients and catalog features. To overcome these problems, some developers, like Iceberg's community [**?**], created also a REST catalog protocol, a common API for interacting with ant catalog. The advantages of such protocol are are multiple: (1) one client implementation, for new languages or engines, can support any catalog; (2) secure table sharing is enabled, using credential vending or remote signing; (3) the amount of failures is reduced, since server-side deconfliction and retries are supported.

When implementing a new data lakehouse, the catalog decision depends on the specific integrations and feature proper of each catalog option, also considering how this can be configured on the top of the previously added layer, as shown in Table 2.3.

2.3.4 Duck DB

DuckDB [**?**] is an open-source, embeddable, OLAP DBMS designed for efficient processing of small-scale datasets (from 1GB to 100GB) within the same process as the application using it. This embedded, in-process operation, inspired by SQLite's success, simplifies deployment and eliminates the overhead of inter-process communication, leading to high responsiveness and low latency. DuckDB requires no external dependencies and compiles into a single amalgamation file, enhancing portability across major operating systems and architectures, including web browsers via DuckDB-Wasm. It offers API for various languages, such Java, C, C++, Go, Rust and Python and supports complex SQL queries, including window functions and ACID properties through Multi-Version CC.

Data is stored in persistent, single-file databases, and secondary indexes enhance query performance. DuckDB's columnar-vectorized query execution engine, a key design choice for OLAP workloads, processes data in batches (vectors), significantly improving performance compared to traditional row-by-row processing systems. While optimized for analytical queries processing significant portions of datasets, DuckDB is not designed for massive data volumes (1TB or more) that require disk-based processing, as its core processing relies on in-memory operations. However, its extensible architecture allows for adding features like support for Parquet, JSON, and cloud storage protocols as extensions.

2.3.5 Arrow Flight

Arrow Flight is an high-performance framework designed for efficient data transfer over networks, mostly utilising Arrow tables [**?**]. This protocol facilitates the transmission of large data volumes stored in a specific format, such as Arrow tables, without the necessity of serialisation or deserialisation for transfer. This significantly accelerates the data transfer, making Arrow Flight highly efficient. Arrow Flight is engineered for cross-platform compatibility and supports many programming languages, including C++, Python, and Java. The protocol further facilitates parallelism, enhancing transmission speeds by using numerous nodes in parallel networks. The Arrow Flight protocol is constructed upon gRPC, facilitating standardisation and simplifying the construction of connectors.

2.4 Application - Hopsworks

This Section describes the application layer, the uppermost layer presented in Figure 2.1, which takes advantage of the data stack described above. The software described is the Hopsworks feature store, which this project contributes to. This software is part of the broader Machine Learning Operations (MLOps) platform offered by Hopsworks AB, the company that hosted this master thesis.

2.4.1 Machine Learning Operations (MLOps)

MLOps are a full set of practices related to the development and automation of ML workflows. Through the MLOps lenses, a ML workflow is logically separated in smaller steps, and considered from a data, code and model

perspective. Differently from a classical software application, where only the code needs versioning, in the context of ML applications, it is key that all three of data, code and model are versioned. Perhaps, different data might be used by models in different moments, as well as new model might be trained on the same data, to truly understand their improvements.The novel challenges are thus related to data validation, ML artifacts versioning, ML workflow orchestration, and infrastructure management, given higher computation and storage capabilities needed by those workflows [**?**, **?**].

The need of the described features saw new solutions emerging, like the Hopsworks AI data platform [**?**]. In the context of data versioning, the specific solution is called feature store [**?**], which consists in a centralized fast-access storage for both real-time and batch data.

A simple architecture following MLOps principle is presented in Figure 2.5. As first step, data are gathered either streaming (real-time) or batch data sources. A feature pipeline process those the data, performing model-independent transformations [**?**], and saves them in the feature store. A training pipeline runs now model-dependent transformations on features and labels retrieved from the feature store, and train the respective model, saving the trained model in the model registry. The last pipeline, the inference pipeline, extract at its need a specific model from the model registry and access the features, over which a inference will be conducted, from the feature store. The output of this pipeline, which normally is embedded directly in the consuming application, are the predictions of the model over the selected features, altogether with logs describing the performance of the model according to a pre-selected measure. All the pipeline are decoupled and could thus work asynchronously, making the whole ML workflow scalabile, maintainable and effective.

2.4.2 Hopsworks AI Data Platform

As briefly said above, the feature Store is a key data layer in an ML application built over the MLOps principles. The feature store enables feature reusability, as well as multi-user collaboration and a centralized authenticated access to the feature. The Hopsworks feature store organizes features in feature groups. Those are mutable collection of features, over which developers can perform CRUD operations, accessing them via the Hopsworks APIs.

The Hopsworks feature store supports both batch data sources and real-time data streaming. This hybrid system is possible thanks to the dual implementation of an offline and an offline feature store. The offline feature

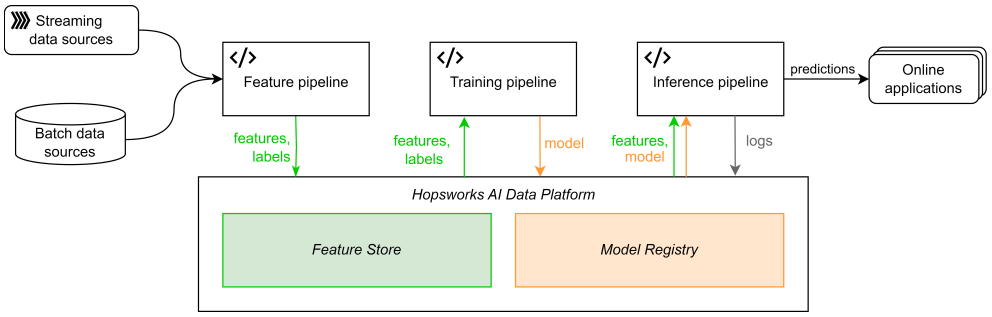

Figure 2.5: MLOps pipeline using a feature store and a model registry. Diagram inspired by Hopsworks documentation available at https://www.hopsworks.ai/dictionary/feature-store.

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 real-time row-based, key-value data storage based on RonDB, thus enabling low latency and real-time processing. To maintain consistentency between those two stacks, the Hopsworks feature store has a unique point of entry for data, which is Kafka, explained in detail in 2.3.2. This guarantess that each message is delivered to both storages, which acts are consumers subscribed to the Kafka topics.

The model registry works under the same principles of the feature store, proving a centralized access and a standard deployment for all the models saved in it. It also saves information about model performance along time, model schemas and training feature logs, to ease model reproducibility and tracking.

2.5 System architectures

This Section describes the architectures of the legacy, based on Hudi the system implemented during this thesis work, based on Iceberg, and the system implemented in a related thesis work [**?**], based on Delta Lake. Those are the system that will be run and measured in teh experimental part of this thesis work. This Section is divided into six Subsections, according to the system and the operation run over it. For each Subsection, a chart presents the operation protocol step by step.

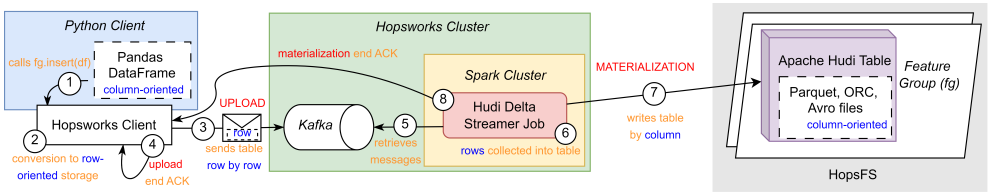

Figure 2.6: Legacy system writing a Pandas DataFrame from a Python client to the Hopsworks offline feature store. Each step is represented with a number. The table format conversion is outlined in blue, i.e., from columns to rows and then from row to columns. Steps from one to four represent the upload 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.

2.5.1 Legacy system - Hudi - writing

Figure 2.6 * 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.

*For enhanced visualization, refer Figure B.1.

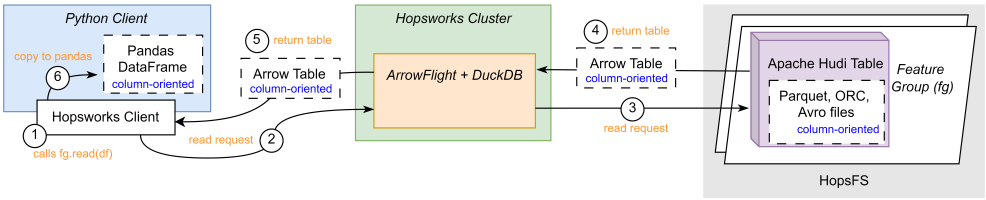

Figure 2.7: Legacy system reading a table from the Hopsworks offline feature store and loading it into the Python client's local memory. The process is streamlined using Arrow Tables that avoid table conversion. Diagram inspired by the Hopsworks feature store paper [**?**].

2.5.2 Legacy system - Hudi - reading

Figure 2.7 * shows 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.

2.5.3 New system - Pyiceberg - writing

Figure 2.8 shows how the PyIceberg library writes on an Iceberg table instanced on top of HopsFS. If first requests metadata information about the existing table to Iceberg Catalog, implemented using SQLite in the example. Once the JSON file containing the metadata is received, it reads the table location and request to read the table. The PyIceberg library streamlines the process without passing from a server instance (Spark), removing the middle-tier from the process.

2.5.4 New system - Pyiceberg - reading

Figure 2.9 shows how the PyIceberg library reads on an Iceberg table instanced on top of HopsFS. If first requests metadata information about the existing table to Iceberg Catalog, implemented using SQLite in the example. Once the JSON file containing the metadata is received, it reads the table location and proceed to write (append), by column, on the Iceberg Table. The PyIceberg library streamlines the process without passing from a server instance (Arrow Flight), removing the middle-tier from the process.

*For enhanced visualization, refer Figure B.2.

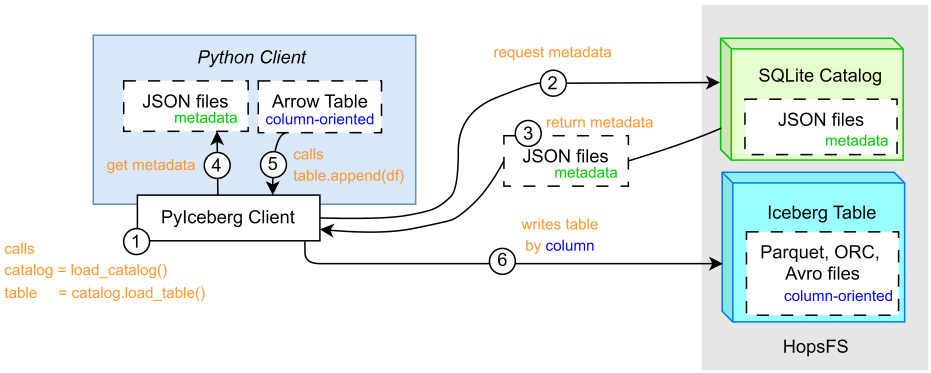

Figure 2.8: PyIceberg library writing an Arrow Table from a Python client to an Iceberg Table stored on HopsFS.

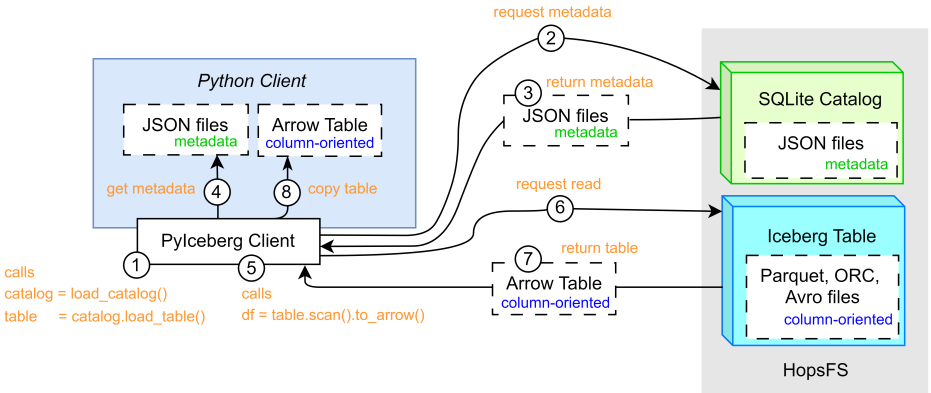

Figure 2.9: PyIceberg library reading an Iceberg Table stored on HopsFS and loading it into memory.

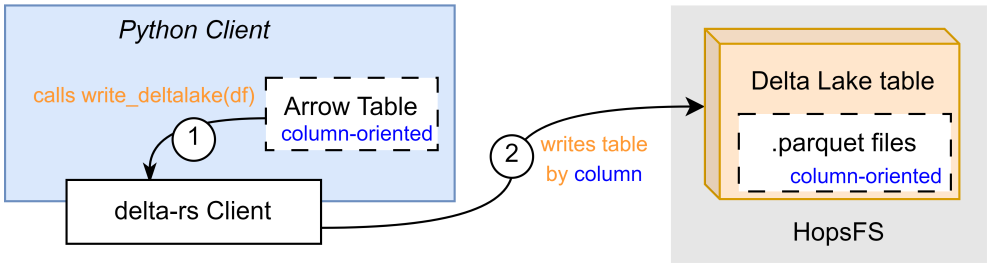

Figure 2.10: Delta-rs library writing an Arrow Table from a Python client to a Delta Lake table stored on HopsFS.

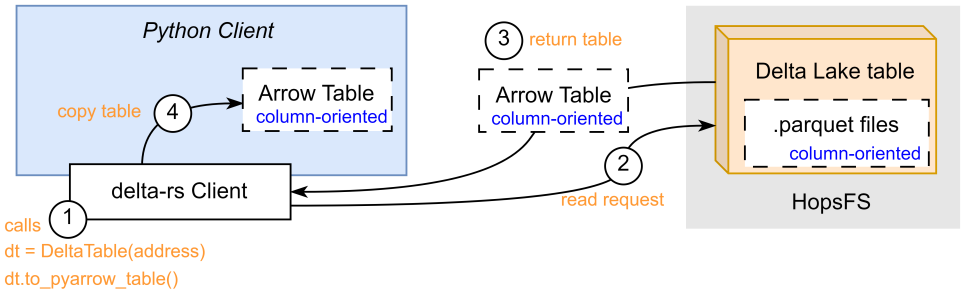

Figure 2.11: Delta-rs library reading a Delta Lake table stored in HopsFS and loading it into memory.

2.5.5 New system - delta-rs - writing

Figure 2.10 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), removing the middle-tier from the process. In this system, the only file format supported is Parquet, while the other systems supported also ORC and Avro.

2.5.6 New system - delta-rs - reading

Figure 2.11 shows how the delta-rs library reads a Delta Lake table instanced on top of HopsFS. The delta-rs library streamlines the process without passing from a server instance (Arrow Flight), removing the middle-tier from the process. In this system, the only file format supported is Parquet, while the other systems supported also ORC and Avro.

Chapter 3

Method

This chapter defines three methodologies that will be applied sequentially in this project, answering the two RQs defined in Section 1.2.1. Section 3.1 defines the system integration process that outputs part of D3, i.e., the integration detail. This output will enable the Hudi vs. Iceberg system evaluation defined in Section 3.2, which will output D1, i.e., the results of the experiments. The Iceberg experiment results (D1-partial) will enable the Iceberg vs. Delta Lake system evaluation, defined in Section 3.3, which will output D2, i.e.,the results of the comparative experiments. The analysis of Ds1–2 will be delivered in D3, i.e. this comprehensive thesis report.

It has to be noticed that the system evaluation processes, described in Sections 3.2-3.3, are inspired by the system evaluation process of a related work [**?**]. That related work was hosted by the same company which hosted this thesis work, Hopsworks AB, and it focused on answering RQs on some intent similar to the ones described in Section 1.2.1. The related work enabled write and read operations on Delta Lake tables stored on HopsFS, and investigated performance differences between that system and the current Hopsworks legacy system. Following similar system evaluation processes will not only allow to answer this project's RQs, but will also enable the reader to fairly compare the three technologies investigated in these two thesis works. For the same reason, it easen the comparative process described in Section 3.3, and will make its results more fair and consistent.

3.1 System integration

This Section explains the method and principles used for the system integration process, between PyIceberg and HopsFS. This Section is divided

into three Subsections: Integration process, describing the activities to be conducted to integrate PyIceberg and HDFS; Requirements; and Development environment, detailing the tools and resources that will be used during the integration process.

3.1.1 Integration process

The integration development process will follow an iterative in Figure 3.1. This project will require numerous interactions with HopsFS maintainers (i.e., the industrial supervisors), since PyIceberg library has to interact with HopsFS. This review process creates the need for a feedback loop, allowing the system to fit all the stakeholder requirements, described in Subsection 3.1.2. This process partially answer RQ1, to which Gs1–2 are associated, as described in Section 1.4. The relationships between each process activity and Gs are here explained:

1. **Set requirements collaboratively**: this activity is key for the completion of Gs1-2, as it is an initial system analysis, performed together with the industrial supervisors, who are knowledgeable on Hopsworks' infrastructure. This task sets the project requirements and investigates what needs to be integrated at a high-level abstraction.

2. **Analyze system and tools**: this activity solves G1 performing low-level code analysis on system and PyIceberg function calls, understanding what needs to be developed to integrate HopsFS and PyIceberg. This activity is the first of an iterative loop accounting for the requirements fulfillment.

3. **Develop integration**: this activity partially G2, designing and developing a code solution from the information gathered during the analyses above.

4. **Test integration**: this activity partially solves G2, verifying the solution validity with integration tests. Failed integration tests or unfulfilled requirements will triggen a new loop iteration. Otherwise, the integretion process will be concluded.

This process will produce a part of D3, which is the code implementation for PyIceberg and HopsFS integration, including the consideration about the specific tools selected or discarded.

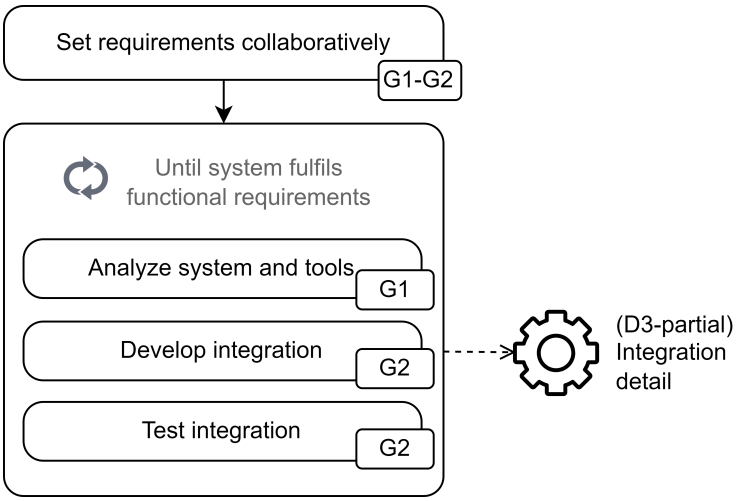

Figure 3.1: Diagram of the system integration process partially answering RQ1. Each activity is associated to specific Gs. The process produces the integration detail (D3 partial). The loop iterates until the functional requirements, defined in Section 3.1.2, are fulfilled.

3.1.2 Requirements

A series of requirements are defined in agreement with Hopsworks AB, to create a solution that could be later used by the company, in a production environment. The **functional requirements** are:

1. **Write Iceberg Tables**: the solution should allow to write Iceberg tables on HopsFS via the PyIceberg library.
2. **Read Iceberg Tables**: the solution should allow to read Iceberg tables on HopsFS via the PyIceberg library.

