Degree Project in the Field of Technology Computer Science and the Main Field of Study Data Science

Second cycle, 30 credits

Accelerating Feature Store performance with Apache Iceberg

A Python-based alternative delivering speed and scalability without Spark dependency

SEBASTIANO MENEGHIN

Accelerating Feature Store performance with Apache Iceberg

A Python-based alternative delivering speed and scalability without Spark dependency

SEBASTIANO MENEGHIN

Master's Programme, ICT Innovation, 120 credits
Date: February 4, 2025

Supervisors: Sina Sheikholeslami, Fabian Schmidt, Jim Dowling
Examiner: Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: Snabbare prestanda för Feature Store med Apache Iceberg
Swedish subtitle: Ett Python-baserat alternativ som ger snabbhet och skalbarhet utan Spark-beroende

Abstract

Keep in mind that most of your potential readers are only going to read your `title` and `abstract`. This is why the abstract must give them enough information so that they can decide if this document is relevant to them or not. Otherwise, the likely default choice is to ignore the rest of your document.
An abstract should stand on its own, i.e., no citations, cross-references to the body of the document, acronyms must be spelled out, ...
Write this early and revise as necessary. This will help keep you focused on what you are trying to do.

Write an abstract that is about 250 and 350 words (1/2 A4-page) with the following components:

- What is the topic area?
- (optional) Introduces the subject area for the project.
- Short problem statement
- Why was this problem worth a Master's thesis project? (*i.e.*, why is the problem both significant and of a suitable degree of difficulty for Master's thesis project? Why has no one else solved it yet?)
- How did you solve the problem? What was your method/insight?
- Results/Conclusions/Consequences/Impact: What are your key results/ conclusions? What will others do based on your results? What can be done now that you have finished - that could not be done before your thesis project was completed?

Keywords

Machine Learning, Feature Store, Spark, Apache Iceberg, Python, Read/write latency

Formatting the keywords:

- The first letter of a keyword should be set with a capital letter and proper names should be capitalized as usual.
- Spell out acronyms and abbreviations.

- Avoid "stop words" - as they generally carry little or no information.
- List your keywords separated by commas (",").

Sammanfattning

Nyckelord

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

Sommario

Parole Chiave

5-6 parole chiave

Acknowledgments

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 [1]. 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 [2] 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 [3]. This surge is also strongly related to the AI adoption acceleration [4], which put pressure on storage solutions and data processing capabilities [1]. 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 [5].

From the first appearance of data lakehouses [6], three primary applications arose [7]:

1. **Apache Hudi**: introduced by Uber in 2017, then supported by Alibaba, Bytedance, Uber and Tencent [8].

2. **Apache Iceberg**: open sourced by Netflix footnoteNetflix first implementation at `https://github.com/Netflix/iceberg` in 2018, now used by Airbnb, Apple, Expedia, LenkedIn, Lyft [9].

3. **Delta Lake**: open sourced by Databricks in 2019 [10], supported now by Databricks and Microsoft.

Apache Iceberg (from now on, Iceberg) was originally developed as an alternative to Apache Hive (from now on, Hive), a data warehousing system

that enabling 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, which both go beyond Hive's capabilities [11]. This is possible thanks to the Iceberg metadata management layer, that enables data warehouse-like capabilities via cloud object storage. Iceberg has consolidated its position in 2024 [12], being recently included in arguably all the major big data vendors' solutions [13].

Apache Hudi (from now on, Hudi), Iceberg and Delta Lake were initially designed to support a specific engine, respectively Hive, Trino and Apache Spark (from now on, Spark) [7]. These three solutions have been improved to better address industry needs and flexibility issues, and now provides integrations for several engines [14]. This evolution, while beneficial, highlights a potential limitation: even solutions which have gained widespread adoption might not be the optimal solution in all the cases. For example, integrating the system with Spark, a data query and processing engine [15], works well for processing massive datasets (1 TB+) on the cloud, however it is yet unclear if it works well for processing smaller datasets (1 GB to 100 GB) [16].

Limitations of Spark are also brought to light by technologies such as the Python library Polars [17] and the Data Base Management System (DBMS) DuckDB [18]. A Spark cluster performs worse than other options when processing data locally with lesser quantities. Thus, employing Spark eventually results in higher expenses and computation time [19] [20]. With small-scale (from 1 GB to 100 GB) scenarios, different solutions would greatly increase performance.

Remaining in the data science domain, Python has arguably become the de-facto standard programming language. Perhaps, Python is currently the most popular general-purpose programming language [21, 22, 23], and the go-to option for Machine Learning (ML) and Artificial Intelligence (AI) applications [24]. Among the most used libraries by ML application developers, we find NumPy and Pandas [22], while PyTorch and TensorFlow are the first two for Neural Network (NN) libraries. Accessing data lakehouse solutions via Python client would be the favorite option for most of ML and AI developers, so they would not have to resort to alternatives, such as Spark and its Python Application Programming Interface (API) (PySpark).

Thus, 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 *. This also implements resource-efficient and point-in-time join between datasets, adding historical retrieval features to saved data [25].

This project seeks to reduce latency (seconds), thus increasing throughput (rows/second), of reading and writing data on Iceberg tables, as offline feature store in Hopsworks. Currently, the writing pipeline is Spark-based, supported by Hudi's table format: the project's main hypothesis is that a quicker non-Spark alternative is possible. If proved to be successful, 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, while 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 [26].

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

This work's primary contributions are:

- **TO BE DISCUSSED:** The evaluation of the possible integration between Iceberg and Hopsworks feature store, and the selection of the best one according to .
- The results of the experiments conducted on the newly integrated and the older Hopsworks feature store, showing differences by latency and throughput, on 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 and read latency by , compared to what experimented using the old system.
- The comparison of the above results with the results of a parallel conducted thesis [26], investigating native Python access for Delta Lake as Spark alternative.

These findings provide a significant addition 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 be used as starting or final point for further exploration and testing in this field.

insert defined requirements

insert the selected one and explain the main reasons

add write percentual improvement

add read percentual improvement

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

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 data management tool, and the rise of Python to the most used programming language in the data science field.

