Degree Project in ?

Second cycle, 30 credits

Faster Delta Lake operations using Rust

How Delta-rs beats Spark in a small scale Feature Store

GIOVANNI MANFREDI

Faster Delta Lake operations using Rust

How Delta-rs beats Spark in a small scale Feature Store

GIOVANNI MANFREDI

Master's Programme, ICT Innovation, 120 credits
Date: October 2, 2024

Supervisors: Sina Sheikholeslami, Fabian Schmidt, Salman Niazi
Examiner: Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: Detta är den svenska översättningen av titeln
Swedish subtitle: Detta är den svenska översättningen av undertiteln

Abstract

Here I will 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 Bachelor's/Master's thesis project? (*i.e.*, why is the problem both significant and of a suitable degree of difficulty for a Bachelor's/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

Canvas Learning Management System, Docker containers, Performance tuning First keyword, Second keyword, Third keyword, Fourth keyword

Sammanfattning

Här ska jag skriva ett abstract som är på ca 250 och 350 ord (1/2 A4-sida) med följande komponenter:

- Vad är ämnesområdet? (valfritt) Presenterar ämnesområdet för projektet.
- Kort problemformulering
- Varför var detta problem värt en kandidat-/masteruppsats? (*i.e.*, varför är problemet både betydande och av en lämplig svårighetsgrad för ett kandidat-/masteruppsats-projekt? Varför har ingen annan löst det än?)
- Hur löste du problemet? Vad var din metod/insikt?
- Resultat/slutsatser/konsekvenser/påverkan: Vilka är dina viktigaste resultat/
  slutsatser? Vad kommer andra att göra baserat på dina resultat? Vad kan göras nu när du är klar - som inte kunde göras innan ditt examensarbete var klart?

Nyckelord

Canvas Lärplattform, Dockerbehållare, Prestandajustering Första nyckelordet, Andra nyckelordet, Tredje nyckelordet, Fjärde nyckelordet

Sommario

Qui scriverò un abstract di circa 250 e 350 parole (1/2 pagina A4) con i seguenti elementi:

- Qual è l'area tematica? (opzionale) Introduce l'area tematica del progetto.
- Breve esposizione del problema
- Perché questo problema meritava un progetto di tesi di laurea/master? (Perché il problema è significativo e di un grado di difficoltà adeguato per un progetto di tesi di laurea/master? Perché nessun altro l'ha ancora risolto?)
- Come avete risolto il problema? Qual è stato il vostro metodo/intuizione?
- Risultati/Conclusioni/Conseguenze/Impatto: Quali sono i vostri risultati chiave/ conclusioni? Cosa faranno gli altri sulla base dei vostri risultati? Cosa si può fare ora che avete finito - che non si poteva fare prima che il vostro progetto di tesi fosse completato?

parole chiave

Prima parola chiave, Seconda parola chiave, Terza parola chiave, Quarta parola chiave

Acknowledgments

I would like to thank xxxx for having yyyy.

Stockholm, June 2024
Giovanni Manfredi

Contents

List of acronyms and abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
AI	Artificial Intelligence
API	Application Programming Interface
BI	Business Intelligence
BPMN	Business Process Model and Notation
CoC	Conquer of Completion
CPU	Central Processing Unit
D	Deliverable
DBMS	Data Base Management System
DFS	Distributed File System
ELT	Extract Load Transform
ETL	Extract Transform Load
G	Goal
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HopsFS	Hopsworks' HDFS distribution
IN	Industrial Need
JVM	Java Virtual Machine
LocalFS	Local File System
ML	Machine Learning
OLAP	On-Line Analytical Processing
OS	Operating System
PA	Project Assumption
PC	Personal Computer

RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RPC	Remote Procedural Call
RQ	Research Question
SDG	Sustainable Development Goal
SF	Scale Factor
SSD	Solid State Drive
SSH	Secure Shell protocol
TLS	Trasport Layer Security
TPC	Transaction Processing Performance Council
VM	Virtual Machine

Chapter 1

Introduction

Lakehouse systems are increasingly becoming the primary choice for running analytics in large-sized companies (that have more than 1000 employees) [1].

This recent architecture design called Lakehouse [2] is preferred over old paradigms, i.e. data warehouses and data lakes, as it builds upon the advantages of both systems, having the scalability properties of data lakes while preserving the Atomicity, Consistency, Isolation and Durability (ACID) properties typical of data warehouses [2]. Additionally, Lakehouse systems include partitioning, which reduces query complexity significantly and provides "time travel" capabilities, enabling users to access different versions of data, versioned over time [3].

Three main implementations of this paradigm emerged over time [4]:

1. **Apache Hudi**: first introduced by Uber, now primarily backed by Uber, Tencent, Alibaba, and Bytedance.

2. **Apache Iceberg**: first introduced by Netflix and now primarily backed by Netflix, Apple, and Tencent.

3. **Delta Lake**: first introduced by Databricks and now primarily backed by Databricks and Microsoft.

While large communities back all three projects, Delta Lake is acknowledged as the de-facto Lakehouse solution [4]. This is mainly thanks to Databricks, which first promoted this new architecture over data lakes among their clients around 2020 [5].

As a data query and processing engine, Delta Lake is typically used with Apache Spark [6]. This approach is effective when processing large quantities of data (1 TB or more) over the cloud, but whether this approach is effective on small quantities of data (100 GB or less) remains to be investigated [7].

DuckDB [8], a Data Base Management System (DBMS) and Polars [9], a DataFrame library, highlighted the limitations of Apache Spark. When the data volume is small (between 1 GB and 100 GB) and the architecture is processing data locally, an Apache Spark cluster performs worse than alternatives. This ultimately brings an increase in costs and computation time [10, 11].

Another aspect to remember is that thanks to its ease of use and high abstraction level, Python has become the most used programming language in the data science space [12]. Python is currently also the most popular general purpose programming language [13, 14] and it is by far the most used language for Machine Learning (ML) and Artificial Intelligence (AI) applications [15], this is mainly thanks to its strong abstraction capabilities and accessibility. This trend can also be observed by looking at the most popular libraries among developers, where two Python libraries make the podium: NumPy and Pandas [14]. In this scenario, creating a Python client for Delta Lake would be beneficial as it would not have to resort to Apache Spark and its Python Application Programming Interface (API) (PySpark). This approach with small-scale (between 1 GB and 100 GB) use cases would improve performance significantly.

This native Python interface for Delta Lake directly benefits Hopsworks AB, the host company of this master thesis. Hopsworks AB develops a Feature Store for ML, a centralized, collaborative data platform that enables the storage and access of reusable features [16]. This architecture also supports point-in-time correct datasets from historical feature data [17].

This project here presented, aims to increase the data throughput (rows/second) for reading and writing on Delta Lake tables that act as an Offline Feature Store in Hopsworks. Currently, the pipeline is Apache Spark-based and the key hypothesis of the project is that a faster non-Apache Spark alternative is possible. If effective, Hopsworks will consider integrating this system implementation into the Hopsworks Feature Store (open source version), greatly improving the experience of Python users working on small quantities of data (between 10 GB and 100 GB). More generally, this work will outline the possibility of Apache Spark alternatives in small-scale (between 10 GB and 100 GB) use cases.

1.1 Background

A clear understanding of the background of this project comes from appreciating three different key aspects: Lakehouse development, Apache

Spark relevance and flows, and Python as an emergent language.

Lakehouse is a term coined by Databricks in 2020 [18], to define a new design standard that was emerging in the industry that combined the capability of data lakes in storing and managing unstructured data, with the ACID properties typical of Data warehouses. Data warehouses became a dominant standard in the '90s and early 2000s [19], enabling companies to generate Business Intelligence (BI) insights, managing different structured data sources. The problems related to this architecture were highlighted in the 2010s when the need to manage large quantities of unstructured data rose [20]. So Data lakes became the pool where all data could be stored, on top of which a more complex architecture could be built, consisting of data warehouses for BI and ML pipelines. This architecture, while more suitable for unstructured data, introduces many complexities and costs, related to the need of having replicated data (data lake and data warehouse), and several Extract Load Transform (ELT) and Extract Transform Load (ETL) computations. Lakehouse systems solved the problems of Data lakes by implementing data management and performance features on top of open data formats such as Parquet [21]. Three key technologies enabled this paradigm: (i) a metadata layer for data lakes, tracking which files are part of different tables, (ii) a new query engine design, providing optimizations such as RAM/SSD caching, and (iii) an accessible API access for ML and AI applications. This architecture design was first open-sourced with Apache Hudi in 2017 [22] and then Delta Lake in 2020 [5].

Spark is a distributed computing framework used to support large-scale data-intensive applications [23]. Spark builds from the roots of MapReduce and its variants. MapReduce is a distributed programming model first designed by Google that enables the management of large datasets [24]. The paradigm was later implemented as an open-source project by Yahoo! engineers under the name of Hadoop MapReduce [25]. Spark significantly improved the performance of Hadoop MapReduce (10 times better in its first iteration) [23] thanks to its use of Resilient Distributed Datasets (RDDs) [26]. RDDs is a distributed memory abstraction that enables a lazy in-memory computation that is tracked through the use of lineage graphs, ultimately increasing fault tolerance [26]. This means that Spark avoids going back and forth between storage disks to store the computation results, as represented in Figure 1.1.

