Università degli Studi di Milano

Facoltà di Scienze e Tecnologie

Corso di Laurea in Sicurezza dei sistemi e delle reti informatiche

Apache Hive and Apache Druid performance testing for MIND Foods HUB Data Lake

Supervisor: Prof. Paolo Ceravolo, <paolo.ceravolo@unimi.it>

Co-supervisor: Filippo Berto, <filippo.berto@unimi.it>

Graduand: Gabriele D'Arrigo, <gabriele.darrigo@studenti.unimi.it>

Number: 909953

Academic Year: 2021-2022

Abstract

Big Data, which defines significant in volume, various in structure and shape, and fast-paced data, is no longer an academic or technology domain. To stay competitive, organizations of all types and sizes now need to adapt their business to deal with this vast amount of information that traditional processing tools cannot treat.
Comparing, testing, and choosing between different databases suited to work with large amounts of data is not an easy operation since multiple characteristics should be considered: scalability, operation and maintenance costs, performances and, not least, the type of business analysis that needs to be supported.
An international, interdisciplinary project, MIND Foods HUB employs a Data Lake infrastructure to store and analyze data for an innovative plant-phenotyping process. Its main component, Apache Hive, an open-source Data Warehouse system that allows querying distributed datasets with SQL, does not satisfy two essential requirements of the project: maintainability and performance. As a possible, performant solution, I identified Apache Druid, a real-time analytics database designed to support sub-seconds aggregation queries and high concurrency APIs.
In this paper, I describe how I tested the performance of these two Big Data platforms following a rigorous, thorough and repeatable methodology.
The test results show how Apache Druid is a strong, better alternative to Apache Hive, outperforming it in every test scenario.

Keywords: Big Data, Data Lake, Performance Testing, Apache Hive, Apache Druid

Table of contents

List of Figures

List of Tables

1. Introduction

MIND Foods HUB [1] is an international, interdisciplinary project led by various public and private subjects (including Università Degli Studi di Milano, TIM and others) that operates in the context of the Milan Innovation District. Start-ups, organizations, and public institutions work on innovative and sustainable projects in this multifunctional space.
The goal of MIND Foods HUB is to "implement a computational infrastructure to model, engineer and distribute data about plants phenotyping" [2].
The project, split into different stages, plans to:

1. Identify crops with optimal nutritional profiles.
2. Creation of a Germplasm bank of the selected species.
3. Farm the selected species in protected environments (greenhouses and vertical farms) .
4. Apply high-throughput phenotyping applications on the plant's cultivations.
5. Create a computing infrastructure by employing smart devices and sensors, 5G communication networks, and Big Data platforms.
6. Analyze the collected data to study the selected species' nutritional properties and bioactive components and develop prediction models of nutritional and functional value.

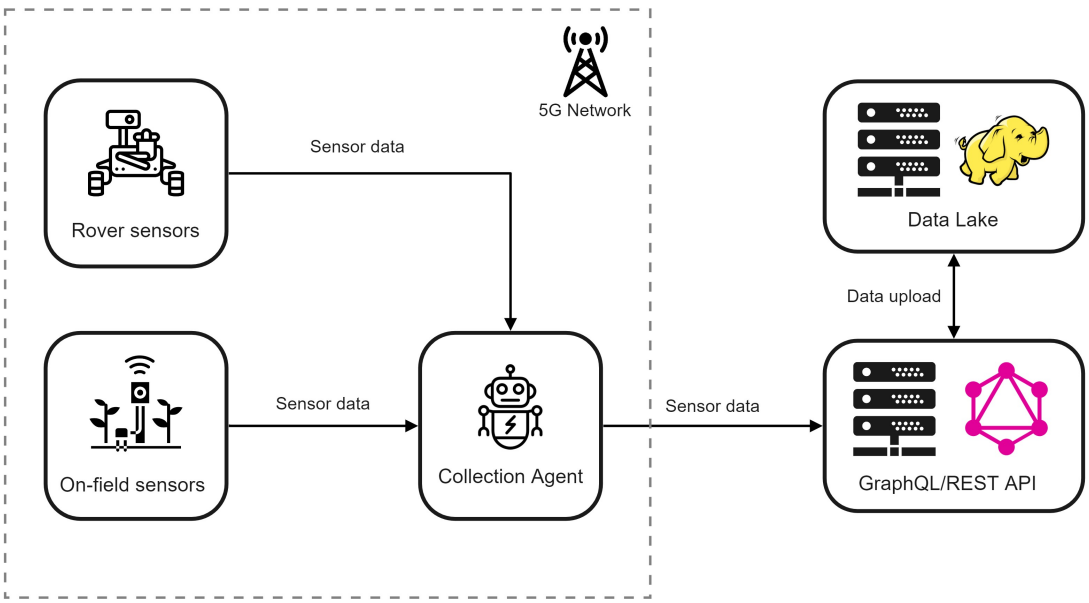

Figure 1: MIND Foods Hub computing infrastructure

The computing infrastructure is the core component of the MIND Foods HUB project and, as figure 1 illustrates, is comprised of different modules. The phenotyping process is managed by a rover deployed on the greenhouse.
The rover, equipped with various phenotyping sensors on a robotic arm, autonomously takes measurements on plant cultivations. Along with the rover, various on-field sensors register weather or environmental data, like air temperature and humidity, soil temperature, solar radiation, etc.
The greenhouse is supplied with a 5G communication network; rover and on-field sensors measurements are collected by two software agents and sent through the network to a hybrid GraphQL/REST API that persists the data in the MIND Foods HUB Data Lake. The Data Lake is an essential element of the computing infrastructure; it stores historical sensor data and supports various ad-hoc data analysis processes. Also, Data Lake serves a client application, FoodsApp, through the GraphQL/REST API.
For example, greenhouse operators could use FoodsApp to scan a QRcode associated with each cultivated species and visualize raw and processed data of the phenotyping process.

The implementation of the computing infrastructure involved various technological partners; TIM, the first Italian provider of fixed, mobile and cloud infrastructures, developed the 5G network, the on-field sensors, and the client application.
Multiple departments of Università Degli Studi di Milano are actively involved in the MIND Foods HUB project.
The Dipartimento di Scienza Agrarie e Ambientale (DISAA) designed and implemented the rover, its sensors, and the algorithms that analyze the nutritional properties of the chosen species.
The SEcure Service-oriented Architectures Research Lab (SESAR Lab) planned and implemented all software modules of the computing infrastructure: the collection agents, the GraphQL/REST API, the data model to structure sensor's data and the Data Lake platform.
The MIND Foods HUB Data Lake platform is based, among other components, on Apache Hadoop [3], a framework for parallel and distributed processing of large data sets and Apache Hive [4]. This software allows reading, writing, and managing large datasets in distributed storage using SQL.

1.1 Research goals

In the MIND Foods HUB project, the Data Lake platform is a core component of the computing infrastructure. It must support various stakeholders and use cases: the uploading of raw sensor data, the algorithmic analysis of the collected data, the serving of the client application through the hybrid GraphQL/REST API. So, like every architectural core component, the Data Lake platform must conform to non-functional requirements in terms of quality attributes [5]: scalability, maintainability and, not least, performance.

Scalability determines the ability of a system to perform gracefully as the client's requests increase over time [6]. *Maintainability* is defined as the ease with which a system can be modified to improve its performance, correct defects, or adapt to new requirements [7]. Finally, *performance* is defined as the amount of work a system must perform in a unit of time within given constraints, such as speed, accuracy, or memory usage [7].

When this research started, the SESAR Lab team was not pleased with at least two of these properties: Apache Hive reading performance has proven poor even on simple aggregations queries. Also, the maintainability of the Data Lake infrastructure was not satisfying since it relies on a complex multi-container Docker application, using custom images with various bash scripts and configuration files to define the Hadoop and Apache Hive services. So, SESAR Lab started investigating for faster and more maintainable alternatives to Apache Hive, including Apache Druid [8], a real-time database that supports modern analytics applications.
The requirements that the alternative platform must satisfy are:

1. Scalability: adding more resources to the platform to support an increase in the average workload should be effortless

2. Maintainability: the platform must be easy to configure and deploy on the MIND Foods HUB Hadoop cluster

3. Performance: the platform should provide sub-second aggregations queries

This research aims to study Apache Druid as a viable, more performant solution to Apache Hive; to accomplish this goal, I tested the performance of the two platforms with Apache JMeter [9], an open-source Java application designed to measure the performance of various systems and protocols. Performance testing, which is a type of non-functional testing, evaluates the functioning of a system by simulating a variety of standard and abnormal load conditions. However, in testing the performance for Big Data technologies, an essential factor should be considered: the variety and volume of the data set involved for testing [10].

That is why to test Apache Hive and Apache Druid, given the requirements of the MIND Foods HUB Data Lake platform, I applied a strict, thorough and reproducible methodology described in the following sections of this paper.
Section 2 defines Big Data and Data Lake systems and describes Hadoop, Apache Hive, and Apache Druid, focusing on their architecture and functionalities. Section 3 demonstrates the testing methodologies employed to test Apache Hive and Apache Druid with JMeter; Section 4 discusses the testing results, while section 4 concludes this paper.

2. Big Data technologies

We live in the age of data: mobile phones, smart devices, sensors, social networks, etc., all concur in a scenario where all information is digital and needs to be processed, stored and analyzed.
According to statistics, a person produces 1.7 MB of data per second [11]; globally, 44 zettabytes of data are generated daily. Furthermore, based on research [12] led by Seagate and IDC, the amount of global data will reach 175 zettabytes by the end of 2025.
This exponential explosion of data generation directly impacts organizations facing the challenge of managing a massive amount of data, coming from multiple sources at an ever-increasing rate; days are gone when a single database serves as the primary source of truth for an entire business. That is where new Big Data technologies can make the difference in how organizations make business decisions based on their data to stay competitive.

Big Data is a universal term [13] that refers to data that is "too big, too fast, or too hard" [14] to be processed or analyzed by traditional methods or technologies. To be so considered, Big Data is characterized by the "three Vs", a concept formulated by Doug Laney from Meta (now Gartner) in 2001 [15]: *Volume*, *Velocity*, *Variety*.

- *Volume* is a property that determines the size of the data, usually expressed in petabytes.
- *Velocity* refers to the speed at which data is generated and processed. Generally, data processing happens in batch, where a large volume of data within a specific time is processed, or in a stream mode, where data is processed in real-time as it is produced.
- *Variety* is a property that refers to different types of data produced by multiple sources; usually, types fall in one of these three categories: structured data (relational data with an enforced data model, often stored in tables), semi-structured data (like XML or JSON), and unstructured data (data not arranged according to a data model, for example, text, images, video, etc.)

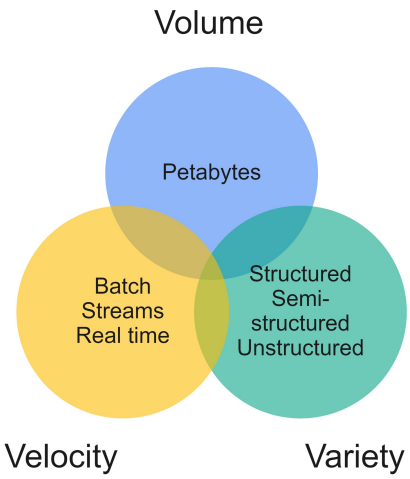

Figure 2: Big Data three Vs

Over time the three Vs concepts of Big Data have been complemented by two additional properties: *Veracity* [16] and *Value*.

- *Veracity* is a property that refers to the degree of reliability associated with certain data types. For example, some data is inherently uncertain or unpredictable (sentiments in social media, weather, economics) regardless of the data cleansing method applied; yet, these data types still contain valuable information.
- *Value* is a property that refers to the social or economic value generated from the data. Value is the desired outcome of Big Data processing since data has no inherent value without any process that analyzes the data to understand how to employ them.

As described in the next section, this model of distinguishing Big Data over traditional data models has revolutionized the way organizations deal with business analytical processes.

2.1 Data Warehouse and Data Lake systems

Big Data heavily changed the shape of traditional data management systems, like Data Warehouses, that now need to support parallel data processing, complex analytical workflow, resilient and flexible storage capable of efficiently persisting data from several sources. Also, with the rise of new Big Data technologies, further models to store and analyze data have been developed; one for all: Data Lake systems.