The **non-functional requirements** are:

1. **Consistency**: the solution should be consistent with the current open-source codebase.
2. **Maintainability**: the solution should minimize the need for maintenance and support.
3. **Scalability**: the solution should handle read or write operations on Iceberg Tables, up to 100 GB, to support small-scale dataset scenarios.

3.1.3 Development environment

The system implementation will be developed using the following technologies:

- **Computing resources**: the system integration will be developed on a Virtual Machine (VM) accessed via Secure Shell protocol (SSH) from a computer terminal. Local development will be avoided, due to low reproducibility and complexity of mounting HopsFS on a local machine.
- **Code versioning and shared development**: GitHub will be used for versioning and sharing the developed solution.

3.2 System evaluation - Hudi vs. Iceberg

This Section explains the method and principles used for the system evaluation processes, measuring and comparing the performance (latency, in seconds, and throughput, in rows/second) of reading and writing Hudi and Iceberg tables on HopsFS. Those operations are conducted respectively on the current legacy system and on the PyIceberg-based system, integrated in this thesis work.

3.2.1 Evaluation process - RQ1

This evaluation process will follow a sequential approach described in Figure 3.2. Each step of this process is related to one of the Gs3–6 associated with the RQ1 in Section 1.4, to which this process partially answer. The relationships between each process activity and Gs are here explained:

1. **Design experiments**: this activity maps perfectly to G3, 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 PyIceberg library-based access to Icbeerg Tables in HopsFS.
2. **Perform experiments**: this activity maps perfectly to G4, using the integration detail (D3-partial) to develop and conduct the designed experiments on the analyzed systems. Here, data is collected as latency, expressed in seconds.

3. **Transform data according to metrics**: this activity is requisite to fulfill G5, and G8 of RQ2. The activity is conducted because throughput is not directly measured, but it is computed from latency following the formula here below:

$$Throughput\ (rows/second) = \frac{Number\ of\ rows\ (rows)}{Latency\ (seconds)}$$

4. **Visualize results**: this activity maps perfectly to G5, visualizing the experiments' result according latency, measured in seconds, and throughput, measured in rows/second. This activity also generates D1, the experiment results complemented with tables and histograms, presented in Chapter 5.

5. **Analyze results**: this activity maps perfectly to G6, analyzing and interpreting the results delivered in D1. This activity contributes to D3, generating the analysis of experimental results, presented in Chapter 5.

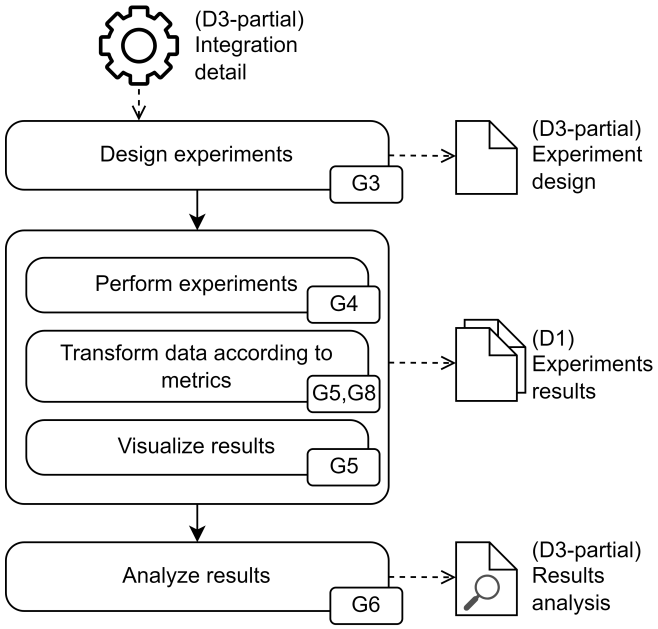

Figure 3.2: Diagram of the system evaluation process partially answering RQ1. Each activity is associated to specific G. The process produces two Ds, the experiments results (D1) and a results analysis (D3-partial).

3.2.2 Industrial use case

Several choices must be made for a system evaluation: which data will be used, which environment will run the experiments, and which metrics will be used to evaluate the system. Those choices depends on the scenarios for which the system is created. Thus, this Subsection describes the typical use case for this system. Follwing, the other Subsections describe which were the decision taken accordingly. While conducting their research in Hopsworks, the author outlined a typical industrial use case for the Hopsworks feature store, by reading internal documentation and discussing with several employees on their customer needs and trends. The use case is described by:

- **Table size**: most of the Hospworks' customers' workloads were limited (from 1M to 100M rows), with only few clients needing support for massive workloads (more than 1B rows). Thus, this project opted to improve performance for the smaller workloads (from 100k to 100M rows). The datatset selected are presented in Section 3.2.3.
- **Type of data**: the Hopsworks feature store works only with structured data (e.g., numbers, strings), thus experiment design and selected datasets embody this scenario.
- **Rows over storage size**: In the experimental part of this thesis, in order to have a reliable unit measure for table size, the number of rows will be over the storage size (bytes). Perhaps, in the structured data domain of this use case, storage size (bytes) is neither linked to a table structure nor to a storage structure, arguably making this unit measure not reliable (i.e, table with a lot of rows and few columns, and a table with few ros and a lot of columns occupies the same memory).
- **Client configuration**: the client configuration is modeled to reflect typical customers' clients' computational and storage capabilities. Thus, the configuration are limited between one and eight CPU cores, RAM just sufficient for system needs and common SSD storages. The experimental environment is further detailed in Section 3.2.5.

3.2.3 Experimental data

The datasets that will be used to perform read and write experiments come from TPC-H benchmark suite *. TPC-H is a decision support benchmark

*Benchmark suite website available at https://www.tpc.org/tpch/

by Transaction Processing Performance Council (TPC), that consists in a collection of business-oriented queries in specific industry sectors [**?**], which became the de-facto standard for experiments on data storage system. Perhaps, it has been used in related studies [**?**, **?**, **?**].

The TPC-H benchmark contains eight tables, and any part of the data can be generated via the TPC-H data generation tool *. The two tables that will be used are the SUPPLIER and the LINEITEM, respectively, the smallest (10k rows) and largest (60M rows) tables. The size (number of rows) of a table depends on the Scale Factor (SF), that can be varied to progressivly change in the table size. 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
  - L_LINENUMBER : integer
  - L_QUANTITY : decimal
  - L_EXTENDEDPRICE : decimal
  - L_DISCOUNT : decimal
  - L_TAX : decimal

*Available at https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

- 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

Considering the different structure of the two tables used (i.e., number of columns and data types) comparison across different tables cannot be done using the the selected metrics, i.e., latency (seconds) and throughput(rows/second). For this reason, the system evaluation will only consider same tables on different configuration. This project used five table variations to benchmark the system integration. SF was varied to obtain a table at each significant order of magnitude from 10k to 60M 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 = 6000000 rows
5. *lineitem_sf10*: size = 60000000 rows

Lastly, despite no information are given about the row and storage size (bytes) ratio, in accordance to Section 3.2.2, this Section presentes comprehensive information on the data, including how to retrieve it and how it is composed. These gives to the reader the ability of calculating the storage occupancy of each table, depending on their prefered unit of measure.

3.2.4 Experimental design

The experiments aim to highlight the differences between the current legacy system and the the newly implemented system based on the PyIceberg library. Two experimental pipelines were designed to isolate the performance of those two systems:

1. **Legacy pipeline**: is the current Hopsworks feature store which stores data in 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. This is described in Sections 2.5.1-2.5.2.

2. **PyIceberg - HopsFS**: is the system implemented in Chapter 4, which stores data in Iceberg Tables. This systems uses a pipeline based on PyIceberg, SQLite, and Arrow Flight to both write to and read from the Iceberg tables, saved on HopsFS. This is described in Sections 2.5.3-2.5.4.

The experiments will verify how the performance will change based on different CPU resources provided: one, two, four, and eight cores, according to the typical Hopsworks use case (Section 3.2.2). Each time, the experimental environment will be modified, creating a new Jupyter server where the host the experiments. The data used for experiments, as described in Section 3.2.3, will come from two different tables, modified according to a SF, for a total of five times for each table. For the writing experiments conducted on the legacy system, different parts of the whole writing process will be measured, to verify how different parts of the legacy system will scale in the different environments described above. Those two parts are the upload and the materialization, which are explained in Section 2.5.1. This will also help defining whether and what part of the legacy pipeline is a bottleneck.

In conclusion, the experiments conducted will be a total of two (pipelines) times four (CPU configurations) times five (tables) times two (read and write operations), thus eighty experiments, performed each fifty times, to ensure statistical significancy to the results.

3.2.5 Experimental environment

The experimental environment consists of a physical machine in Hopsworks AB' offices, 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 dedicated during the experiments. The machine mounts about 5.4 TBs of SSD memory, allowing for fast read and write speed, respectively 2.7 GB/s, and 1.9 GB/s (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 2048MB, which will be adjusted during the experiments, according to the needs (see Section 5.1.4).

Notica that, despite this experimental environment is virtualized in isolation, it runs on shared resources, thus experiments result might vary depending on the machine's load. To mitigate this, all experiments will be run when the machine load is low (less than 50% of CPU and RAM usage).

Listing 3.1: Output of a *lscpu* bash command on the machine.

```
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)
```

3.2.6 Evaluation framework

The system evaluation framework will evaluate the different system on three key aspects:

1. **Functional requirements**: will be measured by verifying the success or failure of running an experiment. This will not happen by design, since the system integration phase stops only when all functional requirements are met. Those are escribed in Section 3.1.2:

2. **Non-functional requirements**: Consistency and maintainability are mainly addressed during integration, while scalability is measured during the system evaluation experiments. The metric used for measuring this requirement is the throughput, as defined in RQ1.

3. **How does the legacy pipeline compare to the PyIceberg pipeline?** this question answers directly RQ1, measuring the throughput of the pipelines defined in Section 3.2.4. Results are then compared using a visual approach.

3.2.7 Reliability and validity

Experiments result are significant according to their reliability and validity. Each experiment will be performed fifty times to ensure the reliability of the its results on the system performance. This number was agreed to balance the results' consistency and resource efficiency (i.e., time and computing resources), as described in Section 1.5. Secondly, due to the complex nature of the pipelines used, the data distribution of results might vary from one experiment to the other. Thus, a bootstrapping technique will be employed to restore the validity of the data collected. Data will be resampled with substitutions a thousand times, then for each experiment an average measure, altogether with a confidence interval.

3.3 System evaluation - Iceberg vs. Delta Lake

This Section firstly explains the steps of this system evaluation processes, aimed at measuring and comparing the performance (latency, in seconds, and throughput, in rows/second) of reading and writing Iceberg and Delta Lake tables on HopsFS. Those experiments will be conducted respectively on the PyIceberg-based system, integrated in this thesis work, and on the delta-rs-based system, implemented in a related work [**?**]. Then, it presents the evaluation framework used by in this process.

3.3.1 Evaluation process - RQ2

This evaluation process will follow a sequential approach described in Figure 3.3. Each step of this process is related to one of the Gs7–9 associated with the RQ2 in Section 1.4. The relationships between each process activity and Gs are here explained:

1. **Analyze related work results**: this activity maps perfectly to G7, analyzing the results coming from the related work taken as input by this process.

2. **Visualize results**: this activity maps perfectly to G8. It takes as input also PyIceberg experiments results (D1-partial), visualizing thus the experiments' result according to latency and throughput. This activity generates D2, the comparative experiments results complemented with tables and histograms, presented in Chapter 5.

3. **Analyze results**: this activity maps perfectly to G9, analyzing and interpreting the results delivered in D2. This activity contributes to D3, generating the comparative analysis of the experiment results, presented in Chapter 5.

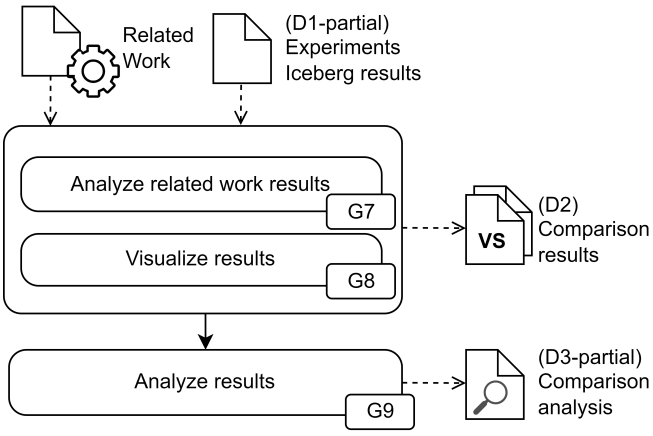

Figure 3.3: Diagram of the system evaluation process partially answering RQ2. Each activity is associated to specific G. The process produces two Ds, the comparative expertiments results (D2) and a comparative results analysis (D3-partial).

3.3.2 Evaluation framework

This system evaluation framework is similar to the system evaluation framework described in Section 3.2.6, but will not focus on functional requirements, as they will be already assessed in previous Section, for PyIceberg, and they have been already assessed in the related work, for delta-rs. Thus, this evaluation framework will evaluate the different system on two key aspects:

1. **Non-functional requirements**: Consistency and maintainability are mainly addressed during integration, while scalability is measured during the system evaluation experiments. The metric used for measuring this requirement is the throughput, as defined in RQ2.

2. **How does the PyIceberg pipeline compare to the delta-rs pipeline?** this question answers directly RQ2, measuring the throughput of the PyIceberg pipeline defined in Section 3.2.4 and the delta-rs pipeline [**?**]. Results are then compared using a visual approach.

Chapter 4

Implementation

This Chapter follows first the system integration process, defined in Section 3.1, describing which integration choices were taken, which components were selected and their usage. Thus it explains how was conducted the experimental part of this thesis, presented in Section 3.2.

4.1 Integration design and usage

The first step of the system integration process consisted of analyzing the Hopsworks system and the PyIceberg tools, as described in Section 3.1. This permitted to identify which parts had to be integrated to satisfy the requirements, thus to be able to read and write on Iceberg Table hosted on HopsFS, via the PyIceberg library. This step outlined the need of a catalog, a query engine, and a FileIO. While the formers are fundamental components of any Data lakehouse architecture explained in Section 2.2.2, the latter is a pluggable module for performing CRUD operations on files, specifically required by PyIceberg.

PyIceberg, being a rather recently developed library, does not provides yet all the features integrations of older Iceberg libraries [**?**], such the Java API. In addition, despite HopsFS expose the same methods of HDFS, the environment where HopsFS was mounted, described in Section 3.2.5, did not allow to use some catalogs. Sections 4.1.1-4.1.2 describe the choices taken for both catalog (SQL Catalog) and query engine (DuckDB), describing the reason behind those choices and the problem encountered. Regarding the FileIO, since there was a single compatible option (PyArrowFileIO), this component could not be subject of any design comparison. Lastly, Section 4.1.3 describe how instanciate the integrated system, and how to perform operations over it.

4.1.1 Catalog choice

At time of development, the available catalogs were:

- **REST**: it is supported by HopsFS. However, since this would have need to develop the interface from scratch, thus was discarded as not fulfilling the maintainability not-functional requirement, described in Section 3.1.2.
- **AWS Glue**: it is supported by HDFS, but it did not pass the integration test with HopsFS, thus was discared. Additionally, since it is a proprietary solution of AWS, this would have lowered the reproducibility of the experiments later conducted, due to the additional costs of this solution.
- **AWS DynamoDB**: it is supported by HopsFS. Was however discarded, for the same reproducibility reason explained above.
- **HMS**: it is supported by HopsFS. HMS is however a complex tool, developed to be tightly integrated with MapReduce and Spark environment, and perhaps perform its best in large-scale data scenarios. This was discared since not matching with purpose of avoiding Spark and its environment, and the industrial use case described in Section 3.2.2. Furthermore, this did not fulfill the maintainability not-functional requirement.
- **SQL Catalog**: it is supported by HopsFS, and it could be instanciated on a SQLite database supported by PyIceberg. This was the choice for the system integration, as it is an open-source catalog, the most light-weight option among the alternatives, and needs few lines of code to be used, fulfilling both the not-functional and the functional requirements. This solution suits perfectly small-scale scenarios such this thesis' use case, but it does not fit a large-scale scenario. Furthemore, the specific SQLite database is not built for concurrency.

4.1.2 Query engine choice

In the choice of the query engine, all the candidates are known to be suitable for both HopsFS, since this all of these engines are supported by Hopsworks AI Data Platoform, which uses HopsFS are data storage layer. At time of development, the available query engines were:

- **AWS Athena and Snowflake**: both were discarded since they are proprietary solutions. This would have lowered the reproducibility of the experiments later conducted, due to the additional costs of this solution.
- **Spark**: this was discarded since it direclty violates the purpose of this project, i.e. create a Spark alternative to read and write data on OTF stored on HopsFS.
- **Presto, Trino, Flink**: were discarded as designed to perform their best in large-scale data scenarios, thus it did not suit the industrial use case described in Section 3.2.2. Additionally, it did not fulfill the maintainability not-functional requirement, describe in Section 3.1.2.
- **DuckDB**: was the chioce for the system integration, as it is a portable open-source OLAP-DBMS, which proved to be the best performing engine in related work on small-scale scenarios [**?**, **?**].

4.1.3 Usage

Once selected PyArrowFileIO as FileIO, SQLite as support to SQL Catalog, and DuckDB as query engine, all the libraries and dependencies are directly managed by the installation of the PyIceberg library, using the command `pip install pyiceberg[pyarrow,duckdb,sql-lite]`, as described on PyIceberg documentation [**?**]. This integration supports all PyIceberg methods, but this Section will focus only on the methods used for the experiments. Listing 4.1 describe how to instanciate an Iceberg catalog using SQLite, to enable metadata management on HopsFS (or HDFS), and how to create a namespace and a table within the namespace. Following this, an example of write operation on HopsFS (or HDFS) is described in Listing 4.2, while Listing 4.3 provides an example of read operation on HopsFS (or HDFS).

Listing 4.1: Instanciating an Iceberg catalog using SQLite

```
from pyiceberg.catalog.sql import SqlCatalog

catalog = ("default",**{
        "uri" : "sqlite:///catalog_path",
        "warehouse" : "hdfs_path",
        "hdfs.host" : "hdfs.host" })
catalog.create_namespace("ns")
table = catalog.create_table(
                "ns.table",
                schema=your_df.schema,
                location="hdfs_path",)
```

Listing 4.2: Writing a DataFrame with PyIceberg on an Iceberg Table stored on HopsFS (or HDFS).

```
import pandas as pd
from pyiceberg.catalog import load_catalog

df = pd.DataFrame({"num": [1, 2, 3],
                   "letter": ["a", "b", "c"]})
catalog = load_catalog()
table   = catalog.load_table("ns.table")
table.append(df)
```

Listing 4.3: Reading a table with PyIceberg from an Iceberg Table stored on HopsFS (or HDFS).

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog()
table   = catalog.load_table("ns.table")
df      = table.scan().to_arrow()
```

4.2 Experimental setup

As defined in Section 3.2.4, experiments run different system configurations with five tables fifty times per experiment. The run time of the experiments,

i.e., the read and write latency, was measured using two different approaches. This decision is motivated by the need to measure accurate results, while it was impossible to use 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 4.4, 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, and was used to conduct the experiments about the read operations of legacy system, described in Section 2.5.2.

Listing 4.4: Timeit usage to measure the time to read from Iceberg table stored on HopsFS.