Spark, which is open-sourced under the Apache foundation as Apache Spark [27] (from now on simply Spark), has seen widespread success and adoption in various applications, becoming the de-facto data-intensive

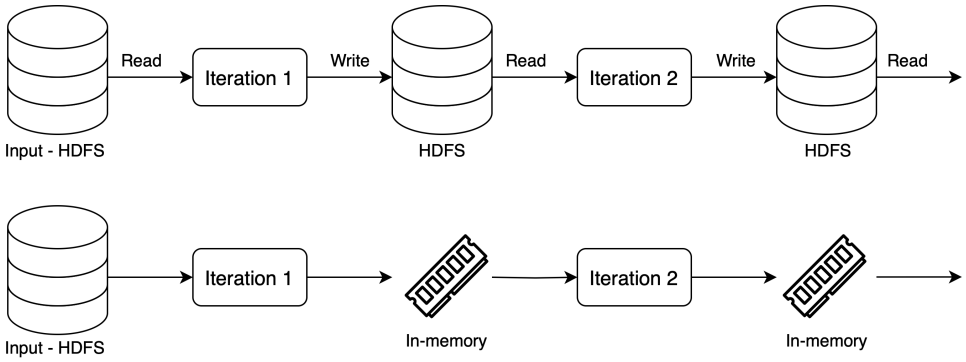


Figure 1.1: Difference between a Hadoop MapReduce execution and an Apache Spark execution. Every step in Hadoop MapReduce must be saved, while Apache Spark operates in memory.

computing platform for the distributed computing world. While Spark is often used as a comprehensive solution [6], different solutions might be better suited for a specific scenario. An example of this is the case of Apache Flink [28], designed for real-time data streams, which prevails over Spark where low latency real-time analytics are required. Similarly, Spark might not be the best tool for lower-scale applications where the high-scaling capabilities of Spark may not be required. This is the case of DuckDB [8] and Polars [9], that by focusing on low scale (10GB-100GB) provide a fast On-Line Analytical Processing (OLAP) embedded database and DataFrame management system respectively offering an overall faster computation compared to starting a Spark cluster for to perform the same operations. This shows the possibility for improvements and new applications that substitute the current Spark-based systems in specific applications such as real-time data streaming or small-scale computation. In this project, the latter application is going to be explored.

Python can be considered the primary programming language among data scientists [29]. Many first adopted Python thanks to its focus on ease of use, high abstraction level, and readability. This helped create a fast-growing community behind the project, which led to the development of many libraries and APIs. So now, more than 30 years after its creation, it has become the de-facto standard for data science thanks to many daily used Python libraries such as TensorFlow, NumPy, SciPy, Pandas, PyTorch, Keras and many others.

Python is also considered to be the most popular programming language, according to the number of results by search query (+*"<language> programming"*) in 25 different search engines [30]. This is computed yearly in

the TIOBE Index [13]. Looking at the 2024 list, it can be noted that Python has a rating of 15.16%, followed by C which has a rating of 10.97%. The index also shows the trends of the last years, clearly displaying the rise of Python over historically very popular languages such as C and JAVA, which were both outranked by Python between 2021 and 2022. This shows the importance of offering Python APIs for programmers and data scientists in particular to increase the engagement and possibilities of a framework.

1.2 Problem

The Hopsworks Feature Store [16] when querying the Offline Feature Store, uses Spark as a query engine, i.e. executes the query (read, write, or delete) on the Offline Feature Store.

If no Spark job was started before (as is always the case in the open-source server-less Hopsworks app), the operation, even if small in size (only retrieving 1 GB of data or less), will take a few minutes (1-2 minutes) to complete. This is mainly due to the overhead of starting a Spark cluster and running a Spark job. This overhead is less relevant for computation on larger quantities of data (1 TB or above), as it composes a smaller part of the overall computation time. Nonetheless, Hopsworks' typical use-case sits between tests on small quantities of data (scale between 1-10 GBs) and production scenarios on a larger scale, but still relatively small (scale between 10-100 GBs). As this overhead is a Spark-specific issue, it grows the need to look for Spark alternatives. Currently, Hopsworks is saving their Feature Store data on Apache Hudi and Delta Lake table formats. Delta Lake supports Spark alternatives for accessing and querying the data, and of particular interest is the delta-rs library [31] that enables Python access to Delta Lake tables, without having the time overhead given by Spark jobs. However, the delta-rs [31] does not support Hadoop Distributed File System (HDFS), and consequently Hopsworks' HDFS distribution (HopsFS) [32].

1.2.1 Research Question

This research project has the ultimate objective to evaluate and compare the performance of the current Spark system that operates on Apache Hudi, to a Rust system that uses delta-rs library [31] operates on Delta Lake, using HopsFS [32]. To achieve this, support for HDFS (and thus also HopsFS) must be added to the delta-rs library [31], so that it can be compatible with the

Hopsworks system. Thus the project addresses the following two Research Questions (RQs):

RQ1: How can we add support for HDFS and HopsFS to the delta-rs library?

RQ2: What is the difference in throughput (measured in rows/second) between a Spark-based operation (read or write) to Apache Hudi compared a the delta-rs library-based operation (read or write) to Delta Lake, in HopsFS?

1.3 Purpose

The purpose of this thesis project is to contribute to reducing the read and write time, and thus increasing the data throughput (rows/second), for operations on the Hopsworks Offline Feature Store. This work will identify which one between a Spark pipeline and a delta-rs pipeline performs better at a small scale, by comparing the differences in read time, write time, and computed throughput (rows/second). As a prospect for future work, if delta-rs is proven to be a more performant alternative (in terms of data throughput, i.e. rows/second), Hopsworks will consider integrating this pipeline into their application.

Overall implications for this thesis work are much wider counting the popularity Spark has in the open source community (more than 2800 contributors during its lifetime [33]). This would enable developers to have a wider range of alternatives when working on "small scale" (1 GB to 100 GB) systems by choosing delta-rs over Spark.

1.4 Goals

The accomplishment of the project's purpose (namely, increasing the data throughput (rows/second) for reading and writing on Delta Lake tables on HopsFS) is bound to a list of Goals (Gs), here set. These are also related to the set of RQs, outlining a clear structure of the various project milestones.

1. Gs aimed to answer RQ1:

   G1: Understand delta-rs library [31] architecture and dependencies.

   G2: Identify what needs to be implemented to add HDFS support to the delta-rs library [31].

G3: Implement HDFS support in the delta-rs library [31].

G4: Verify that HDFS support also works for HopsFS.

2. Gs aimed to answer RQ2:

G5: Design the experiments to be conducted to evaluate the difference in performance between Spark-based access to Apache Hudi compared a the delta-rs [31] library-based access to Delta Lake, in HopsFS.

G6: Perform the designed experiments.

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

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

Associated with these Gs several Deliverables (Ds) will be created.

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

D2: Experiment results on the difference in performance between Spark-based access to Apache Hudi compared a the delta-rs library-based access to Delta Lake, in HopsFS. This D is related to the completion of goals G5–G7.

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

1.5 Ethics and Sustainability

As a systems research project, the focus of this study revolves around software and in particular, developing more efficient data-intensive computing pipelines that find wide application in machine learning and training of neural networks. Software according to the Green Software Foundation [34] can be "part of the climate problem or part of the climate solution" [35]. We can define Green

Software as a software that reduces its impact on the environment by using less physical resources, and less energy and optimizing energy use to use lower-carbon sources [35]. In the context of machine learning and training of neural networks, reducing training time (and so also the read and write time operation on the dataset) has been proven to positively impact the reduction in carbon emissions [36, 37].

This project, by aiming to increase the data throughput (rows/second) for reading and writing on Delta Lake tables on HopsFS, follows the key green software principles reducing CPU time use compared to the previous Spark-based pipeline. This leads to a lower carbon footprint, as less energy is being used.

This project contributes to the Sustainable Development Goals (SDGs) 7 (Affordable and Clean Energy) and 9 (Industry Innovation and Infrastructure) [38], more specifically the targets 7.3 (Double the improvement in energy efficiency) and 9.4 (Upgrade all industries and infrastructures for sustainability). This work achieves this by reducing the read and write time of data on Delta Lake tables, and thus increasing the data throughput. This means that the same amount of data can be read or written in a smaller amount of time, reducing the use of resources (CPU or GPU computing time), thus reducing energy usage. This decrease in energy consumption will lead to a smaller carbon footprint (if the same amount of data is read or written).

Ultimately, this leads to an improvement in energy efficiency and a reduction in the carbon footprint of the data-intensive computing pipelines that find wide application in machine learning and training of neural networks.

Figure 1.2: SDGs to which this thesis contributes to.

1.6 Research Methodology

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

IN1 : the Hopsworks Feature Store has a lower throughput (rows/second) in reading and writing operations when performed on a "small scale" (1 GB - 100 GB) compared to a "large scale" (100 GB - 1 TB). This highlights the potential for improvement in the "small scale" use case.

IN2 : Hopsworks, adapting to their customer needs, supports the Delta Lake table format. Improving the speed of read and write operations on this table format, would improve a typical use case for Hopsworks Feature Store users.

