Research & Development: Enterprise Architecture

By Krai Chamnivikaipong

Advisor:
Dr. Akkarit Sangpetch
Dr. Orathai Sangpetch

Table of Contents

Motivation 3

Problem 4

Evaluation / Analysis 6

1. Feasibility Evaluation 7

Minio 7

HDFS 10

Apache Spark 13

Delta Lake 14

Apache Hudi 15

Apache Iceberg 16

2. Integration Evaluation 17

Tools Combination 17

Example Use-Case 18

3. Performances Evaluation 19

Create a Test Environment 19

Generate a Necessary Testing Data (Loads) 27

Create a Testing Script 28

Result Analysis 32

Finding / Insight 49

Comparison Table: Data Storage 49

Comparison Table: Writing Format 51

Conclusion 52

GitHub Repository 52

References 53

Motivation

An author is assigned to work on the project named Enterprise Architecture. Given the client's current data warehouse design is aggregating all the data via Apache Kafka and storing it in PostgreSQL, there is still room for improvement in several aspects such as processing speed, scalability, and availability.

However, the design and tools for a new Enterprise Architecture in the infrastructure layer and data ingestion layer were already settled which are Kubernetes and Apache Kafka. Therefore, the author's focus remains on the data storage layer and data computation layer.

Inspired by the opportunity, the author went through several existing solutions for data warehouse, data lake, and data lakehouse [1]. Apparently, the concept of data lakehouse [2] is started to be spread around 2020 and there are several advertised tools out there in the community. However, there are still not many of the examples and published use-cases by the community on the available open-source tools. Moreover, **combining these tools together to bring up the most efficiency** in terms of processing power, availability, fault tolerance, and more attributes from these tools is considered to be challenging or even something new to the industries. As there usually is a tradeoff between attribute for another. Hence, an author is decided to explore the tools based on the rough idea of following metrics.

- Performance: processing speed, availability, fault tolerance, etc.
- Configuration Complexity: deployment, network configuration, etc.
- Integration Complexity: integration with other available tools in the ecosystem
- Learning Curve: documents, tutorials, syntax, knowledge to be consumed
- Community: popularity, helps and supports from the internet
- Existing Use-Cases: industrial use-cases, demonstration use-cases, etc.

The author then decided the **solution that is able to balance all the attributes in the metrics without showing any sign of significant negative impact** (configuration failure, low fault tolerance, no support documents, etc.) **toward the user later**. Though, weights for the performance, configuration complexity, and integration complexity will be heavier than other attribute in the metrics.

Problem

Hence, in order to **find the most efficient combinations** out of the existed tools, the following steps are required for each tool to be evaluated:

1. **Feasibility**: if the tools can be configured or simple enough to configure given the circumstances of every party in holistic views

2. **Integration**: if the tools can be integrated with other tools with an acceptable configuration complexity

3. **Performance**: if the performance of the tools is acceptable in terms of processing power in comparison to traditional methods or the tradeoff for additional features

The tools will be separated into 3 categories which are data storage, computational & analytical tools, and writing format tools. The following diagram represents a high-level architecture of data lakehouse solution and the tools of their category.

Data storage (file distributed storage) is one of the powerful tools that allows the user to store any type (or format) of files into it. Given such a property, it can be referred (worked) as a lake of data where any kind of data can be stored (dumped) in it. However, the data storage itself is insufficient in handling and organizing the stored data, this is

where the computational tools and writing formats came in. The writing formats help reducing the size of the csv files by approximately 10 times. With a help of computational tools that able to process and transform a very large size of data in a short period of time, it can easily transform data into desire writing formats in no time. Some computational tools also able to use for the analytical purpose as well. In addition to the general writing format (Parquet), several producers have developed new writing formats which created a metadata or additional schema on top of the Parquet to ensure the ACID properties and extra features such as time-traveling too. It helps in organizing, indexing, and handling the lake of data which leads to the concept of data lakehouse eventually.

Despite several tools being mentioned out there in the community, there are some tools that have a strong disclaimer or are heavily mentioned in the area of their own use. Thus, the following tools are selected by the author to be evaluated:

Data Storage

- **Minio** [3]: high-performance, S3 compatible object storage. Native to Kubernetes, it is the only object storage suite available on every public cloud, every Kubernetes distribution, the private cloud, and the edge. It is software-defined and is 100% open source under GNU AGPL v3.

- **HDFS** [4]: is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject.

Data Computation

- Apache Spark [5]: a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Data Writing Formats

- Delta Lake [6]: an open-source project that enables building a Lakehouse architecture on top of data lakes. Delta Lake provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing on top of existing data lakes, such as S3, ADLS, GCS, and HDFS.

- Apache Hudi [7]: brings transactions, record-level updates/deletes, and change streams to data lakes. It is a rich platform to build streaming data lakes with incremental data pipelines on a self-managing database layer while being optimized for lake engines and regular batch processing.

- Apache Iceberg [8]: an open table format for huge analytic datasets. Iceberg adds tables to compute engines including Spark, Trino, PrestoDB, Flink, and Hive using a high-performance table format that works just like a SQL table.

Evaluation / Analysis

As is previously mentioned in the problem section that there are 3 steps for the tools to be evaluated which are **feasibility**, **integration**, and **performance**. Despite there being a single main idea for the evaluation process which is trial & error, there is still a detail for different solutions and methods toward each step that can be summarized as the followings:

1. Feasibility Evaluation

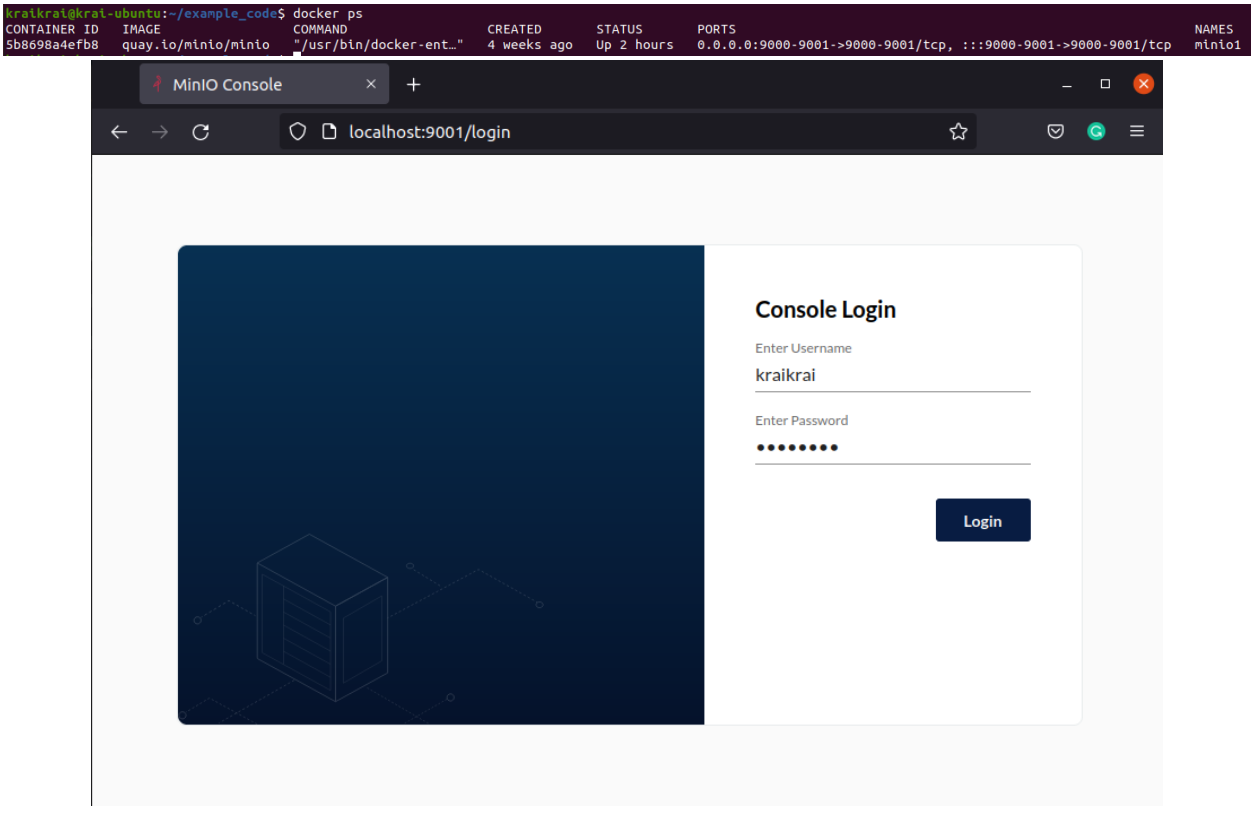
Minio

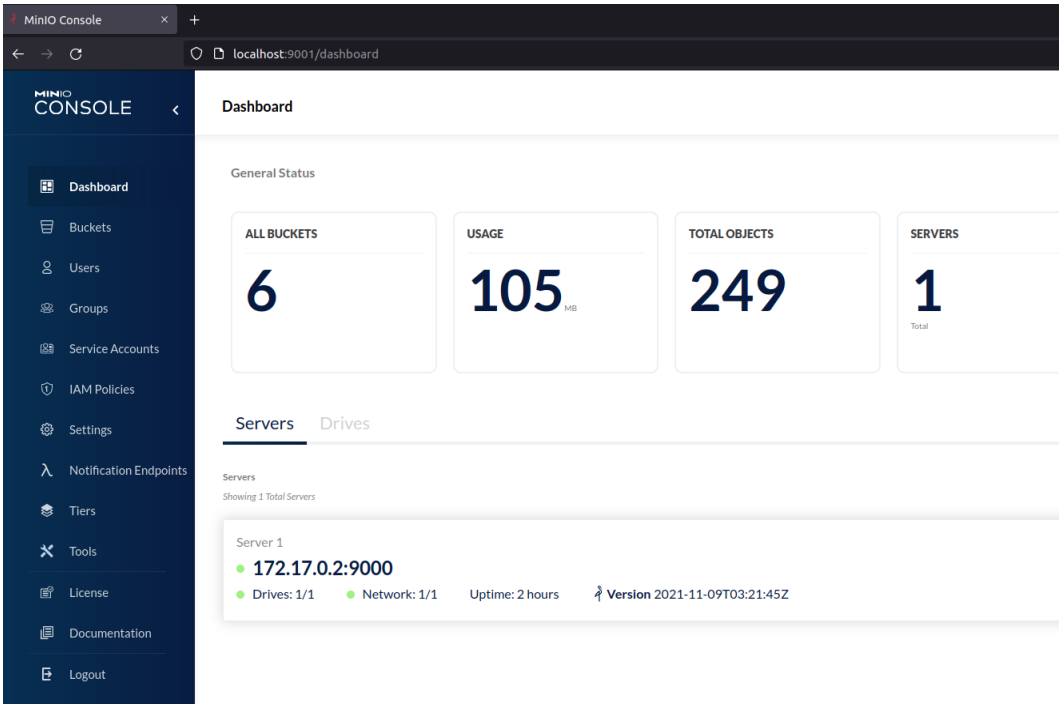
Setting Up Minio

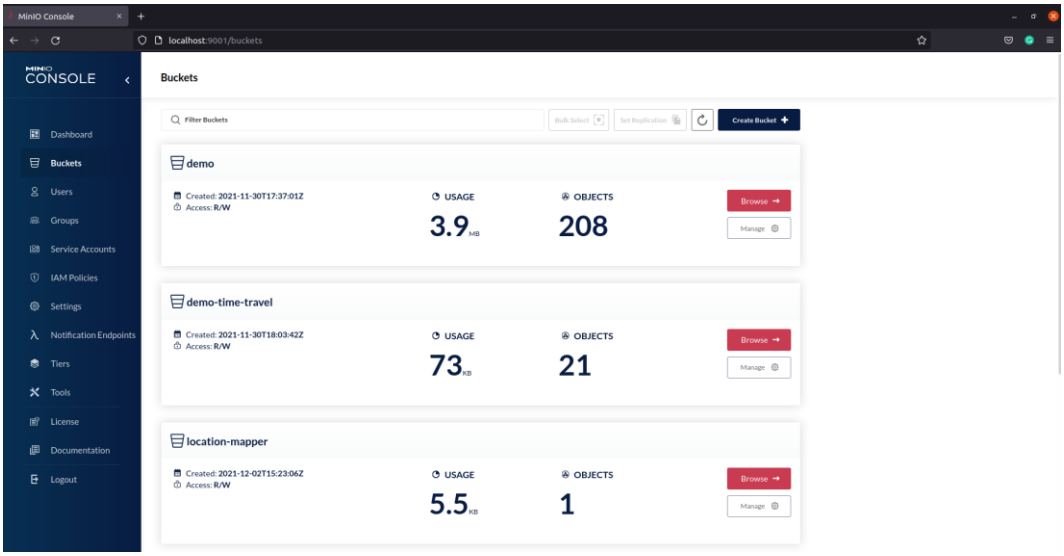
There are several methods to deploy Minio [9], though, an author figured that there are two simplest methods for deploying for proof of concept purpose which is deploying Minio using docker or helm chart.

Deploying Minio via Docker [10]:

```
docker run \
  -p 9000:9000 \
  -p 9001:9001 \
  --name minio1 \
  -v /Users/kraichamnivikaipong/Documents/temp_minio:/data \
  -e "MINIO_ROOT_USER=kraikrai" \
  -e "MINIO_ROOT_PASSWORD=kraikrai" \
  quay.io/minio/minio server /data --console-address ":9001"
```







note: please ensure that docker disk space is larger than 4Gb as it is a default setting for deploying Minio.

This is the quick simplest method to deploy standalone(single-node) Minio and get exposure to the tools. It is the best way to verify if the tool is able to integrate with other tools without a need of configuring any network by the user him/herself as well.