```
import timeit
SETUP_CODE= '''
from pyiceberg.catalog import load_catalog '''

TEST_CODE= '''
catalog = load_catalog()
table   = catalog.load_table("ns.table")
df      = table.scan().to_arrow() '''

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

Running in an isolated Python environment, the timeit approach had two limitations: (1) it was not possible to breakdown the two main steps performed by the legacy system when performing a write operation, (2) it was not possible to separate the timings of the creation of a new catalog and of the read operations in the new Iceberg system, since timeit cannot access variables declared elsewhere (this catalog creation/deletion was necessary at every step, to the machine to perform caching). Thus, for those two cases, see Sections 2.5.1 and 2.5.4, a second approach was implemented. This approach consisted in recording the time before and after the script run, using the function time, from the Python standard library called time, and an example of this approach is showed in Listing 4.5. The usage of two different approaches 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 4.5: A simple time difference approach the time to read from Iceberg table stored on HopsFS.

```
import time
from pyiceberg.catalog import load_catalog

catalog = load_catalog()

before = time.time()
table  = catalog.load_table("ns.table")
df     = table.scan().to_arrow()
after  = time.time()

reading_time = after - before
```

Chapter 5

Results and Analysis

This Chapter is the output of both system evaluation processes defined in Section 3.2–3.3. It starts with Section 5.1, which presents the results of both the experiments and the comparison in tables, histograms, and written descriptions. Then, Section 5.2 complements the chapter by analyzing and discussing the major findings.

5.1 Major results

This Section presents the main results of the one eighty experiments performed as defined in Section 3.2.4, and the fourty experiments performed in the related work [**?**], conducted with the same experimental desing and environment. 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 (measured in seconds) and throughput (measured in rows/secodns), defined in RQ1.

It must be noted that the results are visualized using a log scale for clarity, as results differing from more than one significant figure would not be clearly interpreted using a linear scale, in a histogram representation. For each measurement, a 95% confidence interval was calculated using the bootstrapping technique mentioned in Section 3.2.7. For better readibility, this interval is not reported in the histograms of this Section, but it is reported in Appendices C–D, with a histogram and a table for each experiment expressed in both metrics.

Of the two metrics, only latency was measured during the experiments, while throughput was calculated with the formula present in Section 3.2.1. Latency and throughput are inversely related by a constant factor, as all

experiments were conducted with fixed-size tables. This implies that halving the latency results in a doubling of throughput, while reducing latency to a quarter results in a quadrupling of throughput. Since the results are described according to both metrics, this introduces redundancy. To minimize repetition, trends will primarily be discussed in terms of latency, with throughput trends highlighted only when they exhibit significantly different behavior.

5.1.1 Write experiments

Figures 5.1–5.2, and Tables 5.1–5.2 report the results of the write experiments performed. The results are expressed in latency in Figure 5.1 and Table 5.1, while are expressed in throughput in Figure 5.2 and Table 5.2. The experiments were performed on the two systems defined in Section 3.2.4 and on the delta-rs based system defined in the related work [**?**]. The five tables of different sizes being written were defined in Section 3.2.3.

Both histograms, i.e., Figures 5.1–5.2, report the results of the experiment performed with one CPU core. Instead, Tables 5.1–5.2 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.

Legacy pipeline vs. PyIceberg

The latency on write operations measured using the PyIceberg pipeline is more than fourty times lower than the latency measured using the legacy pipeline, in all the experiments. This trend is more prominent for bigger tables (6M and 60M rows), where latency measured using the PyIceberg pipeline is less than one hundredth of the latency measured usign the legacy system.

PyIceberg vs. delta-rs

The latency on write operations measured using the PyIceberg pipeline is lower than the latency measured using the delta-rs pipeline, in all the experiments. This trend is more prominent for the biggest table (60M rows), where latency measured using the PyIceberg pipeline is around seven times lower than the latency measured using the delta-rs pipeline, while it is lesser prominent the smaller are the tables, with approximatively equal latency for the smallest table (10K rows).

Table 5.1: Write experiment results expressed as latency. Experiments performed with multiple CPU cores 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)
Hudi Legacy	10K	50.237 94	−0.98	−2.06	−1.97
	100K	59.556 05	−0.37	0.07	−1.23
	1M	112.177 73	3.22	3.00	2.49
	6M	511.849 08	7.52	5.84	7.01
	60M	2 716.209 39	13.81	13.63	14.41
Iceberg PyIceberg	10K	1.204 06	0.75	0.73	−1.18
	100K	1.265 24	5.19	0.83	1.32
	1M	1.717 70	−0.22	5.01	−0.08
	6M	3.576 49	0.28	0.88	1.23
	60M	24.609 89	4.48	2.46	4.01
Delta Lake delta-rs	10K	1.250 88	−0.95	2.72	−8.70
	100K	1.368 00	4.44	2.30	5.54
	1M	9.382 31	9.19	10.27	11.59
	6M	19.751 49	17.47	17.86	20.39
	60M	177.194 58	24.29	29.99	31.21

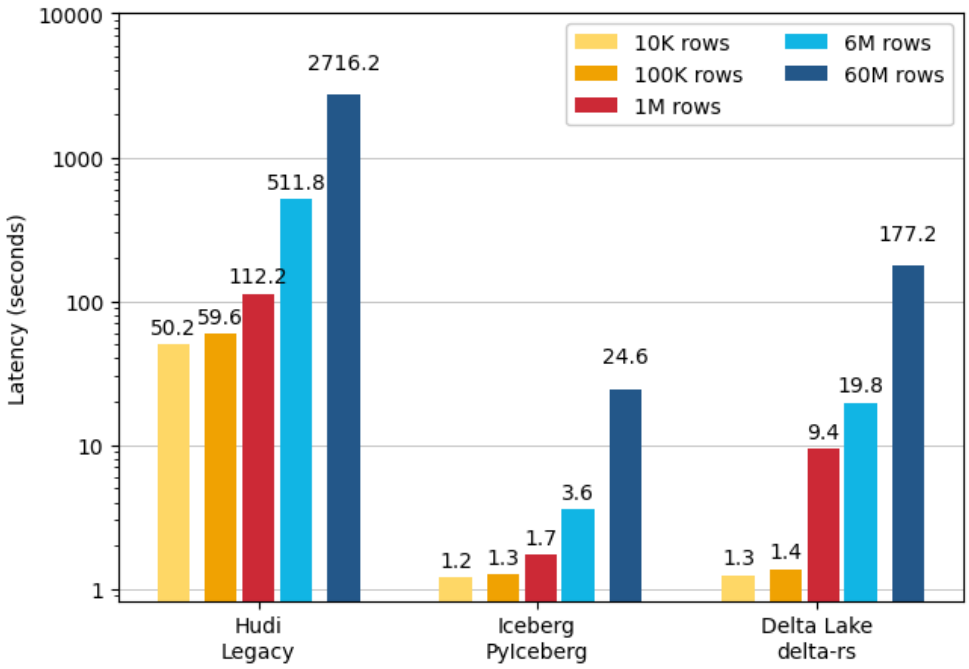

Figure 5.1: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with one CPU core.

Table 5.2: Write experiment results expressed as throughput. Experiments performed with multiple CPU cores 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)
Hudi Legacy	10K	0.199 05	−0.97	−2.02	−1.93
	100K	1.679 09	−0.37	0.07	−1.21
	1M	8.914 43	3.33	3.10	2.56
	6M	11.722 21	8.13	6.20	7.54
	60M	22.089 61	16.02	15.78	16.84
Iceberg PyIceberg	10K	8.305 21	0.75	0.74	−1.16
	100K	79.036 58	5.47	0.84	1.33
	1M	582.172 95	−0.22	5.28	−0.08
	6M	1 677.621 61	0.28	0.89	1.25
	60M	2 438.044 52	4.69	2.52	4.18
Delta Lake delta-rs	10K	7.994 35	−0.94	2.80	−8.00
	100K	73.099 24	4.65	2.35	5.87
	1M	106.583 58	10.12	11.45	13.11
	6M	303.774 53	21.16	21.75	25.61
	60M	338.610 80	32.08	42.83	45.37

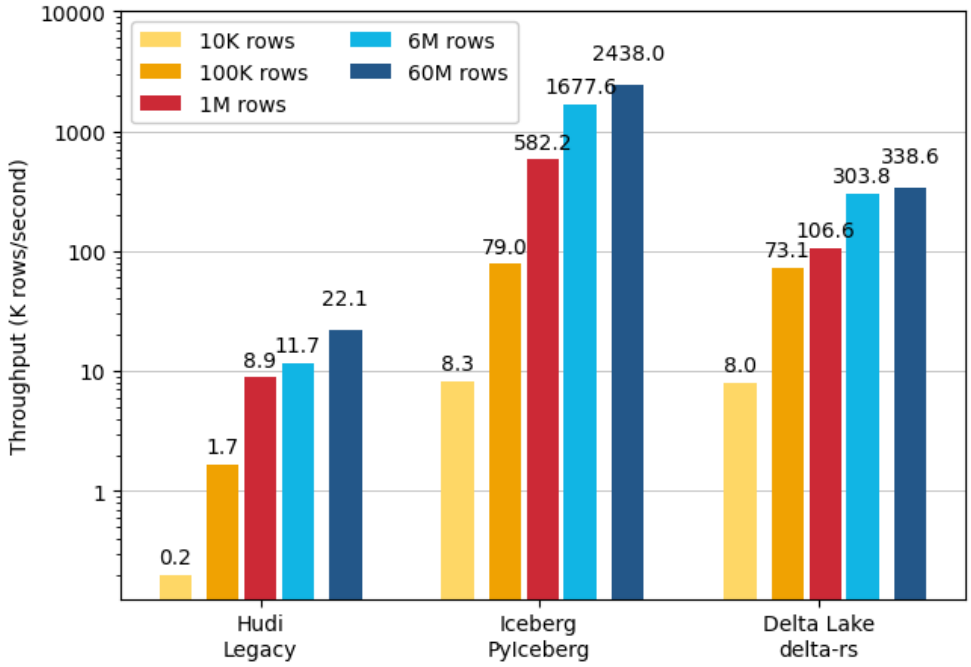

Figure 5.2: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with one CPU core.

Change of performance as the CPU cores increase

The experiments with increased CPU cores showed different results across the different pipeline technologies. For the legacy system, significant latency reductions were only observed for the largest table (60M rows), with a decrease of approximately 14%. Smaller tables (10K and 100K rows) actually saw slight increases in latency, and even the larger tables (1M and 6M rows) only improved by a maximum of 7%. PyIceberg saw some benefit from additional CPU cores, but the improvement was limited, with only a maximum of 5% descrease in latency for some tables (100K, 60M). The most substantial gains were seen with delta-rs, where the larger tables (6M and 60M rows) experienced considerable improvements of 20-30%, and smaller tables (100K and 1M rows) only saw modest latency decreases of 5-10%.

5.1.2 Read experiments

Figures 5.3–5.4, and Tables 5.3–5.4 report the results of the read experiments performed. The results are expressed in latency in Figure 5.3 and Table 5.3, while are expressed in throughput in Figure 5.4 and Table 5.4. The experiments were performed on the two systems defined in Section 3.2.4 and on the delta-rs based system defined in the related work [**?**]. The five tables of different sizes being read were defined in Section 3.2.3.

Both histograms, i.e., Figures 5.3–5.4, report the results of the experiment performed with one CPU core. Instead, Tables 5.3–5.4 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.

Legacy pipeline vs. PyIceberg

The latency of read operation measured using the PyIceberg pipeline results from 55% up to sixty times lower than the latency measured using the legacy pipeline. The most prominent reduction (sixty times) is obtained in the table containing 100k rows, while the less prominent (55%) is obtained with the largest table (60M rows). Regarding the other tables, the PyIceberg pipeline has around fifteen times lower read operation latency than the legacy system.

PyIceberg vs. delta-rs

The latency on read operations measured using the PyIceberg pipeline is completely comparable with the latency measured usign the delta-rs pipeline,

Table 5.3: Read experiment results expressed as latency. Experiments performed with multiple CPU cores 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)
Hudi Legacy	10K	0.631 44	1.05	−0.76	0.65
	100K	2.650 43	−0.49	0.40	−0.45
	1M	8.592 96	−0.16	−1.76	2.80
	6M	33.525 80	0.44	0.23	0.26
	60M	33.690 31	0.18	0.09	1.63
Iceberg PyIceberg	10K	0.017 23	−6.36	−72.94	1.53
	100K	0.046 87	7.89	8.62	8.95
	1M	0.430 18	37.37	45.75	46.04
	6M	2.123 31	38.96	52.50	58.34
	60M	21.819 55	38.07	53.66	59.49
Delta Lake delta-rs	10K	0.053 45	22.90	18.50	19.29
	100K	0.057 63	0.73	3.78	5.18
	1M	0.538 78	56.62	64.81	67.69
	6M	1.949 47	53.41	72.76	74.50
	60M	22.981 86	50.35	75.72	87.19

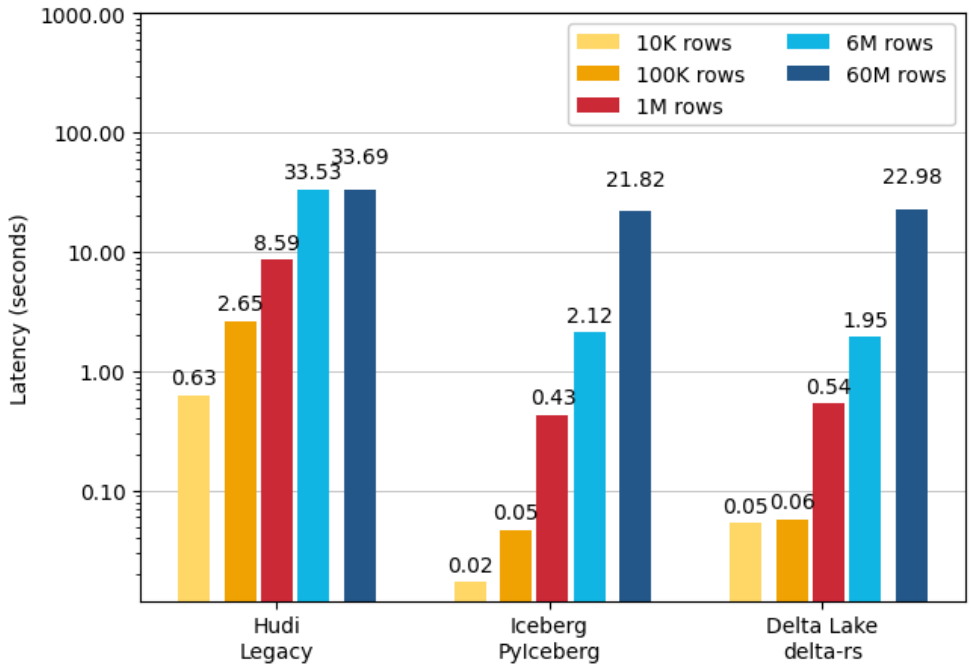

Figure 5.3: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with one CPU core.

Table 5.4: Read experiment results expressed as throughput. Experiments performed with multiple CPU cores 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)
Hudi Legacy	10K	15.836 73	1.06	−0.75	0.66
	100K	37.729 79	−0.48	0.40	−0.45
	1M	116.374 31	−0.16	−1.73	2.88
	6M	178.966 62	0.44	0.23	0.26
	60M	1 780.927 64	0.18	0.09	1.65
Iceberg PyIceberg	10K	580.480 03	−5.98	−42.18	1.56
	100K	2 133.403 62	8.57	9.43	9.83
	1M	2 324.629 63	59.66	84.33	85.33
	6M	2 825.776 91	63.83	110.53	140.02
	60M	2 749.827 53	61.47	115.82	146.85
Delta Lake delta-rs	10K	187.081 71	29.70	22.69	23.89
	100K	1 735.080 31	0.74	3.93	5.47
	1M	1 856.033 25	130.51	184.19	209.48
	6M	3 077.758 01	114.64	267.09	292.19
	60M	2 610.755 20	101.42	311.87	680.55

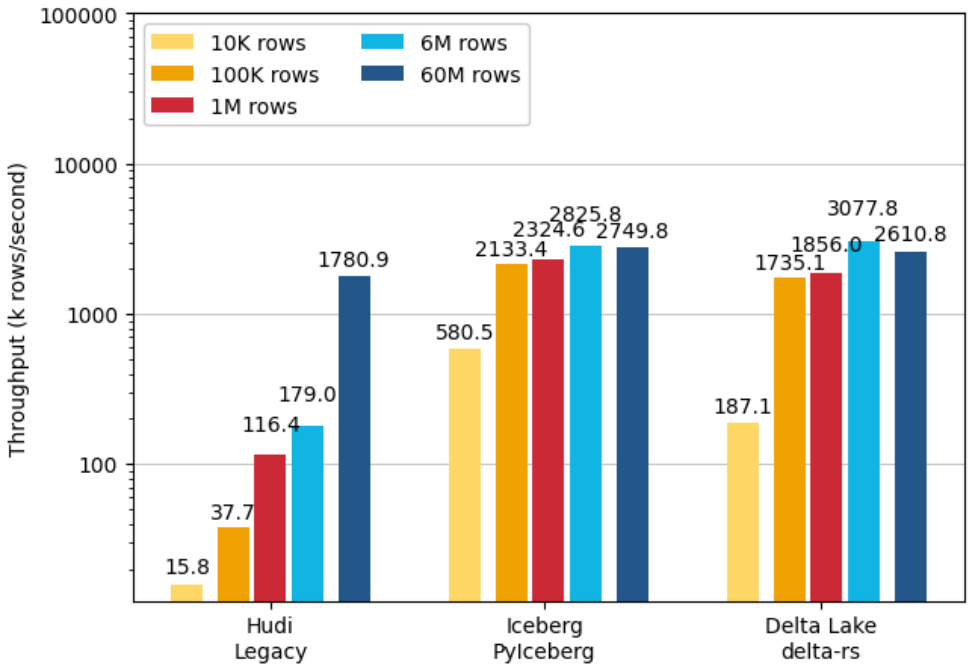

Figure 5.4: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with one CPU core.

if not for the smallest table (10K), where PyIceberg pipeline latency was three times lower than delta-rs pipeline latency. For the other tables, The latency on write operations measured using the PyIceberg pipeline is lower than the latency measured using the delta-rs pipeline, in all the experiments. This trend is more prominent for the biggest table (60M rows), where latency measured using the PyIceberg pipeline is around seven times lower than the latency measured using the delta-rs pipeline, while it is lesser prominent the smaller are the tables, with approximatively equal latency for the smallest table (10K rows).

Change of performance as the CPU cores increase

The experiments with increased CPU cores showed different results across the different pipeline technologies. In the legacy pipeline, increasing the number of CPU cores had minimal impact on latency reduction. The observed improvements remained below 1% for most table sizes, with some cases even showing slight increases. PyIceberg saw some benefit from utilizing multiple CPU cores. While the smallest table (10K rows) showed a slight increase in latency, all other tables benefited from latency reductions. For tables larger than 100K rows, the reduction was more pronounced, reaching around 39% for mid-sized and large tables (1M, 6M, and 60M rows). Delta-rs showed the most substantial gains in read latency with additional CPU cores. For small tables (10K and 100K rows), the letancy reductions remained below 5%, but for tables larger than 100K rows, the reduction was significant, reaching around 55% for mid-sized table (1M rows) and around 50% for larger tables (6M and 60M rows). The trend continued with additional cores, further enhancing performance as table sizes increased.