The term "data lakehouse" was used by Databricks in 2020 [6] to characterise 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 bacame the standard in the 1990s and early 2000s [27], 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 [28]. Data lakes were thus introduced, as a more flexible and scalable solution, serving as a central repository for all data. They also 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 [29]. 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 [8], followed by Netflix in 2018, with Iceberg [9], and ultimately by Databricks, with Delta Lake in 2020 [10].

Spark is a distributed computing platform designed to facilitate large-scale, data-intensive applications [30]. 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 [31], 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 [18] and Polars [17] offer compelling alternatives. These solutions, optimized for smaller datasets, provide high-performance OLAP capabilities and DataFrame operations within an embedded environment, delivering significantly faster results compared to initiating a Spark cluster for equivalent tasks. This project will investigate 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 [32]. Its user-friendliness, high-level abstraction, and emphasis on clear code expression initially attracted a large and enthusiastic community [22]. 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 [21] and 2024 GitHub report [23] underscore the current trends,

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

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 emphasise the need of 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 [33] 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 better cope with customer's needs. In the legacy system, Spark serves as the query engine, executing queries (read, write, or delete) on the offline feature store. The system demonstrated that a write operation on a tiny dataset, including 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).

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 [16] for data ingestion in their Hudi system, and to search for alternatives to the Hudi-Spark tandem [26]. Iceberg provides support for several [14] 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) [34], and several implementations are possible [11].

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

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 Rust delta-rs library *, which operates on Delta Lake tables, and the PyIceberg library *, which operates on Iceberg 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 following two 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 and the delta-rs alternatives?

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 on the offline feature store 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 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 [35]. Opting using PyIceberg or delta-rs instead of Spark provides developers with a wider array of options for

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

managing small-scale data (1 GB - 100 GB). This thesis work also provides them of 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 seeks to aims 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 also connected to the collection of RQs, delineating a distinct framework of the different project milestones.

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.

Add links to GitHub to include tests conducted on different implementations

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 machine learning and neural network 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 [36]. In the realm of ML and NN training, minimising training duration—and therefore the read and write latency associated with the dataset—has been shown to significantly decrease carbon emissions [37, 38]. 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 time utilisation relative to the prior system. Minimising CPU utilisation 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 those reasons, this project also supports the Sustainable Development Goals (SDGs) † Affordable and Clean Energy (Goal 7) and Industry Innovation and Infrastructure (Goal 9), particularly the objectives 7.3, "Double the improvement in energy efficiency", and 9.4, "Upgrade all industries and infrastructures for sustainability".

*Foundation's website available at https://greensoftware.foundation/
†SDGs website available at https://sdgs.un.org/

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

Furthermore, the experiments created to satisfy G3 has been carefully design to minimize the number of trails necessary for statistical relevance, pursuing resource effiency. All the data throughly explained and all the codes are and provided in this thesis report, 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. Hopsworks's INs are:

IN1 : The Hopsworks 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 domain.

IN2 : Hopsworks actively looks to customer needs and software's integration 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. This popularity means that high-performance Python libraries will typically be preferred over alternatives (even more efficient) that are Java Virtual Machine (JVM) or other environments -based.

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

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 [34], 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, measured in seconds, and throughput, measured in rows per second. Those two metrics were chosen as they most affect the computational time of accessing table formats, and can thus be used as a fair comparator 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 consideration drawn from these provides 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 readers with 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 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 and the various CPU settings, as well as results of related works. The chapter includes a discussion Section that allows readers to understand the major findings and implication 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 [39]. 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 [40, 41, 9]. The following Sections of this report defines the data stack for this use case, which is is illustrated in Figure 2.1.

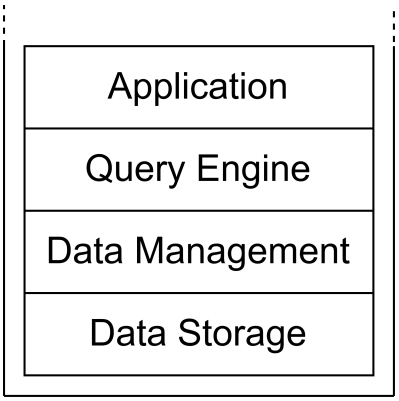

Figure 2.1: Data stack abstraction for this project.

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 firsty what a data storage is and which typologies of storages exist. Thus this project's data storage layer, HopsFS, is presented, starting from its evolution from Hadoop Distributed File System (HDFS), going in 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) [42, 43, 44] according to the author's understanding.

Table 2.1: Data storage features comparison. Table inspired by major cloud providers articles [42, 43, 44].

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 [42, 43, 44].

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 [45].

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 [34], 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 [33]. Despite its scalability