Deploying Minio via Helm Chart [11] on Kubernetes:

- Minikube (single-node standalone)

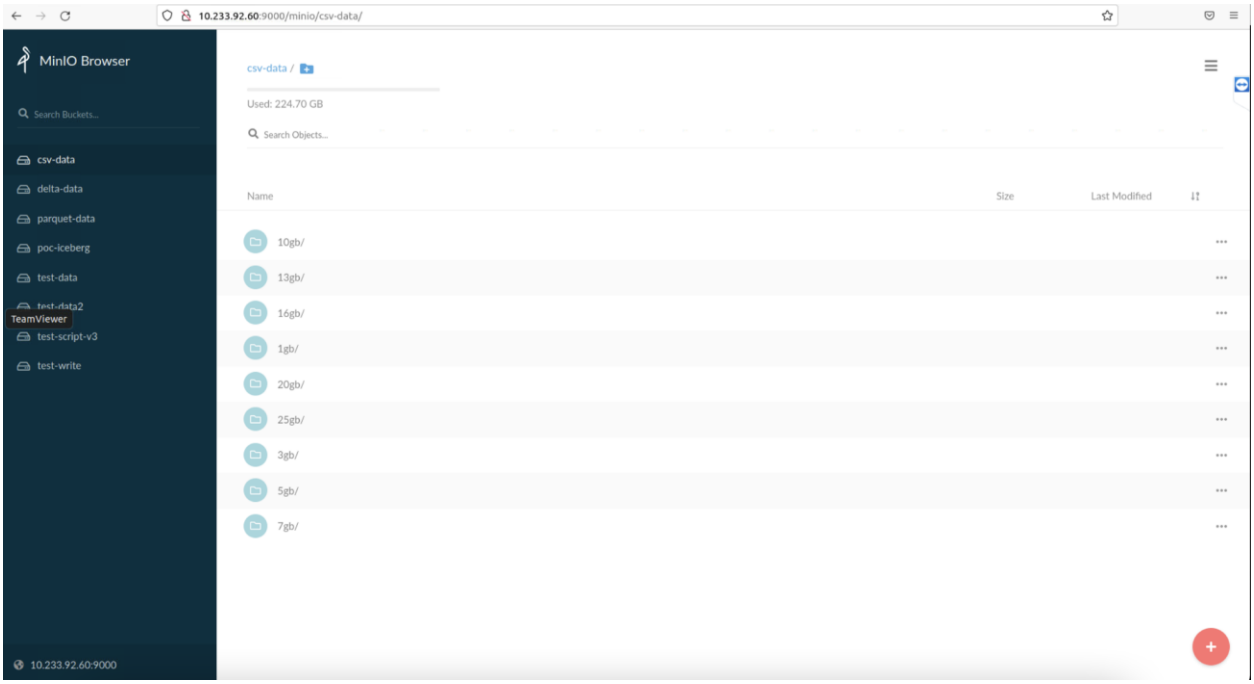
```
helm install minio-single-node --set accessKey=minio,secretKey=minio123 minio/minio
```

note: for this case, docker(Minikube) will handle all the persistence volume-related issues.

- Kubernetes Cluster (distributed)

```
helm install krai-distributed-minio --set
mode=distributed,accessKey=minio,secretKey=minio123,persistence.storageClass=openebs-
hostpath,service.type=NodePort,persistence.enabled=true,replicas=4 minio/minio
```

note: for simplicity in deploying the Minio on the Kubernetes cluster,OpenEBS, Local PV Hostpath [12] is recommended as a persistence volume provider for this case.



These are the simplest methods for deploying both standalone(single-node) and distributed Minio on the Kubernetes, and all the important configurations for proof of concept purpose are also presented as parameters in the command as well. Plus, it is painless to update and remove the deployed Minio with Helm Chart (except for the case that the user already set up Minio credential for Prometheus for a monitoring purpose).

Feasibility Result of Minio

After a successful deployment using several methods, the following features have been tried and proved that Minio is working without any concern:

- insert/update/delete/get the bucket
- uploading/removing the file through GUI
- write/read/update/delete file (single-node/distributed) through a python script
- set up credentials for monitoring with Prometheus

Hence, the feasibility of Minio can be confirmed that it **should be able to be used as storage** for a planned data lakehouse design.

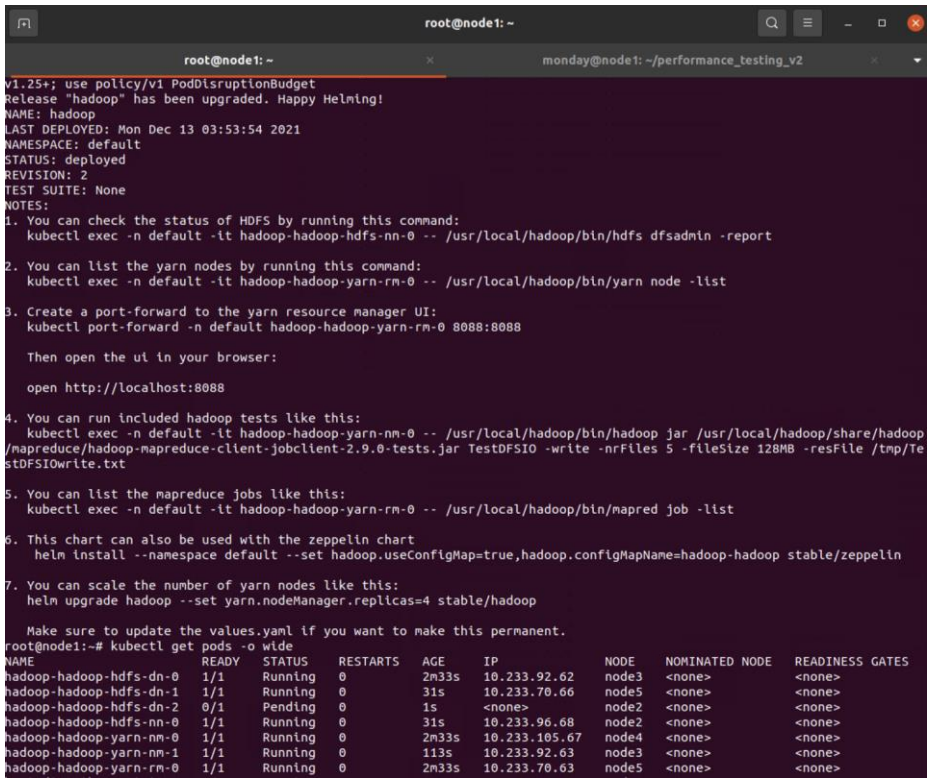
HDFS

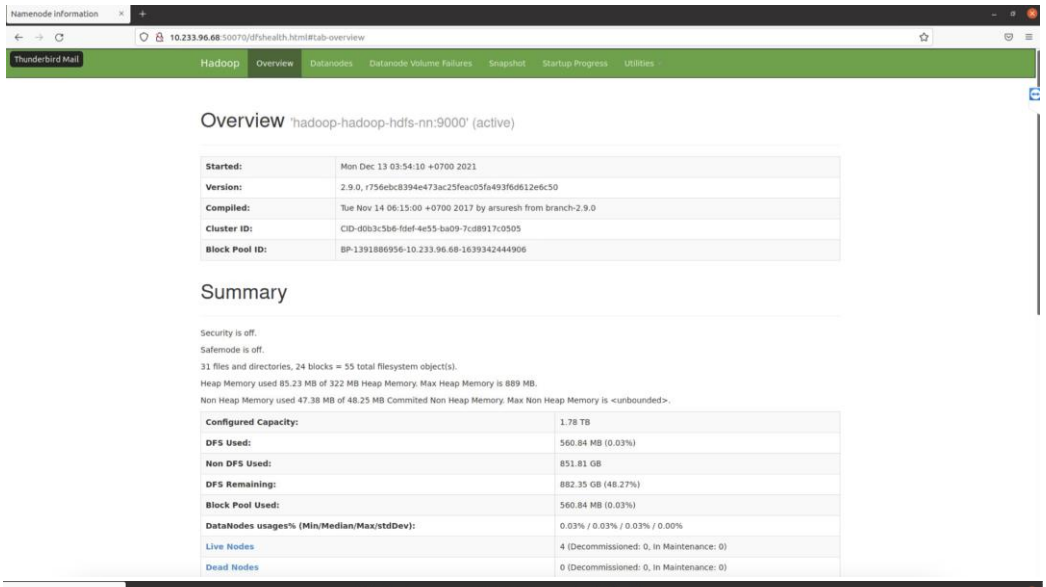
Setting Up HDFS

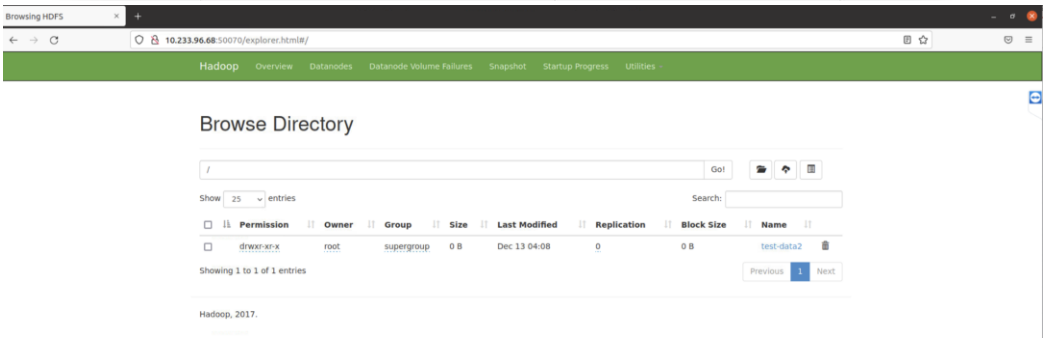
Unfortunately, due to the limited timeframe, an author didn't get much exposure to the HDFS. Nonetheless, a quick simple deployment method on the Kubernetes for the proof of concept purpose that is selected by the author is via Helm Chart deployment as the followings:

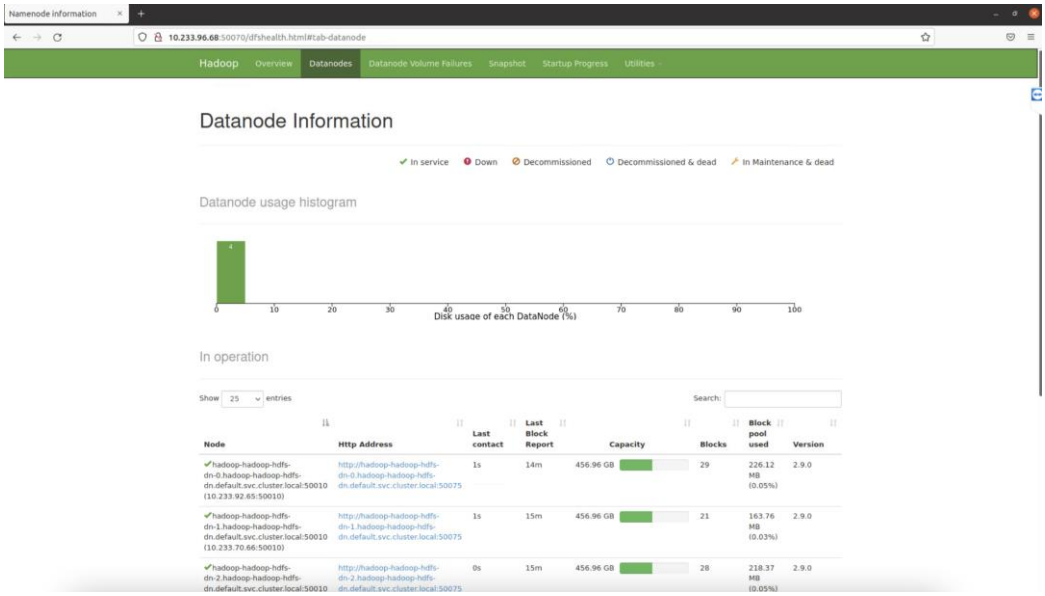
```
helm install hadoop --set persistence.nameNode.enabled=true --set
persistence.nameNode.storageClass=openebs-hostpath --set
persistence.dataNode.enabled=true --set
persistence.dataNode.storageClass=openebs-hostpath stable/hadoop
```

```
helm upgrade hadoop --set hdfs.dataNode.replicas=4 stable/hadoop
```


root@node1: ~
root@node1: ~
monday@node1: ~/performance_testing_v2
v1.25+; use policy/v1 PodDisruptionBudget
Release "hadoop" has been upgraded. Happy Helming!
NAME: hadoop
LAST DEPLOYED: Mon Dec 13 03:53:54 2021
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
1. You can check the status of HDFS by running this command:
kubectl exec -n default -it hadoop-hadoop-hdfs-nn-0 -- /usr/local/hadoop/bin/hdfs dfsadmin -report
2. You can list the yarn nodes by running this command:
kubectl exec -n default -it hadoop-hadoop-yarn-rm-0 -- /usr/local/hadoop/bin/yarn node -list
3. Create a port-forward to the yarn resource manager UI:
kubectl port-forward -n default hadoop-hadoop-yarn-rm-0 8088:8088
Then open the ui in your browser:
open http://localhost:8088
4. You can run included hadoop tests like this:
kubectl exec -n default -it hadoop-hadoop-yarn-nm-0 -- /usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.9.0-tests.jar TestDFSIO -write -nrFiles 5 -fileSize 128MB -resFile /tmp/TestDFSIOwrite.txt
5. You can list the mapreduce jobs like this:
kubectl exec -n default -it hadoop-hadoop-yarn-rm-0 -- /usr/local/hadoop/bin/mapred job -list
6. This chart can also be used with the zeppelin chart
helm install --namespace default --set hadoop.useConfigMap=true,hadoop.configMapName=hadoop-hadoop stable/zeppelin
7. You can scale the number of yarn nodes like this:
helm upgrade hadoop --set yarn.nodeManager.replicas=4 stable/hadoop
Make sure to update the values.yaml if you want to make this permanent.
root@node1:~# kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hadoop-hadoop-hdfs-dn-0 1/1 Running 0 2m33s 10.233.92.62 node3 <none> <none>
hadoop-hadoop-hdfs-dn-1 1/1 Running 0 31s 10.233.70.66 node5 <none> <none>
hadoop-hadoop-hdfs-dn-2 0/1 Pending 0 1s <none> node2 <none> <none>
hadoop-hadoop-hdfs-nn-0 1/1 Running 0 31s 10.233.96.68 node2 <none> <none>
hadoop-hadoop-yarn-nn-0 1/1 Running 0 2m33s 10.233.105.67 node4 <none> <none>
hadoop-hadoop-yarn-nn-1 1/1 Running 0 113s 10.233.92.63 node3 <none> <none>
hadoop-hadoop-yarn-rm-0 1/1 Running 0 2m33s 10.233.70.63 node5 <none> <none>


Namenode information
10.233.96.68:50070/dfshealth.html#tab-overview
Thunderbird Mail
Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Overview 'hadoop-hadoop-hdfs-nn:9000' (active)
Started: Mon Dec 13 03:54:10 +0700 2021
Version: 2.9.0, r756ebc8394e473ac25feac05fa493f6d612e6c50
Compiled: Tue Nov 14 06:15:00 +0700 2017 by arsuresh from branch-2.9.0
Cluster ID: CID-d0b3c5b6-fdef-4e55-ba09-7cd8917c0505
Block Pool ID: BP-1391886956-10.233.96.68-1639342444906
Summary
Security is off.
Safemode is off.
31 files and directories, 24 blocks = 55 total filesystem object(s).
Heap Memory used 85.23 MB of 322 MB Heap Memory. Max Heap Memory is 889 MB.
Non Heap Memory used 47.38 MB of 48.25 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.
Configured Capacity: 1.78 TB
DFS Used: 560.84 MB (0.03%)
Non DFS Used: 851.81 GB
DFS Remaining: 882.35 GB (48.27%)
Block Pool Used: 560.84 MB (0.03%)
DataNodes usages% (Min/Median/Max/stdDev): 0.03% / 0.03% / 0.03% / 0.00%
Live Nodes 4 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes 0 (Decommissioned: 0, In Maintenance: 0)


Browsing HDFS
10.233.96.68:50070/explorer.html#/
Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Browse Directory
/
Go!
Show 25 entries
Search:
Permission
Owner
Group
Size
Last Modified
Replication
Block Size
Name
drwxr-xr-x
root
supergroup
0 B
Dec 13 04:08
0
0 B
test-data2
Showing 1 to 1 of 1 entries
Previous
1
Next
Hadoop, 2017.



note: in order to control the later experiment the author is using [OpenEBS Local PV] for this deployment as well.