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 means that Python libraries with high performance will typically be preferred over alternatives (even more efficient) that are Java Virtual Machine (JVM) or other environments based.

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

These assumptions will be validated in Chapter 2.

The project aims at fulfilling the INs with a system implementation approach. First, a HDFS storage support will be written for the delta-rs library to extend the Rust library support to HopsFS [32]. Then, an evaluation structure will be designed and used to compare the performances of the old Spark-based system and the new Rust-based pipeline. The two approaches will be tested with datasets of different sizes (between 1 GB and 100 GB). This is critical to identify if the same tool should be used for all scenarios or if they perform differently. The critical metrics that will be used to evaluate the system are read and write operations data throughout (the higher, the better) measured in rows per second. These were chosen as they most affect the computation time of pipelines accessing Delta Lake tables.

1.6.1 Delimitations

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

1.7 Thesis Structure

Once the thesis is written, provide an outline of the thesis structure

Chapter 2

Background

This project works on a layered data stack, that handles Big Data, i.e. large volumes of various structured and unstructured data types at a high velocity. The data stack handles how data is stored, managed, and retrieved to enable applications built on top of it. As there is no single data architecture that is generally accepted, i.e. different approaches use different architectures [39, 40], thus this project defines a data stack, then focuses on improving its parts, namely the query engine. The data stack of the project is illustrated in Figure 2.1.

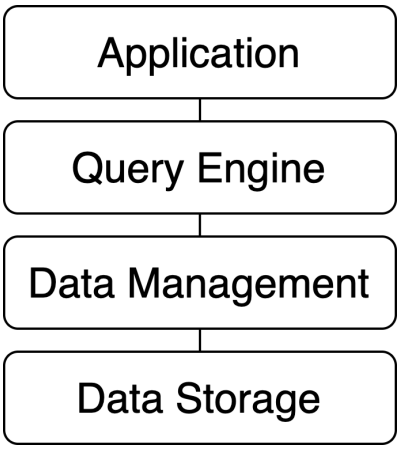


Figure 2.1: Data stack for how is considered in the project.

The data stack is divided into four sections:

1. **Data Storage**: handles how the data is stored. The data storage layer might be centralized or distributed, on-premise or on the cloud, storing

data in files, objects, or blocks.

2. **Data Management** : handles how the data is managed. The data management layer might offer ACID properties, data versioning, support open data formats, and/or support structured and/or unstructured data.

3. **Query Engine**: handles how data is queried, i.e. accessed, retrieved and written. The query engine might offer caching, highly scalable architectures, and/or API support for multiple programming languages.

4. **Application**: a system that will take advantage of the data stack. In the case of this project, only the software of the host organization (Hopsworks' Feature Store) will be described.

After explaining the data stack, a section on programming languages complements the Background by explaining the roles different programming languages play in the data stack (namely, Python, Java, Scala, and Rust).

2.1 Data Storage

This section aims to describe the data storage layer of this project, namely HopsFS. HopsFS is Hopsworks's evolution of HDFS, a distributed file system. HopsFS complexity will be broken down into parts, providing not only a great understanding of the tool but also a comparison with common alternatives, namely cloud object stores.

2.1.1 File storage vs. Object storage vs. Block storage

Data can be stored and organized in physical storages, such as Hard Disk Drives (HDDs) and/or Solid State Drives (SSDs), in three major ways: (1) Files, (2) Objects, and (3) Blocks. For each one, the technique is briefly described and then a comparative table is shown (Table 2.1). This subsection is a rework according to the author's understanding of three articles coming from major cloud providers (Amazon, Google, and IBM) [41, 42, 43].

File storage

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

directories (called "subdirectories"). In many file storage systems directories are called "folders".

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

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

Advantages

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

Disadvantages

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

Object storage

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

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

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

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

Advantages

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

Disadvantages

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

Block storage

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

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

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

Advantages

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

Disadvantages

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

Table 2.1: Comparison between data storage different characteristics.

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

2.1.2 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a Distributed File System (DFS), i.e. a file system that uses distributed storage resources while providing a single namespace as a traditional file system. HDFS has significant differences compared with other DFSes.HDFS is highly fault-tolerant, i.e. it is resistant to hardware failures of part of its infrastructure, and can be deployed on commodity hardware. HDFS also provides high throughput access to application data and it is designed to be highly compatible with applications with large datasets (more than 100 GB) [25].

HDFS architecture consists of a single primary node called Namenode and multiple secondary nodes called Datanodes. The Namenode manages the filesystem namespace and regulates access to files by clients. On the other hand, Datanodes manage the storage attached to the nodes they run on and they are responsible for performing replication requests when prompted by the Namenode. HDFS exposes to users a file system namespace where data can be stored in files. Internally, a file is divided into one or multiple blocks and these

blocks are stored in a set of Datanodes. The blocks are also replicated upon the first write operation, up to a certain number of times (by default three times, with at least one copy on a different physical infrastructure). The Namenode keeps track of the data location, matching it with the filesystem namespace. It is also responsible for managing Datanode reachability (through periodical state messages sent by Datanodes), and providing clients with the locations of the Datanodes containing the blocks that compose the requested file. If a new write request is received, it is still the Namenode that needs to provide the locations of available storage for the file blocks.

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

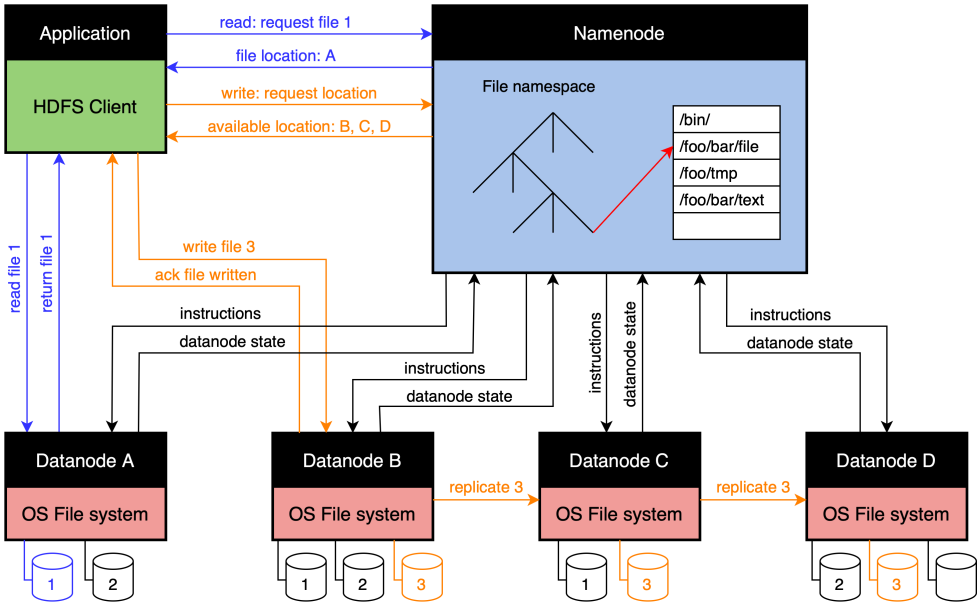


Figure 2.2: Hadoop Distributed File System (HDFS) architecture displaying in different colors basic operations: read (blue), write (orange) and Namenode-Datanodes management messages (black). Note: for representation simplicity files are not segmented into blocks.

Chapter 3

Method

3.1 System implementation – RQ1

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

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

3.1.1 Development process

The development process will follow an iterative and incremental development approach described in Figure 3.1. This methodology will be applied as it allows flexibility while creating incrementally a working system [44]. This project, due to the need to work on HopsFS, will require numerous interactions with HopsFS maintainers (i.e. the industrial supervisor). This creates the need for a feedback loop, which will allow the system to fit all the requirements according to all stakeholder's expectations.

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

1. **Identify problems collaboratively**: this activity solves partially G1–

G2, as it is an initial system analysis, performed together with the industrial supervisor, who is knowledgeable on Hopsworks' infrastructure (in particular HopsFS). This task fixes the initial requirements of the project and investigates what needs to be implemented at a high-level abstraction.

2. **Analyse system**: this activity solves partially G1–G2 each time is re-iterated, as it performs low-level code-based analysis of how the system works and what needs to be implemented to support HDFS in delta-rs. This activity also starts an iterative loop that will end once the system fulfills the requirements described in Section 3.1.2.

3. **Design software**: this activity solves partially G3, as the first part of the software development. In this activity, the system analyzed before is considered to design a solution.

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

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

6. **Verify system integration**: this activity solves partially G4, as the second part of the tests performed to verify the solution validity, via integration tests. Failed integration tests will trigger a new development loop iteration, where this failure will be considered as first starting point in the system analysis. On the other hand, if the integration test succeeds, the loop will be restarted if the system does not yet fit a requirement, or finished if the system fulfills all requirements described in Section 3.1.2.