Data Warehouse concepts

There are two types of operations in data processing at the core: transactional and analytical.
Usually, organizations handle daily operations using multiple Online Transaction Processing (OLTP) systems, or rather relational SQL databases with ACID properties,

supporting the classical Create, Read, Update, Delete (CRUD) routines. Historically, organizations that want to apply analytical processes to their data employed a Data Warehouse system.
A *Data Warehouse* is a system used for reporting and data analysis that store historical, time-variant collection of data in a single repository [17], typically operated by business end-users to support management's decision.
A fundamental property of Data Warehouses is non-volatility; differently from OLTP systems, a Data Warehouse does not allow update or delete operations on the data. This property allows the historical comparison and analysis of data across extensive data ranges.
At its core, a Data Warehouse collects data from different OLTP data sources into a star schema. A star schema comprises a central *fact* table that stores measurable quantitative values for an event, like sales, revenues, costs, and multiple *dimension* tables that store the attributes of the fact data, like products, stores, and others.
The most used data processing paradigm to ingest data into a Data Warehouse and populate the fact and the dimension tables is Extract, Transform, Load (ETL).

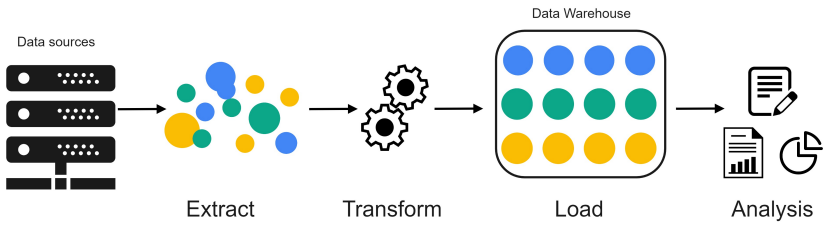

Figure 3: Extract, Transform, Load

In ETL, data are extracted from different sources, transformed to adhere to the star schema, and loaded into the Data Warehouse; as Data Warehouses are built for analytical purposes, usually data is loaded in batch mode at regular time intervals. Later, data is gathered with ad-hoc queries for reporting and mining activities as part of the organization's business intelligence process. This process to store data in a Data Warehouse, also known as *schema-on-write*, allows efficient and fast reads operations at the cost of designing and maintaining a schema (the star schema) that describes the shape of the data.

Data Lake concepts

Data Lake is a term coined in 2011 by James Dixon [18], founder and former CTO of Pentaho and successively extended by several studies [19]: "*A Data Lake uses a flat architecture to store data in their raw format. Each data entity in the lake is associated with a unique identifier and a set of extended metadata, and consumers can use purpose-built schemas to query relevant data, which will result in a smaller set of data that can be analyzed to help answer a consumer's question*".

In essence, a Data Lake is a repository that stores large quantities and varieties of data in their raw format, independently from their source or structure. Differently from a Data Warehouse, a Data Lake could store structured, semi-structured and unstructured data, regardless of their format, types, or structure. With this approach, the understanding of the data stored in a Data Lake is left to the consumer when data is later processed for business analysis.
Unlike traditional Data Warehouses, which pursue an ETL approach, Data Lake systems use an Extract, Load, Transform (ELT) data processing paradigm.

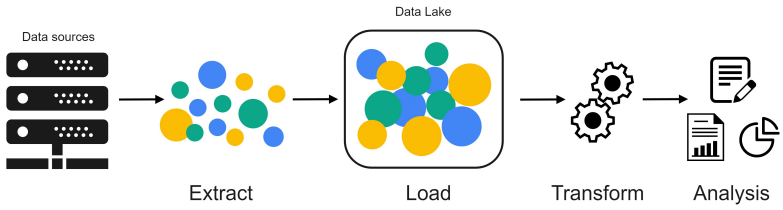

Figure 4: Extract, Load, Transform

In ELT, data is extracted from various sources in an untransformed or nearly transformed state and ingested into the Data Lake as soon as it is available. Data transformation is applied later during business analysis, giving the data a structure suited for that specific analysis on what is known as a *schema-on-read* process.
The advantages of a Data Lake over a Data Warehouse are that the upfront cost for data ingestion is extensively lower since ingestion does not require any prior schema design to store the data, nor any complex preprocessing or transformations. Also, data ingested into a Data Lake often happens in streaming mode, allowing organizations to extract business insights in real-time to make effective decisions.
Usually, many implementations of Data Lake systems are based on Apache Hadoop, which is described in detail in the following section.

2.2 Hadoop

Hadoop is an open-source framework that enables reliable, scalable, distributed computing.
Started in 2002 as an open-source project by Doug Cutting and Mike Cafarella, that was developing a search engine capable to index 1 billion pages, the first version of Hadoop implemented two powerful computing concepts introduced by Google in 2003 and 2004: Google File System (GFS) [20], a scalable distributed file system for large distributed data-intensive applications, and *MapReduce* [21], a programming model for processing and generating large data sets.
So, at its core, Hadoop is composed of two modules: storage and processing. The storage component is the Hadoop Distributed File System (HDFS), while the processing component is MapReduce.

HDFS

HDFS is a highly fault-tolerant, scalable, and distributed file system designed to run on commodity hardware; HDFS provides high throughput access to application data and is suitable for applications that need to compute large data sets.
The goals of HDFS are:

1. Support the processing of extremely large files, from multiple gigabytes to petabytes. HDFS supports small files but is not optimized for them.

2. Enable data streaming to applications that run on HDFS, allowing high throughput of data access rather than low data access latency.

3. Implement a simplified data coherence model. Once a file on HDFS is created, written and closed, it cannot be changed except for append or truncate operations. This assumption simplifies data coherence across the file system and allows high throughput data access.

4. Capability to execute the file system on low-cost hardware; an HDFS cluster can run on hundreds or thousands of server machines, each storing part of the file system's data, making hardware failure the norm rather than an exception.

5. Move the computation near the data to avoid the network transfer of large data sets, which is a notoriously slow and costly operation.

6. Portability of the file system across heterogeneous hardware and software platforms; since HDFS is implemented in Java, any machine that supports its runtime can operate HDFS.

HDFS is based on a master/slave architecture.
The master server is called *NameNode*, and it manages the file system namespace and regulates client access to files. In HDFS, a file is split into one or more blocks of equal size (default is 128 megabytes); these blocks are stored and replicated in a set of *DataNodes*.
NameNode's responsibility is to handle the mapping of blocks between DataNodes; also,

NameNode manages file system namespace operations, like opening, closing, and renaming files and directories, instructing DataNodes on their execution.

DataNodes manage the data and the storage attached to the nodes on which they run; they are responsible for serving read and write requests from the file system's clients. Also, DataNodes, when instructed from a NameNode, perform block creation, deletion, and replication.
Data replication across the DataNodes of the cluster guarantee reliability and fault-tolerance to HDFS. A client application can specify the desired number of replicas per file, which is three by the default policy; this arrangement means that a Hadoop cluster should have at least three DataNodes. To implement replication, DataNodes communicate with the NameNode sending a periodical *heartbeat* message. When the NameNode detects the absence of the heartbeat caused by a disk failure or a network partition, it marks the DataNode as no more available to HDFS and does not forward any new I/O requests to it. Eventually, the NameNode starts replicating the affected blocks across the remaining DataNodes.
A typical HDFS cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster since each DataNode may execute multiple application tasks simultaneously.

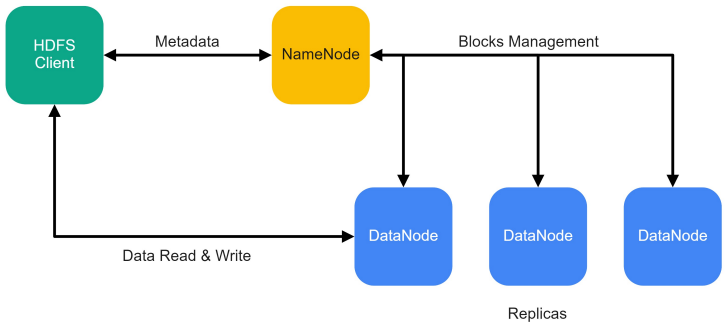

Figure 5: HDFS Architecture and data flow

Figure 5 shows the HDFS architecture and the flow of I/O operations.

1. An HDFS client wishing to read a file contact the NameNode to determine where the actual data is stored

2. The NameNode returns: the relevant block ids and the location where the blocks are held (i.e., on which DataNodes)

3. The client then contacts the DataNode to retrieve the data

As we can see, to guarantee high data throughput, data never flows through the NameNode but is directly served from the DataNodes.

MapReduce

MapReduce is a programming model for processing and generating large data sets. Developed by Google and implemented by Hadoop, its goal is to enable the parallelization of heavy computation tasks, data distribution and failure handling of large amounts of data across hundreds or thousands of machines.