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

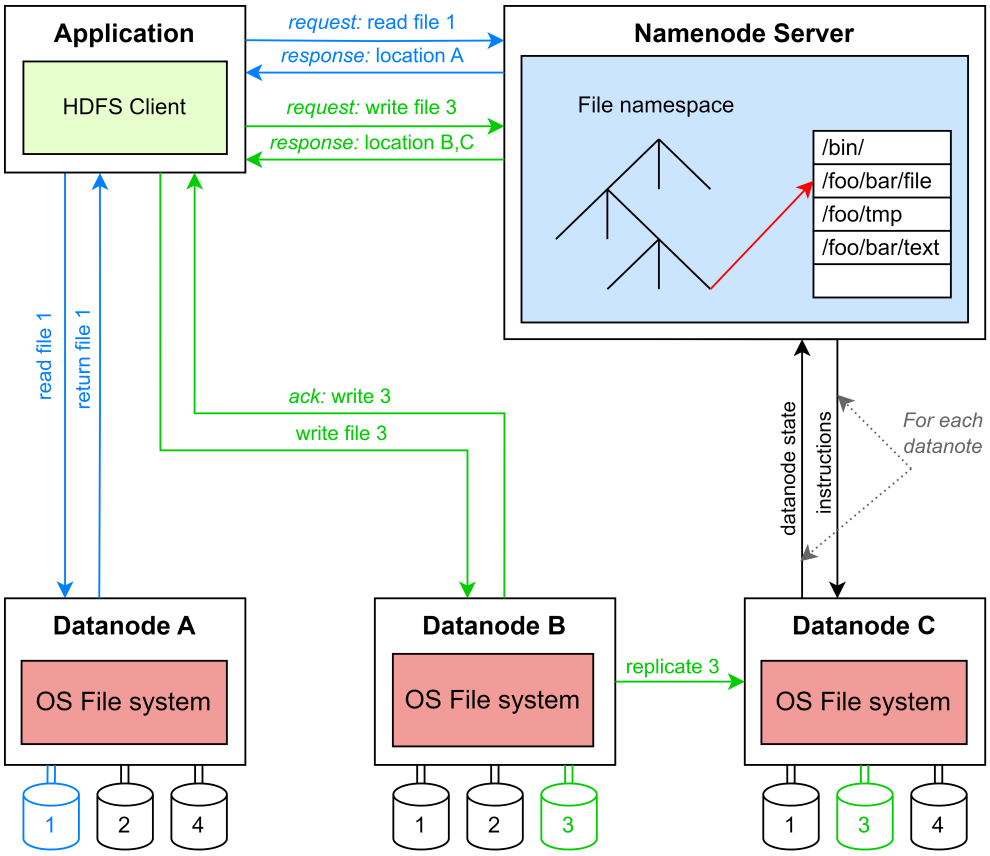

Figure 2.2: Hadoop Distributed File System (HDFS) architecture displaying in different colors basic operations: read (blue), write (green) and Namenode-Datanodes management messages (black). Note: for representation simplicity, files are not segmented into blocks and a single Namenode-Datanode message 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 [46]. 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 [47].

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 [28, 48]. While on one side, this proved to be of enormous value, on the other this posed several challenges [49] on data architectures, which had to evolve to cope with these new needs. Data lakehouse frameworks, like Hudi, Iceberg and Delta Lake [8, 9, 10] 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 On-Line Analytical Processings (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) [3], 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 [50]. All these capabilities and features make lakehouses an ideal solution for enterprise-scale data processing. Table 2.3 provides a comparison of data warehouses, data

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

Table 2.3: Comparison of key features of Data Warehouses, Data Lakes, and Data Lakehouses [51].

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.

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 [50]. 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 Open Table Format (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. Additionally, recent introduction of tools like Apache XTable * demonstrates the trend towards a universal compatibility between OTFs.

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

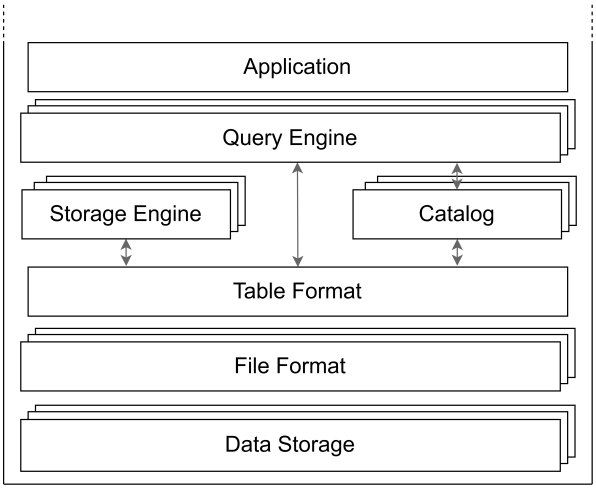

Figure 2.3: Abstraction of architecture of a data lakehouse. Inspired by "Open table formats in perspective" article on OneHouse Website, available at https://www.onehouse.ai/blog/open-table-formats-and-the-open-data-lakehouse-in-perspective

Open Table Formats

OTFs provide an abstraction layer on top of data lakes, enabling database-like functionalities, enhancing data management capabilities significantly [3, 52]. 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 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 [8], Iceberg [9] and Delta Lake [10]. 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 [53]. 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.

Maybe add some schema about the metadata layer for each technology.

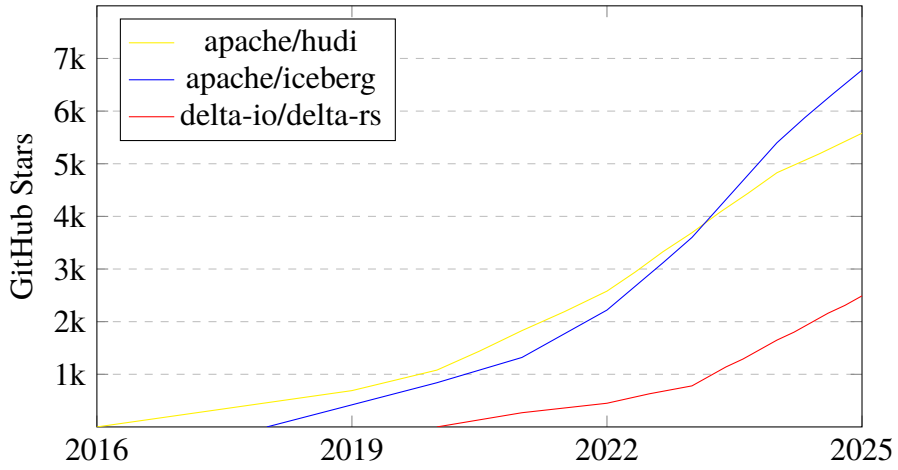

Figure 2.4: Trend of GitHub stars of following repositories https://github.com/apache/iceberg, https://github.com/apache/hudi, https://github.com/delta-io/delta-rs.

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

Apache Hudi

Apache Hudi, open-sourced by Uber in 2017 [8], 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 [54]. 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. 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 [55]. 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 [56].

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

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

support for some technologies, Hudi is currently still JVM-centric.

Apache Iceberg

Apache Iceberg, open-sourced by Netflix in 2018 [9], 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 [11, 57]. 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 [55]. 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 [12, 57]. The most common way to interact with Iceberg is through SparkSQL or other query engines like Trino and Presto, which support Iceberg natively, making Java and Scala the dominant languages in these environments. However, Iceberg offers more than just Spark bindings. PyIceberg [58] 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 [10], 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 [59]. 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 [53, 60]. 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 [59], 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 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 [61], a Java library providing low-

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

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 2.4 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 [15]. It builds upon MapReduce, a distributed programming model developed by Google for handling massive datasets [62], later adapted into Hadoop MapReduce by Yahoo! engineers [63]. Spark enhances this model using RDDs[64], 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 [64] 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 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

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

Table 2.4: Comparison of key features of Apache Hudi, Apache Iceberg, Delta Lake [54, 57, 59].