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 two different steps explained in Section 2.5.1. The breakdown is proposed for all five tables defined in Section 3.2.3. 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 (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 (10K and 100K) rows, while it grows following a similarly linear pattern

Table 5.5: Contributions to the write latency of the upload and materialization steps in the legacy pipeline. Experiments performed with multiple CPU cores 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	2.50	47.73	4.50	−1.28	4.53	−2.46	4.65	−2.32
100K	3.67	55.91	6.36	−0.79	5.56	−0.30	6.28	−1.68
1M	22.59	89.60	17.51	−0.37	14.88	0.01	16.49	−1.03
6M	244.60	267.25	15.83	−0.10	13.47	−1.15	15.09	−0.39
60M	2 438.22	278.16	15.34	0.43	15.17	0.20	15.96	0.86

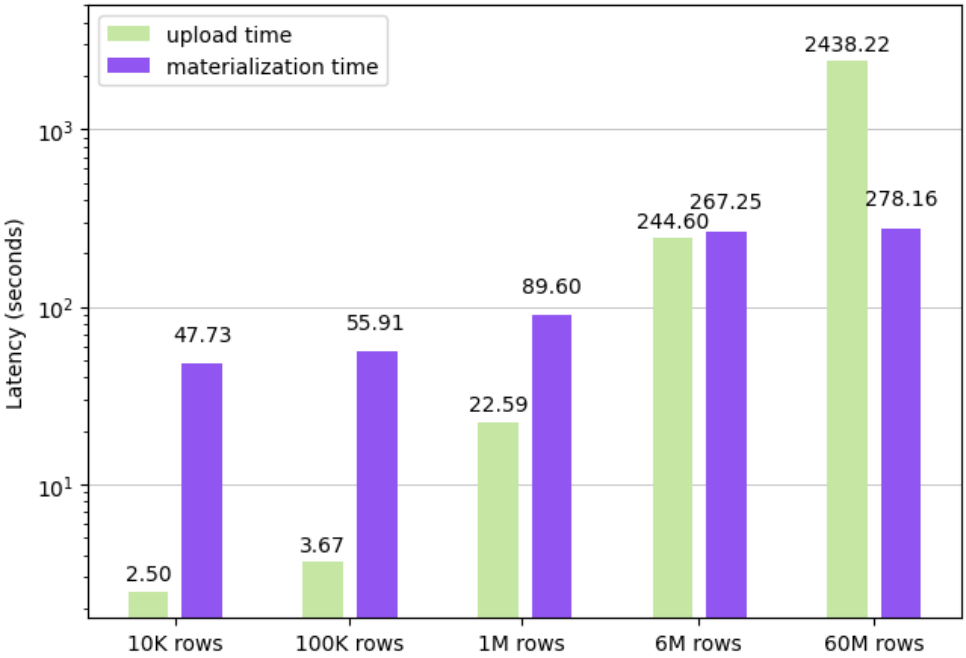

Figure 5.5: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with one CPU core.

in larger tables (100K, 6M, and 60M rows). This radically changes the proportion between the upload and materialized time contributions, making the upload time 90% of the total write latency for the largest table (60M rows). Differently, the materialization time contribution makes up 95% of the total write latency for the smallest table (10K rows), but then its absolute value does not increase by more than a significant figure even if the table size is increased by three significant figures.

Analyzing the impact of multiple CPU cores, the upload time benefits significantly from increased core availability, particularly for larger tables, where latency decreases by 15% compared to just 4% for smaller tables. In contrast, the materialize time remains largely unaffected, showing only minor fluctuations, with latency varying by 1-2%.

5.1.4 In-memory resources usage

The resources of the experimental environment, defined in Section 3.2.5, were adjusted to match computational needs. The write operations were demanding on the available RAM resources, requiring up to 24 GBs to operate with the larger tables (6M and 60M rows). Thus, on were allocated 32768 MBs of RAM, to avoid slowing down operations.

5.2 Results analysis and discussion

This Section discusses the major results, described in detail in Section 5.1, explaining what they mean, which are their implication for Hopsworks AB, and for the research area. Following, this Section contains the author's considerations over the legacy system and the PyIceberg library, which were developed while conducting this thesis research. Additional consideration regarding the delta-rs library can be found in the related work's discussion [**?**].

5.2.1 Discussion on major results

The results presented in Section 5.1 disclose a substantial difference in the performance (both latency and throughput) of the legacy system, and the newly implemented system using the PyIceberg library. Despite the two systems behaved were tested on two different tasks, i.e. write and read operations, the PyIceberg-based system integratedin this thesis shown from 15 to 140 times lower latency than the legacy system, in all experiments conducted using tables

from 10K to 6M rows. Furthermore, the results reveal that the PyIceberg and delta-rs systems differs generally much less, with the PyIceberg system outperforming the delta-rs system only in write operations with table bigger than 1M rows, where the system latency was up to seven times lower than its counterpart. These findings confirm the hypothesis that alternatives to a Spark-based system exist, and that those are preferable when operating on small tables (from 10K to 60M rows). Furthermore, the large performance differences observed indicate substantial room for optimization in the legacy system, on both read and write operations.

Going more in depth in the results of the write experiments, the newly implemented Iceberg system, has a latency from from 40 to 140 times lower than the legacy system on all the tables sizes. This system performed almost identically to the delta-rs-based system on smaller tables (10k and 100k rows), while outperformed it on larger tables (1M, 6M and 60M rows), with a latency up to seven times lower than its counterpart. In the experiments using multiple CPU cores, the delta-rs system experienced latency decreases up to 30% , while the PyIceberg system latency decreased of maximum 5%. However, despite the better scalability results, the delta-rs system still underperform compared to the PyIceberg, for all the table sizes and CPU settings used in this thesis research. The above findings clearly demonstrate that the new Iceberg-based system, in the industrial use case defined in Section 3.2.2, is a better alternative to legacy Spark-based system, while also confirming the need for Spark alternatives when operating on small tables (from 10K to 60M rows). Those alternatives could not only increase the performances, but also reduce computational costs, system complexity, and deployability, since managing a Spark cluster for small size data is expensive and more complex, and requires as well JVM dependencies, which cannot be deployed from any Python environment manager, such as pip, or conda.

Considering now the read experiments, the results show smaller performance differences, between the legacy system and the new Iceberg system. When considering the largest table (60M rows), the new system exhibits only 55% latency reduction, however for all the other tables the Iceberg system has a latency from 15 to 56 times lower than the legacy system latency. The delta-rs pipeline follows a very similar trend, despite slighlty underperforming the Iceberg pipeline, with a maximum of 10% latency increase. The reason behind this different trend between new system and legacy system's performance, compared to what seen for the write operation, is that the legacy system, despite it is already using an alternative to Spark, cannot efficiently scale down resources when reading small size tables. The Spark alternative employs

a combination of Arrow Flight and DuckDB, which is instanciated within the Hopsworks cluster with a predefined amount of dedicated resources. Furthermore, in the experiments using multiple CPU cores, the delta-rs system scales better than the PyIceberg system, experiencing latency decreases of respectively 55% and 39%, while the legacy system is not impacted at all. The above outcomes shows that Hopsworks AB has still room for improvement even for the read operation, where employing PyIceberg or delta-rs system would increase the system overall flexibility to different scales of data and to multiple CPU cores settings, at the price of moving computation on the client side.

All the above findings impact significantly Hopsworks AI, direclty affecting the Hopsworks feature store, which is their main product. The results recommend the adoption of one of those two new systems, over the current system, using Apache Hudi and Spark to implement an offline feature store, or the extend this system with one or both the new systems. This would allow the users of the feature store, depending on their use case (data scale), their computational resources (CPUs), and their habits, to choose the preferred datalakehouse format to use. This would resonate with the Hopsworks AB objective of having products accessible from and integrable with multiple tools, and could be the starting point of possible further integration with currently developing technologies, such Apache XTable [*]. For a complete replacement of the legacy system across different use cases, further experiments should be conducted. The author envisions that beyond a certain data threshold, the Spark-based system may still outperform the new solutions due to its scalable distributed nature.

Perhaps the industrial use case described in Section 3.2.2 limits the results and findings to specific data scales, computational resources, and operations. Modifying one or more of these variables could lead to different outcomes, meaning the findings cannot be fully generalized. Additionally, since this thesis was conducted in collaboration with the company developing the evaluated product, some degree of bias is inevitable. Nonetheless, the results align with research expectations, reinforcing the idea that Spark alternatives should be considered in scenarios of small size (10K to 60M rows) data processing. Additionally, these findings highlight the need for further research and experimentation on Spark alternatives, to which thousands developers already contributes everyday.

[*]XTable repository available at https://xtable.apache.org/

5.2.2 Considerations on the legacy system

The legacy system, described in Sections 2.5.1-2.5.2, consists of a multi-layered architecture that employs Spark as compute engine and Hudi as OTF. This system is modeled to support multiple users, each having different use cases and resources. This could create several bottlenecks, depending on the scenario, that goes far beyong the high latency of instanciating a Spark cluster. An example for this issue is uses Kafka, which is used as single data entry-point in the Hopsworks feature store. Using Kafka as single entry-point ensures data consistency between the online feature store, working with streaming data, and the offline feature store, working on batch data. However, this creates a performance bottleneck for this infrastructure as data increases, needing to make a piece the full table in single rows, to save each of those as a single message. This was clearly visible in the experiments conducted in this thesis work, described in Figure 5.5 and Table 5.5, where Kafka accounted for more than 95% of the latency of the write operations. Now that these experiments outlined this problem, more research is needed to find an effective solution. Some approaches might reduce the overall latencies: (1) implement micro-batching mechanism, altogether with using columns-based format for data exchange; (2) enable client multi-processing, to support multiple settings with multiple CPU cores; (3) refine trade-off between partitions, topic replicas and brokers.

5.2.3 Considerations on the PyIceberg library

The system integration between Iceberg and HopsFS, described in Section 4.1, was possible thanks to the PyIceberg library, developed by the Iceberg open-source community for Python access to Iceberg Table. It was possible to create a Spark alternative system thanks to the high level of integrability of the library and the overarching OTF (compared to its counterparts, as highlighted in A.2), and thanks to several other online and book sources documenting this framework and library. The author's recommendation on this regard are: (1) the Iceberg and PyIceberg maintainers should revise their current documentation and integrate it with new findings, thus developers can directly access all the information needed from Iceberg webpage, without the need of buying a book, or of relying on documentation created by software companies that integrates Iceberg in their products; (2) intensively investigate solution for efficient local resources management, that might make Iceberg libraries better adaptable to changing local environment, as delta-rs shown to be able to, during the experiments described in Section 5.1.

Chapter 6

Conclusions and Future work

The conclusions of this thesis are presented in this Chapter. The RQs are revisited and discussed in relation to the experimental findings in Section 6.1, highlighting the contributions of this work. Section 6.2 details the limitations of the project, while Section 6.3 explores potential future research directions.

6.1 Conclusions

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

RQ1: What are the differences in read and write latency and throughput on the Hopsworks offline feature store, between the existing legacy system and the PyIceberg alternative?

RQ2: What are the differences in read and write latency and throughput on the Hopsworks offline feature store, between the new PyIceberg system and the delta-rs alternative?

The work conducted in this thesis answered the first RQ by performing a system integration, and then evaluating the newly implemented system. Thus, the second RQ was answered by evaluating the newly implemented system against the delta-rs alternative.

The first step was integrating HopsFS with the PyIceberg library. To complete this task, the technologies supported by HopsFS and PyIceberg were evaluated and the promising ones tested, according to the functional and non-functional requirements, until the best technology set was selected. This contributed to the creation of the integration details, where the reasoning and findings behind the author's choice for SQLCatalog, DuckDB and

PyArrowFileIO are explained. This can be used by developers to districate in similar complex system architectures, together with the thorough data lakehouse comparison tackled in the Section 2.2.3. Additionally, most of the significance of this work lies in the creation of a production-ready solution, which overcame system dependencies and use case limitations, while fulfilling all specified requirements. The integration of this system into the production Hopsworks feature store shortly after the thesis's release perhaps serves as a powerful testament to its practical value and real-world applicability.

The second and third steps were comparing the legacy system with the newly implemented PyIceberg system, and the latter with the delta-rs alternative. Experiments were designed and conducted to measure the performance of read and write operations on those system, according to two metrics, i.e., latency (in seconds) and throughput (in rows/second). The results presented in Section 5.1 showed the new system accessing Iceberg tables on HopsFS has a latency at least fifteen times lower, for both read and write operations, for the tables contained from 10K to 6M rows. The write experiments demonstrate that the PyIceberg system significantly outperforms the legacy system in all cases, with write latency more than forty times lower across the board. This trend becomes even more striking for larger tables (6M and 60M rows), where the PyIceberg pipeline achieves write latencies that respectively are 140 and 110 times lower of those observed in the legacy system. Thexe experiments were key to identify a critical bottleneck in current Hopsworks feature store architecture, where Kafka accounted for 95% of the wrtie latency. In the other hand, the decrease of performance gap with the growth in size of the tables suggests that for even larger tables a Spark-based system might perform better, but more experiments are needed to verify this hypothesis. A separate comparison between PyIceberg and delta-rs reveals that while PyIceberg consistently achieves lower latencies, the extent of this advantage depends on table size. For the smallest table (10K rows), the two pipelines perform almost identically, whereas for the largest table (60M rows), PyIceberg exhibits a performance edge of nearly seven times lower latency. Similarly, read performance experiments highlight considerable advantages of the PyIceberg system over the legacy system. The latency reduction ranges from 55% for the largest table (60M rows) to as much as sixty times lower for the 100K rows table, with a consistent trend across all table sizes. This observed reduction in the performance gap is likely a result of the legacy system's use of a Spark alternative for reading, which already provides some improvements, but does not optimize resource usage as effectively as PyIceberg, especially at smaller data scales. In contrast, a comparison

between PyIceberg and delta-rs shows that their read performance is nearly identical, except for the smallest table (10K rows), where PyIceberg achieves three times lower latency. This indicates that while PyIceberg is superior for write operations, its advantage diminishes in read operations, with delta-rs performing similarly. Experiments with increasing CPU cores reveal notable differences in scalability between the three systems. The legacy system shows minimal gains from additional CPU resources, while PyIceberg significantly benefits from multi-core execution only on reading operations, reaching 39% performance gains with the larger tables (6M and 60M rows), and a modest 5% for writing operations. The delta-rs, however, scales significantly better, achieving 20-30% performance gains in read operations with the larger tables (6M and 60M rows), as well as latency reductions of 55% for the 1M rows tables and 50% for the larger tables (6M and 60M rows) in write operations.

Overall, these findings reinforce the recommendation to adopt the newly integrated PyIceberg system in the defined use case, as it provides a substantial improvement over the legacy system, either as a replacement or as an extension for Iceberg table users. At the same time, the comparison between PyIceberg and delta-rs suggests that PyIceberg represents overall a better option for single-core machines, particularly for write operations, but delta-rs might be preferable in scenarios where multi-core scalability is crucial.

6.2 Limitations

The limitations of this study primarily stem from constraints in resources, both in terms of time and computational resources, and from in scope, which is closely tied to the defined industrial use case, described in Section 3.2.2.

This research focuses on reducing the read and write latency of the Hopsworks offline feature store, which dictated the choice of technology for implementation and experimentation. As a result, the findings are intrinsically influenced by the specific tools and infrastructure used, particularly given the collaboration with the company developing this technology, lowering their generality. Furthermore, the thesis' scope was shaped by the predefined use case, described in Section 3.2.2, which determined the CPU and data loads used in the experiments. While this helped establish a clear boundary for the research contributions, it also limits the broader applicability of the results, needing further research to validate these findings in different contexts.

Despite access to substantial computational resources, their availability was restricted due the shared nature of the development environment, where other critical workloads had to be prioritized. Additionally, time constraints

limited the number of experiments to reach 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 findings and limitations of this thesis open multiple avenues for future research. The scope of this work was primarily constrained by the specific industrial use case and the available computational resources. Relaxing one or more of these constraints could provide further insights and help generalize the findings across broader applications.

Future experiments should investigate the system's behavior with significantly larger datasets (1B rows, 10B rows, or more). The current study focused on tables up to 60M rows, but exploring a broader data scale could help determine whether there is a threshold where a Spark-based system begins to outperform the proposed alternatives due to its distributed nature. Similarly, additional experiments on different data formats, workloads, and storage solutions–such as AWS S3 or GCS– could further validate the results. This thesis was conducted in collaboration with Hopsworks AB, focusing on optimizing the Hopsworks offline feature store. While the findings are valuable within the specific use case context, described in Section 3.2.2, testing the proposed approaches on other feature store implementations–such as Databricks or Feast–would help assess their broader applicability. As well, new libraries supporting the OTF evaluated in this thesis, like iceberg-rust, should be investigated, as they might further improve performances and scalability, better exploiting local resources and various computational settings.

Lastly, while this thesis focused on PyIceberg and delta-rs as alternatives to the legacy Spark-based system, other OTFs are currently under development, such as Apache Paimon, and performing similar experiments and comparisons with these technologies would broaden the knowledge on the possible alternatives to Spark.

Appendix A

Technology comparisons

This appendix contains the comparative tables for the technologies presented in Section 2.2.1–2.2.3, respectively between the different DBMs developed along their history, and the different OTFs presented in this thesis report.

Table A.1: Comparison of key features of Data Warehouses, Data Lakes, and Data Lakehouses [**?**]. Technologies presented and discussed in details in Section 2.2.1.

Feature	**Data Warehouse**	**Data Lake**	**Data Lakehouse**
Primary Use	Business Intelligence, SQL Analytics	Unstructured data storage, AI/ML workflows	Unified storage for structured/unstructured data, SQL + AI/ML processing
Data Type	Structured	Unstructured	All data types
Query Performance	Optimized for SQL-based queries, fast indexed access	Slow for structured queries, requires extensive data preparation	High performance for both SQL and ML workloads
Schema Enforcement	Strict schema-on-write	Schema-on-read	Flexible schema with enforcement and evolution support
ACID support	Fully supported	Not supported	Fully supported
Scalability	Limited scalability due to high storage costs	High scalability, low-cost storage	Highly scalable with structured query optimizations
Storage Cost	High due to proprietary formats	Low due to object storage	Moderate, optimized for efficiency
Governance	Strong governance, access control, and security	Weak governance, risk of data swamps	Fine-grained governance with access control and security
Key Takeaways	Best for structured data and BI queries. High-performance SQL analytics.	Ideal for cost-effective big data storage. Difficult to manage and query efficiently	Combines the best of warehouses and lakes. Supports both SQL and AI/ML use cases efficiently.

Table A.2: Comparison of key features of Apache Hudi, Apache Iceberg, Delta Lake [**?**, **?**, **?**]. Technologies presented and discussed in details in Section 2.2.3.

