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: September 18, 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

RDD	Resilient Distributed Dataset
RQ	Research Question
SDG	Sustainable Development Goal
SF	Scale Factor
SSD	Solid State Drive
SSH	Secure Shell protocol
TLS	Trasport Layer Security
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]. This paradigm was enabled by three key technologies: (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

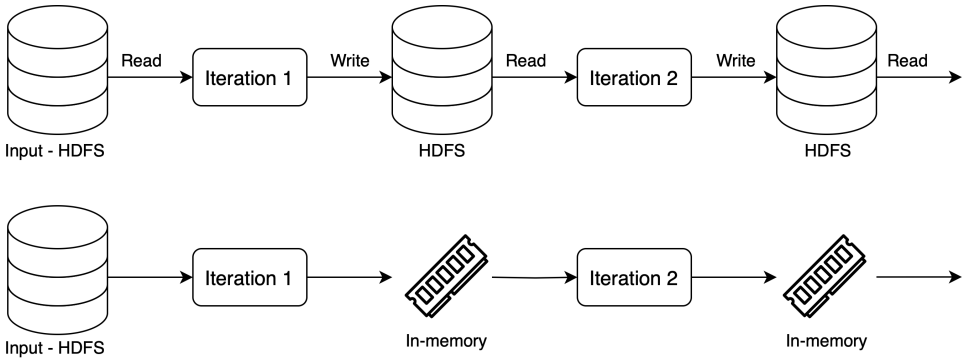

Figure 1.1: Difference between a Hadoop MapReduce execution and an Apache Spark execution. Every step in Hadoop MapReduce must be saved, while Apache Spark operates in memory.

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]. Python was first adopted by many 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 a great number of 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 current Spark system that operates on Apache Hudi, to a Rust system that using 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 performance between a Spark-based access to Apache Hudi compared a the delta-rs library-based access 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 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, providing more detail on the implementation, design decisions, expected performance and analysis of the results. This D is a comprehensive report of all the thesis project, 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.

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.

Figure 1.2: SDGs to which this thesis contributes to.

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 a outline of the thesis structure

Chapter 2

Background

This project works on 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 which is generally accepted, i.e. different approaches use different architectures [39, 40], thus this project defines a data stack, then focusing on the improving of its parts, namely the query engine. The data stack of the project is illustrated in Figure 2.1.

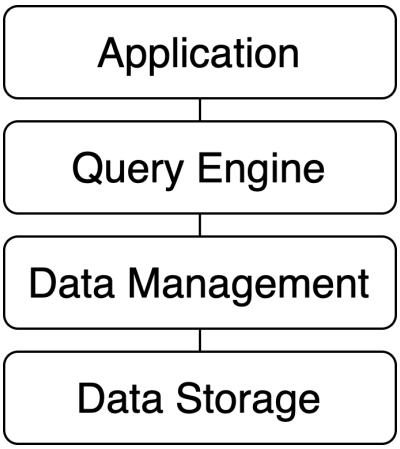

Figure 2.1: Data stack for how is considered in the project.

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 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, a highly scalable architectures and/or API support for multiple programming languages.

4. **Application** : 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 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, (3) Blocks. For each one, the technique is briefly described and then a comparative table is showed (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 an 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 storages 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 lacks 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, 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 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 an 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, 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 are 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 to 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 operating at a low level on physical systems, so the cost of the architecture is strictly bound on the storage and servers used, not allowing to scale the architecture on the demands needs.

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 resistent 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 store 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.