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)

other frameworks, making it not the best solution for small-scale datasets processing [20].

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 [65]. 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:

Maybe add here schema of Kafka Architecture

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 [11]. 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 [66]. 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 [8, 11], 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 [12].

Data lakehouse technologies support different Catalogs, as shown in Table 2.4. 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 [61].

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 [66], 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 [18] 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 [67]. 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 [4, 39].

The need of the described features saw new solutions emerging, like the Hopsworks AI data platform [68]. In the context of data versioning, the specific solution is called feature store [69], 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 [70], 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.

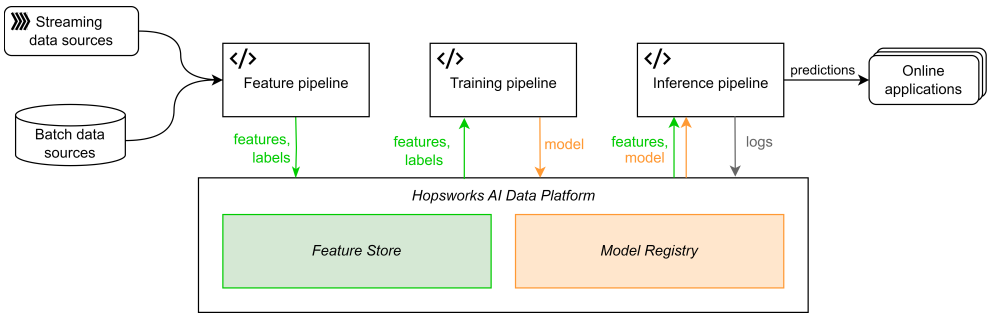

Figure 2.5: MLOps pipeline using a feature store and a model registry. Diagram inspired by Hopsworks documentation available at https://www.hopsworks.ai/dictionary/feature-store.

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 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 [26], 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.

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

Add reference to appendix

*For enhanced visualization, refer Figure.

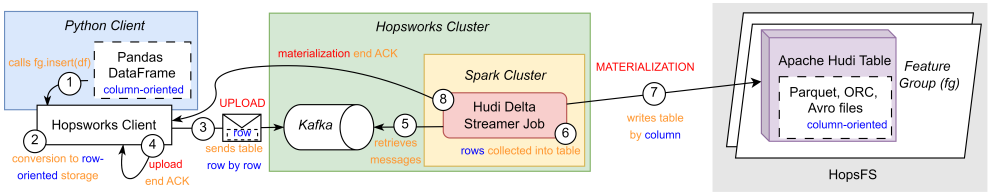

Figure 2.6: Legacy system writing a Pandas DataFrame from a Python client to the Hopsworks offline feature store. Each step is represented with a number. The table format conversion is outlined in blue, i.e., from columns to rows and then from row to columns. Steps from one to four represent the upload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

at a time. Then, when the upload is completed, the client will be notified. Asynchronously, a Spark job, the Hudi Delta Streamer, has been running in the cluster since the Hopsworks cluster was started. This job periodically retrieves messages from Kafka, and then once it retrieves a full table, it writes the table in a column-oriented format to Apache Hudi, which sits on top of a HopsFS system. Once the materialization is completed, the Python client will be notified of completion.

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

2.5.2 Legacy system - Hudi - reading

Add reference to appendix

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.

*For enhanced visualization, refer Figure.

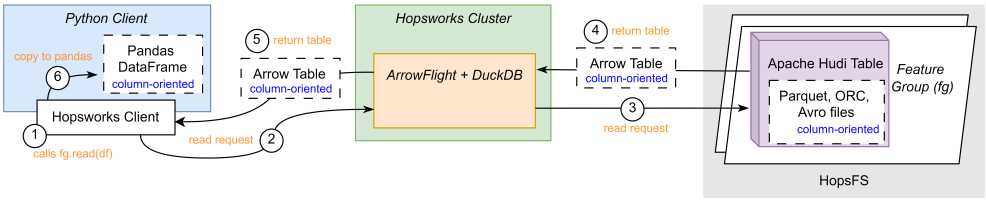

Figure 2.7: Legacy system reading a table from the Hopsworks offline feature store and loading it into the Python client's local memory. The process is streamlined using Arrow Tables that avoid table conversion. Diagram inspired by the Hopsworks feature store paper [33].

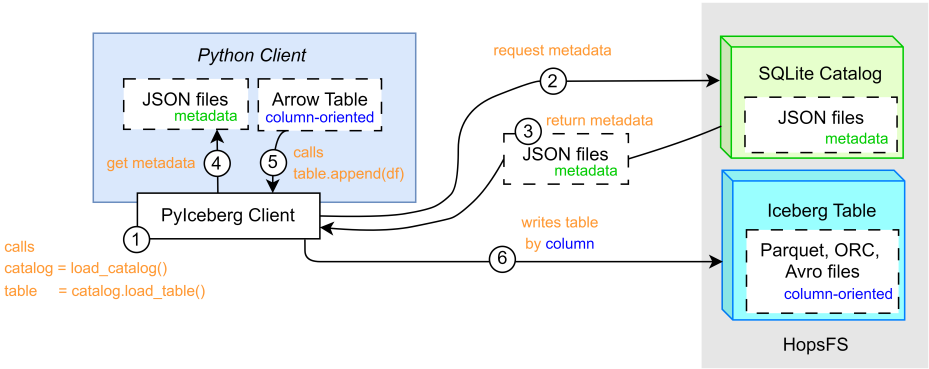

Figure 2.8: PyIceberg library writing an Arrow Table from a Python client to an Iceberg Table stored on HopsFS.