Feasibility Result of HDFS

After a successful deployment, the following features have been tested and proved that Minio is working without any concern:

- insert/update/delete/get the directory
- uploading/removing the file through GUI
- write/read/update/delete file (single-node/distributed) through a python script

A concern regarding authorization has been raised by an author, as it requires a proper authorization configuration to edit the data (which in this case the author is using ‘root’). Though, the feasibility of HDFS can be confirmed that it **should be able to be used as storage** for a planned data lakehouse design.

Apache Spark

Setting Up Apache Spark

It is presented that Apache Spark is available for popular operating systems such as Windows, macOS, and Linux. Thus, an author had an opportunity to try setting up Spark environment on all of the operating systems. Knowledge has been learned that it is highly recommended for all user who has less experience in the computer system and Apache Spark to set the Spark environment on Linux, as configuring classpath can be more or less troublesome.

However, the details on setting up the Spark environment on Linux (Ubuntu 20.04) will be provided in the later section, on performance evaluation.

Feasibility Result of Apache Spark

After successful setting up the Spark environment, the following features have been tried and proved that Spark on the host machine is working without any concern:

- spark-submit / PySpark
- insert/update/delete/get from the data source (local directory / local databases)
- machine-learning library by Spark
- other spark data transforming features

Hence, the feasibility of Spark can be confirmed that it **can be used as a computational tool** in a planned data lakehouse design.

Delta Lake

Setting Up Delta Lake for Apache Spark

As an author is only focusing on Spark as a computational tool, hence, the feasibility evaluation for the Delta Lake will only be based on configuring it so that it can be used by Spark. The configuration steps are simple and straightforward as presented in its document [13]. Please keep in mind to install the delta-spark [14] library for PySpark users. An example code on configuration Delta Lake with PySpark will be presented in the next section, on performance evaluation.

Feasibility Result of Delta Lake for Apache Spark

The following features have been tested and proved that Apache Iceberg is working without any concern:

- Create Delta Table
- Read/Write/Append Data Table
- Get History Metadata Table
- Time traveling

Hence, the feasibility of Delta Lake can be confirmed that it **can be used as a data writing format tool** for a planned data lakehouse design.

Apache Hudi

Setting Up Apache Hudi for Apache Spark

As an author is only focusing on Spark as a computational tool, hence, the feasibility evaluation for the Apache Hudi will only be based on configuring it so that it can be used by Spark.

An author has tried setting up a combination Apache Hudi version 0.9.0 and 0.8.0 against Apache Spark version 2.4.8, 3.0.3, and 3.1.2 given the quick start examples provided by Apache Hudi [7]. In addition, an author also tried a Scala and SparkSQL based examples as well.

Feasibility Result of Apache Hudi for Apache Spark

Unfortunately for Apache Hudi (0.9.0), despite several attempts to follow the quick start guide for all the possible languages (Scala, Python, SparkSQL) with a different version of Spark (2.48, 3.03, 3.12), none of the combinations is worked. It presented to the author that Apache is trying to call the java (Spark) class that doesn't exist. However, in search for the given class, it only presented a single reference with a similar issue [15] which has been solved by pulling a new configuration released or downgrading the Spark version (which doesn't work for the author).

However, facing complexity in both setup configurations and determining the writing formats in the code, not working examples from documents, and little information from the community it is decided that **Apache Hudi is not feasible as a writing format for a planned data lakehouse design**.

Apache Iceberg

Setting Up Apache Iceberg for Apache Spark

As an author is only focusing on Spark as a computational tool, hence, the feasibility evaluation for the Apache Iceberg will only be based on configuring it so that it can be used by Spark. For Apache Iceberg, an extra .jar file is required to be downloaded and stored in the jars directory. An example of configuration steps will be discussed in detail in the next section, on performance evaluation.

Feasibility Result of Apache Iceberg for Apache Spark

The following features have been tested and proved that Apache Iceberg is working without any concern:

- Create Iceberg Table
- Insert/Select/Update/Delete Data from Iceberg Table
- Get Snapshots Metadata Table
- Get History Metadata Table
- Get Files Metadata Table
- Get Manifest Metadata Table

Hence, the feasibility of Apache Iceberg can be confirmed that it **can be used as a data writing format tool** for a planned data lakehouse design.

2. Integration Evaluation

As the feasibility of all the tools has been evaluated, the next step is to combine the selected tools together and ensure that all of the combinations are able to work together properly. It is an important step to verify if there are any complexities during integration or further workaround required toward the solutions.

Tools Combination

The followings are all the combinations of the provided tools in a prior step and the working examples of a configuration for each combination will be presented below:

- Spark (PySpark) + CSV
- Spark (PySpark) + HDFS
- Spark (PySpark) + Minio
- Spark (PySpark) + Delta Lake
- Spark (PySpark) + Delta Lake + Minio
- Spark (PySpark) + Delta Lake + HDFS
- Spark (PySpark) + Apache Iceberg
- Spark (PySpark) + Apache Iceberg + Minio
- Spark (PySpark) + Apache Iceberg + HDFS

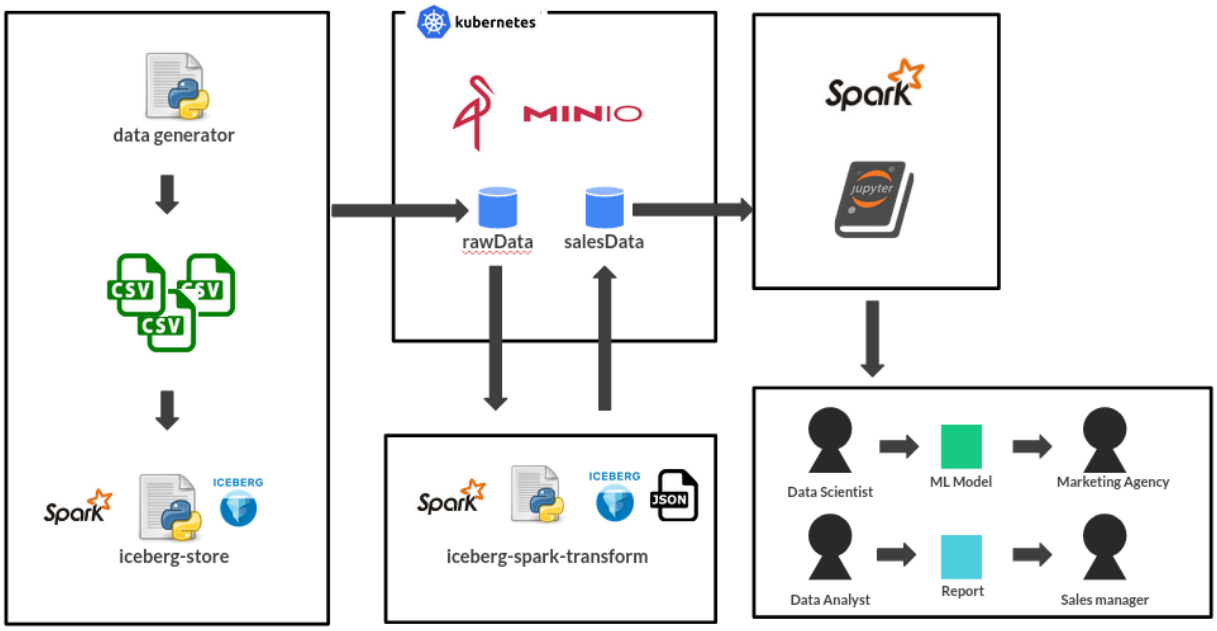
note1: for setting up the Spark environment, please refer to the later section, on performance evaluation.
note2: for detailed codes and use-cases please visit the GitHub repository

In addition, a concern has been raised regarding the combination of PySpark, Apache Iceberg, and HDFS. HDFS is needed to run as root for writing the data and it requires a workaround configuration to direct to the jar files.

Example Use-Case

Furthermore, the author also developed an example of analytical and machine learning use-cases in order to gain better exposure to the tools and their real-world use-case as well. The following use-cases will be escalated further for measuring the performances of the Spark features in handling large sizes of data in the future research.

However, the following diagram is the brief design of the example use-cases.



An example use-case consists of 6 parts that are created to imitate an actual scenario of the data pipeline in daily life situations. A table is created to store the data, the has been generated, the raw data has been https://github.com/CMKL-University/TB-Enterprise-Arch/blob/main/example-use-case-pyspark-iceberg-minio/e-verify_committed_information.pystored in a raw database, the raw data has been transformed and stored in a specific-use database, and the transformed data has been used for a specific task. The scripts can be sequentially executed using main.py as imitating the running data pipeline. The followings are the detail of each script's tasks:

- a-create_iceberg_table.py: delete an existing iceberg table in the Minio and create new tables for rawData and salesData.

- b-data_generator.py: generating the mock sales data for each month and storing them in the local directory in a CSV format.

- c-store_raw_chunk.py: store raw sales data into the rawData iceberg table, this script is duplicated into 4 files in order to imitate the real-life scenario of storing the sales data for each quarter of the year. It also helps the iceberg to keep a record of the different timestamps for a time-traveling feature as well.

- d-transform_raw_store_salesData.py: It filters, transforms, and mapped the raw sales data from the rawData table then stores the transformed data into the salesData table.

- e-verify_committed_information.py: It verifies if all the operation runs successfully and presented the metadata of the updated timestamp on each iceberg table.

note: for the implementation detail please visit the following GitHub repository.

3. Performances Evaluation

Create a Test Environment

It is important to ensure that the performance results are in a controlled environment. Hence, the following are the setup for the performance testing.

Hardware Specification

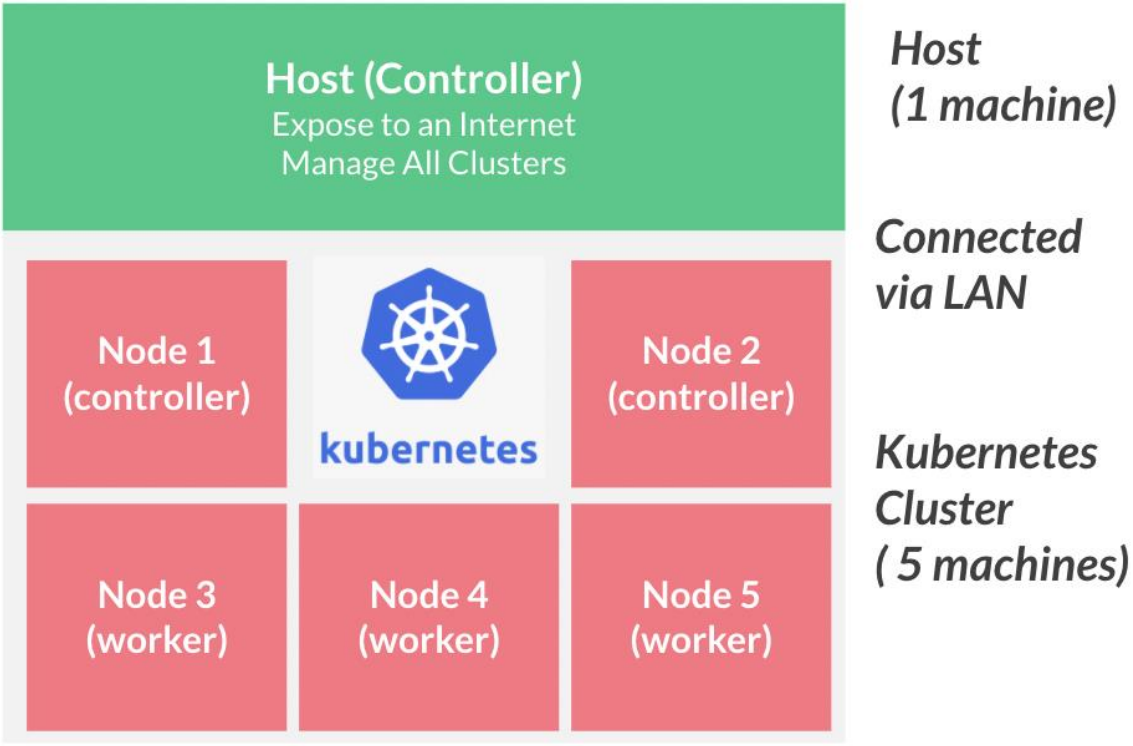
To summarize, there are 6 machines connected together with a LAN network with the following specifications:

- CPU - Intel(R) Core(TM) i5-7500T CPU @ 2.70 GHz
- RAM - 16Gib

For a detailed hardware specification, please follow this link.

Deploying Kubernetes Cluster

A Kubernetes cluster is designed to deploy into the provided machines as the following design:




```
root@node1:~# kubectl get nodes -o wide
NAME    STATUS   ROLES                  AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
node1   Ready    control-plane,master   24d   v1.22.3   192.168.0.2   <none>        Ubuntu 20.04.3 LTS   5.11.0-41-generic   docker://20.10.9
node2   Ready    control-plane,master   24d   v1.22.3   192.168.0.3   <none>        Ubuntu 20.04.3 LTS   5.11.0-41-generic   docker://20.10.9
node3   Ready    <none>                 24d   v1.22.3   192.168.0.4   <none>        Ubuntu 20.04.3 LTS   5.11.0-41-generic   docker://20.10.9
node4   Ready    <none>                 24d   v1.22.3   192.168.0.5   <none>        Ubuntu 20.04.3 LTS   5.11.0-41-generic   docker://20.10.9
node5   Ready    <none>                 24d   v1.22.3   192.168.0.6   <none>        Ubuntu 20.04.3 LTS   5.11.0-41-generic   docker://20.10.9
root@node1:~#
```

All of the machines are installed with Linux (Ubuntu 20.04), an author just realized that the server. The followings are the reference on how to install Linux (Ubuntu 20.04) on the machine:

- how to install Linux [17]

There is an alternative method to install Linux on the machine as well, it is known as Metal As A Service (MAAS) [18] and it will be explored in the future research.