The MapReduce model is based on two simple functional programming primitives: *Map* and *Reduce*.

```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

The Map function is written by the user and takes a key/value pair as input, and generates a set of intermediate key/value pairs in output . All intermediate values with the same intermediate key are grouped and passed to the Reduce function.
The Reduce function accepts an intermediate key and a set of values for that key. Then, it merges these values to form a smaller set of values; typically, zero or one output value is produced per Reduce invocation.
The intermediate values are supplied to the user's Reduce function via an iterator that allows handling lists of too large values to fit in memory.

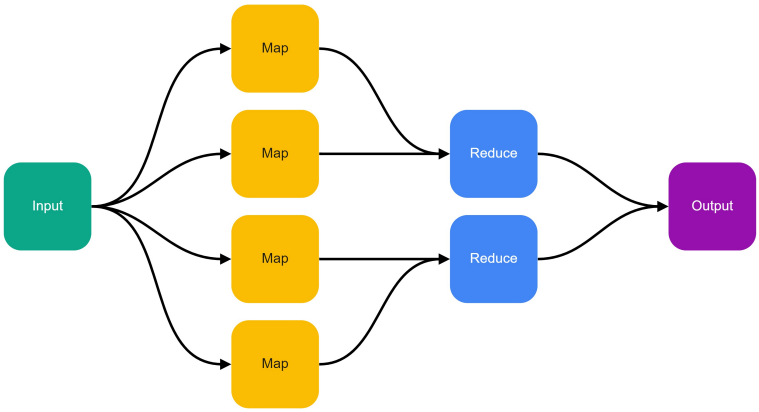

Figure 6: MapReduce flow

The power of the MapReduce model is that it enables the parallel execution of computation since the Map invocations are distributed across multiple machines of a cluster.
Figure 6 shows the typical execution flow of a MapReduce application. First, the input data is split into a set of *M* splits; each input split is processed in parallel by different Map functions, running on different machines. Next, the Map functions' intermediate key/value pairs are passed to the *R* available Reduce functions. The number *R* of Reducer is obtained by applying a partitioning function on the intermediate key (for example, `hash(key) mod R`).
The user specifies the number of partitions *R* and the partitioning function; the use of the partitioning function guarantee that the output of the various Map invocations is evenly distributed across the Reducers.

Hadoop implements MapReduce as a framework to process data stored on HDFS.
A MapReduce job is typically implemented in Java; it splits the input dataset into independent chunks processed in parallel by the Map tasks. Next, the framework sorts the outputs of the Maps, which are then input to the Reduce tasks. Usually, both the job's input and output are stored on HDFS.
The computing and storage nodes are the same, so the MapReduce framework and HDFS run on the same set of nodes; this allows the framework to take advantage of data locality, moving the computation where the data already resides. As a result, data locality helps minimize network congestion because it avoids the transfer of large data sets across the cluster and improves the overall computation throughput.

YARN

MapReduce jobs are scheduled by Hadoop's "Yet Another Resource Negotiator" (YARN) [22]. This Hadoop layer decouples the programming model from the resource management infrastructure and delegates many scheduling functions to per-application components.

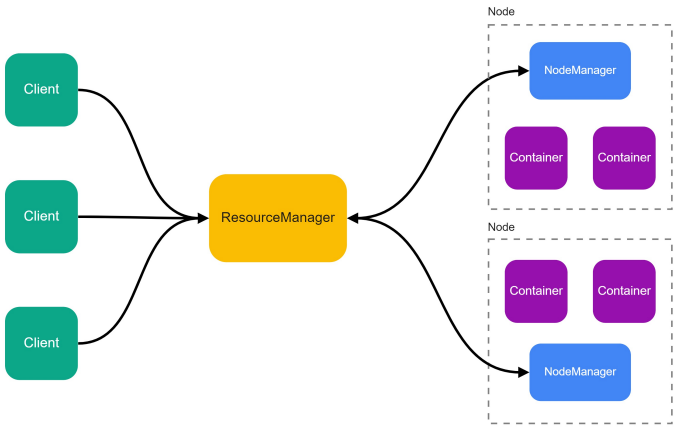

Figure 7: YARN Architecture

YARN, at its core, is composed of two processes: the *ResourceManager* and the *NodeManager*.
The ResourceManager is a process that runs on a dedicated node and is responsible for arbitrating cluster's resources among various contending jobs and scheduling job execution depending on resources availability.
YARN resources are called *containers*, a logical set of resources (for example, 2 GB of RAM, 1 CPU) bound to a specific node of the cluster and reserved for executing a MapReduce job.
A ResourceManager accepts jobs submissions from the clients and asks the NodeManager to allocate the required resources for their execution. NodeManager processes, one per each Hadoop node, are responsible for container lifecycle: they launch the container, monitor its execution and its resources usage (in terms of CPU, memory, disk, network), and reports the information to the ResourceManager. The ResourceManager, collecting information from all NodeManagers, can assemble the global status of the cluster and schedule jobs execution.

It is worth mentioning another component of the YARN architecture: the *TimelineServer* (previously known as the Application History Server), which runs on a dedicated node of the cluster.
The TimelineServer collects the historical states for each completed MapReduce job and provides various YARN related metrics accessible via REST APIs, including:

- The number of Map and Reduce tasks employed for a specific job
- The elapsed time for the job execution
- The final status of the job (undefined, succeeded, failed, or killed)
- The allocated memory and virtual cores per container

2.3 Apache Hive

Initially developed in 2009 by Facebook to access data on their Hadoop cluster, *Apache Hive* is an open-source Data Warehouse system designed for querying and analyzing large datasets stored in Hadoop. Hive provides standard SQL functionality with HiveQL, a declarative language that enables users to do ad-hoc querying, aggregation and data analysis.
Apache Hive offers many advantages over working directly on Hadoop; first, SQL is familiar to many developers and analysts. Instead of writing verbose, complex MapReduce jobs in Java, a user can submit SQL queries to Hive to compute data stored on HDFS. Then, Apache Hive is a mechanism to impose structure, using relational database abstractions, like tables, on various data formats. Finally, since HDFS can hold structured, semi-structured and unstructured data, Hive can read and store table data in various formats, including comma and tab-separated values (CSV/TSV), JSON, Apache Parquet, Apache ORC, and others.

Apache Hive is not designed for OLTP workloads or to support real-time analytics due to its batch-oriented nature. It serves better on traditional data warehousing tasks: data analysis and distributed processing of massive volumes of data. Since it is based on Hadoop, Apache Hive is engineered to be scalable: more machines can be dynamically added to the Hadoop cluster as the storage needs of an organization increase over time. Also, Hive exploits HDFS peculiarity in fault-tolerance since data is automatically replicated across the DataNodes and available even in the presence of failures. As described in the following "Ingestion" section, Apache Hive is loosely coupled with its input formats because it can ingestvarious types and shapes of data.

Design

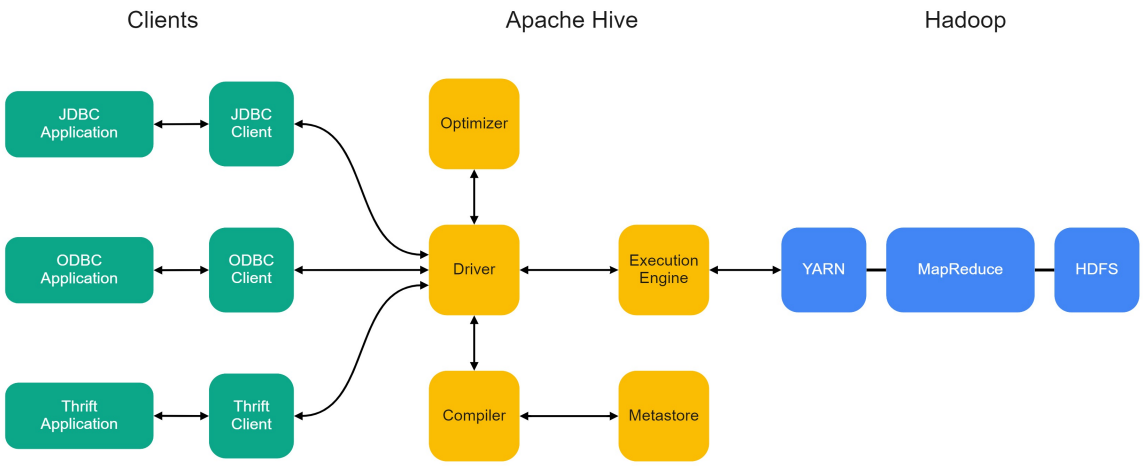

Figure 8: Apache Hive architecture

Figure 8 shows the architecture of Apache Hive and its main components:

- **Hive Client**: any kind of client that wants to interact with Hive by running HiveQL queries. Clients usually employ Apache Thrift, JDBC or ODBC drivers to connect and communicate with Hive.
- **Driver**: the component that receives those queries and handles the communication with the *Compiler*, the *Optimizer* and the *Execution engine.* It internally uses *HiveServer2*, a process that enables multi-client concurrency and client authentication. HiveServer2 uses a thread pool, allocating a worker thread for each TCP connection instantiated by a client.
- **Compiler**: the component that parses the queries, apply semantic analysis on each query block and expression and, by retrieving table and partitions metadata from the *Metastore*, eventually generates a query execution plan in the form of a Directed Acyclic Graph (DAG).
- **Optimizer**: the component that applies further transformations on the execution plan to optimize query execution.
- **Metastore**: the component that stores all schema's tables, partitions and buckets metadata, including the list of columns, columns type, data location on HDFS, serialization and deserialization strategies required to read and write data from and to tables. Metadata is usually stored on an external relational database, like MySQL or PostgreSQL and not directly on HDFS.
- **Execution engine**: the component which instructs Hadoop to perform the execution plan.

- **SerDe**: "Serializer and Deserializer" (SerDe) is the primary interface used by Hive to perform I/O operations on HDFS. One of the core principles of Hive is that it does not own HDFS file format; a user could store data in HDFS in various file formats, and Hive uses SerDe to read and write data back and forth to HDFS supporting that specific file format.

Query execution

When the Driver receives a query from a client, it creates a session handle for that client and forwards the query to the Compiler:

1. The Compiler retrieves the necessary metadata from the Metastore; metadata is used to type-check the query expressions and eventually prune partitions data based on query predicates.
2. The Compiler produces a query execution plan in the form of a DAG, where each vertex is a MapReduce job to be executed.
3. The Optimizer applies columns pruning and other optimizations to the query execution plan.
4. The Execution engine submits each MapReduce job of the query execution plan to Hadoop for parallel processing.
5. The SerDe deserializer associated with the table reads the HDFS files' rows and returns data to the client in each task.

Data Model and Storage

In order of granularity, data in Apache Hive is organized into:

- **Databases**: namespaces that group tables, views, partitions, and columns.
- **Tables**: similarly to relational databases tables, they are homogenous data units within the same schema. All table data is stored in `/user/hive/warehouse/databasename.db/tablename/` directory of HDFS; Apache Hive also supports *external* tables, or instead tables created on pre-existing files or directories in HDFS.
- **Partitions**: each table can have one or more partition keys that determine how the data is stored on HDFS; all data with the same partition key are held together into the same partition. For example, in a table T, with a `date` partition, all data for a particular date is stored in the `T/data=<date>` directory on HDFS. Partitions allow users to retrieve data that satisfies specific predicates efficiently.
  For example, a query on T that satisfies the predicate `T.date='2022-02-02'` would only look at files stored in the `T/data=2022-02-02/` directory on HDFS.

- **Buckets**: data in each partition may be further divided into buckets based on the hash of a column in the table; each bucket is stored as a file in the partition directory. Bucketing allows the system to efficiently evaluate queries that depend on a sample of data.

SQL capabilities

HiveQL supports all essential Data Definition Language (DDL), Data Query Language (DQL) and Data Manipulation Language (DML) operations to work with databases, tables and partitions. To name a few: the ability to select only specific columns using a `SELECT` clause, rows filtering using a `WHERE` clause, equi-join between tables, data aggregations with multiple `GROUP BY` columns, store the results of a query into another table, manage databases, tables and partitions with the `CREATE`, `DROP` and `ALTER` statements.

Ingestion

There are multiple ways to ingest data into Apache Hive using SQL statements, but since the Compiler translates these into MapReduce jobs, all data ingestion occurs in a batch mode.
So, for example, to load data into Apache Hive, a user can:

1. Create an external table that points to a specified location within HDFS and copy the data into any other Hive table. As previously mentioned, Apache Hive can store table data in various file formats; when a user creates a table, could specify the format of the files stored on HDFS.
   For example, the following statements create an external table `T` that points to the `/user/data` directory in HDFS, which stores JSON files, and load the data into the `V` table.

   ```
   CREATE EXTERNAL TABLE T (
       key string,
       value double,
   )
   STORED AS JSONFILE
   LOCATION 'hdfs://namenode:9000/user/data';

   FROM T
   INSERT OVERWRITE TABLE V
   SELECT *;
   ```

2. Use a `LOAD DATA` statement to load data in various formats from HDFS into a Hive table. The following example shows how `file.json` is loaded into table `V`.

   ```
   LOAD DATA INPATH 'hdfs://namenode:9000/user/data/file.json'
   INTO TABLE V;
   ```

3. Append data into an existing Hive table using the `INSERT` statement. For example, the following statements insert a row into the table `V`:

```
INSERTO INTO V VALUES('Pi', 3.1415);
```

2.4 Apache Druid

Apache Druid is an open-source distributed data store that supports various modern applications, like real-time analytics on large datasets and fast data aggregations for highly concurrent APIs.
Apache Druid offers many advantages over traditional Data Warehouse systems:

- Real-time data ingestion in streaming mode; also batch mode ingestion is supported.
- Low latency real-time queries
- Default time-based partitioning on data, which enables performant time-based queries
- Native support for semi-structured and nested data in different formats, including comma and tab-separated values (CSV/TSV), JSON, Apache Parquet, Apache ORC, Protobuf, and others.

Apache Druid architecture combines ideas from different storage systems (Data Warehouse systems, Timeseries databases, Search engines). At its core, Druid uses a columnar storage format only to load the exact columns needed for a particular query instead of the entire row. Also, ingested data is automatically time partitioned into what are known as *segments*. As we will see in the "Data model and storage section", segments are stored in Druid's distributed storage; time-based queries only access segments that match the query's time range, allowing Druid to provide data to the clients efficiently. Apache Druid is engineered to be horizontally scalable, fault-tolerant, performant, and easy to operate and maintain.

Design

Apache Druid is a multi-process distributed data store designed to be deployed in a cluster. Each system component, ingestion, querying, and coordination, can be configured and scaled independently based on the system's needs; for example, a configuration for a specific workload can dedicate more resources to the query component while giving fewer to the ingestion one. This type of architecture, other than allowing horizontal scalability, enables operational fault-tolerance. For instance, if the ingestion process fails, the query process is not affected, allowing Druid to serve query requests regardless of the failure.
Apache Druid processes are the following:

- **Coordinator**: it is responsible for managing data availability and distribution on the cluster. The *Coordinator* instructs the *Historical* process to load new segments, drop outdated segments, ensure that segments are correctly distributed and replicated across the Historical processes to keep the latter evenly loaded.
- **Router**: it is an optional process that acts as an API gateway. The *Router* receives queries from external clients and routes them to the *Brokers*. Additionally, the

Router runs the Druid Console, a management Web UI to run SQL or native Druid queries and manage the Druid cluster.

- **Broker**: it is responsible for forwarding queries to the Historical processes, depending on the location of the segments.
  A Broker understands what segments exist on what Historical processes depending on the time partitioning, and routes queries to the designed nodes. Also, the Broker merges the results of a query from all individual Historical involved in the load operation into the final result set.

- **Historical**: it stores historical queryable data. Since Druid data is time partitioned into segments, each Historical process, usually multiple per cluster, is responsible for loading and serving data about a specific segment when instructed by the Coordinator or the Broker. Data is generally stored in a distributed file system and loaded into Historical memory every time a read/write operation is requested.

- **Overlord**: it is responsible for accepting ingestion tasks, assigning them to the *MiddleManager* processes and coordinating data publishing on the distributed storage.

- **MiddleManager**: it handles the ingestion of new data into the cluster; MiddleManagers are responsible for reading data from external sources and publishing new Druid segments.

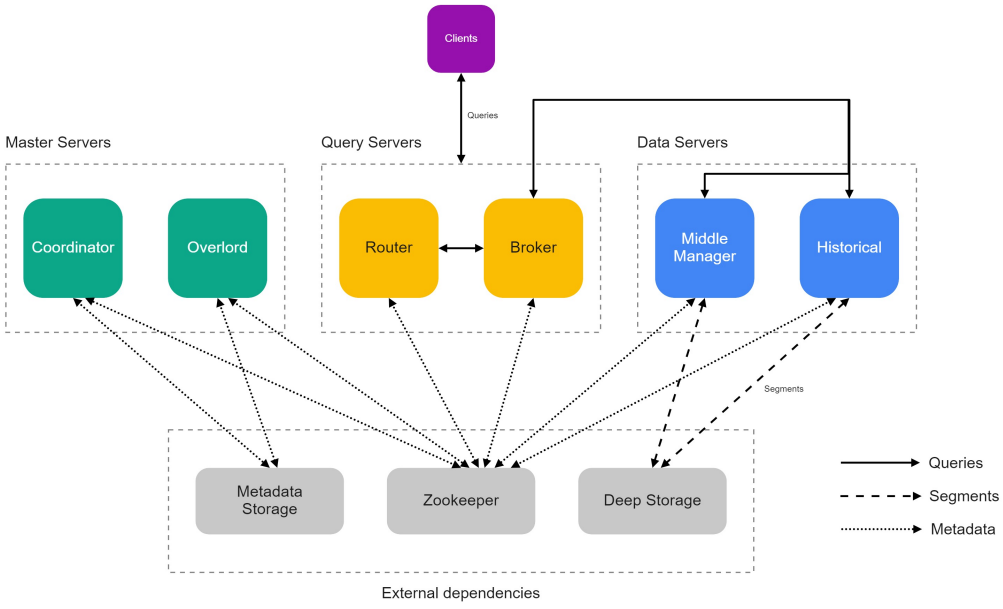

Figure 9: Apache Druid architecture

Druid processes are typically organized into logical units, following a *Master*, *Query* and *Data* server topology; figure 9 shows the architecture of a representative Apache Druid cluster.
Master servers manage data ingestion and availability; they are composed of the

Coordinator and the Overlord processes. Query servers, formed by the Broker and the Router processes, expose the endpoints that users and client applications interact with; they route queries to the Data servers. Finally, data servers execute ingestion jobs and store the data into the distributed storage. Data servers are composed of the Historical and the MiddleManager processes.

Apache Druid also relies on three external dependencies: *Deep storage*, covered in the related section, *Metadata storage*, and *ZooKeeper*. The Metadata storage stores various system metadata, like the number of segments available on the cluster, their usage, and internal task status. It is typically employed as an external relational database on clustered deployments, like MySQL or PostgreSQL.

ZooKeeper is an open-source coordination service [23] used to maintain centralized configuration, naming, and synchronization for distributed applications. For example, Apache Druid uses ZooKeeper for internal service discovery, coordination between the processes, and leader election.

Query Execution

Apache Druid query execution follows a *Scatter/Gather* [24] approach to retrieve data from the Historical processes;
The execution flow is the following:

1. A client submits a query to the Broker (or the Router) via HTTP or using the Apache Avatica JDBC drivers.
2. The Broker identifies which segments are pertinent to the query based on the time interval specified and eventually prune the identified segments according to the `WHERE` clause.
3. The Broker forwards the query to the relevant Historical processes.
4. The Historicals serve each segment in parallel and generate a partial result.
5. The Broker receives partial results from each Historical, merges them into the final result set and returns the data to the caller.

Data model and Storage

Apache Druid data is organized into *datasources*, similar to relational databases tables: homogenous data units within the same schema. A datasource must always include a primary timestamp that Druid stores in the `__time` column of the datasource. The primary timestamp is used to partition, sort, and manage the data across the Historical processes.
Datasources columns are of two types: *dimensions* and *metrics*. Dimensions are columns that Druid stores in their original format. They represent the relevant and descriptive attributes of the data; dimensions can be filtered, grouped, and aggregated at query time. Metrics instead are columns that Druid stores in an aggregated form; during ingestion, a user can apply an aggregation function like count, sum, min/max, to each computed row. Datasources are time partitioned: each time interval, for example, a month, is called a

chunk; a chunk is additionally partitioned into one or more segments. Formally, a *segment* is defined as a collection of rows of data, typically 5–10 million, that span an interval of time [25]. A segment comprises different files that store various data structures to arrange column data so that Apache Druid can extract only those needed for a query.
Depending on the column data type, two compression algorithms, *LZ4* [26] and *Roaring* [27], are used to reduce the cost of storing a column in memory and on disk.
Apache Druid uses Deep storage, an external, shared file system accessible by the cluster to store segments data. Typically, in a clustered deployment, Deep storage is a distributed file system like HDFS or Amazon S3; this design choice enables Druid to be fault-tolerant by design. Even if the cluster is reprovisioned after a network or a hardware failure, as long as the data is persisted in the Deep storage, Druid can recover data and republish segments to new Historicals. Also, as described in the prior sections, a distributed storage like HDFS, among many other characteristics, guarantee data replication, improving the reliability of the entire system.
It is important to note that Druid never accesses the Deep storage at query time: Historical processes load segments from the Deep storage when instructed by the Coordinator and serve them from their memory as well from the local disk when the Broker requests a query.

SQL capabilities

Apache Druid supports two query languages: Druid SQL, a declarative language similar to SQL, and native queries, JSON objects that describe how to filter, group, or aggregate data. Apache Druid automatically translates Druid SQL into a native query every time a query hits a Broker.
Anyhow, Druid SQL capabilities are limited: it only supports `SELECT` queries. This design choice means that DDL statements like `CREATE`, `ALTER` or `DROP`, or typical DML statements like `INSERT` and `UPDATE` are not supported. Also, Druid SQL only supports equality `JOIN` between native datasources.

Ingestion

Apache Druid supports both streaming and batch ingestion; for each ingestion method, the Middlemanager loads the raw data from the source, stores the segments to Deep storage, and publishes each segment by writing a record to the Metadata store.
The Coordinator instead polls the Metadata store periodically, by default every minute; when it finds a new segment, the Coordinator chooses a Historical process and instructs it to load the segment.

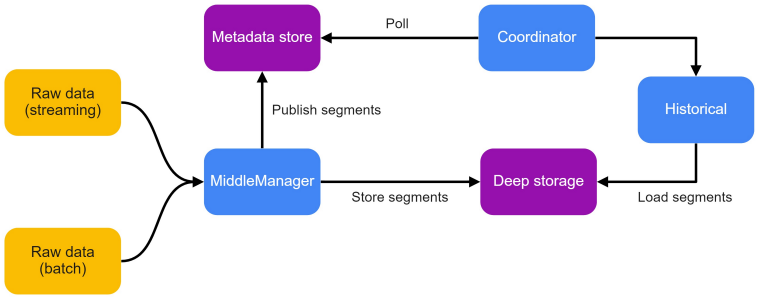

Figure 10: Apache Druid ingestion

With stream ingestion, Apache Druid reads raw data from a data stream and makes segments queryable as soon as they are available. Apache Druid supports two methods of stream ingestion: read from an *Apache Kafka* [28] stream or an *Amazon Kinesis* [29] stream.

With batch ingestion Apache Druid reads raw data from files in various format with a one-time job; segments are queryable only when the ingestion job is completed. Apache Druid supports three methods of batch ingestion: native batch, which reads raw data in parallel using multiple worker threads; native batch simple, which reads raw data with a single thread; and Hadoop based, which reads raw data from HDFS and uses MapReduce jobs to compute the ingestion.

3. Apache Hive and Apache Druid performance testing

In software, performance testing is a type of non-functional testing that measures a system's behaviour under satisfactory and unsatisfactory conditions [30].
The performance of a system, especially for those that use the network to transfer data, is assessed by collecting various time-related metrics, like response time, throughput, and concurrency.
We define *response time* as the measure of time a system takes to respond to a given business request or command. *Throughput* refers to the amount of work, that is, the number of requests, that an application can process in a unit of time, while *concurrency* is defined as the property of a system to respond to several requests that potentially interact with each other, simultaneously.
Usually, various standard statistical measures are calculated for response time, like the median, the average, and the standard deviation. Another valuable metric for response time is Cohen's *d* [31] of the samplers, a commonly recognized way to measure the effect size.
Cohen's *d* is defined as the difference between two means divided by a standard deviation for the data:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s}$$

The magnitude of *d*, namely the difference between the means, is described in the table [32] below:

d	Effect size
0	Similar
< 0.01	Negligible
[0.01-0.19]	Very small
[0.20-0.49]	Small
[0.50-0.79]	Medium
[0.80-1.19]	Large
[1.20-1.99]	Very large
>= 2.0	Huge

Table 1: Magnitude of d

One traditional way to accomplish performance testing is first to collect performance requirements, provided in a concrete, verifiable manner to make the performance testing meaningful.
For example, the requirements of this research are:

1. Scalability: adding more resources to the platform should be effortless.
2. Maintainability: the platform under test must be easy to configure and should integrate well on the actual MIND Foods HUB Hadoop cluster.
3. Performance: the platform under test should provide sub-second aggregations queries.

Then, with the requirements available, it is necessary to develop a benchmark: a workload representative of how the system is used in the field and then run the system on those benchmarks. Designing and implementing a valid performance benchmark is a complex process, often compromising between contrasting goals. To be considered adequate, a benchmark should implement at least two of these five properties, defined by Karl Huppler in his research [33]:

1. *Relevant*: the benchmark should be meaningful, and the collected metrics must be understandable by the performance test's target audience. Also, the benchmark should realistically test the system's features, similar to the typical use of a stakeholder.
2. *Repeatable*: the benchmark should produce the same results if it runs a second time within the same operational context.
3. *Fair*: the benchmark should equally test all systems under consideration.
4. *Verifiable*: the results produced by the benchmark should represent the system's actual performance under test.
5. *Economical*: a relevant, realistic benchmark could be expensive, so the organization that sponsor its development must afford its cost.

To evaluate the maintainability and the performance of Apache Hive and Apache Druid and to achieve a relevant, repeatable and verifiable test, I followed five significant steps, extensively described in the following sections:

1. Provision of each solution
2. Generate synthetic data
3. Ingest data with specific schema optimization
4. Prepare the test queries

5. Run the performance testing using Apache JMeter

The test of the scalability of each platform instead is out of the scope of this research since the actual use of MIND Foods HUB Data Lake does not need robust horizontal scalability given the youth of the project.
Also, the available resources, both in time and commodity (hardware), were not enough to scale the two platforms to support a heavy workload.

3.1 Provision of each solution

The first step was to provision Apache Hive and Apache Druid and seamlessly integrate them with a replica of the Hadoop cluster used in production by MIND Foods Hub. For the deployment of each platform, I used a Vmware virtual machine hosted on the SESAR Lab infrastructure. The virtual machine has the following specifications:

- 12 vCPU
- 48 GB of memory
- 512 GB storage
- Debian GNU/Linux 11 (bullseye) operating system.

To provision Hadoop, I used *Docker Compose* [34], a tool for defining and running multi-container Docker applications with YAML. The Hadoop services specified in the docker-compose file, and their configuration, are the same as the production cluster:

- A single NameNode
- Three DataNodes
- A ResourceManager
- A NodeManager
- A TimelineServer
- A Zookeeper instance

Using Docker Compose to deploy Hadoop, Hive and Druid on the same machine is suboptimal because even if each service runs in an isolated environment (container), they all concur to the same resource usage. Furthermore, horizontal scalability is fictional with this configuration since adding more nodes to the cluster does not augment its capacity to serve an increasing number of requests. However, during the various test execution, the virtual machine never suffered from resource depletion, so horizontal scalability is just a theoretical problem bound to the hardware resources available for the research.

Apache Hive provisioning

Apache Hive was configured to run on the same Hadoop dockerized cluster.
The setup used for the performance testing consists of the following Docker Compose services:

- A single HiveServer
- A Metastore server
- A MySQL 8.2 instance to persists Metastore's metadata

Apache Hive provisioning was not smooth as presumed. The Apache Software Foundation does not provide any official Docker image for Hive or related documentation to work with Docker. So, each service was defined by using custom Docker images and various bash scripts maintained by the SESAR Lab team and published on their internal registry. This setup led to low maintainability of the platform: each change to the Hive configuration, for example, new Hive versions, or security patches, requires the revision, the rebuild, and the publishing of each image, as well as the integration tests with the existing Hadoop infrastructure. Eventually, it needs to be said that the official Hive documentation is quite fragmented and makes its configuration harder than expected.

After the provisioning, I did a considerable number of preliminary tests, during which I encountered various timeouts and `java.lang.OutOfMemoryError: Java heap space` errors caused by the default Hive's heap size of 1024 MB.
After some research and multiple attempts, the maximum heap size of the Hive Server was increased to 4096 MB; also, Hive CLI JVM heap size was raised to 8192 MB.
Moreover, I configured Apache Hive to enable dynamic partitioning of data and disabled the query result cache not to alter the test samples.

Apache Druid provisioning

Apache Druid deployments could be configured in two ways: single-machine deployment, with each process running on the same machine, or clustered deployment, where Druid processes are distributed with the architecture illustrated in section 2.4. Clustered deployment could be accomplished with Docker since the Apache Software Foundation provides Druid's official Docker images and the related documentation to configure a Druid cluster via Docker Compose.

The Apache Druid setup used for the performance testing consists of the following Docker Compose services:

- A Coordinator
- A Broker
- A Router
- A Historical
- A MiddleManager

- A PostgreSQL instance to persists Druid's metadata

As previously mentioned, Apache Druid can ingest batch data natively or by reading data from a stream.
However, one of the requirements of MIND Foods HUB Data Lake is to take advantage of the already established Hadoop infrastructure. So, to adhere to this requirement, the Druid cluster was configured to work with HDFS using `druid-hdfs-storage` extension.
Integrating Druid with HDFS allows a user to ingest data formerly present on HDFS and store segments in it, exploiting HDFS attributes in terms of distributed storage, scalability, and fault tolerance.
Nevertheless, the choice to use HDFS for batch data ingestion has a few disadvantages: the resulting overall architecture is more complex since Apache Druid depends on an external Hadoop cluster and, more important, the waive to streaming ingestion mode, which enables users to perform real-time analytics on data.

The provisioning of Apache Druid was simple, and the cluster did not encounter any errors or faults during the preliminary performance tests. Furthermore, the overall maintainability of the platform was satisfactory since the official Druid's Docker images worked well out of the box without any custom configuration. Also, Druid documentation covers every use case of the platform in-depth, making its deployment pretty straightforward.

3.2 Data generation

One of the steps of testing database systems that process large volumes of data is to ingest them with a relevant, large dataset and benchmark their performances with a typical workload.
At the time of this writing MIND Foods HUB project was starting to ingest data into the Apache Hive cluster. The relatively small dataset that could be dumped was not suited to test the performances of Big Data platforms like Hive and Druid. Several databases have been studied to verify their capabilities using Star Schema Benchmark (SSB) [35], a benchmark designed to measure the performance of Data Warehouse systems employing a star schema. Instead of using a modified version of SSB, adapted to fit a denormalized schema, I opted to keep the same data structure of the MIND Foods Hub Data Lake and randomly generate an extensive and representative dataset to ingest both Hive and Druid.
The generated dataset should respect the *volume* and *variety* properties of Big Data illustrated in section 2.

MIND Foods HUB data are stored in a single table, named `dl_measurements`, that uses a denormalized data model to avoid expensive join operations.
`dl_measurements` table schema is the following:

```
CREATE TABLE dl_measurements
(
    id                          string,
    double_value                double,
    str_value                   string,
```

```
    unit_of_measure           string,
    sensor_id                 string,
    sensor_type               string,
    sensor_desc_name          string,
    location_id               string,
    location_name             string,
    location_description      string,
    location_botanic_name     string,
    location_cultivation_name string,
    location_latitude         double,
    location_longitude        double,
    location_altitude         double,
    measure_timestamp         timestamp,
    start_timestamp           timestamp,
    end_timestamp             timestamp,
    insertion_agent           string,
    insertion_timestamp       timestamp,
    CONSTRAINT dl_measurements_pk
        PRIMARY KEY (id) DISABLE NOVALIDATE
)
```

Each row of the table represents a *measurement* for a given *location*, or rather cultivation, provided by a specific *sensor*; each measurement reports its *unit of measure* and the corresponding *timestamps* that states when it was taken.
Instead, the insertion *agent* is the software agent that collected the measurement and sent it through the 5G communication network.

MIND Foods HUB *sensors* are of three types:

- *Measurement* sensors register discrete, floating-point values like temperature, humidity, wind speed, etc.
  This data is stored in the `double_value` column, while the measurement time is stored in the `measure_timestamp` column.

- *Phase* sensors register a range of floating-point values in a given period.
  This data is stored in the `str_value` column. The time start and end of the measurement are stored respectively in `start_timestamp` and `end_timestamp` columns.

- *Tag* sensors register string-based values.
  This data is stored in the `double_value` column, while the measurement time is stored in the `measure_timestamp` column.

To randomly generate data for `dl_measurements`, the relation between a sensor type and its measurement should guarantee these logical constraints:

- `double_value` is only populated for float-based measurements while `str_value` is `NULL`.
  `measure_timestamp` is calculated, while `start_timestamp` and `end_timestamp` are `NULL`
- For phase-based measurement, `str_value` is populated, while `double_value` is `NULL`.
  Both `start_timestamp` and `end_timestamp` times are calculated, while `measure_timestamp` is `NULL`
- For tag based measurement, `str_value` is populated, while `double_value` is `NULL`.
  `measure_timestamp` is calculated, while `start_timestamp` and `end_timestamp` are `NULL`

So, to respect these rules, I wrote "MFH measurements generator" [36], a Node.js application to generate random synthetic data for Apache Hive and Apache Druid performance testing. The application source code is hosted on SESAR Lab Github's organization and is released under the Apache 2.0 License.
The application's working is simple: it iterates from 1 to *N*, where *N* is the desired number of rows. Each iteration produces a randomly generated measurement that respects the logical constraints reported above; the application's output is a CSV file containing the produced data.
To provide a certain degree of semantics to the random measurement, I defined some static datasets used as the basis for each generation. Sensors (`sensor_id`, `sensor_type`, `sensor_desc_name`, `unit_of_measure`) and collection agents (`insertion_agent`) were dumped from the MIND Foods HUB Hive production cluster and randomly picked for each generated row.
For the values of `location_name`, `location_botanic_name`, and `location_cultivation_name`, I used Mockaroo [37]. This online service allows generating synthetic data, including commons and scientific plant names, with which I produced a set of 100 locations, randomly picked for each generated row. Ultimately, to simulate a dataset of a Data Lake in operation, all rows were generated computing the `insertion_timestamp` in a temporal range of two years.
Using the "MFH measurements generator", I produced a CSV dataset containing 50 million rows (approximately 15 GB of size) of random, synthetic data to test Apache Hive and Apache Druid.

3.3 Data ingestion

Before ingesting dataset, defining the tables for each platform was necessary.
Unlike the original Hive `dl_measurements` table that holds MIND Foods HUB data, I decided to apply some optimizations to test each platform at the top of their performance capability, using Hive partitions and Druid segments.
So, each *measurement* was stored depending on its `insertion_timestamp`, allowing each system to retrieve data based on temporal criteria efficiently.

Apache Hive table optimization

On Apache Hive, `dl_measurements` has the following schema:

```
CREATE TABLE dl_measurements
(
    id                          string,
    sensor_id                   string,
    sensor_type                 string,
    sensor_desc_name            string,
    double_value                double,
    str_value                   string,
    unit_of_measure             string,
    location_id                 string,
    location_name               string,
    location_description        string,
    location_botanic_name       string,
    location_cultivation_name string,
    location_latitude           double,
    location_longitude          double,
    location_altitude           double,
    insertion_agent             string,
    measure_timestamp           timestamp,
    start_timestamp             timestamp,
    end_timestamp               timestamp,
    insertion_timestamp         timestamp,
    CONSTRAINT dl_measurements_pk
        PRIMARY KEY (id) DISABLE NOVALIDATE
)
PARTITIONED BY (insertion_date string)
STORED AS TEXTFILE
LOCATION
'hdfs://namenode:9000/user/hive/warehouse/mfh.db/dl_measurements
'
TBLPROPERTIES ('bucketing_version' = '2');
```

As we can see, the schema defines an `insertion_date` partition, which determines how to store data into the table; *measurements* with the same `insertion_date` are held together into the same partition, allowing Hive to efficiently retrieve data that satisfies specified criteria based on the `insertion_date`.

Apache Hive ingestion

The first step for ingestion consisted in loading the generated dataset in a temporary HDFS folder (`/tmp/mfh`) on the Hadoop NameNode, to serve as the primary source for the ingestion process.
The second step was the creation of an external table that points to the temporary folder:

```
CREATE EXTERNAL TABLE IF NOT EXISTS dl_measurements_external
(
    id                          string,
    sensor_id                   string,
    sensor_type                 string,
    sensor_desc_name            string,
    double_value                double,
    str_value                   string,
    unit_of_measure             string,
    location_id                 string,
    location_name               string,
    location_description        string,
    location_botanic_name       string,
    location_cultivation_name string,
    location_latitude           double,
    location_longitude          double,
    location_altitude           double,
    insertion_agent             string,
    measure_timestamp           timestamp,
    start_timestamp             timestamp,
    end_timestamp               timestamp,
    insertion_timestamp         timestamp,
    CONSTRAINT dl_measurements_external_pk
        PRIMARY KEY (id) DISABLE NOVALIDATE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 'hdfs://namenode:9000/tmp/mfh'
TBLPROPERTIES ("skip.header.line.count"="1");
```

Finally, data was loaded from the external table into `dl_measurements` with the following SQL statement that computes the belonging `insertion_date` partition starting from the `insertion_timestamp`.

```
FROM dl_measurements_external
INSERT OVERWRITE TABLE dl_measurements PARTITION(insertion_date)
SELECT *, date_format(insertion_timestamp, 'YYYY-MM-dd') AS
insertion_date
DISTRIBUTE BY insertion_date;
```

Apache Druid datasource optimization

On Apache Druid, data was ingested with a *monthly * granularity using `insertion_timestamp` as the primary datasource timestamp; *measurements* registered within the same month are stored into the same time chunk (with a time chunk containing one or more segments).

Apache Druid ingestion

Apache Druid ingestion could be configured by submitting an ingestion task spec to the cluster Coordinator; the overall process was more accessible since it could be obtained by just using Druid's web console.
For instance, the ingestion spec used to load data from the temporary HDFS folder on the NameNode is the following:

```
{
  "type": "index_parallel",
  "spec": {
    "ioConfig": {
      "type": "index_parallel",
      "inputSource": {
        "type": "hdfs",
        "paths": "hdfs://namenode:9000/tmp/mfh/dl-
measurements_1642410970616.csv"
      },
      "inputFormat": {
        "type": "csv",
        "findColumnsFromHeader": true
      }
    },
    "tuningConfig": {
      "type": "index_parallel",
      "partitionsSpec": {
        "type": "dynamic"
      },
      "logParseExceptions": true
    },
    "dataSchema": {
      "timestampSpec": {
```

```
    "column": "insertion_timestamp",
    "format": "auto"
  },
  "dimensionsSpec": {
    "dimensions": [
      "id",
      {
        "type": "double",
        "name": "double_value"
      },
      "str_value",
      "unit_of_measure",
      "sensor_id",
      "sensor_type",
      "sensor_desc_name",
      "location_id",
      "location_name",
      "location_botanic_name",
      "location_cultivation_name",
      "location_description",
      {
        "type": "double",
        "name": "location_latitude"
      },
      {
        "type": "double",
        "name": "location_longitude"
      },
      {
        "type": "double",
        "name": "location_altitude"
      },
      "insertion_agent",
      "measure_timestamp",
      "start_timestamp",
      "end_timestamp",
      "insertion_timestamp"
    ]
  },
  "granularitySpec": {
    "queryGranularity": "none",
    "rollup": false,
    "segmentGranularity": "month"
  },
```

```
    "dataSource": "dl_measurements",
    "transformSpec": {
      "transforms": [
        {
          "type": "expression",
          "name": "insertion_timestamp",
          "expression": "timestamp_format(__time)"
        }
      ]
    }
  }
}
```

Ingestion performance

Database	Partitions	HDFS data size	HDFS replication size	Ingestion time
Apache Hive	1462 partions	14.6 GB	43.9 GB	02:55:8
Apache Druid	51 segments	9.9 GB	29.8 GB	01:17:15

Table 2: Ingestion numbers

Table 1 reports the number of partitions, the data size on HDFS, the total disk space used for data replication, and the computed ingestion time for each platform. It is worth observing that Apache Druid consumes significantly less disk space on HDFS since it automatically compresses segment data with LZ4 and Roaring algorithms.
As a result, Apache Druid was 55,8% faster than Apache Hive to import 50 million rows from Hadoop.

3.4 Queries

The benchmark developed to test Apache Hive, and Apache Druid comprises six SQL queries chosen from the ones executed by MIND Foods Hub's front-end. These queries are the most representative of the analysis processes of MIND Foods HUB and make the performance testing similar to an actual production workload.
In addition, the queries are, where feasible, slightly optimized for each platform to make use of Apache Hive table partitions and Apache Druid time segments.

Query 1

Query 1 selects the first 100 measurements of `measurements` sensors in a month time range.

Apache Hive:

```
SELECT *
FROM mfh.dl_measurements
WHERE insertion_date >= '2021-12-01'
AND insertion_date <= '2021-12-31'
AND double_value IS NOT NULL
LIMIT 100;
```

Apache Druid:

```
SELECT *
FROM dl_measurements
WHERE __time >= '2021-12-01'
AND __time <= '2021-12-31'
AND double_value IS NOT NULL
LIMIT 100;
```

Query 2

Query 2 selects the first 100 *tag* sensors measurements in a month time range.

Apache Hive:

```
SELECT *
FROM mfh.dl_measurements
WHERE insertion_date >= '2021-12-01'
AND insertion_date <= '2021-12-02'
AND str_value IS NOT NULL
AND start_timestamp IS NULL
LIMIT 100;
```

Apache Druid:

```
SELECT *
FROM dl_measurements
WHERE __time >= '2021-12-01'
AND __time <= '2021-12-02'
AND str_value IS NOT NULL
AND start_timestamp IS NULL
LIMIT 100;
```

Query 3

Query 3 selects the first 100 `phase` sensors measurements in a month time range.

Apache Hive:

```
SELECT *
FROM mfh.dl_measurements
WHERE insertion_date >= '2021-12-01'
AND insertion_date <= '2021-12-02'
AND str_value IS NOT NULL
AND start_timestamp IS NOT NULL
LIMIT 100;
```

Apache Druid:

```
SELECT *
FROM dl_measurements
WHERE __time >= '2021-12-01'
AND __time <= '2021-12-02'
AND str_value IS NOT NULL
AND start_timestamp IS NOT NULL
LIMIT 100;
```

Query 4

Query 4 groups and count all cultivated locations in "cassoni sx".