2.5.3 New system - Iceberg - 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 - Iceberg - 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

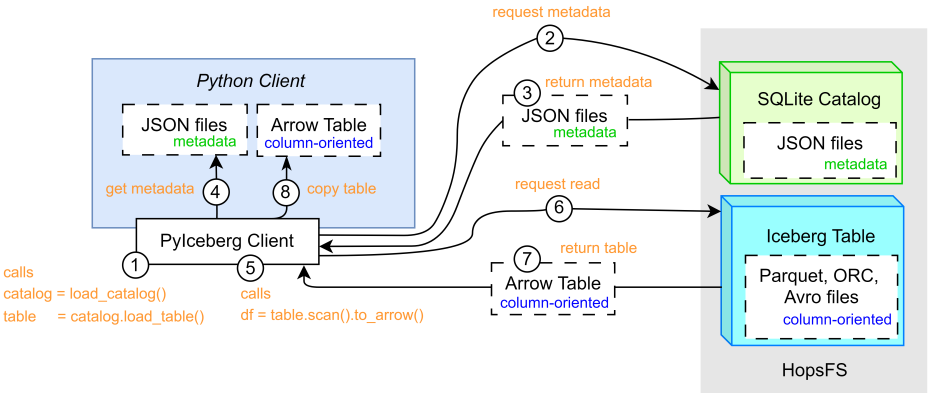

Figure 2.9: PyIceberg library reading an Iceberg Table stored on HopsFS and loading it into memory.

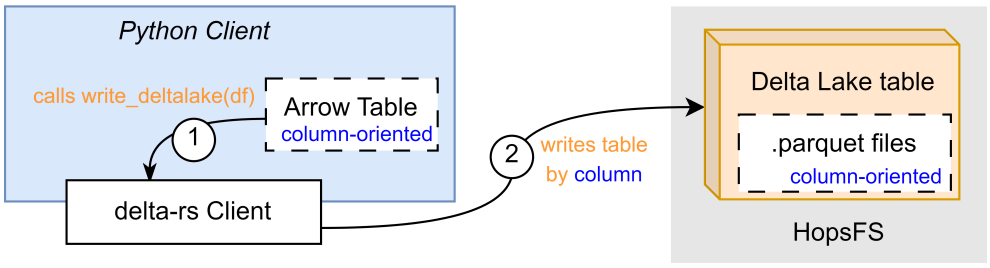

Figure 2.10: Delta-rs library writing an Arrow Table from a Python client to a Delta Lake table stored on HopsFS.

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.

2.5.5 New system - Delta Lake - 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.

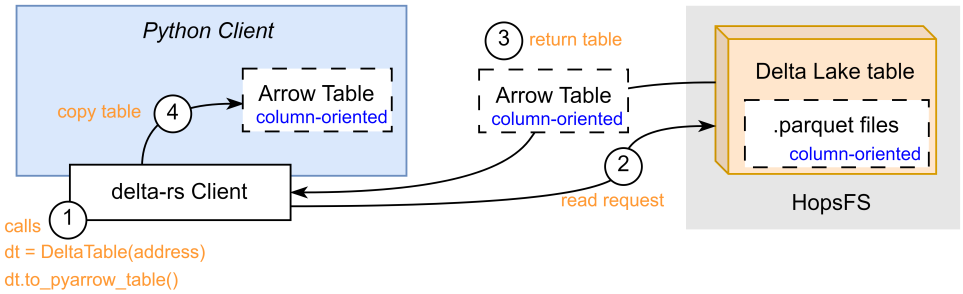

Figure 2.11: Delta-rs library reading a Delta Lake table stored in HopsFS and loading it into memory.

2.5.6 New system - Delta Lake - 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 [26]. 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.

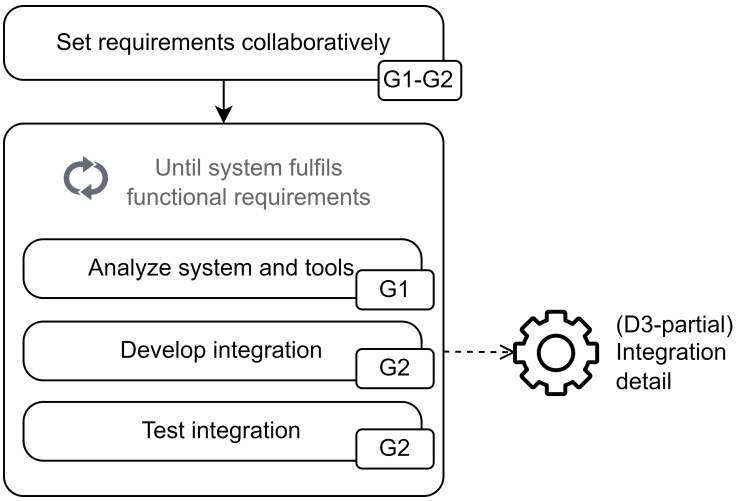

Figure 3.1: Diagram of the system integration process partially answering RQ1. Each activity is associated to specific Gs. The process produces the integration detail (D3 partial). The loop iterates until the functional requirements, defined in Section 3.1.2, are fulfilled.

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.
3. **Avoid Spark usage**: the solution must be an alternative to Spark. This is also implied in the aforementioned requirements.

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 - Hudi vs. Iceberg

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.

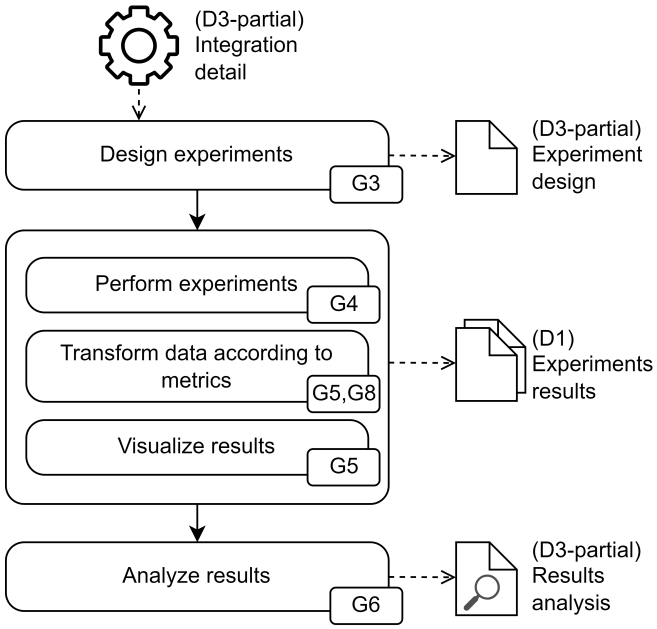

Figure 3.2: Diagram of the system evaluation process partially answering RQ1. Each activity is associated to specific G. The process produces two Ds, the experiments results (D1) and a results analysis (D3-partial).