Kubespray has been used as a tool for the Kubernetes cluster deployment. For the Kubernetes cluster deployment (by Kubespray), the host machine must be able to access (through ssh) all of the remaining nodes as a root without a password required for a monitoring and controlling purpose. The followings are the extra references for preparing the machine before deploying the Kubernetes cluster:

- debug and configuring ssh [19]
- enable root login [20] / allowing ssh root login [21] / set root password [22]
- configuring static IP address on ubuntu [23] / GUI version [24]

The followings are the references for the Kubernetes cluster deployment:

- quick start Kubespary [25]
- video explanation (older Kubespray version) [26]

As the Kubernetes cluster is deployed, the next step is to deploy the data storage for performance evaluation. Please refer to the feasibility evaluation for Minio and HDFS for the deployment steps.

Computational Environment Setup

Python

By default, Linux (Ubuntu 20.04) is installed with a default Python (version 3.8x) and pip (or pip3) is not installed. Hence, the following steps are to:

- Verifying if Python3 existed in the environment:

```
python --version or python3 --version
```

- Installing pip

```
apt install python3-pip python3-dev
```

As it is confirmed that Python3 and pip are installed, the followings are the libraries that are required for the performance evaluation steps which can be installed using the following commands:

- pip install pyspark
- pip install delta-spark
- pip install minio
- pip install matplotlib
- pip install pandas
- pip install numpy
- pip install snakebite-py3

note1: depends on the case, it might be pip3

note2: the author is considering creating a proper requirement.txt for the following libraries in the future research.

Spark

A Spark 3.1.2/hadoop 3.2 is being used for this performance evaluation. The followings are the steps to set up Spark environment on Linux (Ubuntu 20.04):
note: inspired by the following instructions [27]

- downloading Spark

```
apt-get install wget
wget https://dlcdn.apache.org/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz
```

- exracting the Spark zip file and place it to /usr/local/

```
tar xfz spark-3.1.2-bin-hadoop3.2 tgz -C /usr/local/ (or cd /usr/local)
```

- create Spark directory

```
tar xfz spark-3.1.2-bin-hadoop3.2 tgz -C /usr/local/ (or cd /usr/local)
```

- installing java

```
apt install curl mlocate default-jdk   -y

# note: by default JAVA_HOME should be automatically configured
```

- Configure SPARK_HOME

```
spark-submit (should not output anything)
apt-get install vim
vim ~/.bashrc
# Edit mode (press 'i')
# Add  Line "export SPARK_HOME=/usr/local/spark"
# Add Line "export PATH=$PATH:$SPARK_HOME/bin"
# Add Line "export JAVA_HOME=/usr/lib/jvm/java-user-version
Save the changes (esc >> :wq)
source ~/.bashrc
spark-submit (to verify the output)
```

As it is confirmed that the Spark environment is set up. The followings are the extra .jar files that are required for integrating certain tools with a Spark. Thus, it is important to ensure that all the required .jar files are downloaded and placed in the Spark's jars directory.

It can be downloaded using the following commands:

```
# move jar file to Spark's jars directory
sudo mv [ files downloaded directory ] [ filename.jar ]   #cd /usr/local/spark/jars
```

PySpark Writing CSV on S3 Storage:

```
wget https://repo.hortonworks.com/content/repositories/releases/org/apache/spark/spark-hadoop-cloud_2.11/2.4.5.7.2.7.0-87/spark-hadoop-cloud_2.11-2.4.5.7.2.7.0-87.jar
```

Minio:

```
wget https://repo1.maven.org/maven2/org/apache/hadoop/hadoop-aws/3.2.0/hadoop-aws-3.2.0.jar

wget https://repo1.maven.org/maven2/com/amazonaws/aws-java-sdk-bundle/1.11.860/aws-java-sdk-bundle-1.11.860.jar
```

Apache Iceberg :

```
wget https://search.maven.org/remotecontent?filepath=org/apache/iceberg/iceberg-spark3-runtime/0.12.1/iceberg-spark3-runtime-0.12.1.jar
```

Extra Configurations

Git / GitHub

Installing Git

```
apt-get install git
git --version
```

Generating Access Token

- docs [28]

Storing Credentials

```
git config --global credential.helper store
```

Jupyter

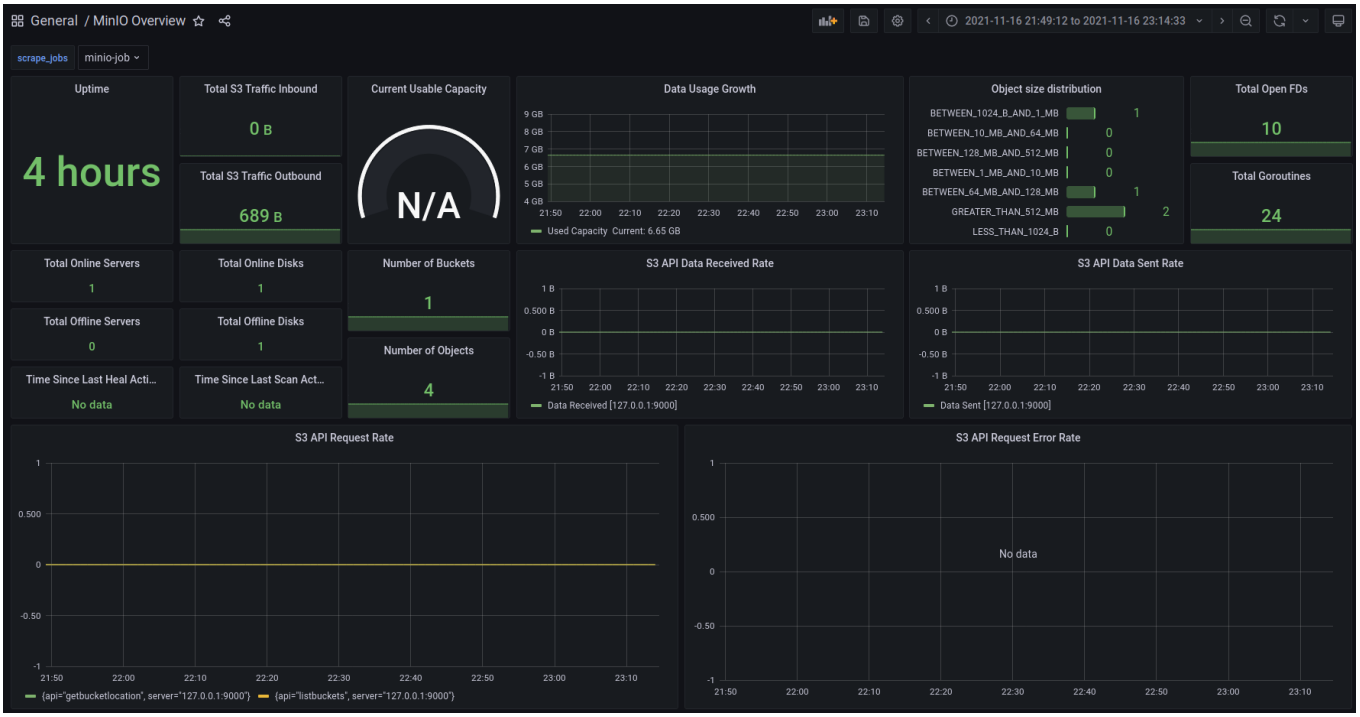
An author finds that Jupyter Notebook is one of the best tools for doing visualizing, data analytics, and machine learning use-cases. However, occasionally, it doesn't update the Spark configuration or computation as well and required the user to restart the Jupyter Notebook session. The following commands are being used for installing and starting a Jupyter Notebook:

```
pip install jupyter
cd [ user's work directory ]
jupyter notebook
```

For an example from an author, please visit the GitHub repository.

Monitoring Minio using Prometheus and Grafana

Minio performances can be monitored, visualized, and exported by integrating it with Prometheus and Grafana. The setup itself is quite simple and straightforward, however, please make sure to remove the Prometheus credentials before uninstalling Minio so that an unexpected ghost credential bug (randomly appearing and unable to get it) won't exist in the Kubernetes cluster.



The followings are the steps to set up Prometheus [29] and Grafana for monitoring Minio:

- (Using helm [11]) configure the parameter metrics.serviceMonitor.enabled to be true during the Minio installation (or using the upgrade command)
- Download Prometheus [30]
- Download mc [31] (minio cli)
- ./mc alias set <alias-name> <endpoint> <access-key> <access-secret>
- ./mc admin prometheus generate <alias-name>
- Get scrape_config, replace it with scrape_config in prometheus.yml
- ./prometheus --config.file=prometheus.yml
- Visit - http://localhost:9090
- Download Grafana here [32] (or here [33])
- sudo systemctl start grafana-server || sudo systemctl status grafana-server
- Visit - http://localhost:3000
- Config > datasource > prometheus > localhost:9090 (Error: Bad Gateway [34])
- Select '+' > import > use json from here [32] (raw [32])

Generate a Necessary Testing Data (Loads)

The data generator script has been created for flexible use in generating loads in different sizes in order to observe the performances of the Spark integrating with other tools against the different sizes of data. The generated data is 12 columns stored in a CSV format as it is presented example below:

As the main memory size of each node is 16 GiB, there is also the assumption that data with a larger size than the main memory might be able to break a Spark program or slow down its operation [35] tremendously as the main memory is occupied.

Given the prior statements, it is decided that the test load would consist of the data with different sizes and some of the data's size in the load would be larger than the main memory size. Throughout the experiment, it also confirmed that converting the CSV file into other writing formats (such as parquet, delta, or iceberg) will also reduce the size of the file by approximately 10 times.

Hence, after trying to generate the data with a size based on a logarithmic model to reduce the running test time, the followings are the sizes of the data in the load:

CSV size for writing formats:

```
data_size =
['1gb','2gb','3gb','4gb','5gb','7gb','10gb','13gb','16gb',
'20gb','25gb','30gb','40gb','50gb']
```

CSV size for writing CSV:

```
data_size =
['1gb','3gb','5gb','7gb','10gb','13gb','16gb','20gb','25gb']
```

However, through this experiment, the throughput (bandwidth) will be measured in the unit of rows per second where 10752 rows are equal to 1Mb.

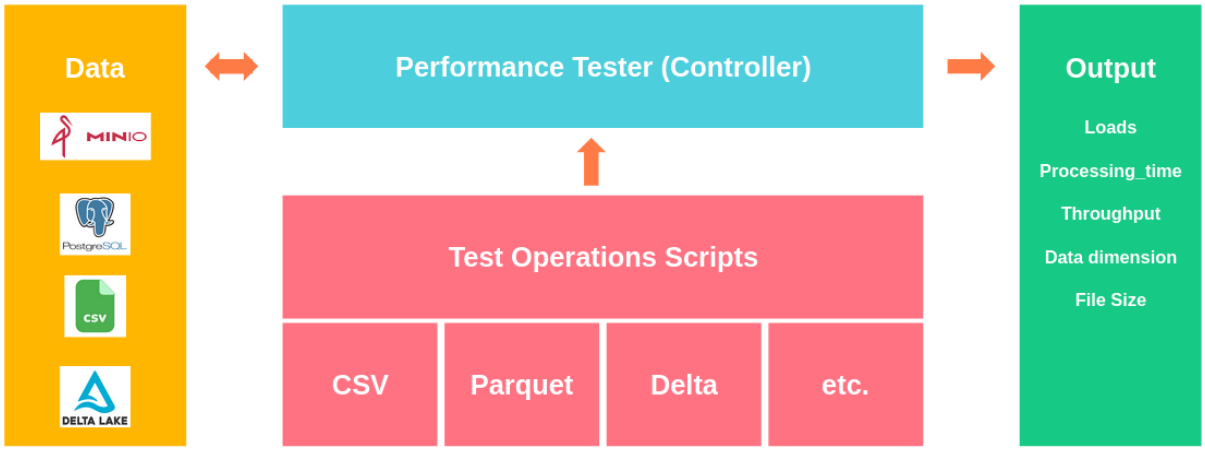
Create a Testing Script

As most of the existing benchmark for the Spark is too old and the usable one has its complexity in configuring the environment, plus there are many more tools and their integration to test which might not actually satisfy the requirement. Therefore, it has been decided by an author to create his own performance evaluation system which has been improved incrementally according to experience and insight that the author has gained through each iteration.

1st attempt:

It is inspired by CODAIT/spark-bench [36] where the user can configure the operation, loads, and others through the self-custom template configuration file and simply run the benchmark and get a number.

Hence, the original design of the performance evaluation system is based on the Spark benchmark from the understanding of the author and it is presented in the following diagram below:



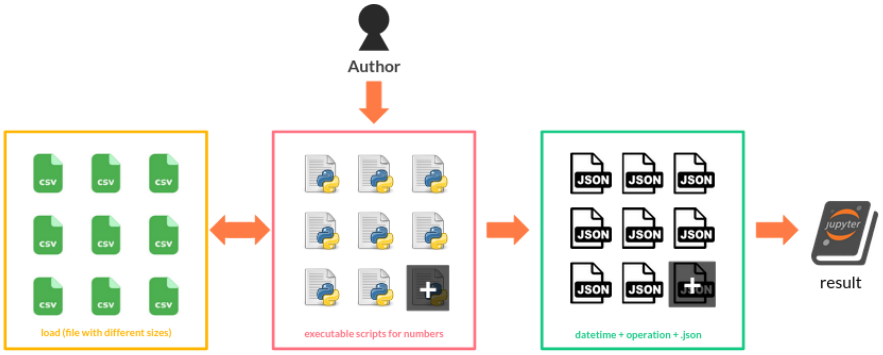
2nd attempt:

An author learned that the 1st attempt code is not flexible enough and there are many static codes that can be improved or turned it to be dynamic variables.