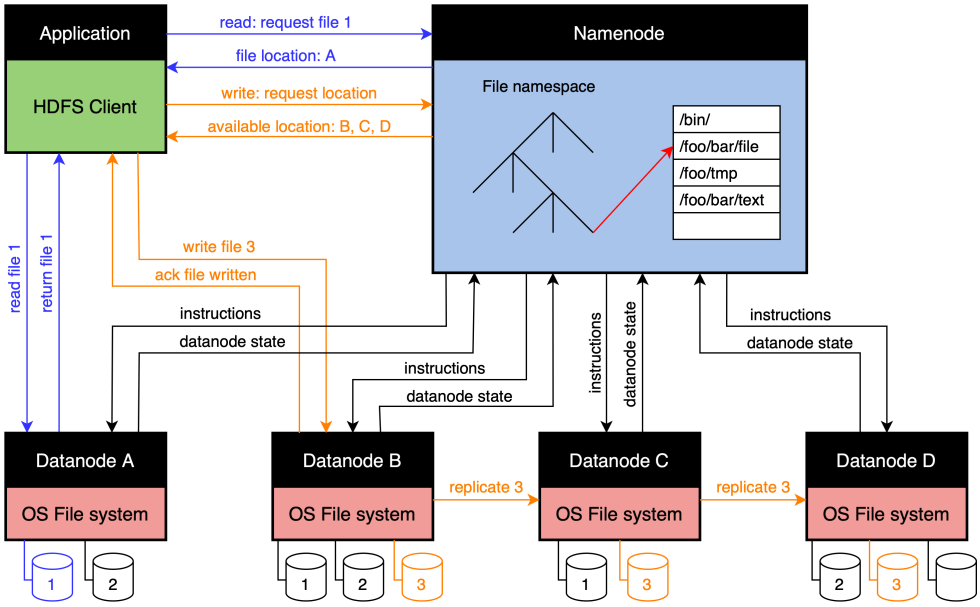

Figure 2.2: Hadoop Distributed File System (HDFS) architecture displaying in different colors basic operations: read (blue), write (orange) and Namenode-Datanodes management messages (black). Note: for representation simplicity files are not segmented into blocks.

Chapter 3

Method

3.1 System implementation – RQ1

The core of this system research thesis resides in the system implementation section which answers RQ1. The development will be carried out following an iterative development approach.

This section details 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.

3.1.1 Development process

The development process will follow an iterative development approach described in Figure 3.1.

The first activity will consist of identifying the system requirements and implementation issues collaboratively with the industrial supervisor, who is knowledgeable on Hopsworks' infrastructure (in particular HopsFS). Then a iterative loop will start, comprised of a in-depth study of the system and its dependencies, a software design and implementation phase, followed by tests conducted on both the implemented library and its system interactions. This loop will be iterated each time an individual test or an integration test will fail, and for each part of the system until the whole pipeline (from the Python interface, to writing on HopsFS) will be working as expected.

Each step of this process is related to one of the goals (G1–G4) associated to RQ1 in Section 1.4. 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.

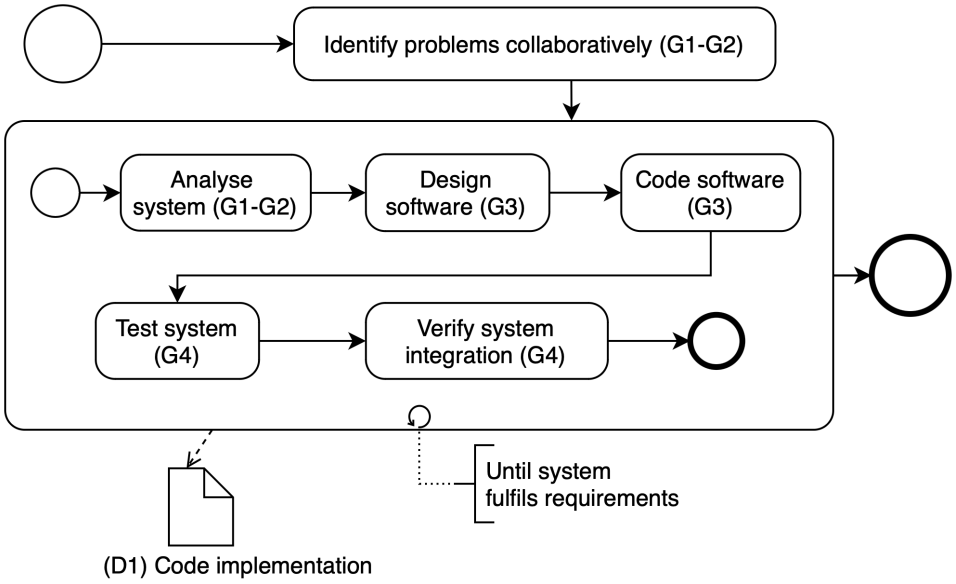

Figure 3.1: Business Process Model and Notation (BPMN) diagram of the System implementation process answering to RQ1. Each activity is associated to a specific Goal (G). The process produces a deliverable (D), in this case a code implementation.

3.1.2 Requirements

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

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

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

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

The **non-functional requirements** are:

1. **Consistent**: the solution should be consistent with 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 GBs) read or written on Delta Tables.