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 [71], which became the de-facto standard for experiments on data storage system. Perhaps, it has been used in related studies [18, 72, 26].

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

add resource usage reference here

with 2048MB, which will be adjusted during the experiments, according to the needs (see Section).

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 Rust-based system, implemented in a related work [26]. Then, it presents the evaluation framework used by in this process.

3.3.1 Evaluation process - RQ2 - Iceberg vs. Delta Lake

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 took as input by this process.

2. **Visualize results**: this activity maps perfectly to G8. It takes as input also Iceberg experiments results (D1-partial), visualizing thus the experiments' result according latenc 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.

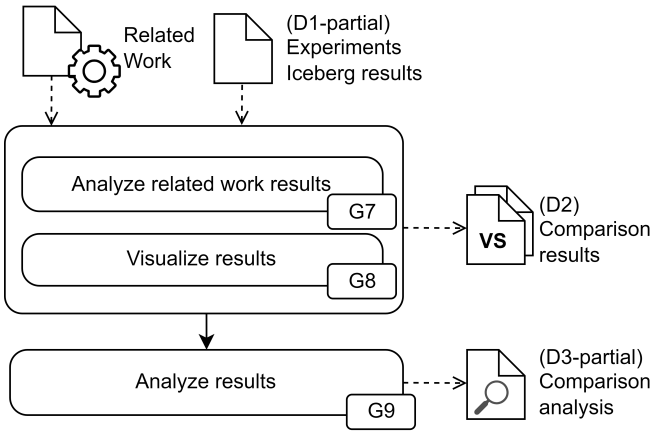

Figure 3.3: Diagram of the system evaluation process partially answering RQ2. Each activity is associated to specific G. The process produces two Ds, the comparative expertiments results (D2) and a comparative results analysis (D3-partial).

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 Iceberg, and they have been already assessed in the related work, for Delta Lake. 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 Lake pipeline?** this question answers directly RQ2, measuring the throughput of the PyIceberg pipeline defined in Section 3.2.4 and the Delta Lake pipeline [26]. 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 Software integration

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 [57], 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 the functional requirements of avoiding Spark, 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 requirerments and 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 [18, 16].

4.1.3 Integration 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,sqlite]`, as described on PyIceberg documentation [57]. This integration supports all PyIceberg methods, but this Section will focus only on the methods used for the experiments. Listing **??** describe how to instanciate an Iceberg catalog using SQLite, to enable metadata management:

Listing 4.1: Instanciating an Iceberg catalog using SQLite

```
from pyiceberg.catalog.sql import SqlCatalog


from deltalake import write_deltalake
import pandas as pd

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

Listing 4.2: Writing a DataFrame on a Delta Table with delta-rs on HDFS or HopsFS.