3rd attempt:

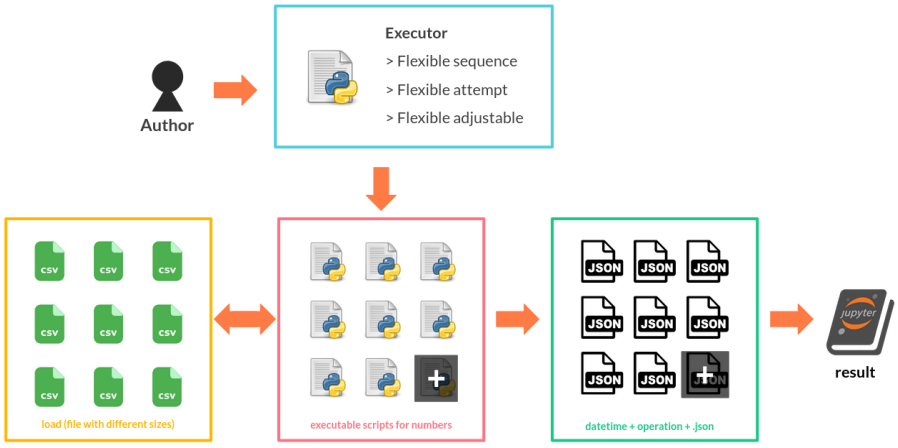
Facing the bug in the data storage connection, an author realized that the controller requires a lot of changes as a workaround for the test operations which aren't integrated with a Spark to work.

Hence, the controller script is abandoned, the performance evaluation is run by each test operation instead as shown on the following diagram:



4th attempt

given that the test operation scripts worked properly in the prior attempts, each of the test operation scripts has been improved to output its own result. All of the test operation scripts are being executed sequentially by an executor script and the attempts (or repetition) can be controlled by both executor script and test operation scripts. A new design is presented in the diagram below.



note: so far, the result is saved in JSON format

4.5th attempt:

As there is an unexpected incident on electrical shortage, it crashes HDFS and one of the Minio's nodes. An incident escalated into the situation where cluster redeployment is with a new version of Kubespray is required.

The lesson has been learned, together with a tight time frame, an author then realized that:

1. The result should be recorded in every successfully executed operation not as a batch when the entire load (different size of data) is done.

2. It would be better for the test operation scripts to write, do an operation with a written file (e.g., count), and delete the file in a single script. It would have a higher tendency to overcome the cache and Spark's optimization algorithm, as well as lessen a maintenance effort.

3. The JSON format is also hard to read/write given that the direction of the data is still unclear and keeps adapting as the experiment progressed.

4. Writing the test operations as an object (class) is also quite complicated to maintain. There will always be a little detail that always made a huge impact on the script and required every file to be fixed and tested.

Therefore, a new prototype script is designed and developed to answer the mentioned issues. It combines the write, count, and delete operations in one script. When each operation is finished, data will be recorded as one line in the text file with the following format:

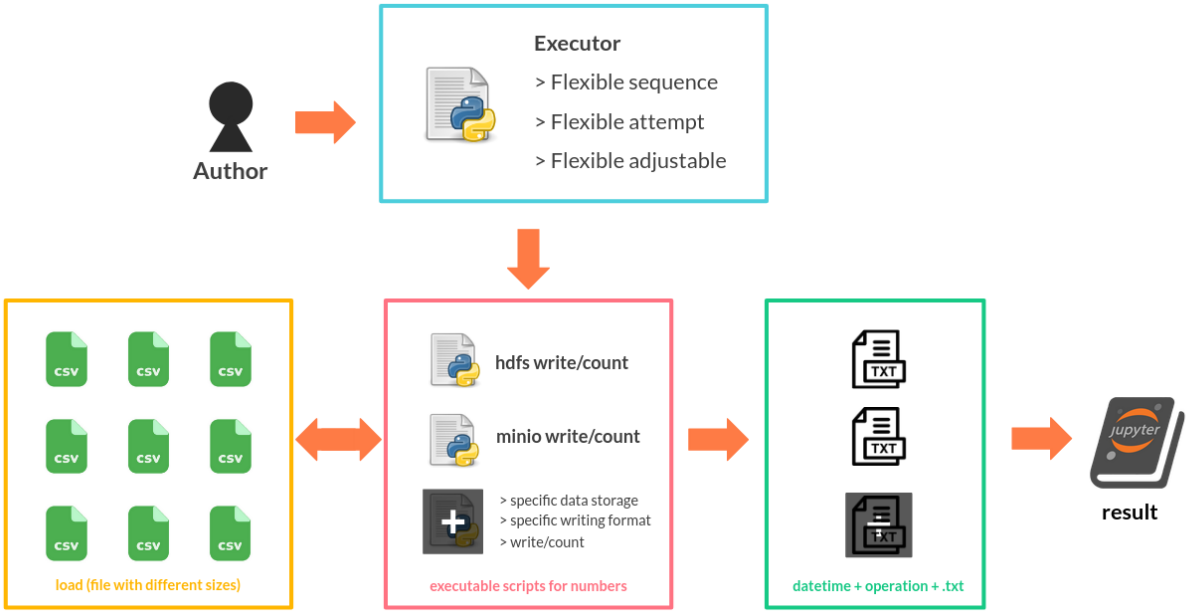
```
operation_name, timestamp, elapsed_time, rows, columns, throughput

Example:
writeMiniocsv,12-12-2021--09-49-41,34.09,11010048,12,322949.79766
countMiniocsv,12-12-2021--09-51-27,10.51,11010048,12,632631.56321
writeMiniocsv,12-12-2021--09-49-41,98.75,22020096,12,542125.62341
countMiniocsv,12-12-2021--09-51-27,32.13,22020096,12,914141.56432
```

5th attempt:

Apparently, the previous script is working pretty well. However, there is always room for improvement and detail polishment. As the previous version is dumping all the data into a single plain text file, it needs to be separated. it is confusing during the information extraction process. The naming convention and new test operations also need to be added.

Hence, the current design of the performance evaluation system is presented in the following diagram:



Future attempt:

The test script will be more convenient if the test operation scripts can be more dynamic. The test operation script should be enabled to receive the configuration parameters from the executor script or configuration file.

Result Analysis

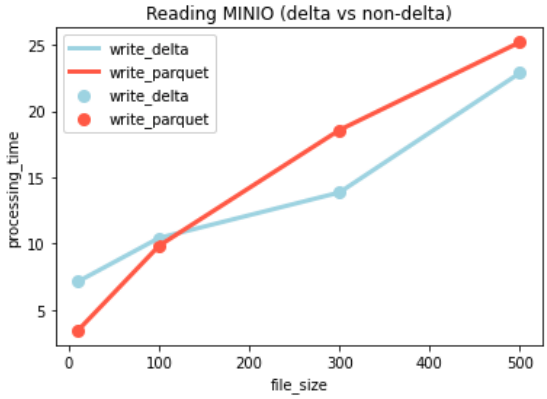
Result Progression:

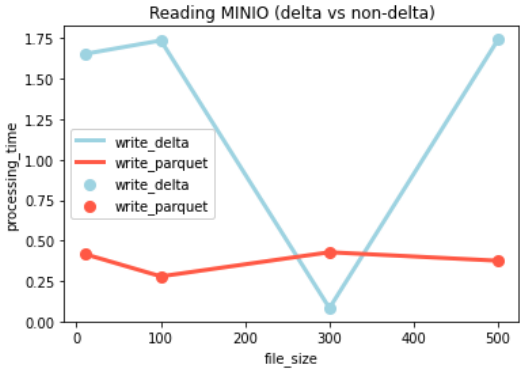
As the performance evaluation system has been improved almost weekly, an author would like to take this advantage to present the progress in an experiment and its result using the attempts of the performance evaluation system improvement instead.

> **Important**: Please note that the performance is evaluated by the **processing time (elapsed time)** against the number of rows in the data and **throughput (bandwidth)** is measured as **rows per second.**

The followings are the progression of the results from each attempt as the performance evaluation system is being improved:

1st & 2nd attempt:





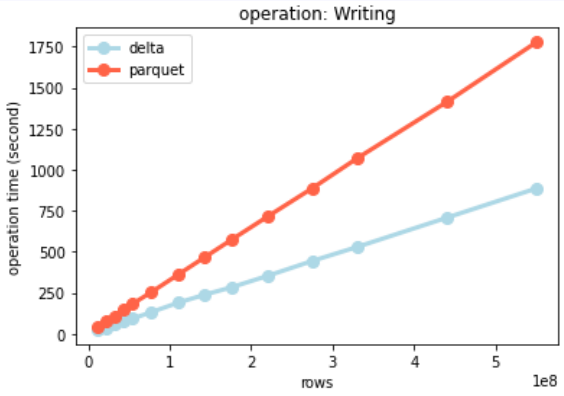
In the 1st & 2nd attempt of the performance evaluation system, the data sizes in the load are 10Mb, 100Mb, 300Mb, and 500Mb. It presented that the load isn't large enough to see the converge trend nor hitting the limitation of the tools for almost every performance evaluation question. Though, for writing data frame in a delta format, the trend is presented that **the performance of writing data frame in a delta format tends to be better than parquet**, which is the baseline for file writing format. However, the read operation result doesn't make sense and it is assumed that the size of the load isn't large enough and the file is being cached.

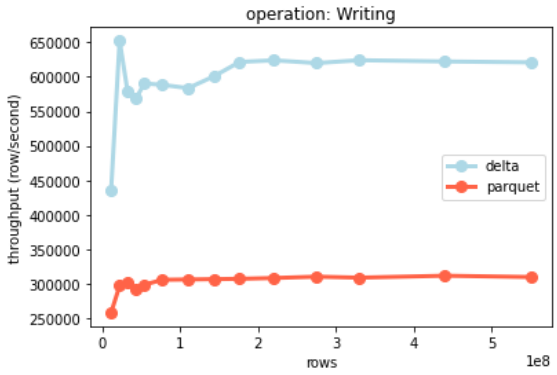
In addition, in these attempts, the test has been conducted with a local machine on a single-node Minio, not a Kubernetes cluster.

3nd attempt:

The data size has been increased to the following:

```
data_size = ['1gb','2gb','3gb','4gb','5gb','7gb','10gb','13gb','16gb',
'20gb','25gb','30gb','40gb','50gb']
```





It is confirmed that delta performs better than parquet by approximately 2 times in a distributed Minio with 4 replicas.

Through the experiment, an author also learned that the read operation doesn't actually go through the data in the data frame but more of establishing the connection for the directory or going through the metadata of the file instead. Thus, count operation is being used for evaluating the performance of the reading operation.

4th attempt:

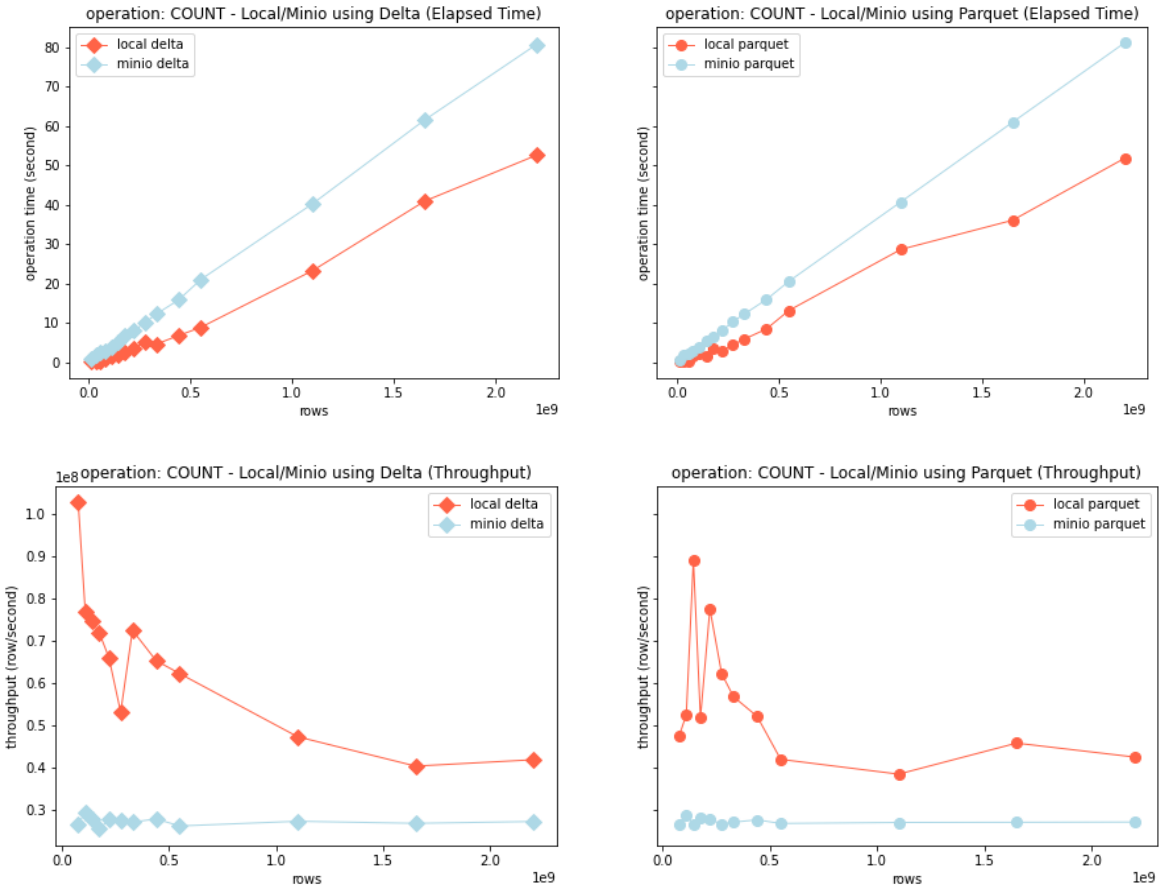
In the 4th attempt, not only a count operation has been replacing the read operation. The question regarding the performance between Minio and local storage had also been asked, as local storage might be used as a based-line for the future result.