```
SELECT location_id, location_name, location_botanic_name,
location_cultivation_name, COUNT(*) AS number_of_measurements
FROM dl_measurements
WHERE location_id = 'cassoni_sx'
GROUP BY location_id, location_name, location_botanic_name,
location_cultivation_name;
```

Query 5

Query 5 groups and count all available sensors.

```
SELECT sensor_id, sensor_type, sensor_desc_name, COUNT(*) AS
number_of_measurements
FROM dl_measurements
GROUP BY sensor_id, sensor_type, sensor_desc_name;
```

Query 6

Query 6 calculates the average of all "TS_0310B473-depth_soiltemperature" sensor measurements.

```
SELECT sensor_id, location_cultivation_name, AVG(double_value)
AS average
FROM dl_measurements
WHERE sensor_id = 'TS_0310B473-depth_soiltemperature'
AND location_id = 'cassoni_sx'
AND location_cultivation_name = 'Rubiaceae'
GROUP BY sensor_id, location_id, location_cultivation_name;
```

3.5 Performance testing using Apache JMeter

To test the performance of Apache Hive and Apache JMeter on an average workload, I used Apache JMeter, an open-source Java application designed to measure the performance of software applications. It can simulate workloads on servers, networks or applications to analyze the overall performance under different load types. In addition, Apache JMeter supports performance testing for various applications and protocols: TCP, HTTP and HTTPS, FTP, Database via JDBC, SOAP and others.

In JMeter, performance testing is defined by a *Test Plan*, which specifies a series of steps that JMeter executes on each run. For each Test Plan, a user can configure one or more *Thread Groups*, the number of threads JMeter will use to complete the performance test. Apache JMeter employs a multi-thread architecture: each Thread Group executes the Test Plan independently of the other threads; multi-threading allows JMeter to simulate concurrent connections from multiple users to the system under test. Thread Groups are composed of one or more *Samplers*. A Sampler is a JMeter component that performs the actual work by sending the requests to the system and collecting the responses and various related samples. Apache JMeter offers different built-in configurable Samplers to test a system across multiple applications and protocols, notably: FTP requests, HTTP requests, JDBC Requests, JDBC Requests, OS Process requests, and others. Another critical component of JMeter is *Listeners*, which provide access to sample results by gathering information while JMeter runs. Various types of Listeners exist for different purposes: "Graph Results" listener plots response times on a graph, while "Aggregate Report" listener reports, for each request various aggregated information, like the Average Response Time, throughput, the percent of requests with errors, and others.

Apache JMeter can run in "Graphical User Interface" (GUI) mode, offering a full-featured interface to configure and run a Test Plan. However, since the GUI and the test Listeners consume a generous amount of memory while the Test Plan is running, it is common to run JMeter in "Command Line Interface" (CLI) mode to reduce resource usage and to avoid altering sample results.
After a Test Plan is complete, JMeter can output the results to a file, usually CSV or XML, containing all registered samples; this file can be successively imported with the GUI for later processing and analysis.

Hive HTTP Proxy

The performance testing was intended to run against each database HTTP API. Sadly, Apache Hive does not expose a set of REST APIs to interact with, contrary to other more recent platforms. Instead, a client is forced to use Hive JDBC drivers or Hive Thrift APIs to perform TCP connections to the database. At the time of this writing, Hive's only available REST API is *WebHCat*, a web application layer on top of HCatalog, a table and storage management layer for Hadoop.
While WebHCat is installed with Hive, starting with Hive release 0.11.0, its capabilities are pretty limited: a client can run a Hive query or set of commands via the `http://hostname/templeton/v1/hive` endpoint, but the response contains only the ID of a job that runs on background on Hadoop, and an optional callback property, that defines a URL called upon job completion. This behaviour makes it very hard to test

Apache Hive with JMeter unless configuring the latter to connect to Hive via JDBC, a not viable option due to the configuration difficulties.
To work around this problem, I decided to write "Hive HTTP Proxy" [38], a Node.js application that functions as an HTTP layer on top of Apache Hive. Hive HTTP Proxy working is simple: internally, it uses *Hive Driver*, a Javascript implementation of the Hive Thrift APIs to connect to Hive, and exposes a single HTTP endpoint on port 10001. A client willing to execute Hive SQL statements via HTTP can do a POST request with a JSON payload, containing the required statement, to Hive HTTP Proxy; the following example shows how to perform a simple SQL statement:

```
$ curl -X POST -d '{ "statement": "SELECT 1" }'
http://localhost:10001
```

Hive HTTP Proxy source code is hosted on SESAR Lab Github's organization and is released under the Apache 2.0 License.

JMeter and test configuration

Data Lake platforms are very different from application databases that support hundreds or thousands of concurrent connections under heavy load. Instead, especially for small to mid organizations, Data Lake platforms run data extraction for analytical and business purposes, consisting of a few concurrent requests, often triggering batch jobs that run from hours to days.
This characteristic led to the decision of configuring JMeter to assess single-user query performance testing, simulating an average workload; only one thread for each query was employed to run the Test Plan.

JMeter ran with the following conditions:

- The query cache for each database was disabled.
- Vectorization was enabled for Apache Druid queries. With vectorization, query execution is faster by processing batches of 512 rows instead of one row at a time. Apache Hive supports vectorized query execution only for data stored in ORC format.
- Single user testing: only one thread per Thread Group was employed.
- Each query was configured to run 10 times (to have 10 samples per query) in a separate Thread Group.
- Each Thread Group ran consecutively (one at a time) to avoid side effects on other requests.

Instead of running JMeter on the same network of the cluster to minimize request latency, the benchmark simulated an everyday use case, or rather a stakeholder of an organization that uses its client machine to extrapolate and analyze data and business insights from its Data Lake.

Therefore, all tests were executed in CLI mode and with all Listeners disabled, on a MacBook Pro Mid 2015 with these specs:

- CPU 2,8 GHz Intel Core i7, 4 core/8 threads
- 16 GB 1600 MHz DDR3 memory
- 512 SSD storage
- macOS 12.1 Monterey operating system

4. Test results

Each JMeter execution produced a CSV dataset containing the test results for each platform, one for Apache Hive the other for Apache Druid. First, to simplify the comparison of the two results, datasets were merged in a single CSV file with the *Merge Results* plugin [39]. Then, the combined results were imported in *Summary Report* JMeter listener to calculate, for each sample:

- Average response time
- Minimum response time
- Maximum response time
- Average response time standard deviation

Also, each query's average response time was compared with the *Results Comparator* [40] plugin to quantify the performance difference between Apache Hive and Apache Druid by calculating Cohen's *d* effect size.
Finally, the combined results were imported into Google Sheets to draw the bar graph that visually compares each query's average response time. The results for each query are illustrated in the following pages.

JMeter test results are downloadable from the "MIND Foods HUB Data Lake Performance Testing" [41] repository hosted on SESAR Lab Github's organization; they are released under the Apache 2.0 License.

4.1 Query 1

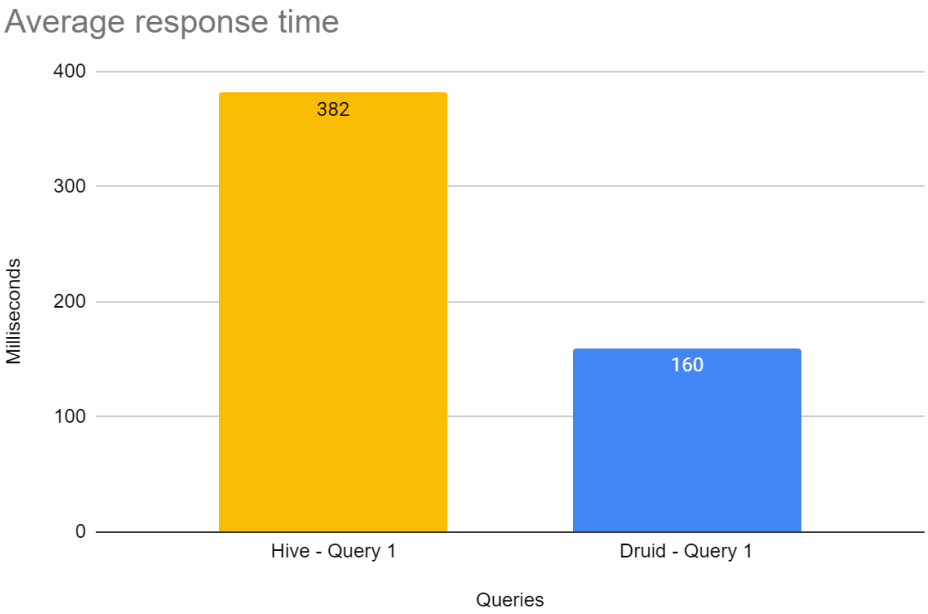

Figure 11: Query 1 Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 1	382	346	457	39,50
Druid - Query 1	160	148	229	23,22

Table 3: Query 1 numbers

Performance	Cohen's d	Average Difference
Query 1	6.87	Huge decrease

Table 4: Performance evaluation for Query 1

Query 1 execution is fast on both platforms since it takes advantage of time partitions on Apache Hive and time segmentations on Apache Druid; the average response time is under 500 milliseconds.
However, we can observe a considerable decrease in average response time value using Apache Druid, making this platform greatly more performant than Hive.

4.2 Query 2

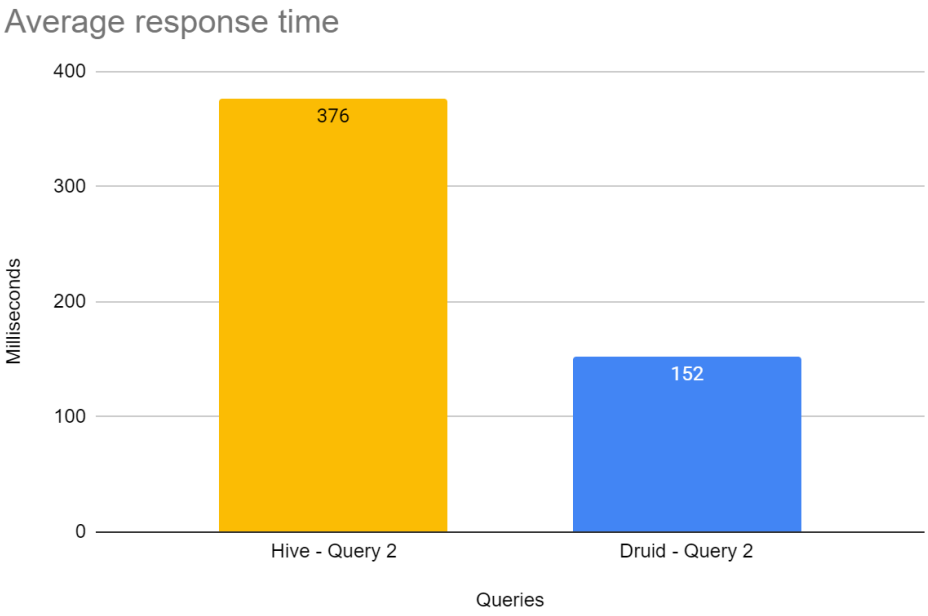

Figure 12: Query 2 Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 2	376	359	401	13,55
Druid - Query 2	152	148	163	4,17

Table 5: Query 2 numbers

Performance	Cohen's d	Average Difference
Query 2	22.37	Huge decrease

Table 6: Performance evaluation for Query 2

Like Query 1, Query 2 exploits time partitioning on each platform, making its execution pretty fast, under a 500 milliseconds threshold. Nevertheless, we can see an extensive decrease in the Average response time value using Apache Druid.

4.3 Query 3

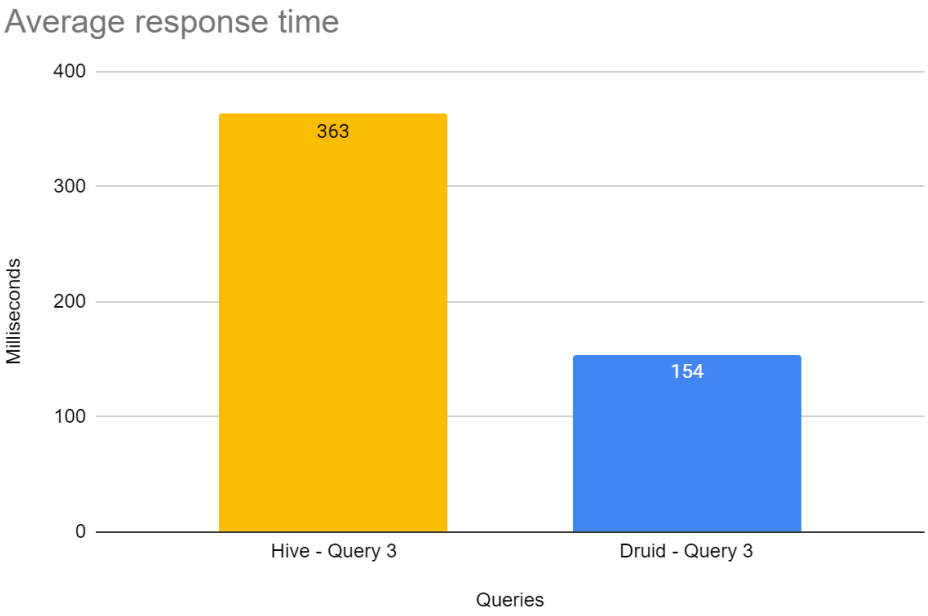

Figure 13: Query 3 Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 3	363	350	383	12,67
Druid - Query 3	154	148	168	5,99

Table 7: Query 3 numbers

Performance	Cohen's d	Average Difference
Query 3	21.15	Huge decrease

Table 8: Performance evaluation for Query 3

Query 3 follows the same trends, achieving a massive decrease in average response time value for its execution on Apache Druid.
We can notice how the behaviour of all time queries is the same on both platforms, with a similar average response time between Query 1, Query 2 and Query 3 per database and minor standard deviation.

4.4 Query 4

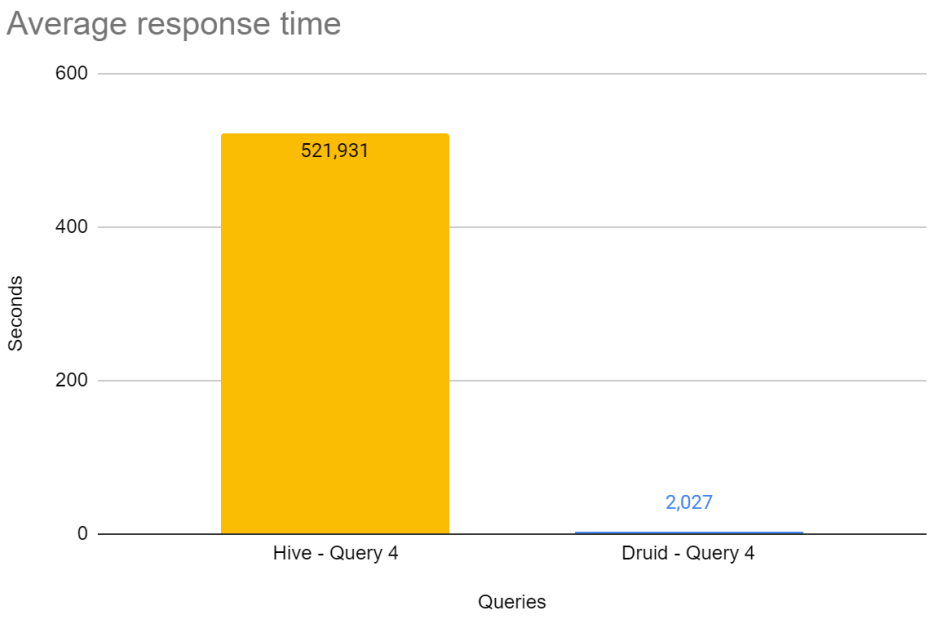

Figure 14: Query 4 Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 4	521931	518235	526918	3157,38
Druid - Query 4	2027	2020	2038	6,53

Table 9: Query 4 numbers

Performance	Cohen's d	Average Difference
Query 4	232.87	Huge decrease

Table 10: Performance evaluation for Query 4

Query 4 groups four different columns, and its execution shows a distinct behaviour between the two platforms. While Apache Druid execution for Query 4 is sensibly slower than time queries, with an average response time of 2 seconds circa, Apache Hive is enormously slower, requesting 8,69 minutes to query the data.
Two factors essentially cause this:

1. The query does not use any partitions, forcing Hive to do a full table scan to select and group the requested values.

2. The query uses 61 Mapper and 62 reducers for its execution. Hence its execution requires a discrete amount of I/O synchronization between the Mappers and Reducers, resulting in multiple disk writing operations that are notoriously slow.

On the other side, as already described in section 2.4, Apache Druid aggregates the results from the Historicals involved in the query in memory, streaming the final result set directly from the Broker to the client.
This behaviour allows Druid to never read data from HDFS at query time, resulting in high-speed query execution.

4.5 Query 5

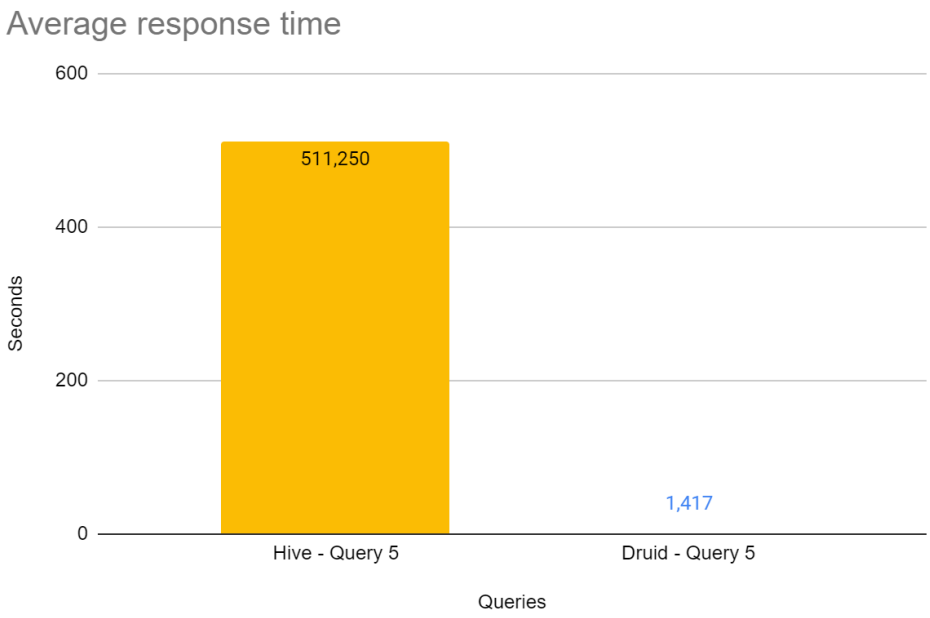

Figure 15: Query 5 Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 5	511250	503896	518260	3898,01
Druid - Query 5	1417	1401	1449	12,75

Table 11: Query 5 numbers

Performance	Cohen's d	Average Difference
Query 5	184.97	Huge decrease

Table 12: Performance evaluation for Query 5

Query 5 shows the exact behaviour of Query 6, with Apache Hive forced to do a full table scan to aggregates sensor's related data. This results in an average response time of 8,52 minutes circa per query execution.
Again, Apache Druid is exceedingly performant, remaining under the 2 seconds threshold for Query 5 completion..

4.6 Query 6

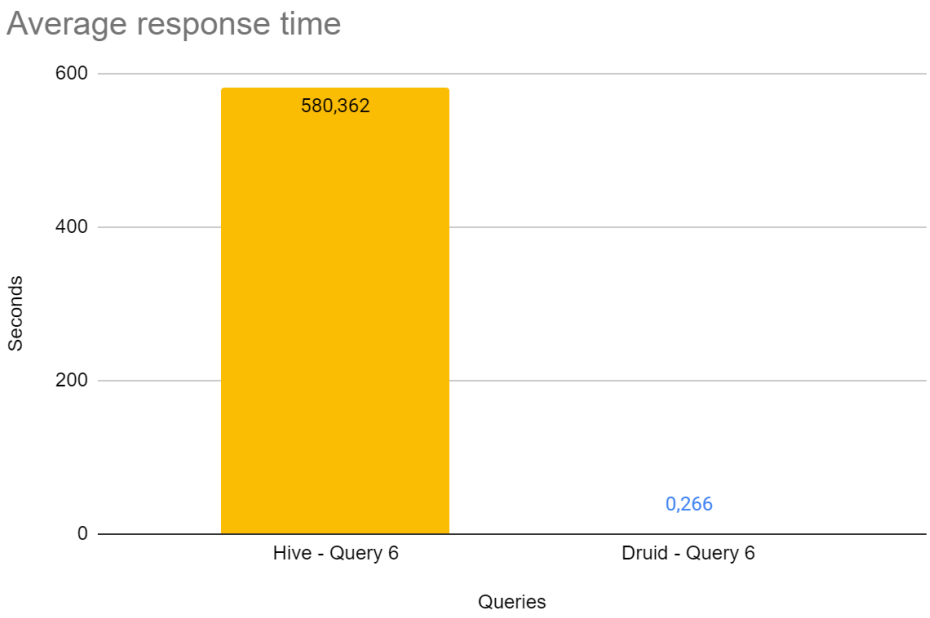

Figure 16: Average response time

Query	Average	Min	Max	Std. Dev.
Hive - Query 6	580362	575520	585806	2952,75
Druid - Query 6	266	264	271	2,01

Table 13: Query 6 numbers

Performance	Cohen's d	Average Difference
Query 6	277.84	Huge decrease

Table 14: Performance evaluation for Query 6

Query 6 behaves the same in Apache Hive, bound to a full table scan that requests 9,67 minutes to accomplish its execution. However, Apache Druid acts differently from Query 4 and Query 5, with an average response time of only 264 milliseconds for Query 6 execution.

5. Conclusions

Big Data heavily changed the shape of business analytical processes. Organizations of all domains and sizes now employ, in one way or another, complex processes that handle large volumes of data on which they base their business decisions. Choosing the right platform to support these processes is a difficult task and can make a real difference in the market competitiveness of these organizations.
MIND Foods HUB, an international, interdisciplinary project, employs a complex computing infrastructure to store and analyze data of an innovative plant-phenotyping process. The core component of this infrastructure is a Data Lake platform comprised of Apache Hadoop, a framework for parallel and distributed processing of large datasets and Apache Hive, a Data Warehouse software that allows reading, writing, and managing large datasets using SQL.
The main goal of this research was to find a valid, more performant alternative to Apache Hive for the MIND Foods HUB Data Lake due to its low maintainability and the inferior performances for even simple aggregations queries.
The alternative platform's requirements are two software qualities: maintainability and performance. The ideal substitute system should be easy to configure, maintain, and integrate well with the existing Hadoop ecosystem on which MIND Foods HUB relies for different use cases. Also, the alternative platform should guarantee sub-seconds performance for aggregation queries. I identified Apache Druid as a proper alternative solution to replace Apache Hive. This open-source distributed data store supports various modern applications, like real-time analytics on large datasets and fast data aggregations for highly concurrent APIs. The table below shows the essential differences between Hive and Druid.

	Apache Hive	**Apache Druid**
Use cases	ETL tasks, reporting, and data analysis in batch mode with SQL.	Real-time analytics, high concurrency and sub-second queries with SQL.
Storage	HDFS.	HDFS, Amazon S3, Google Cloud Storage, Microsoft Azure deep storage, local disk.
Partitioning	With string partition keys, optional.	By default, partition data in time segments.
Ingestion	Batch mode.	Batch and streaming mode.

Table 15: Apache Hive and Apache Druid comparison

5.1 Maintainability and Performance results

Initially developed in 2009, Apache Hive is not designed to work with modern dockerized environments.
During the testing, Hive maintainability has proved to be scarce. The cluster configuration, consisting of ad-hoc, custom Docker images and various bash scripts, was complex, and the official Hive documentation was not always helpful. Also, Apache Hive does not implement many features of more modern platforms, like a REST API to submit queries; this limitation forced the development of an HTTP proxy layer on top of Hive to test the system properly.
Apache Druid instead remarkably satisfied the maintainability requirement. The provisioning of a cluster comprised of all Druid's components was straightforward, thanks to the official Docker images and the related documentation that is well detailed and comprehensive of the various deployment modes of the platform. Also, Apache Druid supports a rich extensions ecosystem to add various functionality at runtime, like the support for Amazon S3, Google Cloud Storage or Microsoft Azure instead of HDFS for segment storage. This interoperability with the aforementioned cloud computing services allows the substitution of Hadoop with immediate advantages in terms of maintainability costs.
However, some limitations should be considered when it comes to the maintainability requirement; the overall maintainability of a software system is usually evaluated on a long-term scenario, working on various operational constraints and for different business requirements. Unfortunately, an evaluation of this type is out of this research's scope, so the maintainability of Apache Hive and Apache Druid was tested only for the ambit of the performance testing.

To test the performance of Apache Hive and Apache Druid, I focused on finding a valid benchmark, that is, a workload representative of how the system is used in the field and then run the system on those benchmarks. To achieve this goal, I followed a rigorous, thorough methodology, comprises five steps: the provisioning of both Apache Hive and Druid, the generation of a large, random, synthetic dataset, the ingestion of the generated data, the definition of the test query and the performance testing with Apache JMeter.