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

3.1.2 Requirements

In the first steps of the system analysis, a series of requirements are defined in agreement with the industrial partner Hopsworks AB, to favor the creation of a solution that could be later used within the company, in a production

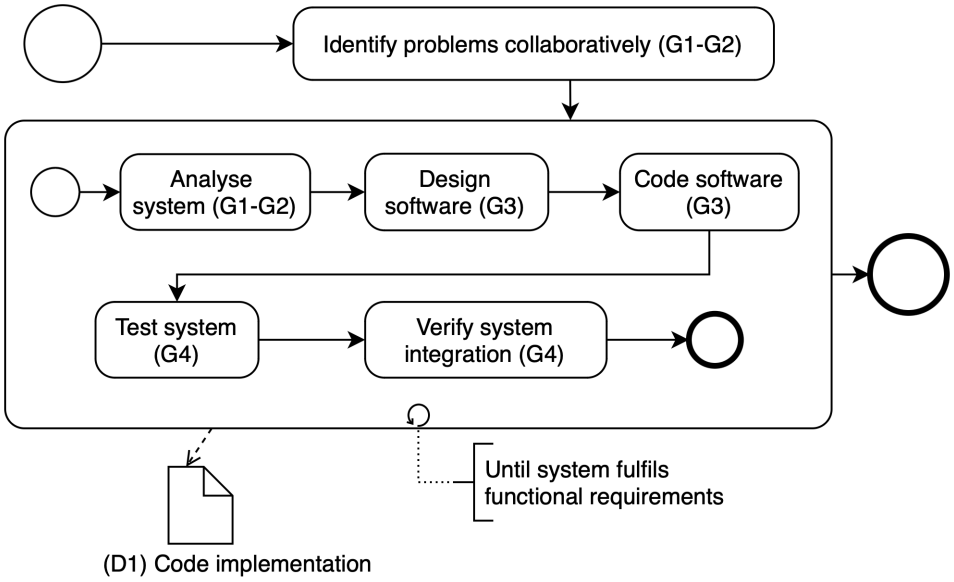


Figure 3.1: Business Process Model and Notation (BPMN) diagram of the System implementation process answering to RQ1. Each activity is associated with a specific Goal (G). The process produces a deliverable (D), in this case, a code implementation. The development loop iterates until the functional requirements (defined in Section 3.1.2) are fulfilled.

environment. These are divided into two categories: functional and non-functional requirements.
The **functional requirements** are:

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

2. **Read Delta Tables**: the solution should allow to read Delta Lake tables on HopsFS via the delta-rs library.

3. **Communicate via TLS**: the solution should interact with HopsFS via Trasport Layer Security (TLS) protocol version 1.2.

The **non-functional requirements** are:

1. **Consistent**: the solution should be consistent with the current open-source codebase used when appropriate.

2. **Maintainable**: the solution should minimize the need for maintenance and support of the codebase in the future, minimizing changes to open-source code. When appropriate, the changes the solution introduces should be compatible with a future upstream merge to the open-source project modified.

3. **Scalable**: the solution should be able to handle larger quantities of data (up to 100 GB) read or written on Delta Tables.

3.1.3 Development environment

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

- **Computing resources**: the system implementation will be developed in a remote environment accessed via Secure Shell protocol (SSH) from a computer terminal. This remote Virtual Machine (VM) is selected as mounting HopsFS on a local machine is non-trivial and developing locally could result in inconsistencies when the solution is reproduced in a virtual environment.

- **Writing code**: the Vim [45] text editor is development tool of choice in combination with Conquer of Completion (CoC) [46] providing language-aware autocompletion and rust-analyzer [47] access for on-code compiler errors.

- **Libraries and dependencies**: for simpler development and test reproducibility, the environment will be set in a Docker container [48].

- **Code versioning and shared development**: GitHub [49] will be used for versioning, collaborating with open-source projects (e.g. delta-rs), and sharing the developed solution.

3.2 System evaluation – RQ2

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

This section details the method and the principles that will be used to carry out the system evaluation process, measuring the performance (throughput,

measured in rows/second) of reading and writing on Delta Lake or Apache Hudi while on HopsFS of Spark-based and Rust-based pipelines.

3.2.1 Evaluation Process and Research Paradigm

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

1. **Design experiments**: this activity maps perfectly to G5, designing the experiments that will be conducted to evaluate the performance difference in performance between Spark-based access to Apache Hudi compared a the delta-rs [31] library-based access to Delta Lake, in HopsFS.

2. **Perform experiments**: this activity maps perfectly to G6, using the code implementation (D1) to conduct the designed experiments on the analyzed systems.

3. **Transform data according to metrics**: this activity is necessary to fulfill G7, as data visualization requires data to be properly formatted. In this activity data is modified, performing a division between the size of the table inputted and the read or write time. This allows data to be in the wanted metric i.e. rows/second.

4. **Visualize results**: this activity maps perfectly to G7, visualizing the experiments' result according to the selected metric, i.e. throughput measured in rows/second. This activity also generates a deliverable (D2) composed of the experiments results complete with tables and histograms, i.e. Chapter 5.

5. **Analyze results**: this activity maps perfectly to G8, analyzing and interpreting the results delivered in D2. This contributes to D3, generating the analysis of results, i.e. Chapter 5.

The research paradigm consists of a quantitative analysis based on repeated runs on the implemented system. The results of the benchmarks are then analyzed performing a visual comparative approach.

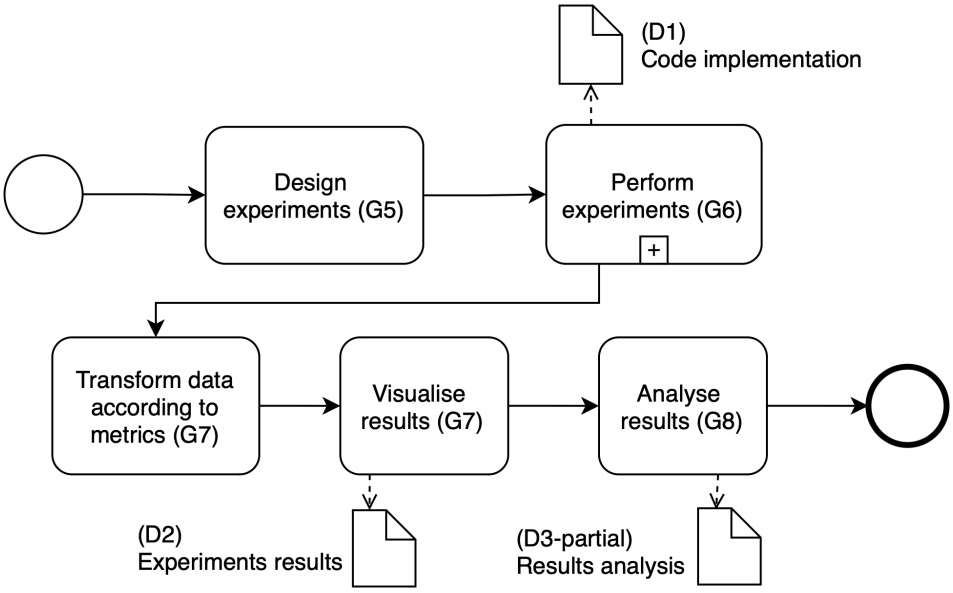


Figure 3.2: BPMN diagram of the System evaluation process answering to RQ2. Each activity is associated with a specific Goal (G). The process produces two deliverables (D), the experiments results (D2) and a results analysis (D3-partial).

3.2.2 Dataset

The data that will be used to perform the read and write operations comes from TPC-H benchmark suite[50]. TPC-H is a decision support benchmark by Transaction Processing Performance Council (TPC). It consists of a series of business-oriented ad-hoc queries designed to be industry-relevant [51]. The data coming from this benchmark suite was used as it provides a recognized standard for data storage systems [52], and it has already been used in similar related work [8, 53]. Any part of the data can be generated using the TPC-H data generation tool [54].

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

The SUPPLIER table has seven columns, while the LINEITEM table has

sixteen. This influences the average size of memory each row occupies. This means the metric selected (throughput, measured in rows/second) cannot be used to compare results across different tables. This is the reason why the comparative evaluation only considers different configurations on the same table.

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

1. *supplier_sf1*: size = 10000 rows
2. *supplier_sf10*: size = 100000 rows
3. *supplier_sf100*: size = 1000000 rows
4. *lineitem_sf1*: size = 60000000 rows
5. *lineitem_sf10*: size = 60000000 rows

3.2.3 Experimental Design

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

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

Furthermore, the experiments will verify how the performances of the three systems will change based on the Central Processing Unit (CPU) resources provided (namely 1 core, 2 cores, 4 cores, 8 cores). Each time

the testing environment will be modified accordingly, creating a new VM where the experiments will run with increasingly more resources. These CPU configurations were chosen together with the industrial supervisor, according to the typical Hopsworks use case.

The data used for experiments, as described in Section 3.2.2, will come from two different tables. These are modified according to a SF, for a total of five times.

In conclusion, the experiments conducted will be a total of 3 (pipelines) times 4 (CPU configurations) times 5 (tables) times 2 (read and write operations), i.e. 120 experiments, performed 50 times each to create statistically significant results.

3.2.4 Experimental environment

The experimental environment consists of a physical machine in Hopsworks' offices, virtualized enabling remote shared development. The CPU details of the machine are present in Listing 3.1, noting that only eight cores at maximum were accessed during the experiments.

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

The experimental environment will be set up with a Jupyter Server of different CPU cores, depending on the experiment (1 core, 2 cores, 4 cores, or 8 cores). The Jupyter server is allocated by default with 2048 MB of Random Access Memory (RAM), out of the .. available on the experimental machine. This amount will be adjusted during the experiments according to the needs of the experiments, up to an available maximum of 192 GB (actually less, as it is a shared environment).