```
from deltalake import write_deltalake
import pandas as pd

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

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

Listing 4.3: Reading a DataFrame on a Delta Table with delta-rs on HDFS or HopsFS. Note: without the last line, the Delta Table is not loaded into memory, as delta-rs has a lazy evaluation approach.

```
from deltalake import DeltaTable

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

4.2 Experimental setup

Chapter 5

Results and Analysis

5.1 Major results

5.2 Related work results

5.3 Results analysis and discussion

Chapter 6

Conclusions and Future work

6.1 Conclusions

6.2 Limitations

6.3 Future work

References

[1] S. A. Jackson, "State of the Data Lakehouse 2025," https://www.dremio.com/. [Page 1.]

[2] "Dremio Report Highlights Surge in Data Lakehouse Adoption for Enhanced Cost Efficiency and Analytics," https://www.bigdatawire.com/this-just-in/new-dremio-report-highlights-surge-in-data-lakehouse-adoption-for-enhanced-cost-efficiency-and-analytics/, 2024. [Page 1.]

[3] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," 2021. [Pages 1, 22, and 25.]

[4] M. , "The state of AI in early 2024," https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai, May 2024. [Pages 1 and 36.]

[5] D. Croci, "Data Lakehouse, beyond the hype," Dec. 2022. [Page 1.]

[6] "What Is a Lakehouse?" https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html. [Pages 1 and 4.]

[7] "Apache Hudi vs Delta Lake vs Apache Iceberg - Data Lakehouse Feature Comparison," https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison. [Pages 1 and 2.]

[8] P. Rajaperumal, "Uber Engineering's Incremental Processing Framework on Hadoop," https://www.uber.com/en-EC/blog/hoodie/, Mar. 2017. [Pages 1, 5, 21, 26, 27, and 33.]

[9] U. Team, "Iceberg Architecture Examples," https://www.upsolver.com/blog/iceberg-architecture-examples, Feb. 2024. [Pages 1, 5, 13, 21, 26, and 28.]

[10] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3411–3424, Aug. 2020. doi: 10.14778/3415478.3415560 [Pages 1, 5, 21, 26, and 29.]

[11] T. Shiran, J. Hughes, and A. Merced, *Apache Iceberg: The Definitive Guide*. O'Reilly, Mar. 2024. ISBN 978-1-09-814861-4 [Pages 2, 6, 28, 32, and 33.]

[12] Dani, "Apache Iceberg: The Hadoop of the Modern Data Stack?" https://blog.det.life/apache-iceberg-the-hadoop-of-the-modern-data-stack-c83f63a4ebb9, Dec. 2024. [Pages 2, 28, and 33.]

[13] L. Clark, "Big data vendors embrace Apache Iceberg," https://www.theregister.com/2024/10/14/apache_iceberg_feature_announcements/. [Page 2.]

[14] "The (Ongoing) Evolution Of Table Formats," https://www.montecarlodata.com/blog-the-evolution-of-table-formats/, Sep. 2024. [Pages 2 and 6.]

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. doi: 10.1145/2934664 [Pages 2 and 30.]

[16] A. Khazanchi, "Faster reading with DuckDB and arrow flight on hopsworks : Benchmark and performance evaluation of offline feature stores," Master's thesis, KTH Royal Institute of Technology / KTH, School of Electrical Engineering and Computer Science (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2023. [Pages 2, 6, and 59.]

[17] R. Vink, "I wrote one of the fastest DataFrame libraries - Ritchie Vink," https://www.ritchievink.com/blog/2021/02/28/i-wrote-one-of-the-fastest-dataframe-libraries/. [Pages 2 and 5.]

[18] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019. doi: 10.1145/3299869.3320212. ISBN 978-1-4503-5643-5 pp. 1981–1984. [Pages 2, 5, 34, 49, and 59.]

[19] "Updates to the H2O.ai db-benchmark! – DuckDB," https://duckdb.org/2023/11/03/db-benchmark-update.html. [Page 2.]

[20] "DataFrames at Scale Comparison: TPC-H," https://docs.coiled.io/blog/tpch.html. [Pages 2 and 31.]

[21] "TIOBE Index," https://www.tiobe.com/tiobe-index/. [Pages 2 and 5.]

[22] "Stack Overflow Developer Survey 2023," https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023. [Pages 2 and 5.]

[23] G. Staff, "Octoverse: AI leads Python to top language as the number of global developers surges," Oct. 2024. [Pages 2 and 5.]

[24] "Python Machine Learning | Data | Print," https://www.packtpub.com/en-us/product/python-machine-learning-9781789955750?type=print. [Page 2.]

[25] A. Pettersson, "Resource-efficient and fast Point-in-Time joins for Apache Spark : Optimization of time travel operations for the creation of machine learning training datasets," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS) / KTH, School of Electrical Engineering and Computer Science (EECS), 2022. [Page 3.]

[26] G. Manfredi, *Reducing Read and Write Latency in a Delta Lake-backed Offline Feature Store : Adding Support for HDFS and HopsFS in the Delta-Rs Rust Library to Reduce Read and Write Latency on the Delta Lake-backed Hopsworks Offline Feature Store*. KTH Royal Institute of Technology, 2024. [Pages 3, 6, 37, 43, 49, 53, and 55.]

[27] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74, Mar. 1997. doi: 10.1145/248603.248616 [Page 4.]

[28] "The 80% Blind Spot: Are You Ignoring Unstructured Organizational Data?" https://www.forbes.com/councils/forbestechcouncil/2019/01/29/the-80-blind-spot-are-you-ignoring-unstructured-organizational-data/. [Pages 4 and 21.]

[29] "Dremel made simple with Parquet," https://blog.x.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet. [Page 4.]

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets." [Page 5.]

[31] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine." [Page 5.]

[32] G. van Rossum, "Python tutorial," no. R 9526, Jan. 1995. [Page 5.]

[33] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, A. A. Ormenisan, S. Niazi, D. Bzhalava, K. Mak, V. Jouffrey, M. Ronström, R. Cunningham, R. Zangis, D. Mukhedkar, A. Khazanchi, V. Vlassov, and J. Dowling, "The Hopsworks Feature Store for Machine Learning," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD/PODS '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024. doi: 10.1145/3626246.3653389. ISBN 9798400704222 pp. 135–147. [Pages 6, 18, and 39.]

[34] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases." [Pages 6, 11, and 18.]

[35] "The Apache Spark Open Source Project on Open Hub," https://openhub.net/p/apache-spark. [Page 7.]

[36] "What is Green Software?" https://greensoftware.foundation/articles/what-is-green-software, Oct. 2021. [Page 9.]

[37] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon Emissions and Large Neural Network Training," Apr. 2021. [Page 9.]

[38] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink," *Computer*, vol. 55, no. 7, pp. 18–28, Jul. 2022. doi: 10.1109/MC.2022.3148714 [Page 9.]

[39] "(PDF) Big Data Processing Stacks," *ResearchGate*, Oct. 2024. doi: 10.1109/MITP.2017.6 [Pages 13 and 36.]

[40] M. Frampton, *Complete Guide to Open Source Big Data Stack*. Berkeley, CA: Apress, 2018. ISBN 978-1-4842-2148-8 978-1-4842-2149-5 [Page 13.]

[41] S. Sakr, "Big Data Processing Stacks," *IT Professional*, vol. 19, no. 1, pp. 34–41, Jan. 2017. doi: 10.1109/MITP.2017.6 [Page 13.]

[42] "Block vs File vs Object Storage - Difference Between Data Storage Services - AWS," https://aws.amazon.com/compare/the-difference-between-block-file-object-storage/. [Pages 14, 15, and 17.]