Apache Druid proved to be more performant than Apache Hive on each tested query; Query 1, Query 2 and Query 3, which take advantage of time partitioning, are all under the 500ms threshold. Furthermore, we can observe a considerable decrease in average response time using Apache Druid. The indisputable performance increment could be observed with Query 4, Query 5 and Query 6, which aggregate data according to different criteria. Apache Hive needs a 9 minutes average response time to satisfy these requests; Apache Druid has always been under the 2 seconds threshold. That is where the design of Apache Hive shows its limits: the dependence on MapReduce to retrieve and aggregate data is suboptimal due to its recurring access to the disk, a notoriously slow operation. It should be noted that Hive could be further optimized by attempting various approaches; for instance, using Apache Tez, a different computation engine for Hadoop, instead of MapReduce could speed up the queries execution. Alternatively, configuring the `dl_measurements` table to use Apache ORC, a columnar file format, could improve the reading performance of HiveQL queries. Nevertheless, each of these methods comes at the cost of configuring and maintaining a complex platform that is not suited to perform real-time analytics tasks.

On the other side, Apache Druid implements a more modern architecture combining column-oriented storage, distributed computing where each node can independently be scaled, and an advanced real-time ingestion system that serves data as soon as it is available. This design allows Druid to compute sub-second queries on datasources with billions of rows.
In conclusion, Apache Druid fully satisfied the goals of this research, proving itself to be a suitable, highly performant solution to serve the actual use cases of the MIND Foods HUB computing infrastructure.

5.2 Future Work

The work done for this research opens on further possible developments. First, the integration of Apache Druid in the MIND Foods HUB Data Lake is not complete. The provisioning described in section 3.1 was achieved on the existing Data Lake architecture, integrating Apache Druid with the Hadoop cluster to ingest data from HDFS. However, the ingestion process, handled by the hybrid GraphQL/REST API, should be revised. The API ingests new measurements, loaded from the collection agents, into the `dl_measurements` table using a simple `INSERT` statement with the corresponding values. As we know, Apache Druid does not support DML operations and allows data ingestion only in batch mode, by loading files from different storages, or in streaming mode, by reading events from Apache Kafka or Amazon Kinesis. So, a natural extension of this work would be the integration of Apache Kafka in the Data Lake cluster. With this configuration, collection agents *publish* measurements to Kafka, while Druid *consumes* them in real-time; in this way, the write operations could be decoupled from the read ones, allowing the API that serves the client applications to be scaled independently from the write architectural layer.
Coming to scalability, as described in section 3, the test of this property was out of the scope of this research: MIND Foods HUB started to ingest measurements data as soon as I joined the project, and the resources available for the research was not adequate to scale both Hive and Druid to support a heavy workload. Therefore, a natural continuation of this work could be the test of Apache Druid with a benchmark designed to quantify its scalability under a peak of multiple, concurrent and high-compute demanding requests. A benchmark of this type should measure, other than time-related metrics, like average response time or throughput, also resource-related metrics: the number of nodes the systems needs to support the peak of load, the CPU and memory consumptions, and the storage needed by Historicals to load segments after ingestions.
Finally, the last development of this research would be the testing of other platforms for the MIND Foods Hub Data Lake. While the SESAR Lab team was investigating various alternatives like Presto (now Trino), a distributed SQL query engine for big data analytics, or Apache Impala, an SQL layer on Hadoop different from Hive, the research on a valid substitute could focus on cloud solutions, like Google Big Query, to remove the cost of configuring and maintaining an in-house architecture.

6. Acknowledgements

A bachelor's degree thesis is only the final step of a route that, for me, endured for four years of hard work and dedication; this work would not have been possible without the many people that, in one way or another, accompanied me in this long journey.
First and foremost, I want to thank teacher Paolo Ceravolo and the SESAR Lab team guys, who introduced me to the MIND Foods HUB project, and especially Filippo Berto, who carefully revisioned each version of this paper and patiently helped me to reach the end of this research.
I should also thank Sabrina, the beating heart of SSRI online, for her precious help with the organization of my studies, and all my course colleagues that every day are committed to completing this challenging path: don't give up, guys, you are one step closer to the goal!
Then, I want to thank my crazy fellows of DNF: Manu, Robs, Matti, Ricky, Frone, Vale, Giorgio, Sepe; these guys are always around for troubles and good times, and our long friendship is invaluable!
Talking about friends, Jeffrey boys, and the magnificent Kek nights have been a source of incredible fun for me: Rprt, Raibaz, Paolone, Memo, Omar, Tired, thank you!
People come and go, so here we are: I want to thank Gab, an incredible mate and a true inspiration for enjoying life.
Robi, and the *compari* from Messina for all the fun during the hot days in Sicily: see you soon, guys!
For her amazing help during these years, I want to acknowledge Doc Sonia Marino: we made it, doc.
And then, love is an important business: I would like to thank Daniela and Deborah, my joyful past, and Chiara, my lovely, beautiful present.
Finally, I have to thank my family for their tireless support during the most challenging time, for their unconditional love and forbearing me despite my hot temper. Teresa, Felice, I really love you!

7. References

[1] "MIND Foods HUB". [Internet]. Available from: https://www.mindfoodshub.com/il-progetto/

[2] DISAA press, "The MIND Foods HUB project is ready to go" [Internet]. Available from: https://disaapress.unimi.it/en/2020/02/17/the-mind-foods-hub-project-is-ready-to-go/

[3] The Apache Software Foundation, "Apache Hadoop". [Internet]. Available from: https://hadoop.apache.org/

[4] The Apache Software Foundation, "Apache Hive". [Internet]. Available from: https://hive.apache.org/

[5] I. Gorton, "Software Quality Attributes", In: Essential Software Architecture. Springer, Berlin, Heidelberg. 2011

[6] A. Bondi, "Characteristics of Scalability and Their Impact on Performance.", Proceedings of the 2nd International Workshop on Software and Performance, ACM, pp. 195–203. 2000.

[7] "IEEE Standard Glossary of Software Engineering Terminology", IEEE, pp. 1–84. 1990.

[8] The Apache Software Foundation, "Apache Druid". [Internet]. Available from: https://druid.apache.org/

[9] The Apache Software Foundation, "Apache JMeter". [Internet]. Available from: https://jmeter.apache.org/

[10] L. Zhenyu, "Research of Performance Test Technology for Big Data Applications", 2014 IEEE International Conference on Information and Automation (ICIA), IEEE, pp. 53–58. 2014.

[11] Arne von See, "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025" 2020. [Internet]. Available from: https://www.statista.com/statistics/871513/worldwide-data-created/

[12] D. Reinsel, J. Gantz and J. Rydning, "Data Age 2025: The Digitization of the World From Edge to Core" International Data Corporation, 2018.

[13] J. S. Ward and A. Barker, "Undefined By Data: A Survey of Big Data Definitions.," CoRR, vol. abs/1309.5821, 2013.

[14] S. Madden, "From Databases to Big Data," IEEE Internet Computing, vol. 16, no. 3, pp. 4–6, 2012.

[15] L. Douglas, "3d data management: Controlling data volume, velocity and variety", Meta Group, 2001.

[16] M. Schroeck, R. Shockley, J. Smart, D. Romero-Morales, P. Tufano, "Analytics: The Real-World Use of Big Data: How Innovative Enterprises Extract Value from Uncertain Data. Executive Report", IBM, 2012. [Internet]. Available from: https://www.bdvc.nl/images/Rapporten/GBE03519USEN.PDF

[17] W. Inmon, "Building the Data Warehouse", John Wiley and Sons, 2005.

[18] J. Dixon, "Pentaho, Hadoop, and Data Lakes", 2010. [Internet]. Available from: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/

[19] A. Hassan, C. Walker, "Personal Data Lake with Data Gravity Pull", 2015 IEEE Fifth International Conference on Big Data and Cloud Computing, IEEE, pp. 160–67. 2015.

[20] S. Ghemawat, H. Gobioff, S. Leung, "The Google File System", "Proceedings of the 19th ACM Symposium on Operating Systems Principles", ACM, Bolton Landing, NY, pp. 20-43. 2003. [Internet] Available from: https://research.google/pubs/pub51/

[21] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", "OSDI 04: Sixth Symposium on Operating System Design and Implementation", San Francisco, CA, pp. 137-150, 2004. [Internet]. Available from: https://research.google/pubs/pub62/

[22] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, "Apache Hadoop YARN: Yet another resource negotiator", Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October. ACM, New York, NY, USA, pp. 5: 1–5:16. 2013.

[23] The Apache Software Foundation, "Apache ZooKeeper". [Internet]. Available from: https://zookeeper.apache.org/

[24] D. R. Cutting, D. R. Karger, J. O. Pedersen, J. W. Tukey, "Scatter / Gather: Browsing A Cluster-based Large Document Collections", Sigir '92, pp. 318–329. 1992.

[25] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, D. Ganguli, "Druid: A real-time analytical data store", Proceedings of the ACM SIGMOD International Conference on Management of Data. 2014.

[26] Y. Collet, "LZ4 - extremely fast compression". [Internet]. Available from: https://lz4.github.io/lz4/

[27] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, "Consistently faster and smaller compressed bitmaps with Roaring", Software: Practice and Experience, 46(11): pp. 1547–1569. 2016.

[28] The Apache Software Foundation, "Apache Kafka". [Internet]. Available from: https://kafka.apache.org/

[29] Amazon, "Amazon Kinesis". [Internet]. Available from: https://aws.amazon.com/it/kinesis/

[30] E.J Weyuker, F.I. Vokolos, "Performance testing of software systems", WOSP '98: Proceedings of the 1st international workshop on Software and performance, pp. 80–87. 1998.

[31] J. Cohen, "Statistical Power Analysis for the Behavioral Sciences", Routledge, 1988.

[32] S Sawilowsky, "New effect size rules of thumb", Journal of Modern Applied Statistical Methods, Vol. 8: Iss. 2, 2009.[Internet]. Available from: https://digitalcommons.wayne.edu/jmasm/vol8/iss2/26/

[33] Karl. Huppler, "The Art of Building a Good Benchmark", In Performance Evaluation and Benchmarking, pp. 18–30. Berlin, Heidelberg: Springer Berlin Heidelberg. 2009.

[34] Docker Inc., "Docker Compose", [Internet]. Available from: https://docs.docker.com/compose/

[35] P. O'Neil, E. O'Neil, C. Xuedong, S. Revilak, "The Star Schema Benchmark and Augmented Fact Table Indexing", In Performance Evaluation and Benchmarking, pp. 237–252. Berlin, Heidelberg: Springer Berlin Heidelberg. 2009.

[36] G. D'Arrigo, "MFH measurements generator". [Internet]. Available from: https://github.com/SESARLab/mfh-measurements-generator.

[37] M. Brocato, "Mockaroo". [Internet]. Available from: https://www.mockaroo.com/

[38] G. D'Arrigo, "Hive HTTP Proxy". [Internet]. Available from: https://github.com/SESARLab/hive-http-proxy

[39] F. Henry, V. Daburon, "Merge Results". [Internet]. Available from: https://jmeter-plugins.org/wiki/MergeResults/#Merge-Results

[40] rbourga, Results Comparator Plugin. [Internet]. Available from: https://github.com/rbourga/jmeter-plugins-2/blob/main/tools/resultscomparator/src/site/dat/wiki/ResultsComparator.wiki

[41] G. D'Arrigo, "MIND Foods HUB Data Lake Performance Testing". [Internet]. Available from: https://github.com/SESARLab/mfh-dl-performace-testing