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

```
Architecture:                    x86_64
  CPU op-mode(s):                32-bit, 64-bit
  Address sizes:                 48 bits physical,
                                 48 bits virtual
  Byte Order:                    Little Endian
CPU(s):                          32
  On-line CPU(s) list:           0-31
Vendor ID:                       AuthenticAMD
  Model name:                    AMD Ryzen Threadripper
                                 PRO 5955WX 16-Cores
```

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

3.2.5 Evaluation Framework

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

1. **Functional requirements**: defined in Section 3.1.2, functional requirements will measured by verifying the success or failure of running an experiment. By design, this will not happen, as the system implementation phase, continues until all functional requirements are met.

2. **Non-functional requirements**: defined in Section 3.1.2, non-functional requirements are: consistency, maintainability and scalability. The first two requirements are mainly addressed during implementation, while scalability is measured during the system evaluation experiments. The metric used for measuring this requirement is the throughput measured in rows/second as defined in RQ2.

3. **How does the system compare to other pipelines?**: this question answers directly RQ2, measuring the throughput (rows/second) of the different pipelines, defined in Section 3.2.3. Results are then compared using a visual approach.

3.2.6 Assessing Reliability and Validity

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

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

Chapter 4

Implementation

4.1 Software design and development

The first step of the development process, defined in Section 3.1.1, consists of identifying what the delta-rs library (version 0.15.x) lacks to satisfy the requirements, more specifically, how to support HDFS and HopsFS in the delta-rs [31] library. This step outlines the library structure (in Rust slang "crate") divided into sublibraries (in Rust slang "subcrates"), which is illustrated in Figure 4.1. As the figure shows, the delta-rs crate has a subcrate for each storage connector, so adding support for different storage is a matter of implementing a new subcrate to the delta-rs crate.

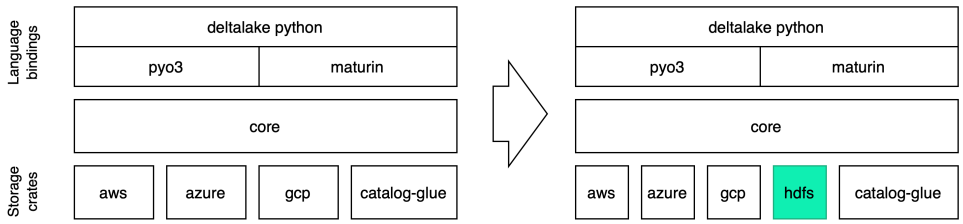


Figure 4.1: Architecture of the delta-rs library (version 0.15.x) before and after support to HDFS was added. The implemented solution is highlighted in green.

The library uses an external interface called object-store defined in the Arrow-rs library [55]. Every storage connector implements this interface, and the other parts of the library interact with the storage layer. It is thus crucial that the new HDFS storage also implements this interface.

The development process was divided into two subsections: a first approach and a final approach. The first approach was carried out by developing a HDFS subcrate for the delta-rs crate from scratch, while the second and final approach modified the support for HDFS added in delta-rs with version 0.18.2 to also support HopsFS.

4.1.1 First approach

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

1. The HDFS support caused the testing pipeline to fail, and no trivia solution was found.
2. The HDFS support had JVM dependencies which weighted a lot on the environment required on delta-rs users (i.e. having Java installed), and this was considered a requirement that would some users from using the library altogether.
3. The community around the library did not have contributors or a large number of users which had HDFS as a use case.

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

This implementation makes use of a C++ library called libhdfs, which contains all the methods required to work as an HDFS client. This library is contained in the Rust library fs-hdfs. The datafusion-objectstore-hdfs makes use of the libhdfs library, to provide an interface for HDFS that implements object-store, the interface used in the delta-rs library to interact with storage connectors.

This approach required first to rewrite the datafusion-objectstore-hdfs as it made use of an older object-store interface version (0.8.0 vs. 0.10.0) and it was not possible anymore to upgrade it easily. Secondly, the JVM dependencies while this project did not have a strict requirement on not having them, being able to have the dependencies consistently work on different development environments proved to be a challenge.

Ultimately, this approach was abandoned following the release of version 0.18.2 of the delta-rs library. This decision was taken to comply with the maintainability requirement defined in Section 3.1.2.

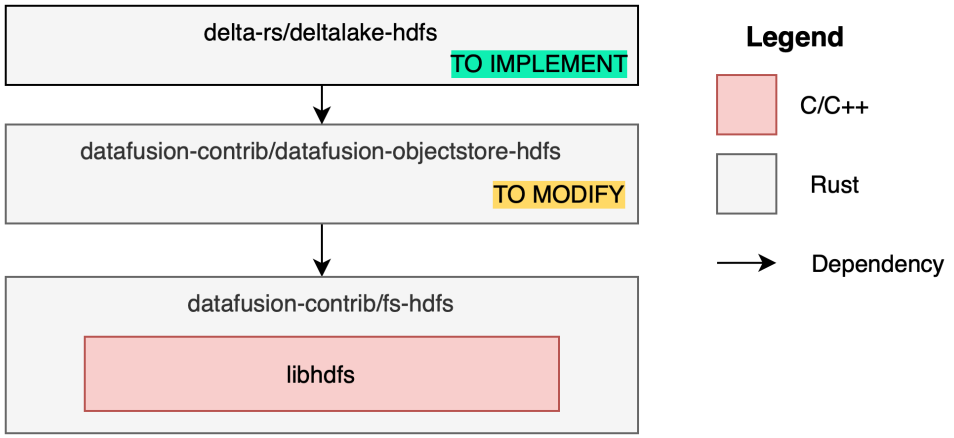


Figure 4.2: Architecture of the first approach to add support to HDFS in the delta-rs library.

4.1.2 Final solution

Version 0.18.2 of the delta-rs introduced support for HDFS via the hdfs-native library [56]. This is a Rust library that re-implements the HDFS client, avoiding to use of libhdfs, and thus has no JVM dependencies. This architecture is illustrated in Figure 4.3.

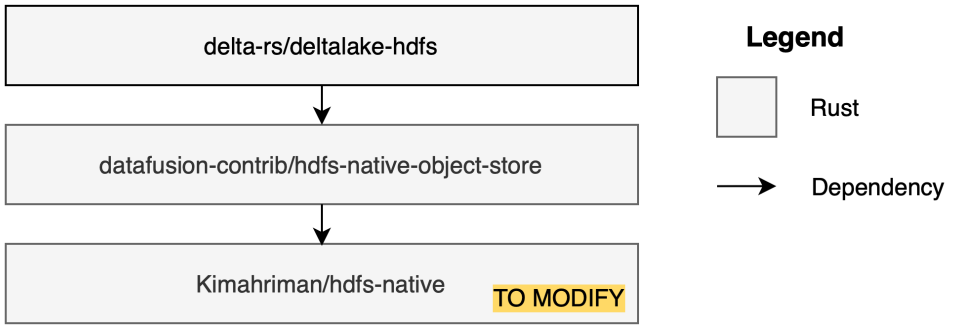


Figure 4.3: Architecture of the second approach to the system solution. Note that here HDFS support is given by the hdfs-native library, but to allow HopsFS support the library needs to be modified.

This section approach, while being used by the delta-rs library, proved to have some incompatibilities with HopsFS. Here below is a list of them:

1. **Different HDFS protocol version**: HDFS makes use of a Remote Procedural Call (RPC) protocol to interact with a HDFS client. Hdfs-native is based on protocol version 3.2, while HopsFS was compatible with version 2.7.

2. **No support for TLS in hdfs-native**: hdfs-native security is based on Kerberos, but it does not secure packet transfers using TLS.

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

1. **Upgrading HopsFS protocol version**: together with the industrial supervisor, responsible for the maintenance of HopsFS, the differences between the two protocol versions (2.7 vs. 3.2) were highlighted, and one by one resolved. This way version 3.2.0.14 of HopsFS was released, adding support to the HDFS RPC protocol version 3.2, making HopsFS compatible with the hdfs-native library.

2. **Adding support for TLS in the hdfs-native library**: TLS support to hdfs-native was added via the use of an external Rust library called tokio-rustls version 0.26 [57].

All changes were applied to copies (in Git slang called "forks") of the open-source repositories related to this project (delta-rs, hdfs-native-object-store, hdfs-native). These changes are available on the author's repository [58, 59, 60].

4.2 Software deployment and usage

Once HDFS and HopsFS support has been added to the delta-rs library it is sufficient to build a python wheel, i.e. a pre-built binary package format for Python modules and libraries, for delta-rs. To do so it is sufficient to follow the instructions already present in the delta-rs library in the README.md file present in the python folder [61].

The usage of the delta-rs library is explained in detail in the delta-rs documentation [31], so in this section, only the method used for the experiments will be shown and explained. Listing 4.1 shows a simple example of writing to HDFS or HopsFS (formally in a Python script only the address changes, as HopsFS is based on HDFS). As is clear from the listing in this case the script is writing a small table of two columns and three rows.