Hence, increasing the load from the prior attempt to:

```
data_size = ["1gb","3gb","5gb","7gb","10gb","13gb","16gb","20gb",
"25gb","30gb","40gb","50gb","100gb","150gb","200gb"]
```

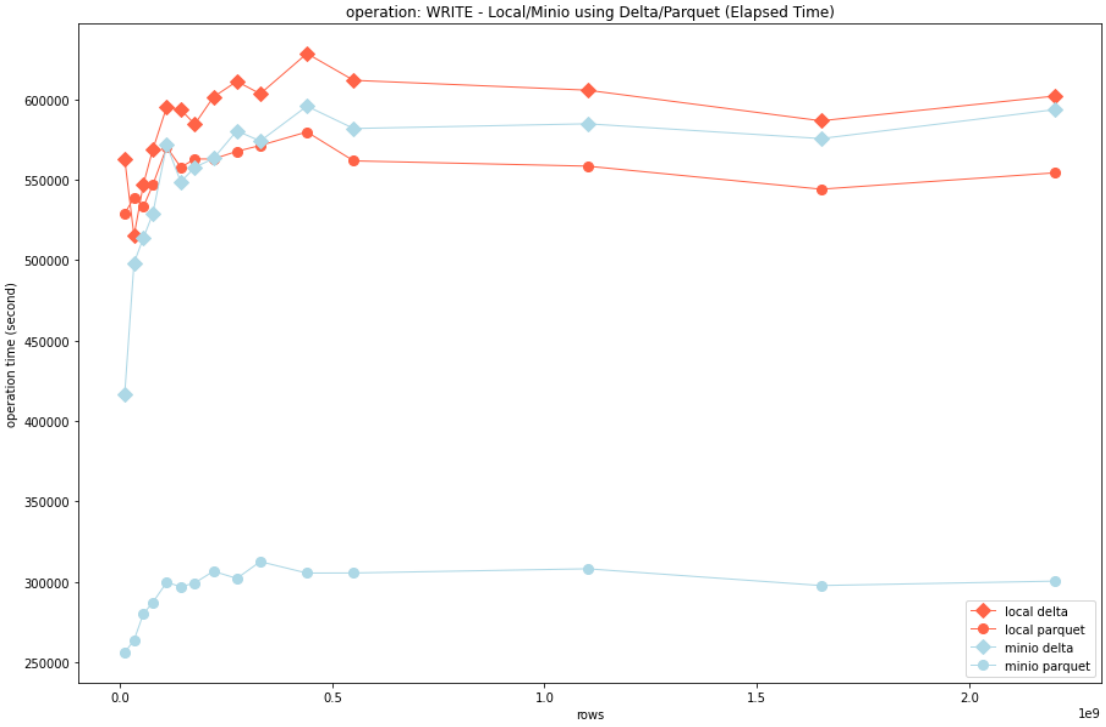
the followings are the result of the 4th attempt of the performance evaluation system.

Counting Operation: local storage against Minio using both Delta and Parquet format



The presented result shows that local storage performs better than a Minio approximately 2 times on the converge trend (last 4 points) despite the fluctuation in performance on the small size of data.

Writing Operation: local storage against Minio using both Delta and Parquet format



Surprisingly, writing the data frame in a delta format on Minio performs better than writing the data frame in a parquet format on the local machine by approximately 1.1 times and slower than writing the data frame in a delta format on a local machine by approximately 0.9 times as well.

However, writing the data frame in a parquet on Minio isn't performing well. Compared to the remaining operations, it performs only half (50%) of the speed of others. Though, it is acceptable considering that Minio has 4 replicas and continuously backing up along the process.

Given the following output, HDFS finally got attention to be tested in terms of performance against the Minio. Also, an iceberg writing format that finally is configurable.

note: the presented trend has been ensured by running the performance evaluation system by more than a single attempt as well.

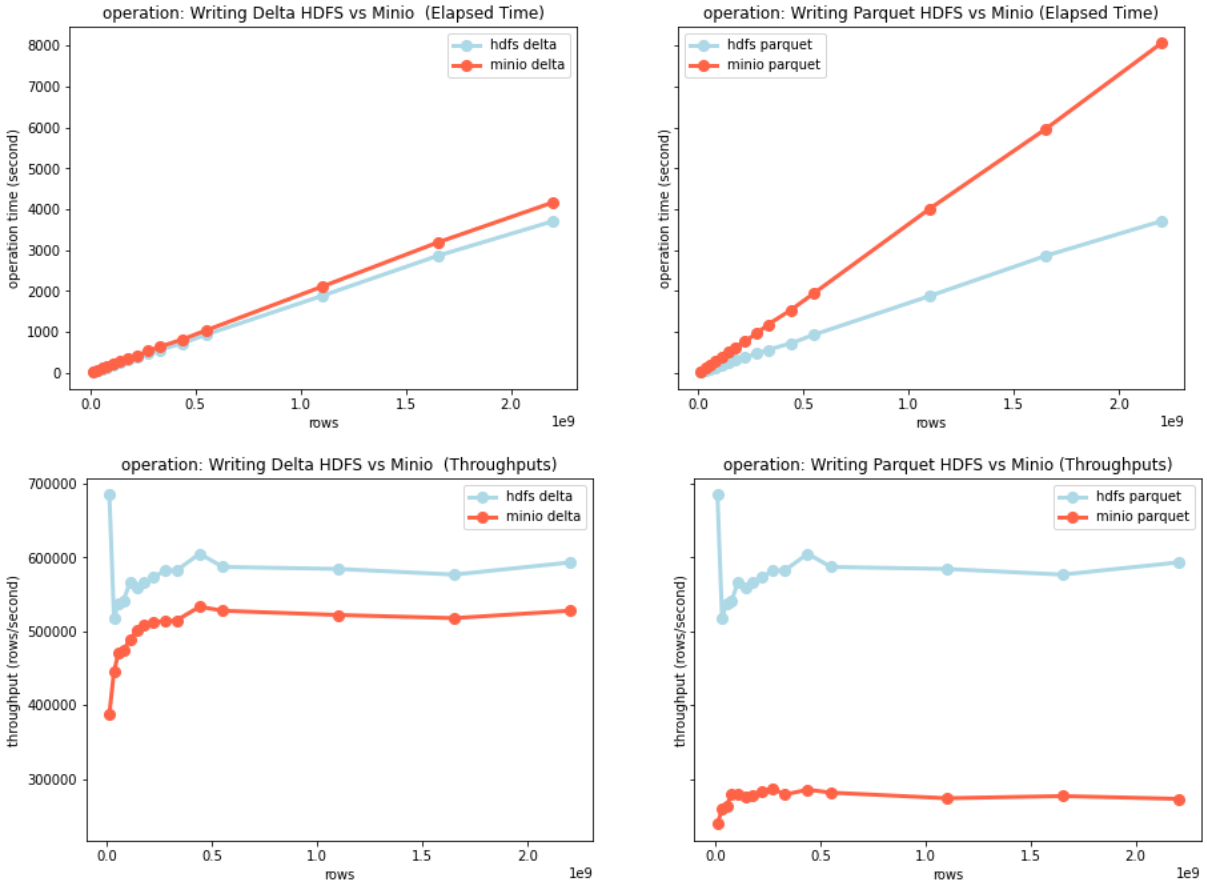
4.5th attempt:

Storage Comparison

Given the following size of load:

```
data_size =
['1gb','3gb','5gb','7gb','10gb','13gb','16gb','20gb','25gb']
```

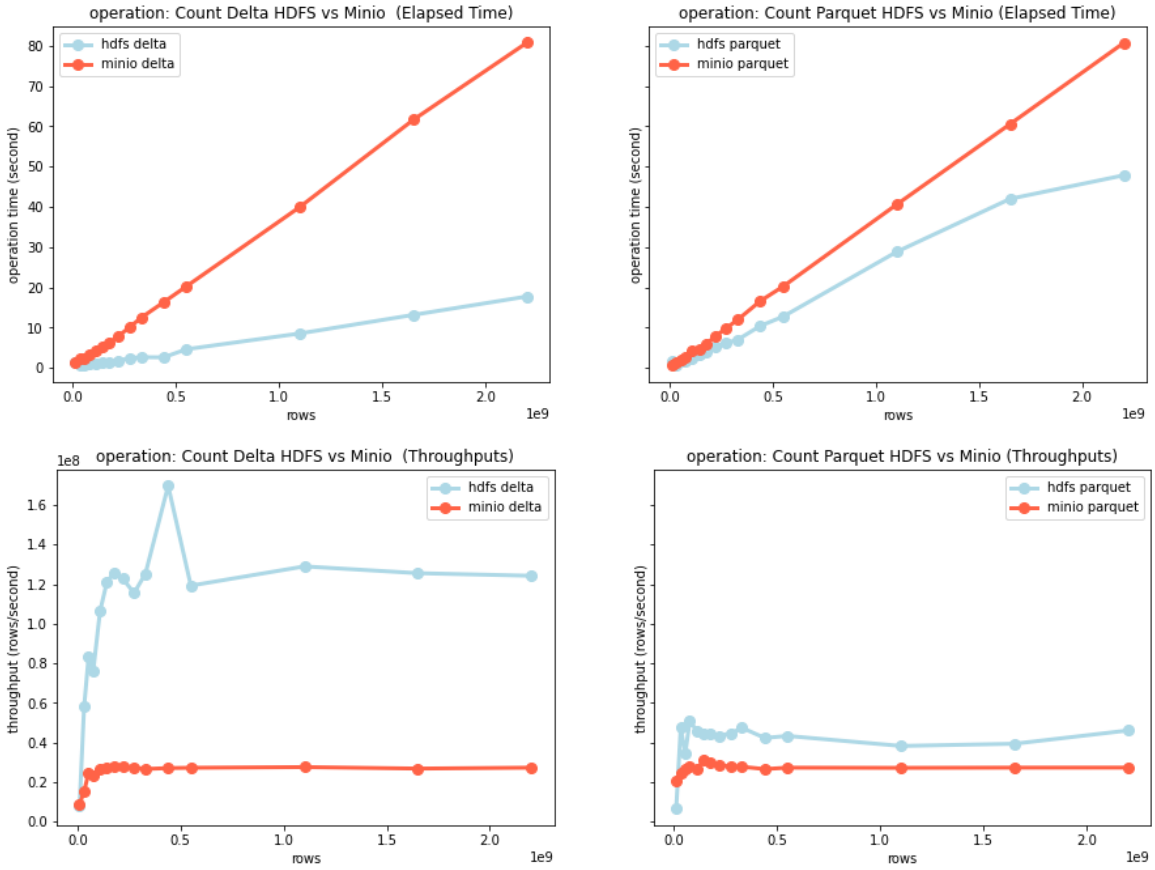
Writing Operation: HDFS vs Minio using Parquet and Delta formats



- HDFS performs **writing** operation better than Minio
- HDFS writing **parquet** performs **better** than **delta**

By writing the data frame in a delta format on both HDFS and Minio, the performance of HDFS is slightly better by approximately 1.15 times. However, there is a significant difference in writing the data in a parquet format for Minio, the performance is dropped a lot and slower than HDFS by more than 2 times.

Counting Operation: HDFS vs Minio using Parquet and Delta formats



- HDFS performs **count** operation better than Minio
- specifically count **delta** performed by **HDFS**

As for the count operation, it is shown that HDFS performs better than Minio for both writing formats. However, in counting operation a data frame in delta format HDFS performs better than Minio by approximately 6 times.

note: the presented trend has been ensured by running the performance evaluation system by more than a single attempt as well.

Summary:

After a long journey of trials and errors, finally, a direction, size of the load, and the numbers of the performance evaluation are clearer to an author. Hence, the results from the latest performance evaluation system (5th attempt) are being used for the following summary.

The result from the performance evaluation system is separated into 2 parts which are:

- **Performance of data storage:** using PySpark writing CSV files into different data storages which are Minio and HDFS, with a different number of replicas which are 2, 4, and 8 given local storage (ext4) as a baseline.

- **Performance of a writing format:** by using PySpark writing a data frame with different data formats, which are delta and iceberg, into a data storage (Minio) given parquet as a baseline

The followings are the result of the latest performance evaluation system with a focus on the given cases.

```
Important: Please note that the performance is evaluated by the
processing time (elapsed time) against the number of rows in the data
and throughput (bandwidth) is measured as rows per second.
```

Performance of Data Storage: Minio vs HDFS

The performance results for this case are a comparison of a throughput (row/second) between Minio and HDFS in terms of writing operation and counting operation against the load (different sizes of files) in a different number of replicas with 4 nodes for each storage.

After discussing with advisor, the result for 2 replicas and 8 replicas can't be used due to an author inexperience in controlling the variables for such a condition. Hence, it will be removed from this report. However, this following experiment will be improved and presented in the future research.

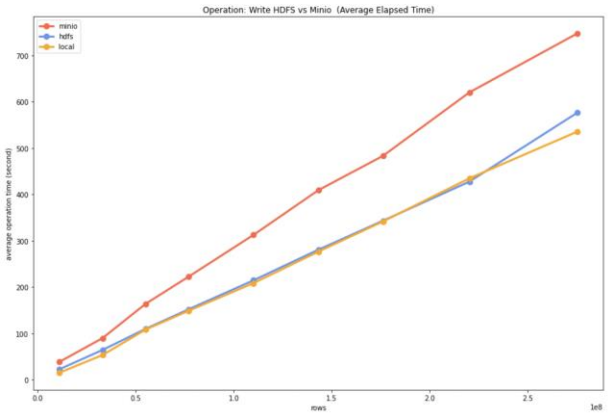
Given that there are multiple attempts for running the result and it has been proved that the approximate standard deviation of the result for each attempt is lesser than 5%. Hence, the average elapsed time of all the attempts is being used to present the information in the following result.

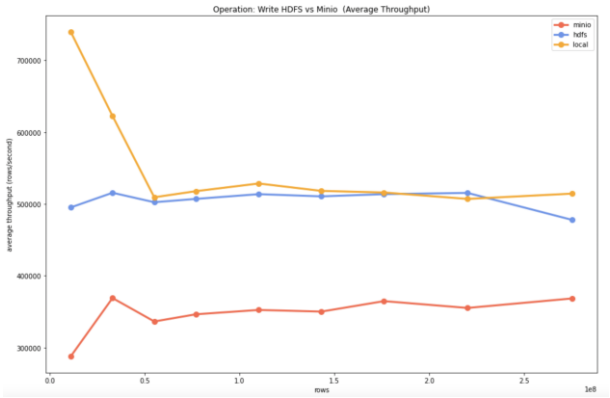
note: for the detail of the result data please visit the following GitHub repository
The following is the size of the load:

```
data_size =
['1gb','3gb','5gb','7gb','10gb','13gb','16gb','20gb','25gb']
```

4 Replicas

Writing Operation

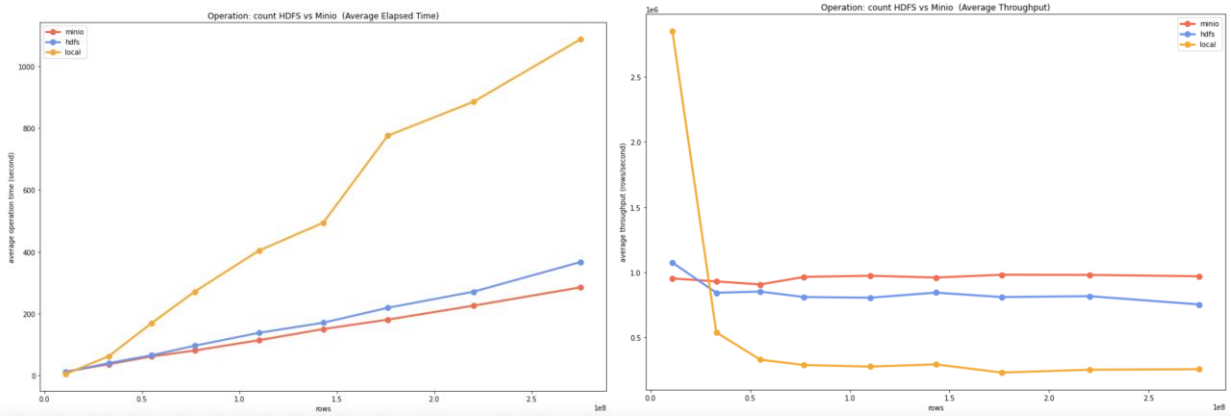




	rows	avg throughput minio(rows/second)	avg throughput hdfs(rows/second)	avg throughput local(rows/second)
0	11010048.0	287882.621623	495101.345426	739498.344830
1	33030144.0	369015.587354	515549.494736	622395.513176
2	55050240.0	336417.879474	502347.101801	509252.256339
3	77070336.0	346438.244876	507004.764821	517790.224280
4	110100480.0	352448.564394	513615.573511	528450.626237
5	143130624.0	350161.165198	510513.351359	518140.948831
6	176160768.0	364735.584899	513623.068010	515920.702122
7	220200960.0	355248.116106	515331.700102	506955.785746
8	275251200.0	368438.933056	477857.141103	514337.836214

According to the diagram (result), given 4 replicas, it presented that HDFS performs better than Minio in the writing operation, with a throughput of approximately 500,000 rows/second and 350,000 rows/second respectively.
HDFS performs approximately 1.43 times faster than Minio in the writing operation.