Feature	**Apache Hudi**	**Apache Iceberg**	**Delta Lake**
Metadata Implementation	Tabular	Hierarchical	Tabular
Time Travel	snapshots	transaction log	incremental commits
Caching	No	Yes	Yes
Optimization	CoW, MoR	CoW, MoR	CoW
Schema Evolution	Limited	Full	Partial
Partition Evolution	Explicit	Hidden	Explicit
Concurrency Control	Optimistic, Multi-version	Optimistic	Optimistic
Storage Engines	HDFS, AWS S3, GCS, Azure Blob Storage		
File Formats	Parquet Avro ORC	Parquet Avro ORC	Parquet
Catalogs	Hive AWS Glue	Hive AWS Glue JDBC REST	Hive AWS Glue Unity
Query Engines	Spark Flink Presto Hive Impala	Spark Flink Presto Athena Trino Snowflake	Spark Presto Athena Redshift Snowflake
APIs	Java (Spark) Python (Spark) Rust Scala (Spark,Flink)	Java (Spark) Python Rust	Java (Spark,Flink) Python (Spark) Rust Scala (Spark)

Appendix B

System architectures

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

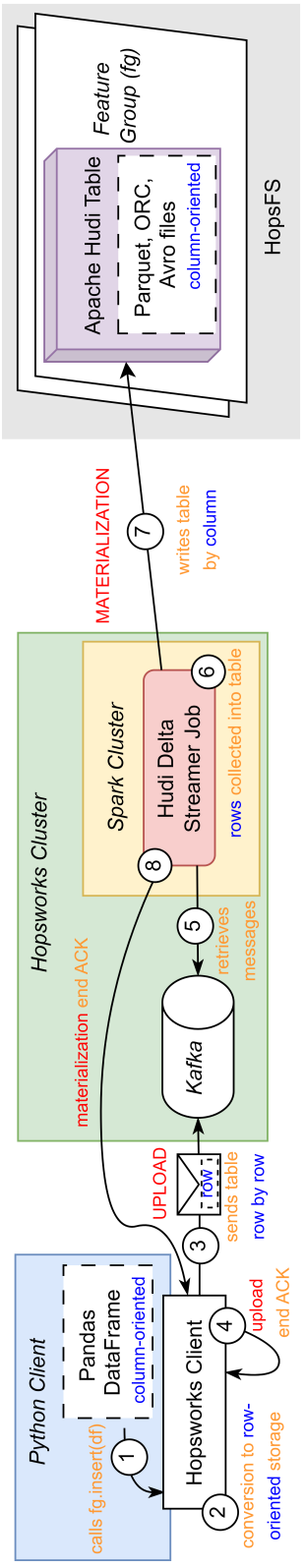

Figure B.1: Legacy system writing a Pandas DataFrame from a Python client to the Hopsworks offline feature store. This image was magnified to enhance visualization.

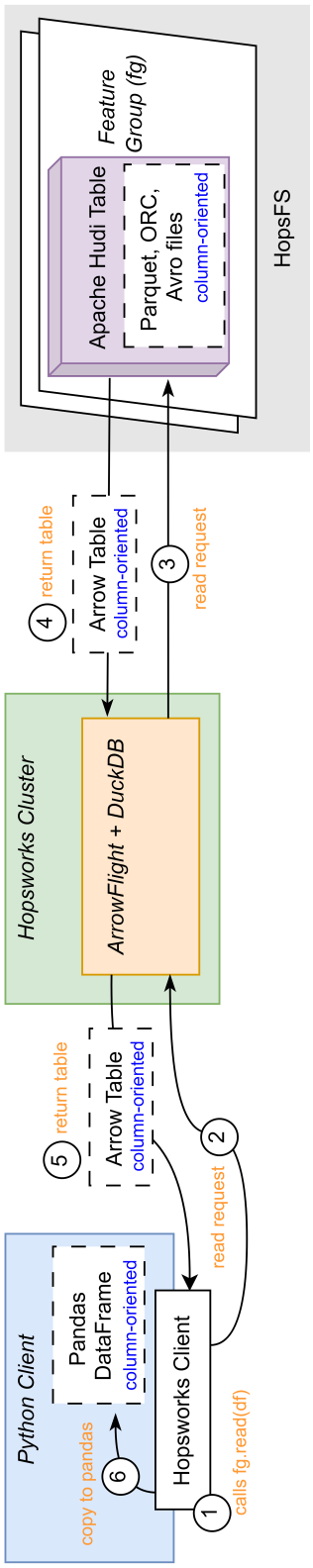

Figure B.2: Legacy system reading a table from the Hopsworks offline feature store and loading it into the Python client's local memory. This image was magnified to enhance visualization.

Appendix C

Write experiments results

This appendix contains all the graphs and the tables related to the write experiments conducted, and of the comparison with the related work. Results are reported first expressed as latency (measured in seconds) and then as throughput (measured in rows/seconds). Latency was directly measured, while throughput was computed from the latency and table size.

Table C.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	high
Hudi Legacy	10K	50.237 94	49.497 73	50.959 89
	100K	59.556 05	58.911 90	60.197 33
	1M	112.177 73	111.398 58	112.961 77
	6M	511.849 08	510.652 80	512.889 00
	60M	2 716.209 39	2 700.807 03	2 732.348 88
Iceberg PyIceberg	10K	1.204 06	1.169 30	1.234 95
	100K	1.265 24	1.221 50	1.299 14
	1M	1.717 70	1.662 75	1.792 99
	6M	3.576 49	3.535 57	3.614 05
	60M	24.609 89	24.453 04	24.785 83
Delta Lake delta-rs	10K	1.250 88	1.238 22	1.264 26
	100K	1.368 00	1.337 73	1.389 56
	1M	9.382 31	9.240 54	9.541 70
	6M	19.751 49	19.386 52	20.111 50
	60M	177.194 58	174.583 94	180.046 79

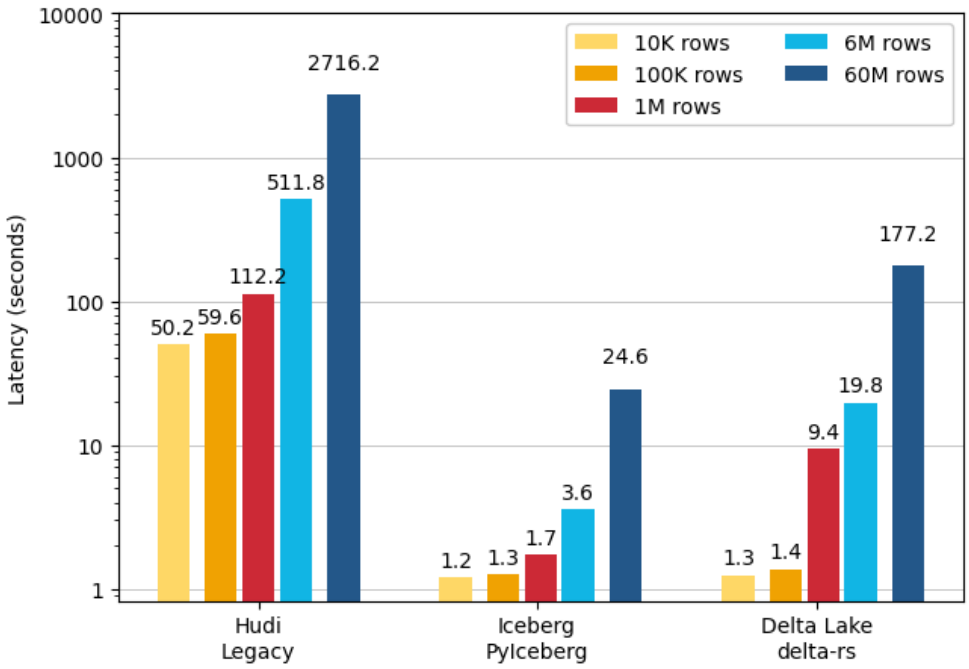

Figure C.1: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with one CPU core.

Table C.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
Hudi Legacy	10K	50.727 79	50.115 08	51.315 97
	100K	59.776 47	59.107 32	60.482 78
	1M	108.564 62	108.016 96	109.069 68
	6M	473.383 10	472.306 35	474.453 90
	60M	2 341.083 59	2 334.158 50	2 347.681 42
Iceberg PyIceberg	10K	1.195 07	1.134 29	1.260 15
	100K	1.199 62	1.141 05	1.252 51
	1M	1.721 48	1.694 02	1.748 27
	6M	3.566 35	3.502 74	3.644 06
	60M	23.507 84	23.464 74	23.546 21
Delta Lake delta-rs	10K	1.262 80	1.251 28	1.276 56
	100K	1.307 21	1.279 33	1.329 99
	1M	8.520 18	8.343 08	8.701 74
	6M	16.301 52	15.920 07	16.669 90
	60M	134.155 67	131.768 12	136.587 94

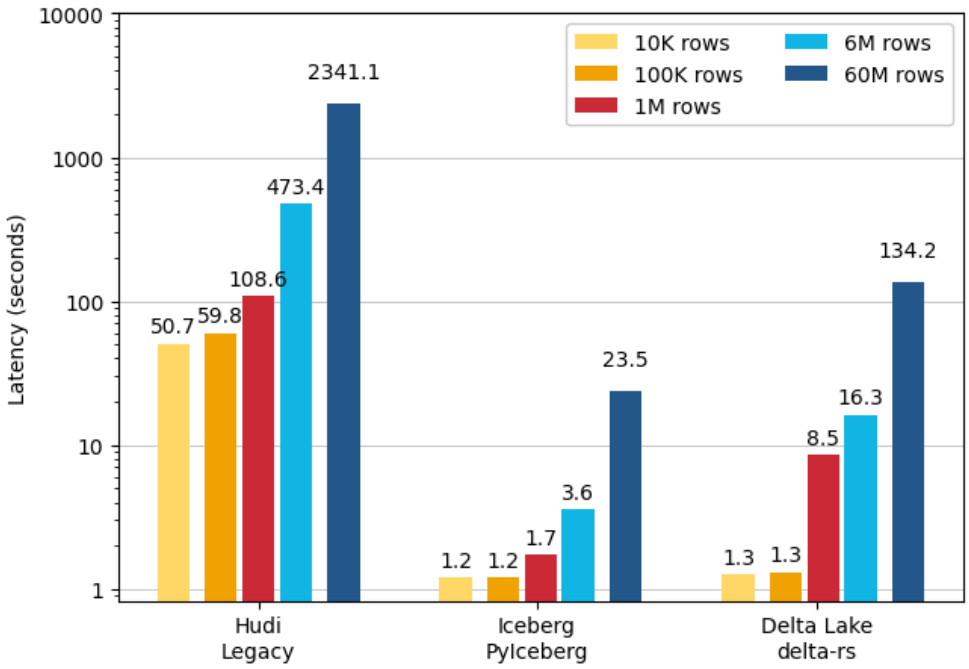

Figure C.2: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with two CPU cores.

Table C.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
Hudi Legacy	10K	51.271 46	50.609 85	51.917 87
	100K	59.514 01	58.855 12	60.137 37
	1M	108.807 23	108.291 44	109.369 79
	6M	481.958 70	480.984 72	482.900 19
	60M	2 346.108 73	2 337.175 80	2 355.212 33
Iceberg PyIceberg	10K	1.195 28	1.161 00	1.226 59
	100K	1.254 75	1.218 73	1.285 07
	1M	1.631 61	1.586 39	1.672 95
	6M	3.544 88	3.514 41	3.571 31
	60M	24.005 35	23.897 06	24.153 70
Delta Lake delta-rs	10K	1.216 84	1.202 38	1.233 06
	100K	1.336 53	1.323 13	1.350 43
	1M	8.418 63	8.245 95	8.603 09
	6M	16.223 45	15.831 16	16.585 56
	60M	124.061 39	121.162 37	126.547 71

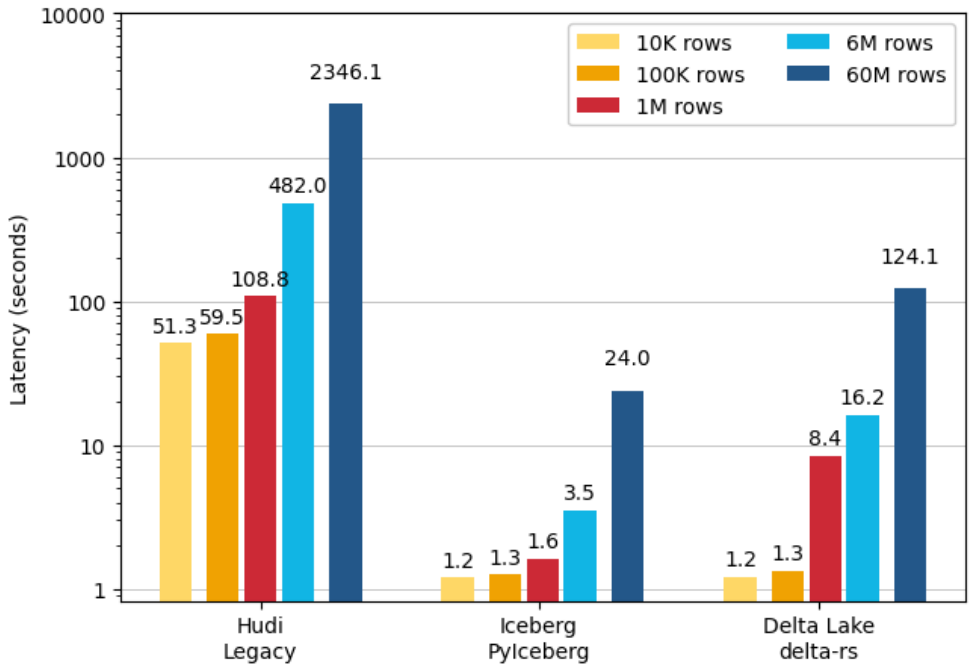

Figure C.3: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with four CPU cores.

Table C.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	high
Hudi Legacy	10K	51.225 98	50.587 30	51.826 30
	100K	60.288 08	59.778 63	60.766 77
	1M	109.381 79	108.911 21	109.852 72
	6M	475.972 84	474.807 88	477.121 64
	60M	2 324.807 40	2 319.160 36	2 331.084 60
Iceberg PyIceberg	10K	1.218 22	1.159 14	1.290 76
	100K	1.248 58	1.219 85	1.276 29
	1M	1.719 08	1.704 10	1.731 50
	6M	3.532 48	3.473 37	3.609 68
	60M	23.623 13	23.560 93	23.700 51
Delta Lake delta-rs	10K	1.359 65	1.248 49	1.570 41
	100K	1.292 20	1.265 57	1.311 20
	1M	8.294 98	8.143 43	8.460 28
	6M	15.724 10	15.239 56	16.176 34
	60M	121.892 14	119.518 99	124.134 25

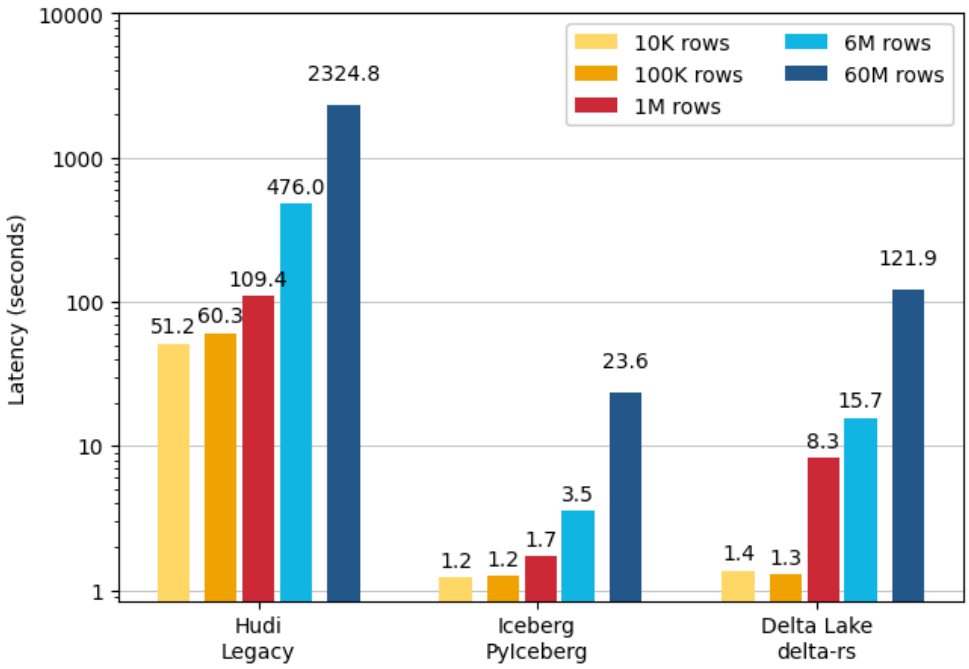

Figure C.4: Histogram in log-scale of the write experiment results expressed as latency. The experiment was performed with eight CPU cores.

Table C.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
Hudi Legacy	10K	0.199 05	0.196 23	0.202 03
	100K	1.679 09	1.661 20	1.697 45
	1M	8.914 43	8.852 55	8.976 78
	6M	11.722 21	11.698 44	11.749 67
	60M	22.089 61	21.959 13	22.215 58
Iceberg PyIceberg	10K	8.305 21	8.097 48	8.552 14
	100K	79.036 58	76.974 09	81.866 64
	1M	582.172 95	557.726 36	601.411 81
	6M	1 677.621 61	1 660.186 13	1 697.038 46
	60M	2 438.044 52	2 420.738 21	2 453.683 12
Delta Lake delta-rs	10K	7.994 35	7.909 77	8.076 11
	100K	73.099 24	71.965 38	74.753 63
	1M	106.583 58	104.803 17	108.218 78
	6M	303.774 53	298.336 72	309.493 33
	60M	338.610 80	333.246 70	343.674 23

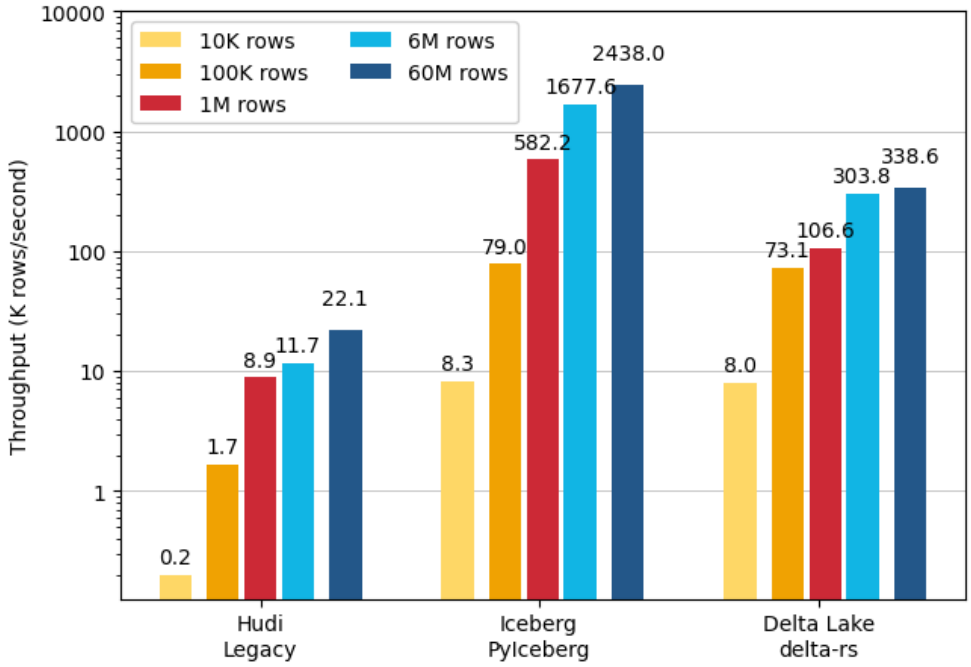

Figure C.5: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with one CPU core.