[43] "How Object vs Block vs File Storage differ," https://cloud.google.com/discover/object-vs-block-vs-file-storage. [Pages 14, 15, and 17.]

[44] "Object vs. File vs. Block Storage: What's the Difference? | IBM," https://www.ibm.com/think/topics/object-vs-file-vs-block-storage, Aug. 2024. [Pages 14, 15, and 17.]

[45] A. Kala Karun and K. Chitharanjan, "A review on hadoop — HDFS infrastructure extensions," in *2013 IEEE Conference on Information & Communication Technologies*, Apr. 2013. doi: 10.1109/CICT.2013.6558077 pp. 132–137. [Page 17.]

[46] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, "Cloud Storage as the Infrastructure of Cloud Computing," in *2010 International Conference on Intelligent Computing and Cognitive Informatics*, Jun. 2010. doi: 10.1109/ICICCI.2010.119 pp. 380–383. [Page 20.]

[47] M. Ismail, S. Niazi, G. Berthou, M. Ronström, S. Haridi, and J. Dowling, "HopsFS-S3: Extending Object Stores with POSIX-like Semantics and more (industry track)," in *Proceedings of the 1st International Middleware Conference Industrial Track*. Delft Netherlands: ACM,

Dec. 2020. doi: 10.1145/3429357.3430521. ISBN 978-1-4503-8201-4 pp. 23–30. [Page 20.]

[48] H. E. Pence, "What is Big Data and Why is it Important?" *Journal of Educational Technology Systems*, vol. 43, no. 2, pp. 159–171, Dec. 2014. doi: 10.2190/ET.43.2.d [Page 21.]

[49] Y. Demchenko, Z. Zhao, P. Grosso, A. Wibisono, and C. de Laat, "Addressing Big Data challenges for Scientific Data Infrastructure," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, Dec. 2012. doi: 10.1109/CloudCom.2012.6427494 pp. 614–617. [Page 21.]

[50] D. Mazumdar, J. Hughes, and J. B. Onofre, "The Data Lakehouse: Data Warehousing and More," Oct. 2023. [Pages 22 and 24.]

[51] B. Inmon, "Five Steps to a successful data lakehouse," vol. 2021. [Page 23.]

[52] "Data Lakehouse: A survey and experimental study," *Information Systems*, vol. 127, p. 102460, Jan. 2025. doi: 10.1016/j.is.2024.102460 [Page 25.]

[53] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia, "Analyzing and Comparing Lakehouse Storage Systems," 2023. [Pages 26 and 29.]

[54] "Apache Hudi | Technical Documentation," https://hudi.apache.org/docs/overview. [Pages 27 and 31.]

[55] O. Katz, "Hudi vs Iceberg vs Delta Lake: Data Lake Table Formats Compared," Apr. 2021. [Pages 27 and 28.]

[56] "Apache Iceberg vs Delta Lake," https://www.starburst.io/blog/iceberg-vs-delta-lake/. [Page 27.]

[57] "Apache Iceberg | Technical Documentation," https://iceberg.apache.org/docs/nightly/. [Pages 28, 31, 57, and 59.]

[58] "PyIceberg," https://py.iceberg.apache.org/. [Page 28.]

[59] "Delta Lake | Technical Documentation," https://docs.delta.io/latest/index.html. [Pages 29 and 31.]

[60] S. Asad, "Apache Iceberg vs Delta Lake vs Apache Hudi: A Comprehensive Comparison," Jan. 2025. [Page 29.]

[61] "Announcing Delta Lake 3.0 with New Universal Format and Liquid Clustering," https://www.databricks.com/blog/announcing-delta-lake-30-new-universal-format-and-liquid-clustering, Wed, 06/28/2023 - 23:53. [Pages 29 and 34.]

[62] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. doi: 10.1145/1327452.1327492 [Page 30.]

[63] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," 2007. [Page 30.]

[64] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." [Page 30.]

[65] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing." [Page 32.]

[66] R. Blue, "Catalogs and the REST catalog," 2024. [Pages 33 and 34.]

[67] wesm, "Introducing Apache Arrow Flight: A Framework for Fast Data Transport," https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/, Oct. 2019. [Page 35.]

[68] "Hopsworks - The Real-time AI Lakehouse," https://www.hopsworks.ai/. [Page 36.]

[69] "Meet Michelangelo: Uber's Machine Learning Platform," https://www.uber.com/en-IN/blog/michelangelo-machine-learning-platform/, Sep. 2017. [Page 36.]

[70] "The Big Dictionary of MLOps," https://www.hopsworks.ai/mlops-dictionary, 2024. [Page 36.]

[71] "TPC-H Benchmark: Decision support, standard verification, revision v3.0.1," https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp, 2022. [Page 49.]

[72] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. Van Bussel, H. Van Hovell, M. Xue, R. Xin, and M. Zaharia, "Photon: A Fast Query Engine for Lakehouse Systems," in *Proceedings of the 2022 International Conference on Management of Data*. Philadelphia PA USA: ACM, Jun. 2022. doi: 10.1145/3514221.3526054. ISBN 978-1-4503-9249-5 pp. 2326–2339. [Page 49.]

Appendix A

System architectures

This appendix contains the system architecture diagrams, for both the legacy system and the Apache Iceberg version.

Appendix B

Write experiments results

This appendix contains the writing diagrams, showing the barplots and the schema with all the information the writing experiments performed on the different systems.

Appendix C

Read experiments results

This appendix contains the reading diagrams, showing the barplots and the schema with all the information the reading experiments performed on the different systems.

Appendix D

Legacy pipeline write latency breakdown results

This appendix contains images about the legacy operation latencies breakdowns

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": "Accelerating Feature Store performance with Apache Iceberg",
"Subtitle": "A Python-based alternative delivering speed and scalability without Spark dependency",
"Language": "eng" },
"Alternative title": {
"Main title": "Snabbare prestanda för Feature Store med Apache Iceberg",
"Subtitle": "Ett Python-baserat alternativ som ger snabbhet och skalbarhet utan Spark-beroende",
"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": "Dowling",
"First name": "Jim",
"E-mail": "jim@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": "1,77"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA – EECS-EX" , "No. in series": "2025:0000" },
"Opponents": { "Name": "Name Surname"},
"Presentation": { "Date": "2022-03-15 13:00"
,"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]": €€€€

Write an abstract that is about 250 and 350 words (1/2 A4-page) with the following components:

- What is the topic area?
- (optional) Introduces the subject area for the project.
- Short problem statement
- Why was this problem worth a Master's thesis project? (*i.e.*, why is the problem both significant and of a suitable degree of difficulty for Master's thesis project? Why has no one else solved it yet?)
- How did you solve the problem? What was your method/insight?

- Results/Conclusions/Consequences/Impact: What are your key results/ conclusions? What will others do based on your results? What can be done now that you have finished - that could not be done before your thesis project was completed?

€€€€,
"Keywords[eng]": €€€€
Machine Learning, Feature Store, Spark, Apache Iceberg, Python, Read/write latency €€€€,
"Abstract[swe]": €€€€
€€€€,
"Keywords[swe]": €€€€
Maskininlärning, Feature Store, Spark, Apache Iceberg, Python, Läs- och skrivlatens €€€€,
"Abstract[ita]": €€€€
€€€€,
"Keywords[ita]": €€€€
5-6 parole chiave €€€€,
}

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}
```