Counting Operation



	rows	avg throughput minio(rows/second)	avg throughput hdfs(rows/second)	avg throughput local(rows/second)
0	11010048.0	950742.558196	1.072933e+06	2.852727e+06
1	33030144.0	928026.449183	8.408020e+05	5.362684e+05
2	55050240.0	904768.147323	8.497848e+05	3.261368e+05
3	77070336.0	962539.698774	8.078793e+05	2.849804e+05
4	110100480.0	971535.568188	8.024930e+05	2.729443e+05
5	143130624.0	957651.375371	8.416996e+05	2.899251e+05
6	176160768.0	979146.830308	8.074867e+05	2.272613e+05
7	220200960.0	977810.010762	8.144007e+05	2.488007e+05
8	275251200.0	967548.912530	7.508466e+05	2.530047e+05

According to the diagram (result), given 4 replicas, it presented that Minio performs better than HDFS at counting operation, with a throughput of approximately 960,000 rows/second and 800,000 rows/second respectively.
Minio performs approximately 1.2 times faster than HDFS in the counting operation.
The data also presented that as the replicas increased, the throughput of both storages in both operations also increased.

Conclusion

The following tables are the throughput approximation for writing operation and counting operation against the different number of replicas:

Writing Operation

Data storage /replicas (approx. throughput)	4 replicas
Minio	350,000 rows/second
HDFS	500,000 rows/second

Counting Operation

Data storage /replicas (approx. throughput)	4 replicas
Minio	960,000 rows/second
HDFS	800,000 rows/second

By observing the result of both storage performing an operation on CSV using PySpark against a different number of replicas there are some assumptions that can be drawn from the given facts:

- There is a chance that the performance of HDFS will be dropped as the data size is increased, as it always presented a huge spike on the last set of data (25Gb CSV file of 2755251200 rows)

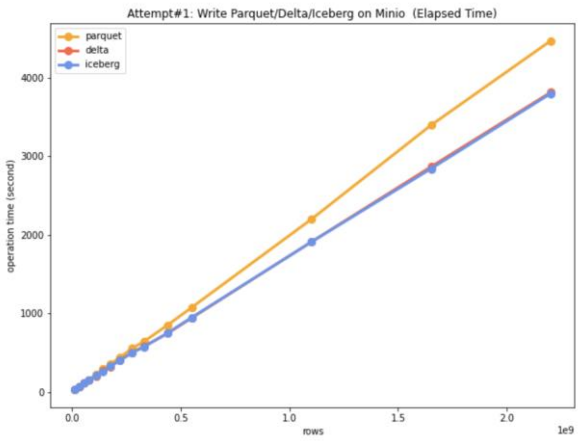
Performance of Writing Format:

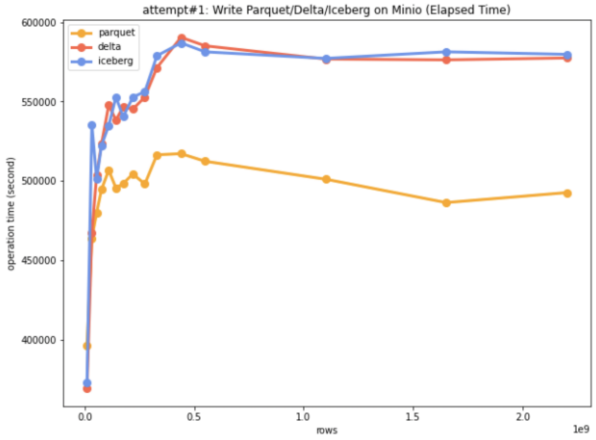
Delta vs Iceberg (on Minio with 4 replicas)

Given the following size of load:

```
data_size = ["1gb","3gb","5gb","7gb","10gb","13gb","16gb","20gb",
"25gb","30gb","40gb","50gb","100gb","150gb","200gb"]
```

Writing Operation

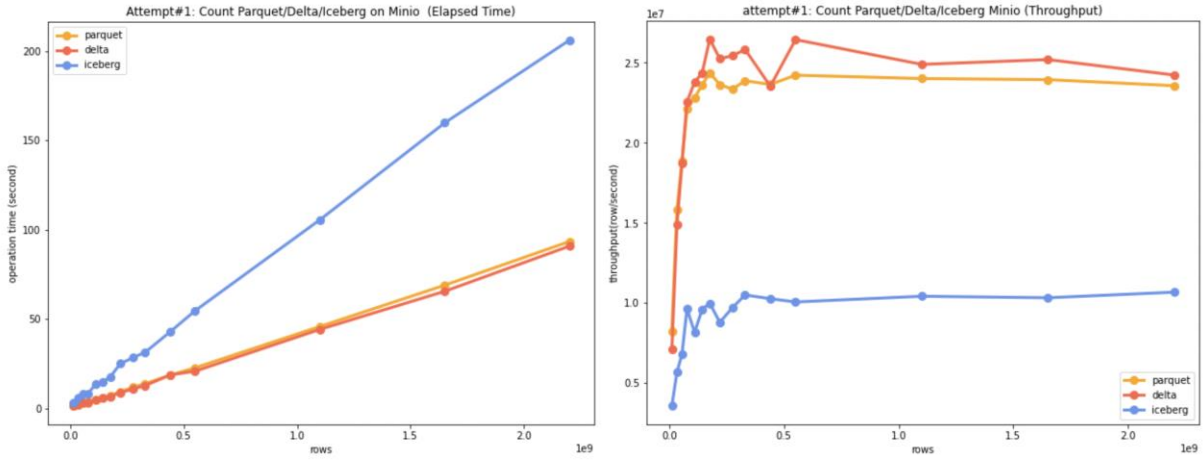




	rows	parquet(row/second)	delta(row/second)	iceberg(row/second)
0	11010048	396626.599005	369418.813185	373394.331937
1	33030144	463991.941186	467244.036250	535620.373512
2	55050240	480150.413624	503595.240568	501153.500000
3	77070336	494632.532701	523177.477996	522390.085856
4	110100480	506854.649775	547770.223126	534842.113398
5	143130624	495602.824537	538383.524404	552848.043866
6	176160768	498430.384844	546664.704418	540466.062831
7	220200960	504470.070185	545270.694336	552782.958662
8	275251200	498133.853698	552863.827405	556320.194558
9	330301440	516521.981989	571399.403621	578846.435911
10	440401920	517241.263878	590473.519359	586971.442308
11	550502400	512440.878047	585131.699785	581335.255314
12	1101004800	501160.483344	576807.029724	577200.524355
13	1651507200	486449.013538	576293.166285	581341.770438
14	2202009600	492717.144122	577469.769932	579766.656017

The performance between writing a data frame in iceberg and delta format is **very similar** which is approximately 580,000 rows/second with an **iceberg is seen to be slightly better as the data size increased**. However, both perform better than writing a data frame in parquet by approximately 1.2 times

Counting Operation



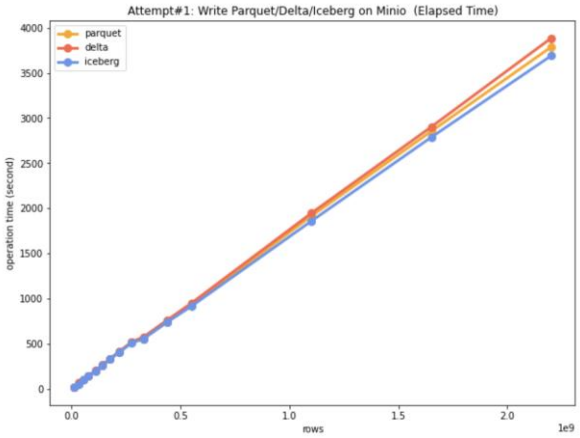
	rows	parquet(row/second)	delta(row/second)	iceberg(row/second)
0	11010048	8.226520e+06	7.109647e+06	3.584724e+06
1	33030144	1.581317e+07	1.488214e+07	5.685757e+06
2	55050240	1.884882e+07	1.872466e+07	6.810205e+06
3	77070336	2.212728e+07	2.256338e+07	9.652137e+06
4	110100480	2.281859e+07	2.378788e+07	8.169023e+06
5	143130624	2.361834e+07	2.437525e+07	9.617445e+06
6	176160768	2.437334e+07	2.647282e+07	9.989530e+06
7	220200960	2.364000e+07	2.525460e+07	8.801241e+06
8	275251200	2.335115e+07	2.543899e+07	9.716044e+06
9	330301440	2.388227e+07	2.583055e+07	1.050951e+07
10	440401920	2.364446e+07	2.353707e+07	1.027193e+07
11	550502400	2.422574e+07	2.644607e+07	1.007011e+07
12	1101004800	2.401472e+07	2.490096e+07	1.042697e+07
13	1651507200	2.394705e+07	2.520012e+07	1.033250e+07
14	2202009600	2.356950e+07	2.423735e+07	1.068124e+07

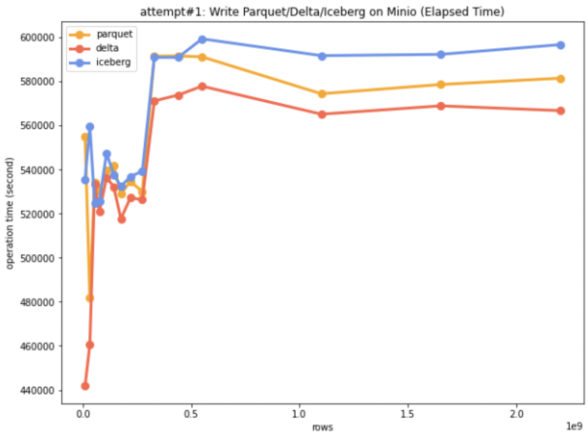
Interestingly, in the counting operation, while the performance of the parquet and delta formats are on par, given that delta is slightly better. The throughput of the delta on Minio is approximately 25,000,000 rows/second while the iceberg is approximately 10,000,000 rows/second.
Delta performs approximately 2.5 times better than the iceberg format.

note: the presented trend has been ensured by running the performance evaluation system by more than a single attempt as well.

Delta vs Iceberg (on HDFS with 4 replicas)

Writing Operation

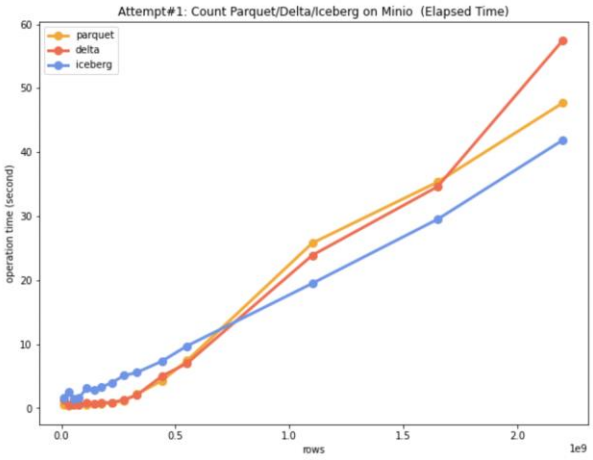


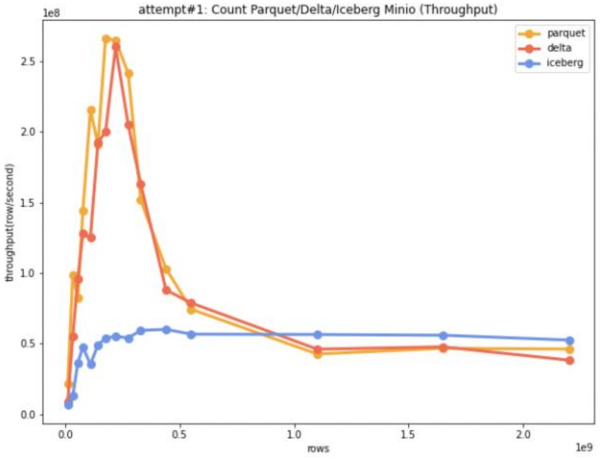


	rows	parquet(row/second)	delta(row/second)	iceberg(row/second)
0	11010048	555072.202936	441803.124402	535643.930204
1	33030144	481898.322737	460833.733501	559692.200541
2	55050240	534359.660151	533363.113218	524732.785352
3	77070336	530303.125511	521066.990808	525862.755751
4	110100480	539879.503132	536454.977904	547544.579223
5	143130624	541854.014845	532287.536864	537809.061782
6	176160768	529199.790633	517757.337182	532470.597263
7	220200960	534625.944528	527267.775473	536889.777357
8	275251200	530109.285967	526381.587580	539396.170019
9	330301440	591574.783983	571273.086028	591035.016108
10	440401920	591690.309825	573940.341704	591038.071909
11	550502400	591285.928146	577960.790637	599454.106050
12	1101004800	574512.669952	565249.794792	591831.809410
13	1651507200	578722.318910	569023.714475	592412.461266
14	2202009600	581575.088862	566868.293149	596826.719380

Apparently, from the data, iceberg performs **slightly better than delta** in writing operation on HDFS. Iceberg is able to write approximately 590,000 rows/second while delta is able to write approximately 560,000 rows/second.
The difference in performance is approximately 1.05 times.