Table C.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	high
Hudi Legacy	10K	0.197 13	0.194 87	0.199 54
	100K	1.672 90	1.653 36	1.691 84
	1M	9.211 10	9.168 45	9.257 81
	6M	12.674 72	12.646 12	12.703 62
	60M	25.629 16	25.557 13	25.705 20
Iceberg PyIceberg	10K	8.367 70	7.935 56	8.816 10
	100K	83.360 01	79.839 44	87.638 80
	1M	580.895 14	571.995 64	590.311 26
	6M	1 682.394 46	1 646.514 35	1 712.942 34
	60M	2 552.340 16	2 548.181 01	2 557.028 17
Delta Lake delta-rs	10K	7.918 88	7.833 56	7.991 83
	100K	76.498 69	75.188 33	78.166 03
	1M	117.368 39	114.919 50	119.859 82
	6M	368.063 92	359.930 15	376.882 77
	60M	447.241 64	439.277 43	455.345 35

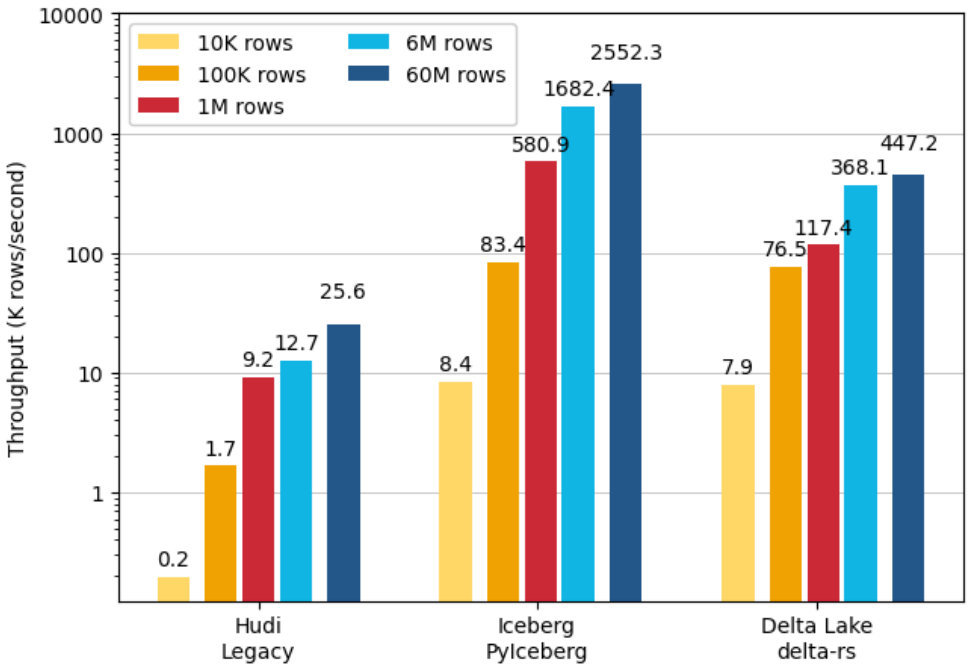

Figure C.6: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with two CPU cores.

Table C.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
Hudi Legacy	10K	0.195 04	0.192 61	0.197 58
	100K	1.680 28	1.662 86	1.699 09
	1M	9.190 57	9.143 29	9.234 34
	6M	12.449 20	12.424 93	12.474 41
	60M	25.574 26	25.475 41	25.672 01
Iceberg PyIceberg	10K	8.366 27	8.152 67	8.613 26
	100K	79.697 15	77.816 50	82.052 81
	1M	612.891 38	597.747 52	630.363 05
	6M	1 692.583 43	1 680.055 53	1 707.255 44
	60M	2 499.443 20	2 484.091 84	2 510.768 77
Delta Lake delta-rs	10K	8.218 03	8.109 89	8.316 84
	100K	74.820 47	74.050 66	75.578 59
	1M	118.784 11	116.237 36	121.271 61
	6M	369.835 13	361.760 41	378.999 43
	60M	483.631 53	474.129 47	495.203 24

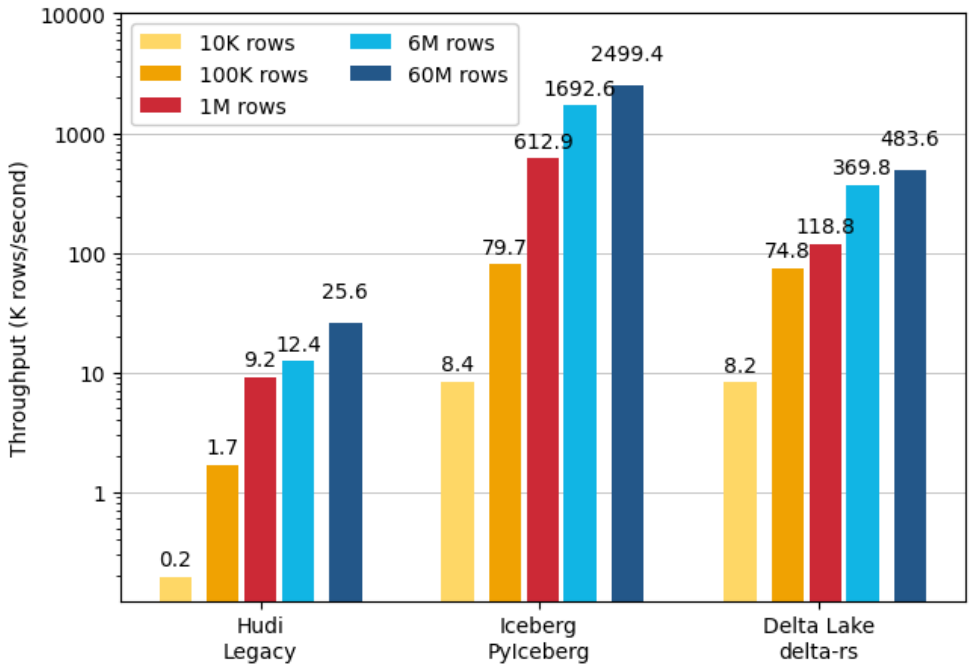

Figure C.7: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with four CPU cores.

Table C.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
Hudi Legacy	10K	0.195 21	0.192 95	0.197 68
	100K	1.658 70	1.645 64	1.672 84
	1M	9.142 29	9.103 10	9.181 79
	6M	12.605 76	12.575 41	12.636 69
	60M	25.808 59	25.739 09	25.871 43
Iceberg PyIceberg	10K	8.208 73	7.747 35	8.627 10
	100K	80.090 89	78.352 19	81.977 34
	1M	581.707 82	577.533 99	586.819 41
	6M	1 698.521 31	1 662.196 40	1 727.430 78
	60M	2 539.884 07	2 531.591 50	2 546.589 00
Delta Lake delta-rs	10K	7.354 81	6.367 77	8.009 70
	100K	77.387 41	76.265 73	79.015 66
	1M	120.554 78	118.199 43	122.798 45
	6M	381.579 81	370.912 17	393.712 18
	60M	492.238 46	483.347 68	502.012 28

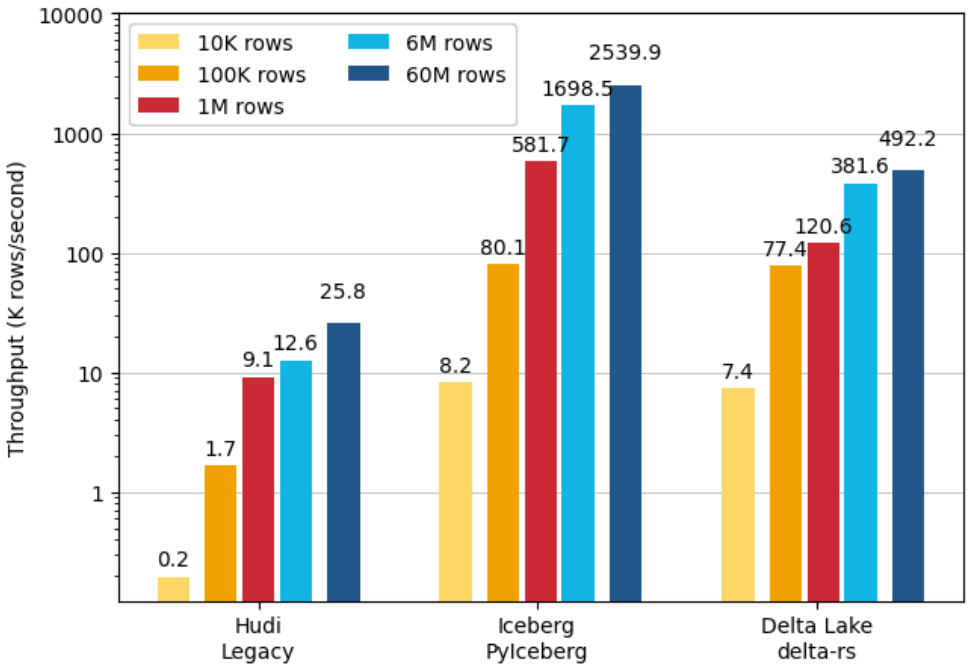

Figure C.8: Histogram in log-scale of the write experiment results expressed as throughput. The experiment was performed with eight CPU cores.

Appendix D

Read experiments results

This appendix contains all the graphs and the tables related to the read experiments conducted and the comparison with the related work. Results are first reported expressed as latency (measured in seconds) and then as throughput (measured in rows/seconds). Latency was directly measured, while throughput was computed from the latency and table size.

Table D.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
Hudi Legacy	10K	0.631 44	0.623 60	0.640 86
	100K	2.650 43	2.642 77	2.658 80
	1M	8.592 96	8.324 32	8.889 65
	6M	33.525 80	33.221 21	33.838 81
	60M	33.690 31	33.342 63	34.035 32
Iceberg PyIceberg	10K	0.017 23	0.016 46	0.017 99
	100K	0.046 87	0.042 83	0.051 07
	1M	0.430 18	0.419 87	0.441 90
	6M	2.123 31	2.107 68	2.139 33
	60M	21.819 55	21.698 84	21.937 05
Delta Lake delta-rs	10K	0.053 45	0.039 17	0.080 56
	100K	0.057 63	0.055 05	0.060 55
	1M	0.538 78	0.524 44	0.553 18
	6M	1.949 47	1.928 55	1.969 58
	60M	22.981 86	22.839 94	23.157 94

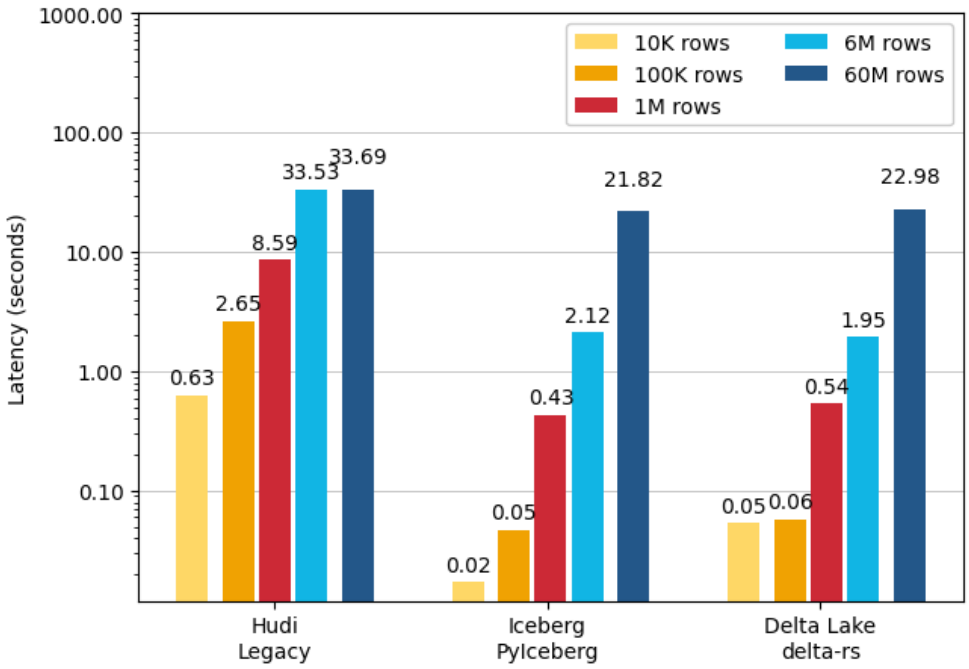

Figure D.1: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with one CPU core.

Table D.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
Hudi Legacy	10K	0.624 82	0.621 81	0.628 02
	100K	2.663 32	2.656 15	2.670 70
	1M	8.606 87	8.301 38	8.979 21
	6M	33.377 40	33.096 84	33.706 81
	60M	33.628 80	33.286 17	34.017 80
Iceberg PyIceberg	10K	0.018 32	0.017 35	0.019 31
	100K	0.043 17	0.041 43	0.045 18
	1M	0.269 43	0.262 17	0.276 64
	6M	1.296 06	1.230 03	1.343 85
	60M	13.512 94	13.410 47	13.624 77
Delta Lake delta-rs	10K	0.041 21	0.039 31	0.043 85
	100K	0.057 21	0.051 10	0.066 94
	1M	0.233 73	0.225 13	0.242 63
	6M	0.908 26	0.899 00	0.918 35
	60M	11.409 75	11.273 01	11.596 08

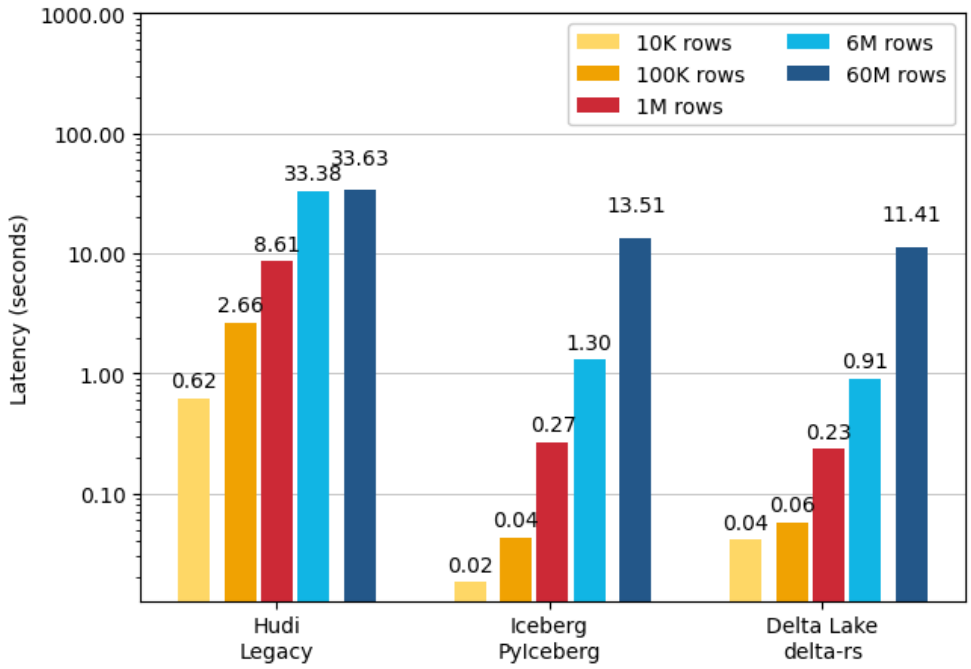

Figure D.2: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with two CPU cores.

Table D.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
Hudi Legacy	10K	0.636 24	0.623 55	0.659 45
	100K	2.639 86	2.633 50	2.646 47
	1M	8.744 36	8.522 35	9.000 47
	6M	33.447 49	33.185 73	33.740 60
	60M	33.661 53	33.263 12	34.090 78
Iceberg PyIceberg	10K	0.029 79	0.015 68	0.059 14
	100K	0.042 83	0.041 13	0.044 74
	1M	0.233 37	0.218 55	0.248 06
	6M	1.008 57	1.001 23	1.016 16
	60M	10.110 29	10.040 90	10.183 32
Delta Lake delta-rs	10K	0.043 57	0.039 31	0.051 02
	100K	0.055 46	0.053 72	0.057 86
	1M	0.189 58	0.150 76	0.255 24
	6M	0.531 06	0.507 52	0.569 20
	60M	5.579 90	5.545 08	5.618 30

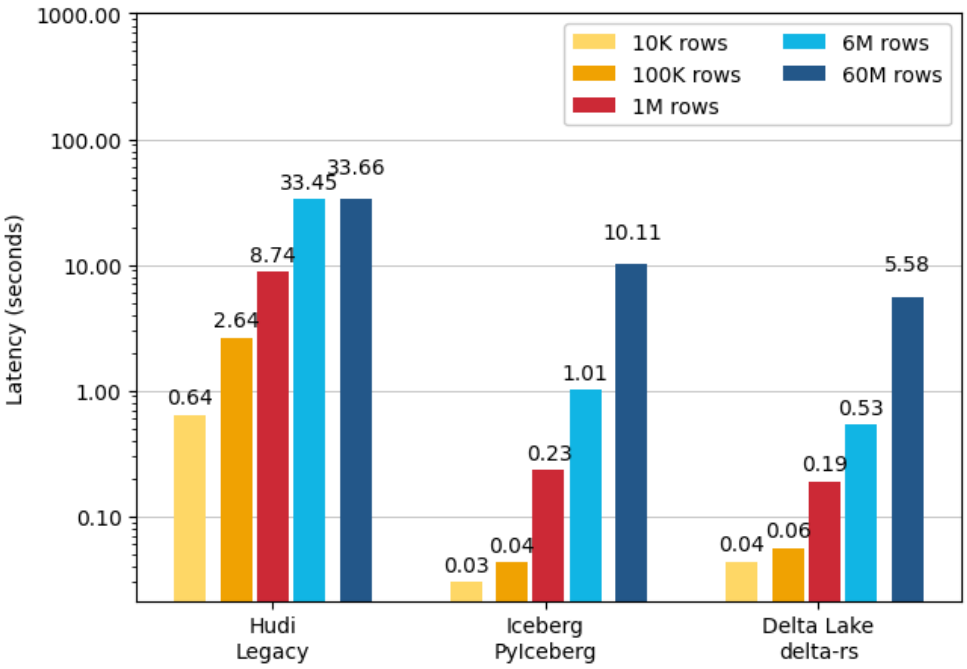

Figure D.3: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with four CPU cores.