Listing 4.1: Writing a data frame 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)
```

In Listing 4.2 an example of a read operation is shown. After being read, the Delta Table is converted to a pyarrow table, then a pydictionary to be easily displayed.

Listing 4.2: Reading a data frame on a Delta Table with delta-rs on HDFS or HopsFS

```
from deltalake import DeltaTable

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

4.3 Experiments set-up

Experiments, as defined in Section 3.2.3, consist of running different system configurations, with different data, fifty times per experiment. Two main approaches were selected to measure the experiment's time, here is explained how to set them up.

The first approach is to use the Python timeit function. As illustrated in Listing 4.3 timeit can be used by defining a SETUP_CODE that runs before the experiment and a TEST_CODE that when running is measured and the time (expressed in seconds) is the return value of the timeit function. This approach was selected as the timeit function provides a clear interface to run and measure a small code script. The script here does not run a repeated number of times as the Delta Table must be deleted before re-running the experiment, and this requires time that shouldn't be included in the experiment time (nor in the setup, as the first time the table is not there). When this approach could not be used (due to more complex scripts, the second approach was used).

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

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

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

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

The second measuring approach was to simply record the time before the script run and after the script run, then calculate the difference. This made it possible to calculate multiple differences without having to recreate the experiment multiple times. Listing 4.4 shows an example of this approach.