Counting Operation





	rows	parquet(row/second)	delta(row/second)	iceberg(row/second)
0	11010048	2.140921e+07	8.791921e+06	6.629236e+06
1	33030144	9.868874e+07	5.493486e+07	1.291901e+07
2	55050240	8.266324e+07	9.588951e+07	3.638619e+07
3	77070336	1.438741e+08	1.280807e+08	4.761645e+07
4	110100480	2.152782e+08	1.252666e+08	3.529161e+07
5	143130624	1.910716e+08	1.925428e+08	4.913166e+07
6	176160768	2.662845e+08	1.999447e+08	5.379464e+07
7	220200960	2.646870e+08	2.601461e+08	5.520430e+07
8	275251200	2.413119e+08	2.049560e+08	5.404246e+07
9	330301440	1.515737e+08	1.627894e+08	5.939416e+07
10	440401920	1.030131e+08	8.799250e+07	6.007760e+07
11	550502400	7.429596e+07	7.879447e+07	5.670695e+07
12	1101004800	4.268335e+07	4.608301e+07	5.646893e+07
13	1651507200	4.673143e+07	4.770235e+07	5.594019e+07
14	2202009600	4.613887e+07	3.828051e+07	5.253231e+07

According to the presented data, despite the spike of the performance presented by counting operation on parquet and delta formats in the early stage, it shows that the iceberg performs way better as the data size increased to the point that exceeds the main memory. Using the last three stable points presented in the graph, the iceberg is counting on HDFS is approximately 55,000,000 rows/second while is delta is counting approximately 43,000,000 rows/second. Delta also showing a sign of a dropping in

performance as the data gets larger on the HDFS by observing the last point where data size is 200Gb (2202009600 rows).
The counting operation of the iceberg on HDFS is approximately 1.28 times better than the delta.

Conclusion

The following tables are the throughput approximation for writing operation and counting operation using a different writing format against the different data storage:

Writing Operation:

Writing Format / Data Storage (approx. throughput)	Minio	HDFS
Parquet	500,000 rows/second	580,000 rows/second
Delta Lake	580,000 rows/second	560,000 rows/second
Apache Iceberg	580,000 rows/second	590,000 rows/second

As a presented result, it seems like writing a data frame with either delta format or iceberg format doesn't make much difference in terms of the performance despite one of the formats might be slightly better than another. However, both of them still perform better than parquet (general format) for Minio, but only the iceberg is better than parquet on HDFS. It also presented a converge trend for the writing performance on the test environment against a large amount of data, which is approximately 580,000 rows/second.

Counting Operation:

Writing Format / Data Storage (approx. throughput)	Minio	HDFS
Parquet	24,000,000 rows/second	45,000,000 rows/second
Delta Lake	25,000,000 rows/second	43,000,000 rows/second
Apache Iceberg	10,000,000 rows/second	55,000,000 rows/second

According to the result, it presented that writing formats perform better at counting operation on HDFS than Minio. It also presented that the iceberg performs really well in HDFS. However, the delta format is up to the standard (parquet format) for both of the data storage despite a slight difference in performance where it performs better in Minio and worsens in HDFS.

Hence, in terms of stability in performance against the storage, it is clear that delta performs better. However, the iceberg is showing a strong sign of compatibility with HDFS.

Finding / Insight

Providing an end-to-end exploration on configuring and deploying evaluated tools, integrating them, developing the performance evaluation system for them, and observing their result. There are many points to be taken for selecting these tools and their combinations.

An author will be providing the comparison table which listed all the evaluation criteria that the author is aware of for the tools and insight based on the author's **experience of the author** regarding the tools.

Comparison Table: Data Storage

	Minio	HDFS
Performance	> Processing speed for input/output is quite stable > High availability > High fault tolerance > Writing speed with CSV: slower than HDFS > Writing speed with file format: expected to reach 580,000 rows/second with current test environment > Counting speed: depends on file formats, replicas, and the configurations > Counting speed: comparable to HDFS	> Processing speed for input/output is quite fluctuate > Low availability: missing data block > Low fault tolerance: easily crashed when electricity shortage > writing speed with CSV: faster than Minio > Writing speed with file format: expected to reach 580,000 rows/second with current test environment > Counting speed: show an increasing trend as the number of replicas increased
Configuration Complexity	> easy to medium	> medium to hard

	> there are several methods and simple examples for configuration > it is a homogeneous system (only Minio)	> still unable to find good simple examples for configuration > it is a heterogeneous system (consist of name node, data node, yarn, etc.)
Integration Complexity	> medium to hard > don't know if it is possible, many tools might still not support > need to look for workaround from Amazon S3	> easy to medium > knowing it's possible as HDFS is like default storage for big data > permission for editing the file is concerned
Learning Curve	> easy to medium > disregarding the integration part, the tool itself is quite straightforward to use > good documents > super user friendly	> medium to hard > given its heterogeneous system architecture there is more information to consume > given its long-time existence, there is more information to screen (old or up-to-date) > need to google on how to open and use the GUI
Community	> certain group of people is aware (or using) it	> huge, everyone in the field knows (or is aware) it > online course, lecture
Popular Existing Use-Cases	> for log aggregation > for object file storage	> for big data (e.g. datalake, dataware house)

Hence, for building the data lakehouse solution, a concern for the author will depend on the client's ability to support and maintain the storage against the client's ability to a workaround for the use-cases and enthusiasm for new technologies.

Comparison Table: Writing Format

Again, provided that Apache Hudi already failed its feasibility evaluation and parquet is just a universal file format to reduce the file size with no ACID property. Hence, an author will only be comparing Delta Lake and Apache Iceberg in the following table:

	Delta Lake	Apache Iceberg
performance	> small file size > extra features such as ACID and time-traveling > performs slightly slower than parquet in HDFS in the writing operation > performs better than parquet in Minio in the writing operation > performs better in Minio	> small file size > extra features such as ACID and time-traveling > performs slightly better than parquet in the writing operation > performs very well in HDFS in the counting operation > performs poorly in Minio, 2.5 times slower in the counting operation
Integration complexity	> Spark friendly, no extra jar file is required > support python, java, scala > good python library > streaming with Spark	> might require extra jar files depending on the use-case > despite several methods for configuring, before the author is able to get it works > support python, java > python library doesn't work for author > streaming with Flink
Learning curve	> good document and tutorials > lower learning curve	> acceptable document > low learning curve, but more complicated than delta
Community	> supported by Databrick	> supported by Netflix

Given the following information, the Apache iceberg is an interesting choice if the user decides to use HDFS as data storage given the performance number.

However, as there isn't much different in terms of complexity between using these 2 tools, Delta Lake also needed to be considered given its own pros. It is simpler, has competitive performance, and more stable in its performance against different types of data storage as well.

Conclusion

From the author's perspective, choosing data storage has already been a major concern in which there is no clear answer to which one is better depending on the use-case, actual loads, and ability of the clients. Thus, Delta Lake is a better tool for the given circumstance as it is simpler, allows more flexibility to the solutions, and Databrick's support team will always be there looking for the customers.

GitHub Repository

https://github.com/CMKL-University/TB-Enterprise-Arch

References

[1] Ben Lorica, Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, "What Is a Lakehouse?," databricks, 30 01 2020. [Online]. Available: https://databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html. [Accessed 15 12 2021].

[2] ยุทธการ อ่างมณีกุล , "what-is-data-lakehouse," sharebyheang, 03 02 2021. [Online]. Available: https://sharebyheang.com/what-is-data-lakehouse/. [Accessed 15 12 2021].

[3] Minio, "min.io," MINIO, [Online]. Available: https://min.io/. [Accessed 15 12 2021].

[4] Dhruba Borthakur, "hadoop.apache.org," Apache Software Foundation, 10 10 2020. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction. [Accessed 15 12 2021].

[5] Apache Spark, "spark.apache.org," Apache Spark, [Online]. Available: https://spark.apache.org/. [Accessed 15 12 2021].

[6] databricks, "docs.delta.io/latest/delta-intro," databricks, 2020. [Online]. Available: https://docs.delta.io/latest/delta-intro.html. [Accessed 15 12 2021].

[7] A. Hudi, "hudi.apache.org," Apache Software Foundation, [Online]. Available: https://hudi.apache.org/. [Accessed 15 12 2021].

[8] Apache Software Foundation., "iceberg.apache.org," Apache Software Foundation., [Online]. Available: https://iceberg.apache.org/. [Accessed 15 12 2021].

[9] MinIO, "docs.min.io/docs/minio-quickstart-guide," MinIO, 2021. [Online]. Available: https://docs.min.io/docs/minio-quickstart-guide.html. [Accessed 15 12 2021].

[10] MinIO, "docs.min.io/docs/minio-docker-quickstart-guide," MinIO, 2021. [Online]. Available: https://docs.min.io/docs/minio-docker-quickstart-guide.html. [Accessed 15 12 2021].

[11] harshavardhana, "github.com/minio/charts," GitHub, 27 11 2021. [Online]. Available: https://github.com/minio/charts. [Accessed 15 12 2021].

[12] OpenEBS, "docs.openebs.io/v250/docs/next/uglocalpv-hostpath," OpenEBS, [Online]. Available: https://docs.openebs.io/v250/docs/next/uglocalpv-hostpath.html. [Accessed 15 12 2021].

[13] databricks, "docs.delta.io/latest/quick-start," databricks, 2020. [Online]. Available: https://docs.delta.io/latest/quick-start.html. [Accessed 15 12 2021].

[14] databricks, "pypi.org/project/delta-spark/," databricks, pypi, 03 12 2021. [Online]. Available: https://pypi.org/project/delta-spark/. [Accessed 15 12 2021].

[15] melin, "github.com/apache/hudi/issues/3554," GitHub, 21 08 2021. [Online]. Available: https://github.com/apache/hudi/issues/3554. [Accessed 15 12 2021].

[16] [Online].

[17] hellotech, "hellotech.com/guide/for/how-to-install-linux-on-windows-10," hellotech, 31 08 2021. [Online]. Available: https://www.hellotech.com/guide/for/how-to-install-linux-on-windows-10. [Accessed 15 12 2021].

[18] Canonical Ltd., "maas.io/docs," Canonical Ltd., 2021. [Online]. Available: https://maas.io/docs. [Accessed 15 12 2021].

[19] K. Buzdar, "linuxhint.com/fix_connection_refused_ubuntu/," linuxhint, 2020. [Online]. Available: https://linuxhint.com/fix_connection_refused_ubuntu/. [Accessed 15 12 2021].

[20] redhat, "preparation_before_the_p2v_migration-enable_root_login_over_ssh," redhat, [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/v2v_guide/preparation_before_the_p2v_migration-enable_root_login_over_ssh. [Accessed 15 12 2021].

[21] L. Rendek, "linuxconfig.org/allow-ssh-root-login-on-ubuntu-20-04-focal-fossa-linux," linuxconfig, 18 04 2020. [Online]. Available: https://linuxconfig.org/allow-ssh-root-login-on-ubuntu-20-04-focal-fossa-linux. [Accessed 15 12 2021].

[22] T. Teri, "confluence.columbia.edu/confluence/display/~ast2163/Ubuntu+Linux+-+How+to+change+the+root+password," Atlassian Confluence, 12 03 2019. [Online]. Available: https://confluence.columbia.edu/confluence/display/~ast2163/Ubuntu+Linux+-+How+to+change+the+root+password. [Accessed 15 12 2021].

[23] P. Kumar, "linuxtechi.com/assign-static-ip-address-ubuntu-20-04-lts/," linuxtechi, 13 03 2020. [Online]. Available: https://www.linuxtechi.com/assign-static-ip-address-ubuntu-20-04-lts/. [Accessed 15 12 2021].

[24] L. Rendek, "linuxconfig.org/how-to-configure-static-ip-address-on-ubuntu-18-10-cosmic-cuttlefish-linux," linuxconfig, 12 11 2021. [Online]. Available: https://linuxconfig.org/how-to-configure-static-ip-address-on-ubuntu-18-10-cosmic-cuttlefish-linux. [Accessed 15 12 2021].

[25] kubespray, "kubespray.io," kubespray, [Online]. Available: https://kubespray.io/#/. [Accessed 15 12 2021].

[26] J. m. a. Opensource, "[Kube 65.1] Kubespray - Kubernetes cluster provisioning," youtube, 16 12 2019. [Online]. Available: https://www.youtube.com/watch?v=8Jh4yZQOVZU&t=921s. [Accessed 15 12 2021].

[27] B. Sheppard, "How to run Spark with Minio in Kubernetes," youtube, 09 10 2020. [Online]. Available: https://www.youtube.com/watch?v=ZzFdYm_DqEM&t=1207s. [Accessed 15 12 2021].

[28] GitHub, "authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token," GitHub, [Online]. Available: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token. [Accessed 15 12 2021].

[29] MinIO, "docs.min.io/docs/how-to-monitor-minio-using-prometheus," MinIO, [Online]. Available: https://docs.min.io/docs/how-to-monitor-minio-using-prometheus.html. [Accessed 15 12 2021].

[30] prometheus, "prometheus.io," prometheus, 2021. [Online]. Available: https://prometheus.io/. [Accessed 15 12 2021].

[31] MinIO, "docs.min.io/docs/minio-client-quickstart-guide," MinIO, [Online]. Available: https://docs.min.io/docs/minio-client-quickstart-guide. [Accessed 15 12 2021].

[32] sinhaashish, "github.com/minio/minio/blob/master/docs/metrics/prometheus/grafana/README.md," GitHub, 18 11 2021. [Online]. Available: https://github.com/minio/minio/blob/master/docs/metrics/prometheus/grafana/README.md. [Accessed 15 12 2021].

[33] Schkn, "devconnected.com/how-to-install-grafana-on-ubuntu-18-04/," devconnected, [Online]. Available: https://devconnected.com/how-to-install-grafana-on-ubuntu-18-04/. [Accessed 15 12 2021].

[34] aximo, "github.com/grafana/grafana/issues/14629," GitHub, 22 12 2018. [Online]. Available: https://github.com/grafana/grafana/issues/14629. [Accessed 15 12 2021].

[35] Apache Spark, "spark.apache.org/docs/latest/hardware-provisioning.html," Apache Spark, [Online]. Available: https://spark.apache.org/docs/latest/hardware-provisioning.html. [Accessed 15 12 2021].

[36] lresende, "github.com/CODAIT/spark-bench," GitHub, 21 11 2018. [Online]. Available: https://github.com/CODAIT/spark-bench. [Accessed 15 12 2021].