Table D.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
Hudi Legacy	10K	0.627 33	0.622 54	0.632 50
	100K	2.662 35	2.653 22	2.672 16
	1M	8.352 14	8.141 63	8.618 32
	6M	33.440 13	33.169 78	33.752 54
	60M	33.142 44	32.900 73	33.405 44
Iceberg PyIceberg	10K	0.016 96	0.015 98	0.018 10
	100K	0.042 68	0.040 96	0.044 78
	1M	0.232 12	0.229 27	0.235 31
	6M	0.884 65	0.875 63	0.894 54
	60M	8.839 20	8.758 94	8.933 90
Delta Lake delta-rs	10K	0.043 14	0.038 83	0.051 29
	100K	0.054 65	0.052 99	0.056 65
	1M	0.174 09	0.169 83	0.178 29
	6M	0.497 07	0.485 62	0.510 95
	60M	2.944 31	2.859 72	3.053 01

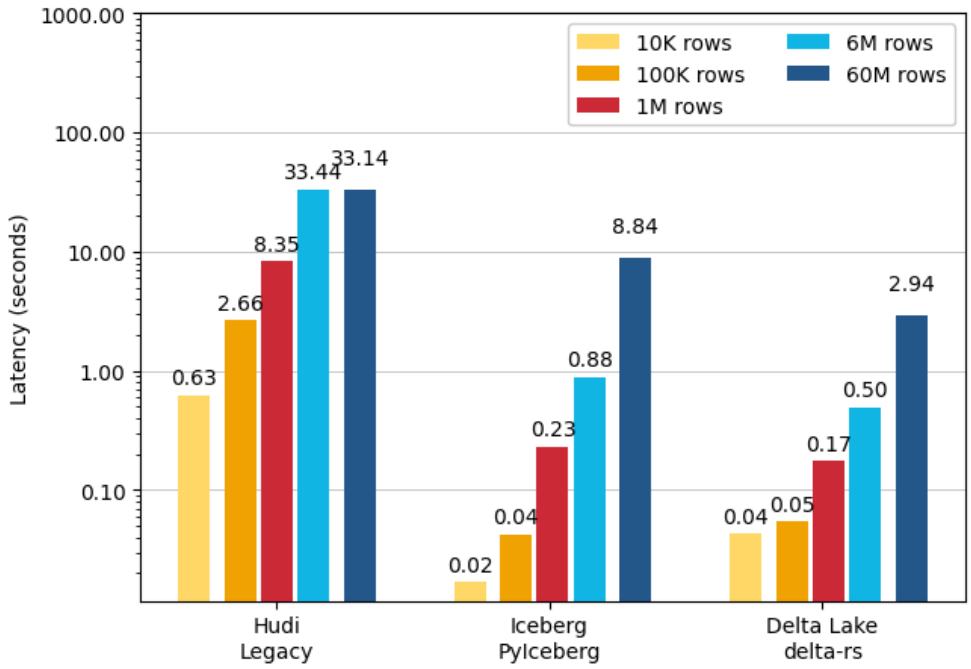

Figure D.4: Histogram in log-scale of the read experiment results expressed as latency. The experiment was performed with eight CPU cores.

Table D.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
Hudi Legacy	10K	15.836 73	15.603 92	16.035 91
	100K	37.729 79	37.611 00	37.839 07
	1M	116.374 31	112.490 38	120.129 97
	6M	178.966 62	177.311 21	180.607 49
	60M	1 780.927 64	1 762.874 34	1 799.498 02
Iceberg PyIceberg	10K	580.480 03	555.595 31	607.645 51
	100K	2 133.403 62	1 958.119 12	2 334.613 16
	1M	2 324.629 63	2 262.967 50	2 381.662 01
	6M	2 825.776 91	2 804.612 49	2 846.730 95
	60M	2 749.827 53	2 735.099 13	2 765.125 11
Delta Lake delta-rs	10K	187.081 71	124.129 41	255.274 25
	100K	1 735.080 31	1 651.502 61	1 816.685 79
	1M	1 856.033 25	1 807.743 07	1 906.811 84
	6M	3 077.758 01	3 046.330 18	3 111.145 48
	60M	2 610.755 20	2 590.903 90	2 626.976 89

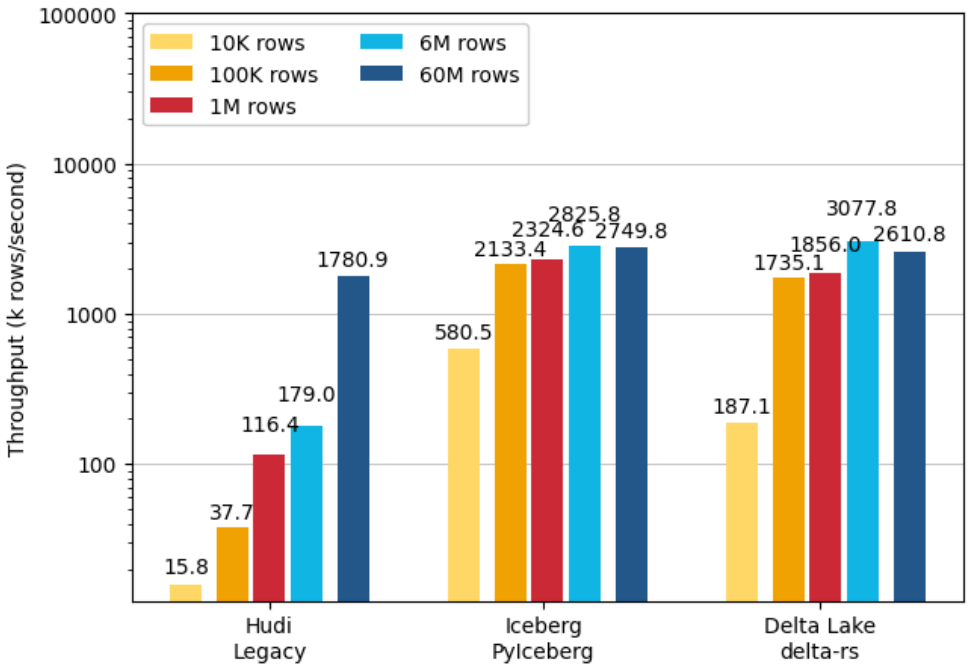

Figure D.5: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with one CPU core.

Table D.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	high
Hudi Legacy	10K	16.004 50	15.922 95	16.082 20
	100K	37.547 12	37.443 31	37.648 42
	1M	116.186 27	111.368 42	120.461 87
	6M	179.762 35	178.005 56	181.286 17
	60M	1 784.185 08	1 763.782 51	1 802.550 61
Iceberg PyIceberg	10K	545.758 37	517.757 02	576.367 42
	100K	2 316.163 25	2 213.607 44	2 413.481 99
	1M	3 711.541 85	3 614.794 02	3 814.373 02
	6M	4 629.432 93	4 464.785 44	4 877.936 41
	60M	4 440.187 28	4 403.743 16	4 474.117 01
Delta Lake delta-rs	10K	242.640 97	228.056 10	254.401 00
	100K	1 747.921 59	1 493.971 60	1 957.125 61
	1M	4 278.422 15	4 121.532 10	4 441.780 08
	6M	6 606.023 33	6 533.473 80	6 674.052 28
	60M	5 258.661 43	5 174.163 60	5 322.447 83

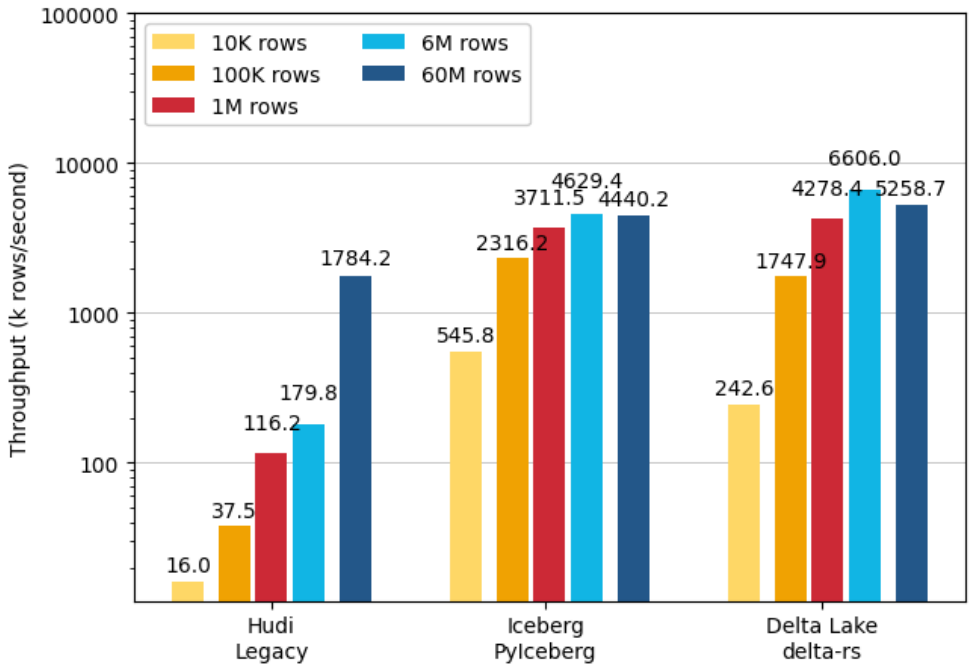

Figure D.6: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with two CPU cores.

Table D.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
Hudi Legacy	10K	15.717 31	15.164 13	16.037 29
	100K	37.880 87	37.786 17	37.972 27
	1M	114.359 39	111.105 33	117.338 56
	6M	179.385 67	177.827 31	180.800 62
	60M	1 782.450 22	1 760.006 76	1 803.799 67
Iceberg PyIceberg	10K	335.657 43	169.104 33	637.821 94
	100K	2 334.622 10	2 235.354 39	2 431.532 16
	1M	4 285.053 39	4 031.244 13	4 575.642 54
	6M	5 949.035 30	5 904.554 74	5 992.630 25
	60M	5 934.546 77	5 891.986 28	5 975.558 76
Delta Lake delta-rs	10K	229.534 46	196.008 45	254.378 02
	100K	1 803.247 82	1 728.356 74	1 861.443 03
	1M	5 274.741 03	3 917.946 20	6 633.050 53
	6M	11 298.148 09	10 541.113 29	11 822.246 55
	60M	10 752.873 37	10 679.378 68	10 820.393 55

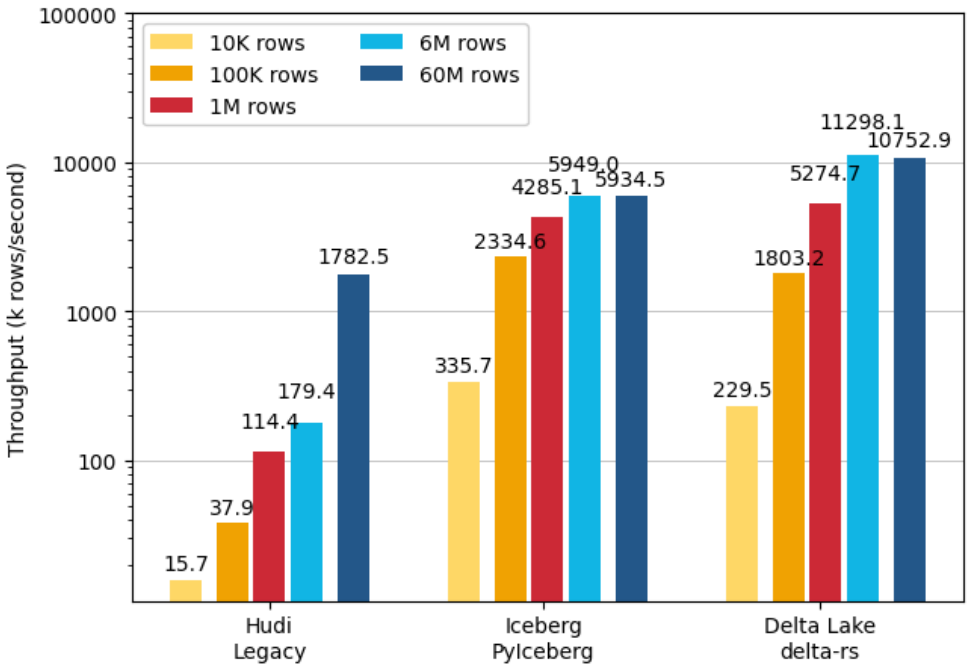

Figure D.7: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with four CPU cores.

Table D.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
Hudi Legacy	10K	15.940 66	15.810 39	16.063 27
	100K	37.560 87	37.422 90	37.689 99
	1M	119.729 81	116.031 88	122.825 54
	6M	179.425 15	177.764 38	180.887 57
	60M	1 810.367 75	1 796.114 54	1 823.667 90
Iceberg PyIceberg	10K	589.518 81	552.402 08	625.871 90
	100K	2 343.108 33	2 233.189 92	2 441.607 41
	1M	4 308.143 74	4 249.775 75	4 361.607 98
	6M	6 782.335 84	6 707.348 51	6 852.171 57
	60M	6 787.943 65	6 715.992 13	6 850.140 93
Delta Lake delta-rs	10K	231.783 14	194.952 45	257.555 78
	100K	1 829.910 95	1 765.222 67	1 887.200 60
	1M	5 744.056 36	5 608.985 88	5 888.367 78
	6M	12 070.777 05	11 742.934 47	12 355.358 82
	60M	20 378.273 02	19 652.710 07	20 981.075 17

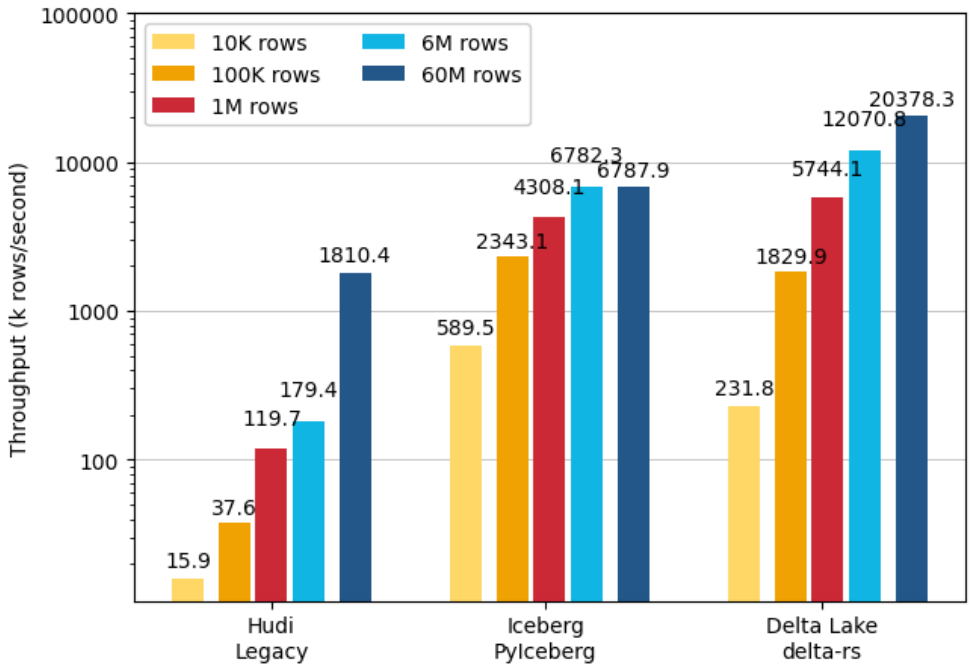

Figure D.8: Histogram in log-scale of the read experiment results expressed as throughput. The experiment was performed with eight CPU cores.

Appendix E

Legacy pipeline write latency breakdown results

This appendix contains all the graphs and the tables related to the write latency breakdown, between upload and materialization steps, for the Hudi-based legacy pipeline. Results are reported expressed as latency (measured in seconds), which was measured directly.

Table E.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	2.486 5	2.389 6	2.626 1
materialize		47.726 2	47.044 5	48.403 1
upload	100K	3.668 4	3.631 0	3.709 8
materialize		55.900 5	55.249 4	56.554 1
upload	1M	22.593 4	22.449 6	22.734 9
materialize		89.575 4	88.828 6	90.304 9
upload	6M	244.612 3	244.036 8	245.190 5
materialize		267.249 0	266.428 7	268.154 9
upload	60M	2 437.784 0	2 422.870 4	2 453.674 6
materialize		278.050 4	276.234 0	280.092 1

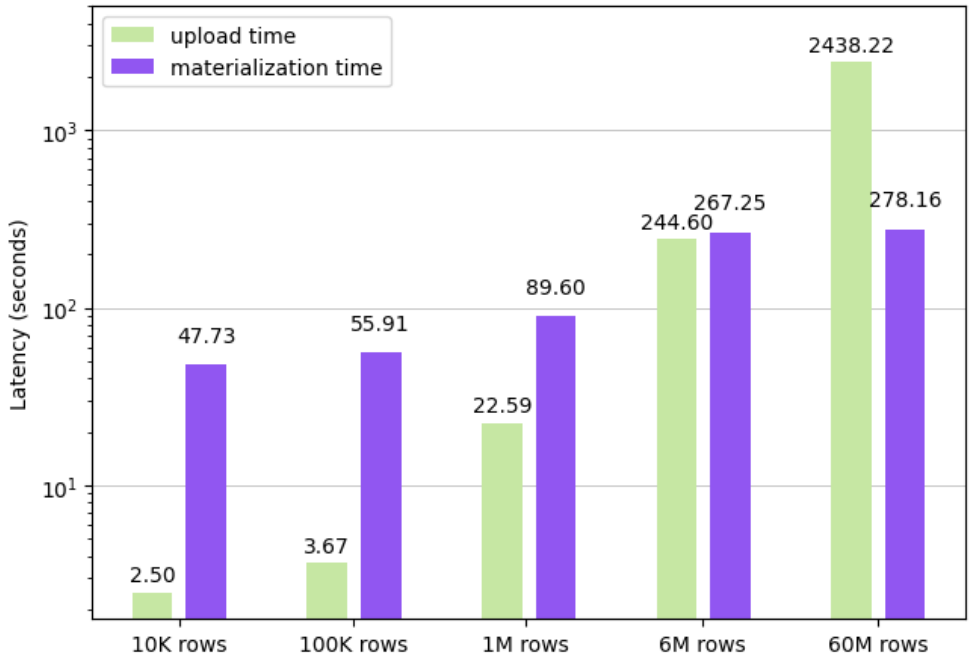

Figure E.1: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with one CPU core.

Table E.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	2.387 3	2.327 6	2.446 6
materialize		48.330 5	47.702 0	48.992 3
upload	100K	3.434 8	3.400 8	3.467 1
materialize		56.336 7	55.562 6	57.112 9
upload	1M	18.634 9	18.567 3	18.710 4
materialize		89.926 7	89.401 2	90.451 4
upload	6M	205.885 4	205.217 7	206.498 4
materialize		267.507 9	266.685 3	268.351 2
upload	60M	2 064.135 7	2 057.639 6	2 070.445 0
materialize		276.960 8	275.715 6	278.284 9

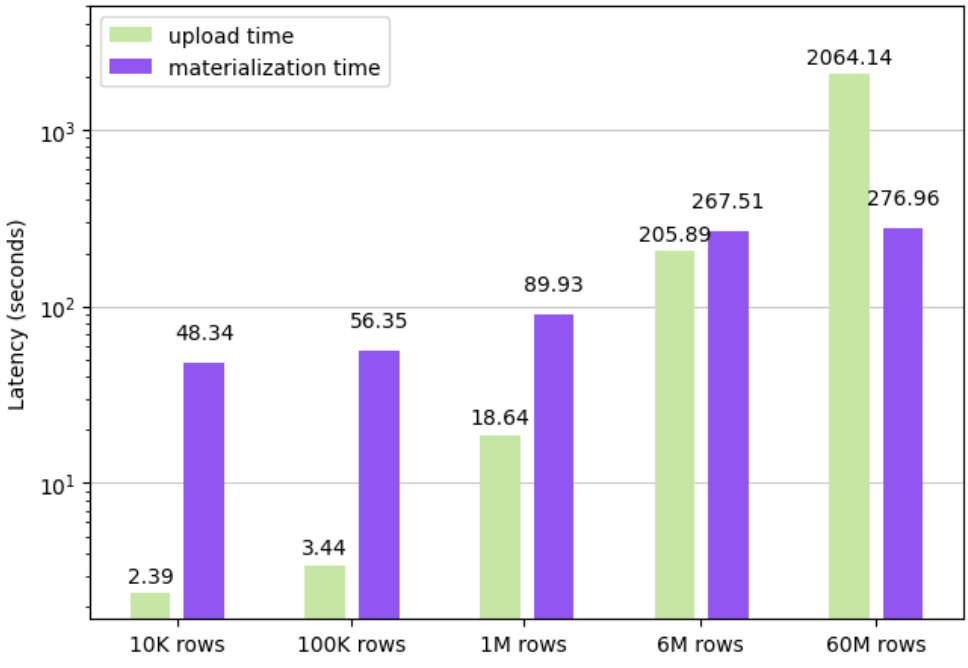

Figure E.2: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with two CPU cores.