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

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

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

write_result = after_writing - before_writing
```

Chapter 5

Results and Analysis

5.1 Major Results

Following the experimental process described in Section 3.2, 120 experiments were performed 50 times each for statistical significance. This section highlights the main results from the performed experiments.

5.1.1 Writing Experiments

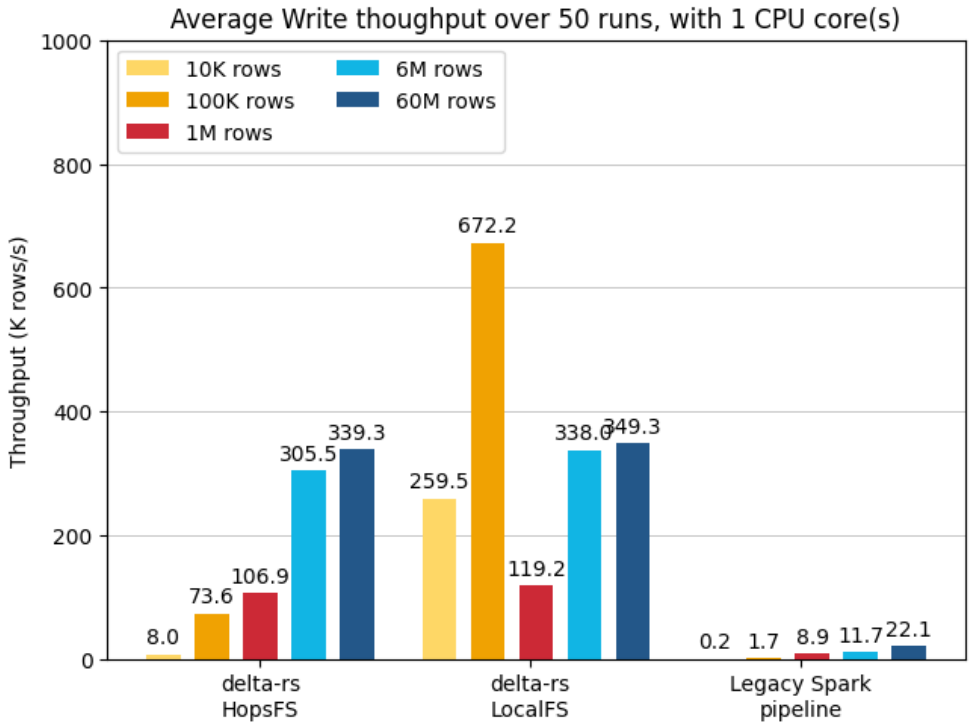
Figures 5.1a and 5.1b show the write throughput of all three pipelines defined in Section 3.2.3 when writing the five different tables defined in Section 3.2.2. This experiment makes use of 1 CPU core.

Each value reported was calculated by measuring the time taken to write the table. An example consists of delta-rs on HopsFS which took 184.12783 seconds to write 60 M rows, so dividing the rows by time the throughput obtained is 325860.566 rows/second. This result is then resampled using the bootstrapping technique, and the average, in this case 339.288 K rows/second is obtained. A 95% confidence interval was also calculated (in this case ±5.05 K rows/second), but it was not displayed in the graph as it would be hardly readable as all results are out of each other's 95% confidence interval.

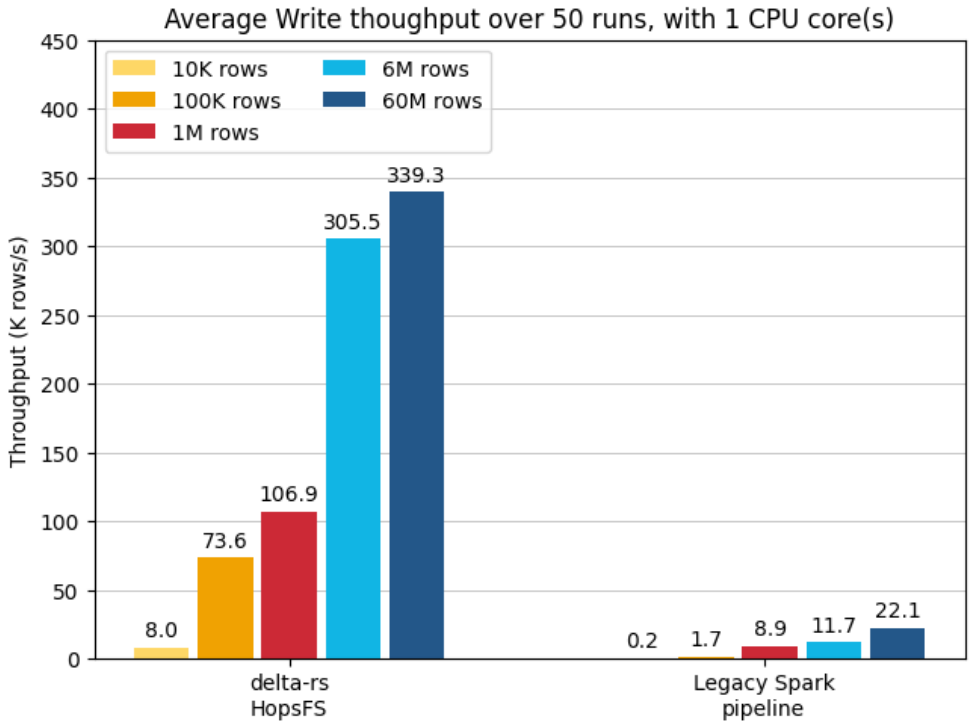
5.1.2 Reading Experiments

Figures 5.2a and 5.2b show the read throughput of all three pipelines defined in Section 3.2.3 when writing the five different tables defined in Section 3.2.2. This experiment makes use of 1 CPU core.

Each value reported was calculated by measuring the time taken to read



(a)



(b)

Figure 5.1: (a) Throughput for writing tables (from the TCP-H benchmark) via the three different pipelines defined in Section 3.2.3. This experiment uses 1 CPU core. (b) As for (a) but showing only the legacy system and newly implemented system.

the table. An example consists of delta-rs on HopsFS which took 0.0378069 seconds to read 60 M rows, so dividing the rows by time the throughput obtained is 1587.0114 M rows/second. This result is then resampled using the bootstrapping technique, and the average, in this case 1929.861 M rows/second is obtained. A 95% confidence interval was also calculated (in this case ±102.7 M rows/second), but it was not displayed in the graph as it would be hardly readable as all results are out of each other's 95% confidence interval.

5.1.3 Experiments with more CPU cores

Experiments reading and writing tables from the TPC-H benchmark in the three different pipelines defined in Section 3.2.3 were repeated with increasingly more CPU cores. The Tables 5.1a and 5.1b display the percentage increase between the first and the increased CPU cores experiment. For example the delta-rs HopsFS pipeline throughput when writing a 60M rows table is 339.288979358 k rows/second, when using 1 CPU core. During the experiment when it was using 8 CPU cores the throughput was 494.7344718446817 k rows/second. Thus the performance increase of the throughput as a percentage is 45.82 % (calculated as the rate between the increase and the 1 CPU experiment multiplied by 100). Note that in some cases the performance increase is negative, meaning that the throughput decreased even if computational resources were increased.

5.1.4 Writing using legacy Spark pipeline – Time breakdown

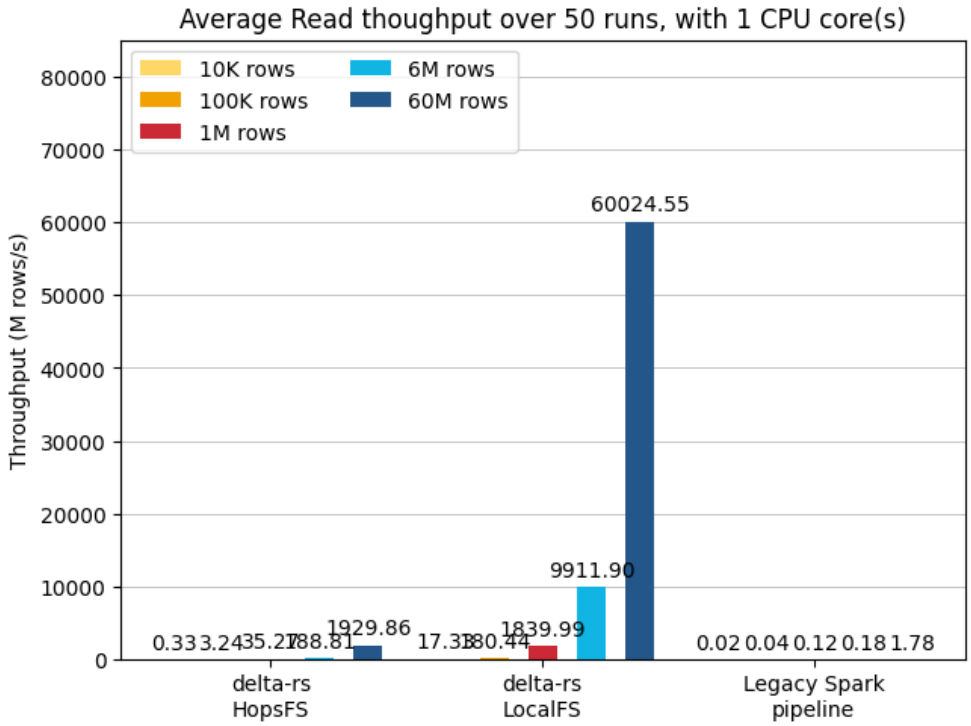
During the writing experiment performed using the legacy Spark pipeline the contribution of different part of the process were measured: namely the upload time and materialization time, dichotomy explained in Section

Ref here time breakdown expl.

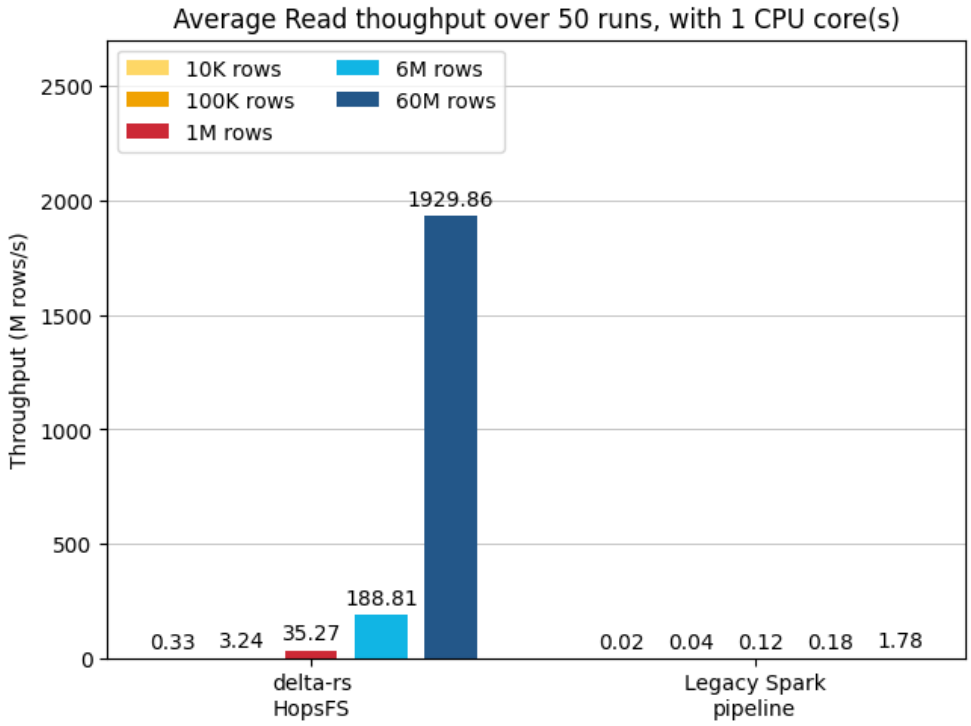
. This served to verify how different part of the legacy pipeline scaled with table sizes, and if Spark was in fact the bottleneck of the architecture.

5.2 Results Analysis and Discussion

This section complements the results section by analyzing the results showed in Section 5.1 one by one.



(a)



(b)

Figure 5.2: (a) Throughput for reading tables (from the TCP-H benchmark) via the three different pipelines defined in Section 3.2.3. This experiment uses 1 CPU core. (b) As for (a) but showing only the legacy system and newly implemented system.

Pipeline	Rows number	1 CPU core throughput (k rows/second)	2 CPU cores (% increase)	4 CPU cores (% increase)	8 CPU cores (% increase)
delta-rs HopsFS	10K	8.005477660	-0.94	2.87	-2.37
	100K	73.550934236	4.77	1.89	5.88
	1M	106.932925730	10.41	11.69	13.20
	6M	305.503687034	21.39	21.93	26.17
	60M	339.288979358	32.46	43.43	45.82
delta-rs LocalFS	10K	259.548165052	-18.62	-12.92	-9.38
	100K	672.239894235	9.15	13.83	10.25
	1M	119.198512205	17.51	17.65	16.84
	6M	338.020789871	17.29	23.20	25.42
	60M	349.252743491	32.89	42.35	43.81
Legacy Spark	10K	0.199571379	-1.03	-2.09	-1.99
	100K	1.681863191	-0.35	0.04	-1.28
	1M	8.922017340	3.28	3.04	2.48
	6M	11.722915952	8.13	6.19	7.56
	60M	22.099807565	15.99	15.75	16.78

(a) Writing Experiment

Pipeline	Rows number	1 CPU core throughput (M rows/second)	2 CPU cores (% increase)	4 CPU cores (% increase)	8 CPU cores (% increase)
delta-rs HopsFS	10K	0.3316539	-3.12	11.85	1.10
	100K	3.237530	14.15	9.27	17.28
	1M	35.26714	-1.87	-5.09	-4.15
	6M	188.8125	9.35	8.21	11.75
	60M	1929.861	2.03	6.53	-1.90
delta-rs LocalFS	10K	17.32663	3.40	8.11	1.33
	100K	180.4406	1.75	0.10	-5.01
	1M	1839.990	-6.14	2.36	-5.58
	6M	9911.899	-0.06	-2.35	-6.69
	60M	60024.55	29.03	17.94	7.89
Legacy Spark	10K	0.01586513	0.90	-0.12	0.57
	100K	0.03773604	-0.49	0.39	-0.44
	1M	0.1176966	0.27	-2.10	2.71
	6M	0.1791499	0.42	0.22	0.25
	60M	1.783280	0.11	0.11	1.62

(b) Reading Experiment

Table 5.1: (a) Table showing the percentage increase in the throughput during the writing experiment with the increase of CPU cores allocated. Experiments are indicated for each table and pipeline defined in Section 3.2.3. The throughput of the 1 CPU writing experiment is reported for reference. (b) As for (a) but for the reading experiment.

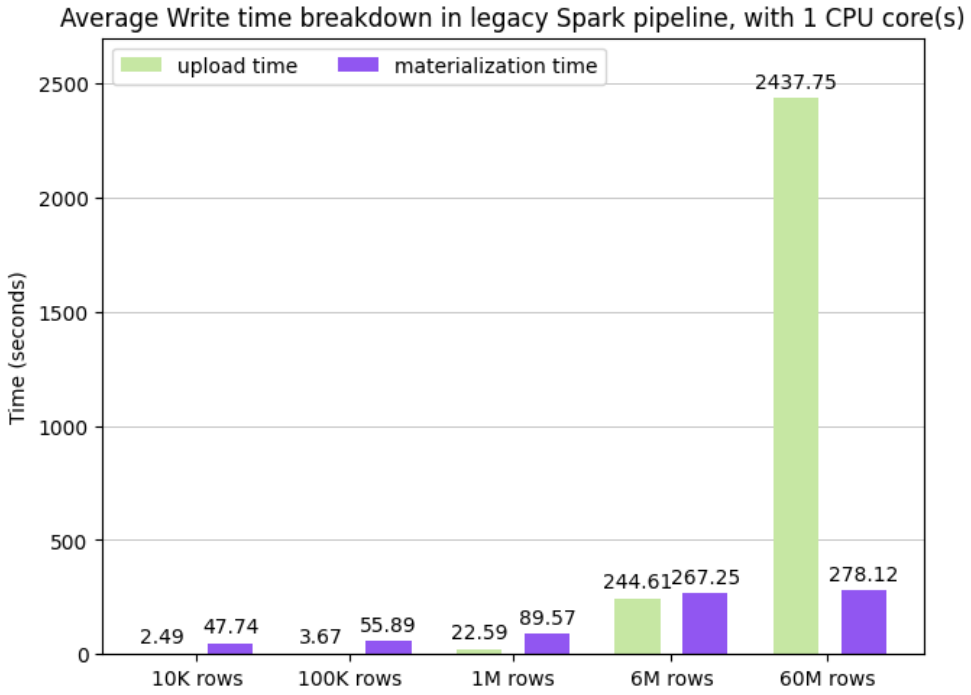


Figure 5.3: Breakdown of the time contribution to the time needed to execute a write operation on the legacy Spark pipeline. To know more about the legacy Spark pipeline and the upload-materialize dichotomy refer to Section

ref to background explaining

5.2.1 Write operations using delta-rs are up to 40 times faster than the legacy Spark pipeline

The writing experiment reveals that the use of the delta-rs library achieves up to 40 times the performance of the Legacy Spark pipeline with smaller tables (10K and 100K rows). The delta-rs pipelines (both on HopsFS and on LocalFS) achieve higher performances even with bigger tables (1M, 6M and 60M rows), but with lower rates (15-26 times better).

This experimental result shows that a Rust pipeline writing data on Delta Lake tables is indeed faster than the legacy Spark pipeline in all writing tests. The rate of this improvement varies (40 times better vs. 15.35 times better when comparing 10K and 60M rows) and it is decreasing with the size of the table. This suggests that a writing a larger table (more than 60M rows) the Legacy Spark pipeline perform better than the delta-rs pipelines. This insight can be verified with further experiments using larger tables (more than 60M

rows).

5.2.2 Read operations using delta-rs are increasingly faster than the legacy Spark pipeline

The reading experiment reveals that delta-rs pipelines achieve increasingly better performance as the size of the table grows. The throughput in the delta-rs on HopsFS is up to 3 significant figures greater than the throughput of the legacy Spark pipeline.