3.1.3 Development environment

The system implementation will be developed making use of a number of 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 [44] text editor is development tool of choice in combination with Conquer of Completion (CoC) [45] providing language-aware autocompletion and rust-analyzer [46] access for on-code compiler errors.
- **Libraries and dependencies**: for simpler development and tests reproducibility, the environment will be setup in a Docker container [47].
- **Code versioning and shared development**: GitHub [48] 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 Research Process

The research process will follow a sequential approach described in Figure 3.2.

The first activity will consist of defining the research objectives for this quantitative system evaluation, then based on these objectives the benchmarks will be defined. Using the code implementation (D1), the benchmarks will be carried out generating a first set of raw data. This data will be modified according to the specified metric (throughput measured in rows/second), enabling the last step of this process: the results visualization.

Each step of this process is related to one of the goals (G5-G8) associated to the RQ2 in Section 1.4. This process will produce two final deliverables which are the experiments results (D2), complete of data visualization and the results analysis (D3-partial).

3.2.2 Research Paradigm

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.

3.2.3 Dataset

The dataset that will be used to perform the read and write operations is the TCP-H [49]. This dataset is used at it provides a recognized standard for data storage systems [50]. Any part of the dataset can be generated using TPC-H data generation tool [51].

The TCP-H dataset 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) table which 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 to 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

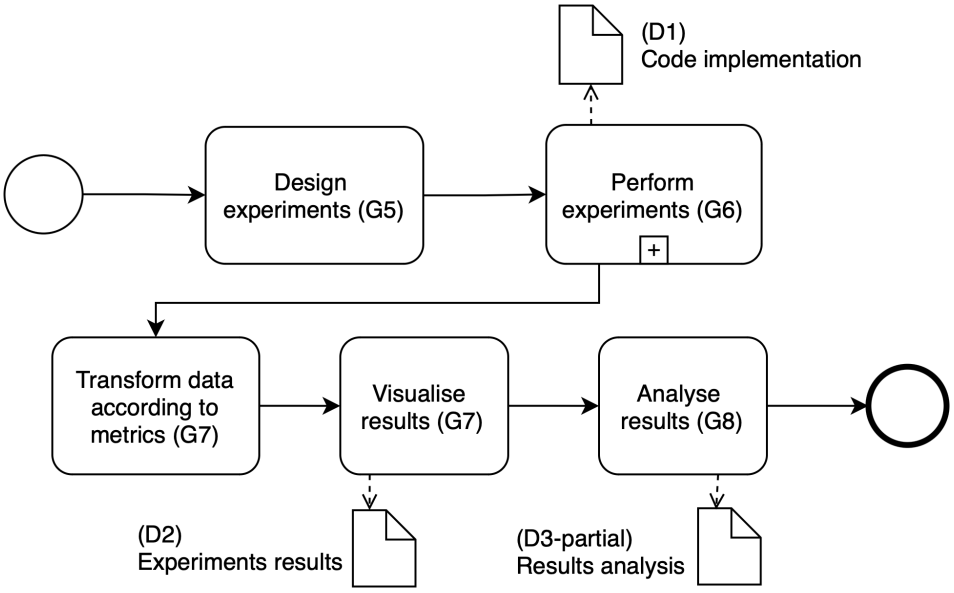

Figure 3.2: BPMN diagram of the System evaluation process answering to RQ2. Each activity is associated to a specific Goal (G). The process produces two deliverables (D), the experiments results (D2) and a results analysis (D3-partial).

means that 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 significate 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

3.2.4 Experimental Design

The experiments aim is 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 **??**. It is composed of a Rust pipeline with Python bindings, that enables 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 performances of three system 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 3.2.3, 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.

Experimental environment

The experimental environment consists of a physical machine in Hopsworks' offices, virtualized for 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).

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 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 on 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 calculating an average and a confidence interval for each experiment. This will benefit the accuracy of the results presented.

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 and 4.]

[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 5.]

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

[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] "Welcome home : Vim online," https://www.vim.org/. [Page 21.]

[45] "Neoclide/coc.nvim," Neoclide, Sep. 2024. [Page 21.]

[46] H. Fann, "Fannheyward/coc-rust-analyzer," Sep. 2024. [Page 21.]

[47] "Docker Build," https://docs.docker.com/build/, 12:14:28 +0200 +0200. [Page 21.]

[48] "GitHub," https://github.com. [Page 21.]

[49] "TPC-H Homepage," https://www.tpc.org/tpch/. [Page 22.]

[50] 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 22.]

[51] "TPC Current Specs," https://www.tpc.org/tpc_documents_current_versions/current_specific [Page 22.]