Table E.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	2.384 6	2.329 9	2.433 5
materialize		48.906 1	48.243 6	49.547 0
upload	100K	3.465 0	3.424 5	3.507 1
materialize		56.052 4	55.368 2	56.682 2
upload	1M	19.229 6	19.145 5	19.316 1
materialize		89.586 4	89.020 9	90.131 3
upload	6M	211.675 8	211.069 4	212.283 9
materialize		270.323 3	269.596 7	270.989 5
upload	60M	2 068.526 0	2 060.335 8	2 077.283 7
materialize		277.600 1	276.006 5	279.145 6

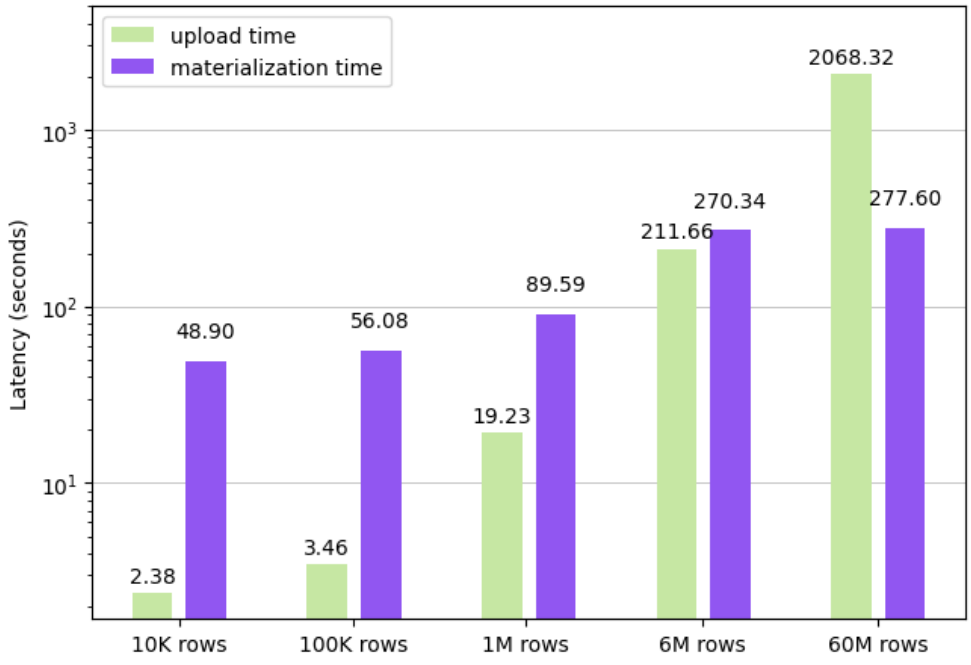

Figure E.3: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with four CPU cores.

Table E.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	2.381 5	2.330 4	2.435 8
materialize		48.848 5	48.197 9	49.446 7
upload	100K	3.439 2	3.408 1	3.470 0
materialize		56.842 8	56.317 7	57.368 5
upload	1M	18.864 2	18.780 8	18.953 2
materialize		90.515 3	90.030 6	90.971 8
upload	6M	207.664 6	207.160 6	208.209 0
materialize		268.275 2	267.356 9	269.245 6
upload	60M	2 049.137 1	2 043.599 1	2 055.478 2
materialize		275.763 6	274.277 3	274.277 3

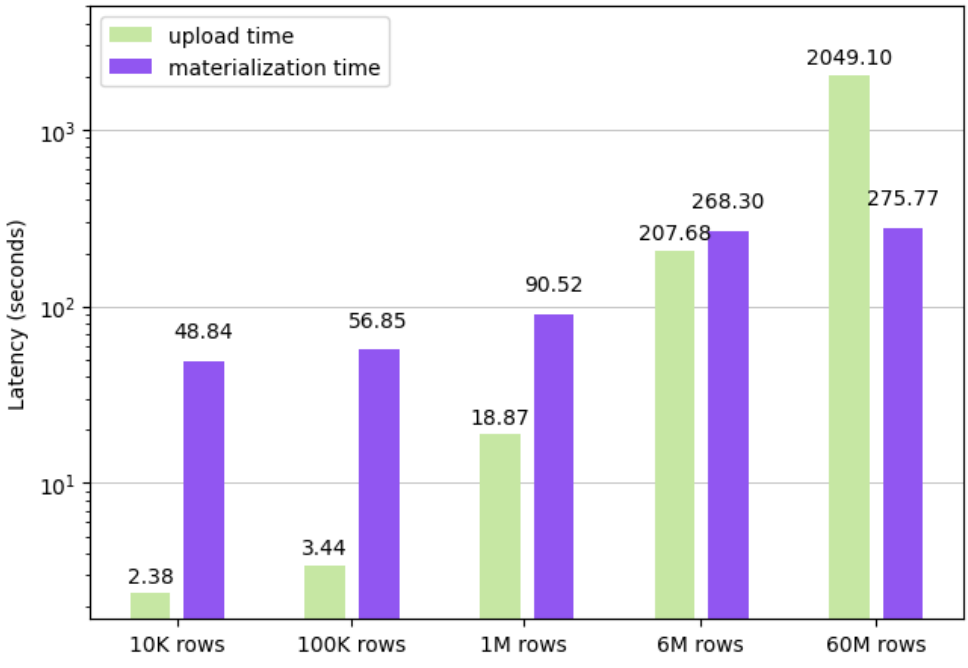

Figure E.4: Histogram in log-scale displaying the contributions to the write latency of the upload and materialization steps in the legacy pipeline. The experiment was performed with eight CPU cores.

TRITA – EECS-EX 2025:0000
Stockholm, Sweden 2025

www.kth.se

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Meneghin",
"First name": "Sebastiano",
"Local User Id": "u16dn7xj",
"E-mail": "meneghin@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA258X",
"Credits": "30.0",
"Degree1": {"Educational program": "Master's Programme, ICT Innovation, 120 credits"
,"programcode": "TIVNM"
,"Degree": "Master's Programme, Distributed Systems and Data Mining for Big Data, 120 credits"
,"subjectArea": "Computer Science"
},
"Title": {
"Main title": "Reducing read and write latency in an Iceberg-backed offline feature store",
"Subtitle": "Integrating HopsFS and PyIceberg Python library to reduce read and write latency on the Iceberg-backed Hopsworks offline feature store, with a comparative analysis of alternative solutions",
"Language": "eng" },
"Alternative title": {
"Main title": "Minska läs- och skrivlatens i en Iceberg-stödd offlinebutik för funktioner",
"Subtitle": "Integrering av HopsFS och PyIceberg Python-bibliotek för att minska läs- och skrivfördröjningen i den Iceberg-stödda offline-lagringen Hopsworks, med en jämförande analys av alternativa lösningar.",
"Language": "swe"
},
"Supervisor1": { "Last name": "Sheikholeslami",
"First name": "Sina",
"Local User Id": "u1znylhw",
"E-mail": "sinash@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"Supervisor2": { "Last name": "Schmidt",
"First name": "Fabian",
"Local User Id": "u1mrsz0u",
"E-mail": "fschm@kth.se",
},
"Supervisor3": { "Last name": "Bzhalava",
"First name": "Davit",
"E-mail": "davit@hopsworks.ai",
"Other organisation": "Hopsworks AB"
},
"Examiner1": { "Last name": "Vlassov",
"First name": "Vladimir",
"Local User Id": "u19yb2c8",
"E-mail": "vladv@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"Cooperation": { "Partner_name": "Hopsworks AB"},
"National Subject Categories": "10201, 10205, 10206",
"Other information": {"Year": "2025", "Number of pages": "1,109"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA – EECS-EX" , "No. in series": "2025:0000" },
"Opponents": { "Name": "Arvid Wennerström"},
"Presentation": { "Date": "2025-02-25 15:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/ddddddddddd"
,"Address": "Isafjordsgatan 22 (Kistagången 16)"
,"City": "Stockholm" },
"Number of lang instances": "3",

"Abstract[eng]": €€€€
The growing need for efficient data management in Machine Learning (ML) workflows has led to the widespread adoption of feature stores, centralized data platforms that supports feature engineering, model training and prediction inference. The Hopsworks' feature store has demonstrated outperformance compared to its alternatives, leveraging Apache Hudi and Spark for offline data storage, but suffers from high write and read latency, even for small quantities of data (1GB or less). This thesis explores the potential of Apache Iceberg as an alternative table format, integrating it with HopsFS (Hopsworks HDFS distribution) and PyIceberg Python library to reduce latencies.
The research begin with an evaluation of potential system integration alternatives, documenting the advantages and limitations of each approach. Then, a PyIceberg-based architecture is implemented and evaluated, benchmarking it against the existing Spark-based solution and an alternative Delta Lake implementation (delta-rs). Extensive experiments were conducted across varying table sizes and CPU configurations to assess write and read performance. Results show that PyIceberg significantly reduces write latency – from 40 to 140 times lower than the legacy system – and read latency – from 55% to 60 times lower than the legacy system. Compared to delta-rs, PyIceberg demonstrates reduced write latency for large tables – up to 7 times lower – and equal read latency, but exhibits lower scaling benefits with additional CPU cores – 20% less than delta-rs.
These findings confirm that alternatives to Spark-based pipelines in small-scale scenarios are possible and are worth of further investigations, and

the system implemented will be included in the Hopsworks feature store. Furthermore, this thesis work and results finally provides a baseline for future work about additional open table formats, alternative languages to mitigate Python's performance overhead, and strategies to improve resource utilization in data management platforms.
€€€€,
"Keywords[eng]": €€€€
Machine Learning, Feature Store, Spark, Apache Iceberg, Python, Read/Write Latency, Open Table Formats €€€€,
"Abstract[swe]": €€€€
Det växande behovet av effektiv datahantering i arbetsflöden för maskininlärning (ML) har lett till en utbredd användning av feature stores – centraliserade dataplattformar som stöder feature engineering, modellträning och inferens. Hopsworks feature store har visat bättre prestanda jämfört med sina alternativ och använder Apache Hudi och Spark för offline-datalagring. Dock lider systemet av hög skriv- och läslatens, även för små datamängder (1 GB eller mindre). Denna avhandling undersöker potentialen hos Apache Iceberg som ett alternativt tabellformat och integrerar det med HopsFS (Hopsworks HDFS-distribution) samt Pyiceberg Python-biblioteket för att minska latensen.
Forskningen inleds med en utvärdering av potentiella systemintegrationsalternativ, där fördelar och begränsningar med varje metod dokumenteras. Därefter implementeras och utvärderas en Pyiceberg-baserad arkitektur, vilken jämförs med den befintliga Spark-baserade lösningen samt en alternativ Delta Lake-implementering (delta-rs). Omfattande experiment genomfördes med varierande tabellstorlekar och CPU-konfigurationer för att bedöma skriv- och läsprestanda. Resultaten visar att Pyiceberg avsevärt minskar skrivfördröjningen – från 40 till 140 gånger lägre än det äldre systemet – och läsfördröjningen – från 55% till 60 gånger lägre än det äldre systemet. Jämfört med delta-rs uppvisar Pyiceberg minskad skrivfördröjning för stora tabeller – upp till sju gånger lägre – och liknande läsfördröjning, men har sämre skalningsfördelar vid ökning av CPU-kärnor (20% mindre än delta-rs).
Dessa resultat bekräftar att alternativ till Spark-baserade pipelines i småskaliga scenarier är möjliga och värda ytterligare undersökningar. Det implementerade systemet kommer att integreras i Hopsworks feature store. Dessutom utgör denna avhandling en baslinje för framtida forskning kring ytterligare öppna tabellformat, alternativa programmeringsspråk för att hantera Pythons prestandabegränsningar samt strategier för att förbättra resursutnyttjandet i datahanteringsplattformar. €€€€,
"Keywords[swe]": €€€€
Maskininlärning, Feature Store, Spark, Apache Iceberg, Python, Läs- och skrivlatens, Open Table Formats €€€€,
"Abstract[ita]": €€€€
La crescente necessità di piattaforme per una efficience gestione dei dati per applicazioni di *Machine Learning (ML)* ha portato a un'ampia diffusione dei *feature store*, piattaforme dati centralizzate che supportano *feature engineering*, addestramento di modelli e inferenza. Il *feature store* di Hopsworks ha dimostrato prestazioni superiori rispetto alle sue alternative, sfruttando Apache Hudi e Spark per il *suo offline feature store*. Tuttavia, questo sisteme soffre di un'elevata latenza di scrittura e lettura, anche per piccole quantità di dati (1GB o inferiori). Questa tesi esplora l'uso di Apache Iceberg come *table format* alternativo, integrandolo con HopsFS (distribuzione Hopsworks di HDFS) e la libreria Python Pyiceberg per ridurre tali latenze.
Questa ricerca inizia con un'analisi delle possibili strategie di integrazione, documentando i vantaggi e i limiti di ciascun approccio. Successivamente, viene implementata e valutata un'architettura basata su Pyiceberg, confrontata con la soluzione esistente basata su Spark e con un'alternative basata su Delta Lake (delta-rs). Esperimenti approfonditi sono stati condotti su tabelle di diverse dimensioni e diverse configurazioni CPU per misurare le prestazioni di scrittura e lettura. I risultati mostrano che Pyiceberg riduce significativamente la latenza di scrittura – da 40 a 140 volte inferiore rispetto al sistema legacy – e la latenza di lettura – dal 55% a 60 volte inferiore. Rispetto a delta-rs, Pyiceberg offre una latenza di scrittura inferiore per le tabelle più grandi – fino a 7 volte minore – e prestazioni di lettura equivalenti, ma mostra minori vantaggi di scalabilità con l'aumento dei CPU cores (20% ridotti rispetto a delta-rs).
Questi risultati confermano che alternative ad architetture basate su Spark, per gestire dati su piccola scala, esistono e sono più efficienti, e dunque il sistema sviluppato verrà integrato nel *feature store* di Hopsworks. Inoltre, questa tesi fornisce una base per futuri studi su nuovi *open table format* e strategie da adottare per ottimizzare l'uso delle risorse nelle piattaforme di gestione dei dati. €€€€,
"Keywords[ita]": €€€€
Machine Learning, Feature Store, Apache Iceberg, Pyiceberg, Latenza in Lettura/Scrittura, Open Table Formats. €€€€,
}

acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym{label}{acronym}{phrase}
%                                      or \newacronym[options]{label}{acronym}{phrase}
% see "User Manual for glossaries.sty" for the  details

\newacronym{KTH}{KTH}{KTH Royal Institute of Technology}
\newacronym{ACID}{ACID}{Atomicity, Consistency, Isolation and Durability}
\newacronym{AI}{AI}{Artificial Intelligence}
\newacronym{ML}{ML}{Machine Learning}
\newacronym{BI}{BI}{Business Intelligence}
\newacronym[shortplural={RDDs}, firstplural={Resilient Distributed Datasets (RDDs)}]{RDD}{RDD}{Resilient Dist
\newacronym{OLAP}{OLAP}{On-Line Analytical Processing}
\newacronym{ELT}{ELT}{Extract Load Transform}
\newacronym{ETL}{ETL}{Extract Transform Load}
\newacronym{HDFS}{HDFS}{Hadoop Distributed File System}
\newacronym{JVM}{JVM}{Java Virtual Machine}
\newacronym[shortplural={INs}, firstplural={Industrial Needs (INs)}]{IN}{IN}{Industrial Need}
\newacronym[shortplural={PAs}, firstplural={Project Assumptions (PAs)}]{PA}{PA}{Project Assumption}
\newacronym[shortplural={APIs}, firstplural={Application Programming Interfaces (APIs)}]{API}{API}{Applicatio
\newacronym{OLTP}{OLTP}{On-Line Transaction Processing}
\newacronym[shortplural={DBMs}, firstplural={Data Base Management Systems}]{DBMS}{DBMS}{Data Base Management
\newacronym[shortplural={Gs}, firstplural={Goals}]{G}{G}{Goal}
\newacronym[shortplural={RQs}, firstplural={Research Questions}]{RQ}{RQ}{Research Question}
\newacronym[shortplural={Ds}, firstplural={Deliverables}]{D}{D}{Deliverable}
\newacronym{CRUD}{CRUD}{Create Read Update Delete}
\newacronym[shortplural={SDGs}, firstplural={Sustainable Development Goals}]{SDG}{SDG}{Sustainable Developmen
\newacronym{AWS}{AWS}{Amazon Web Services}
\newacronym{GCS}{GCS}{Google Cloud Storage}
\newacronym[shortplural={HDDs}, firstplural={Hard Disks Drives}]{HDD}{HDD}{Hard Disk Drive}
\newacronym[shortplural={SSDs}, firstplural={Solid State Drives}]{SSD}{SSD}{Solid State Drive}
\newacronym[shortplural={PCs}, firstplural={Personal Computers}]{PC}{PC}{Personal Computer}
\newacronym[shortplural={OSes}, firstplural={Operating Systems (OSes)}]{OS}{OS}{Operating System}
\newacronym[shortplural={DFSes}, firstplural={Distributed File Systems (DFSes)}]{DFS}{DFS}{Distributed File S
\newacronym{BPMN}{BPMN}{Business Process Model and Notation}
\newacronym{TLS}{TLS}{Transport Layer Security}
\newacronym{SSH}{SSH}{Secure Shell protocol}
\newacronym{VM}{VM}{Virtual Machine}
\newacronym{CoC}{CoC}{Conquer of Completion}
\newacronym{SF}{SF}{Scale Factor}
\newacronym{CPU}{CPU}{Central Processing Unit}
\newacronym{GPU}{GPU}{Graphical Processing Unit}
\newacronym{HopsFS}{HopsFS}{Hopsworks' \glsentryshort{HDFS} distribution}
\newacronym{LocalFS}{LocalFS}{Local File System}
\newacronym{TPC}{TPC}{Transaction Processing Performance Council}
\newacronym{RAM}{RAM}{Random Access Memory}
\newacronym{RPC}{RPC}{Remote Procedural Call}
\newacronym{CIDR}{CIDR}{Conference on Innovative Data Systems Research}
\newacronym{MLOps}{MLOps}{Machine Learning Operations}
\newacronym{LOC}{LOC}{Lines Of Code}
\newacronym{NN}{NN}{Neural Network}
\newacronym{SAN}{SAN}{Storage Area Network}
\newacronym{NAS}{NAS}{Network Attached Storage}
\newacronym{LAN}{LAN}{Local Area Network}
\newacronym{WAN}{WAN}{Wide Area Network}
\newacronym[shortplural={OTFs}, firstplural={Open Table Formats}]{OTF}{OTF}{Open Table Format}
\newacronym{CDC}{CDC}{Change Data Capture}
\newacronym{CoW}{CoW}{Copy on Write}
\newacronym{MoR}{MoR}{Merge on Read}
\newacronym{CC}{CC}{Concurrency Control}
\newacronym{SDK}{SDK}{Software Development Kit}
\newacronym{HMS}{HMS}{Hive Metastore}
\newacronym{JDBC}{JDBC}{Java DataBase Connectivity}
\newacronym{SQL}{SQL}{Structured Query Language}
```