This experiment results shows that a Rust pipeline reading data on Delta Lake tables is indeed faster than the legacy Spark pipeline. Moreover, contrarily to what suggested in the introduction, i.e. that a Spark based pipeline would have faster performance as the size of the table would increase, this is not the case with the analyzed experiments, where the difference in proportion between the two pipelines grows significantly as the table sizes increase (20 times better vs. 1000+ times better when comparing 10K and 60M rows).

5.2.3 Increasing the CPU cores does not increase the read or write performance dramatically

The experiments run with more CPU cores reveals that even if the computational resources increase the throughput does not increase linearly, e.g. no experiment run with 2 CPU cores achieved more than a 33% throughput increase. Moreover, reading experiments did not see major improvements (more than 20%) in throughput with the exception of the experiments performed with delta-rs on the LocalFS. On the other hand, writing experiments did show improvements in throughput, even if limited if compared to the increase in computing power, e.g. the best results was achieved with larger tables where adding a single core resulted in a 32.89% improvement, but this is limited if the fact that computing resources were doubled is considered.

5.2.4 The upload time in the legacy Spark pipeline becomes the bottleneck as table size increases

In the writing experiment on the legacy Spark pipeline, as explained in Section

Ref here time breakdown expl.

, writing time can be broken down into two upload and materialization time. This experiment reveals that as the table size increases, the contribution of the upload time drastically changes (4.96% vs. 89.76% contribution when comparing 10K and 60M rows tables). This suggests that the upload is the phase that limits the system to scale more efficiently, and thus have a higher throughput, as table size increases.

References

[1] "State of the Data Lakehouse," Dremio, Tech. Rep., 2024. [Page 1.]

[2] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, vol. 8, 2021. [Page 1.]

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

[4] "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. [Page 1.]

[5] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta lake: High-performance ACID table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, Aug. 2020. doi: 10.14778/3415478.3415560 [Pages 1 and 3.]

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. doi: 10.1145/2934664 [Pages 1 and 4.]

[7] 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. [Page 1.]

[8] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, Jun. 2019. doi: 10.1145/3299869.3320212. ISBN 978-1-4503-5643-5 pp. 1981–1984. [Pages 2, 4, and 24.]

[9] R. Vink, "I wrote one of the fastest DataFrame libraries," https://www.ritchievink.com/blog/2021/02/28/i-wrote-one-of-the-fastest-dataframe-libraries/, Feb. 2021. [Pages 2 and 4.]

[10] "Benchmark Results for Spark, Dask, DuckDB, and Polars — TPC-H Benchmarks at Scale," https://tpch.coiled.io/. [Page 2.]

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

[12] A. Nagpal and G. Gabrani, "Python for Data Analytics, Scientific and Technical Applications," in *2019 Amity International Conference on Artificial Intelligence (AICAI)*, Feb. 2019. doi: 10.1109/AICAI.2019.8701341 pp. 140–145. [Page 2.]

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

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

[15] M. Raschka and S. Vahid, *Python Machine Learning (3rd Edition)*. Packt Publishing, 2019. ISBN 978-1-78995-575-0 [Page 2.]

[16] "Hopsworks - Batch and Real-time ML Platform," https://www.hopsworks.ai/, 2024. [Pages 2 and 5.]

[17] 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 2.]

[18] "What Is a Lakehouse?" https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html, Thu, 01/30/2020 - 09:00. [Page 3.]

[19] 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 3.]

[20] EDER, "Unstructured Data and the 80 Percent Rule," Aug. 2008. [Page 3.]

[21] "Dremel made simple with Parquet," https://blog.x.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet. [Page 3.]

[22] P. Rajaperumal, "Uber Engineering's Incremental Processing Framework on Hadoop," https://www.uber.com/blog/hoodie/, Mar. 2017. [Page 3.]

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USA: USENIX Association, 2010, p. 10. [Page 3.]

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150. [Page 3.]

[25] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," 2005. [Pages 3 and 15.]

[26] 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," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-82, Jul. 2011. [Page 3.]

[27] "Apache Spark™ - Unified Engine for large-scale data analytics," https://spark.apache.org/. [Page 3.]

[28] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine." [Page 4.]

[29] G. van Rossum, "Python tutorial," Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Tech. Rep. CS-R9526, May 1995. [Page 4.]

[30] "TIOBE Index," https://www.tiobe.com/tiobe-index/programminglanguages_definition/. [Page 4.]

[31] "Delta-io/delta-rs," Delta Lake, May 2024. [Pages 5, 6, 7, 23, 29, and 32.]

[32] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017. ISBN 978-1-931971-36-2 pp. 89–104. [Pages 5 and 9.]

[33] "The Apache Spark Open Source Project on Open Hub," https://openhub.net/p/apache-spark. [Page 6.]

[34] "Green Software Foundation," https://greensoftware.foundation/. [Page 7.]

[35] "What is Green Software?" https://greensoftware.foundation/articles/what-is-green-software, Oct. 2021. [Pages 7 and 8.]

[36] 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," 2021. [Page 8.]

[37] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink," 2022. [Page 8.]

[38] "| Sustainable Development," https://sdgs.un.org/. [Page 8.]

[39] M. Frampton, *Complete Guide to Open Source Big Data Stack*, Jan. 2018. ISBN 978-1-4842-2149-5 [Page 11.]

[40] S. Sakr, "Big Data Processing Stacks," *IT Professional*, vol. 19, no. 1, pp. 34–41, Jan. 2017. doi: 10.1109/MITP.2017.6 [Page 11.]

[41] "Block vs File vs Object Storage - Difference Between Data Storage Services - AWS," https://aws.amazon.com/compare/the-difference-between-block-file-object-storage/. [Page 12.]

[42] "How Object vs Block vs File Storage differ," https://cloud.google.com/discover/object-vs-block-vs-file-storage. [Page 12.]

[43] "Object vs. File vs. Block Storage: What's the Difference?" https://www.ibm.com/blog/object-vs-file-vs-block-storage/, Oct. 2021. [Page 12.]

[44] M. L. Despa, "Comparative study on software development methodologies." *Database Systems Journal*, vol. 5, no. 3, 2014. [Page 19.]

[45] "Welcome home : Vim online," https://www.vim.org/. [Page 22.]

[46] "Neoclide/coc.nvim," Neoclide, Sep. 2024. [Page 22.]

[47] H. Fann, "Fannheyward/coc-rust-analyzer," Sep. 2024. [Page 22.]

[48] "Docker Build," https://docs.docker.com/build/, 12:14:28 +0200 +0200. [Page 22.]

[49] "GitHub," https://github.com. [Page 22.]

[50] "TPC-H Homepage," https://www.tpc.org/tpch/. [Page 24.]

[51] T. P. P. C. (TPC), "TPC-H_v3.0.1.pdf," 1993/2022. [Page 24.]

[52] M. Poess and C. Floyd, "New TPC benchmarks for decision support and web commerce," *Sigmod Record*, vol. 29, no. 4, pp. 64–71, Dec. 2000. doi: 10.1145/369275.369291 [Page 24.]

[53] 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 24.]

[54] "TPC Current Specs," https://www.tpc.org/tpc_documents_current_versions/current_specific [Page 24.]

[55] "Arrow-rs/object_store/README.md at master · apache/arrow-rs," https://github.com/apache/arrow-rs/blob/master/object_store/README.md. [Page 29.]

[56] A. Binford, "Kimahriman/hdfs-native," Aug. 2024. [Page 31.]

[57] “Rustls/tokio-rustls: Async TLS for the Tokio runtime,” https://github.com/rustls/tokio-rustls. [Page 32.]

[58] G. Manfredi, “Silemo/hdfs-native,” Aug. 2024. [Page 32.]

[59] ——, “Silemo/hdfs-native-object-store,” Aug. 2024. [Page 32.]

[60] ——, “Silemo/delta-rs,” Sep. 2024. [Page 32.]

[61] “Delta-rs/python at main · Silemo/delta-rs,” https://github.com/Silemo/delta-rs/tree/main/python. [Page 32.]