A bigdata system with hdfs, spark and airflow

# Introduction

Simplicity, performance and time-sensitivity are three main requirements of bigdata tasks. For this reason, we have built up a bigdata system on which data can be easily sent, results can be easily received and tasks can be easily triggered. Simplicity is achieved by a simple CLI, which allows user to manipulate HDFS, Spark and Airflow with an only dependency of JRE and knowledge of simple operations. Performance is achieved by distributed file system HDFS and distributed computing framework Spark, which distributes the data and tasks on different nodes. Time-sensitivity is achieved by task scheduler Airflow, which triggers the bigdata task every day. If there are many bigdata tasks and they have dependencies on each other, Airflow also helps to manage these dependencies so that developers don’t need to write out the logic of dependency in their code of analysis tasks.

I also wrote a sample analysis task script to analyze the LE data in this bigdata system. For now, this system only supports data of .csv format and task script of .py format.

# User Interface

The user interface of this system is a CLI(Command line interface). This interface is implemented with java and uses maven to do package management. The CLI leverages reflection technique and have all address-related parameters configurable. By changing the address-related parameters, users can easily deploy different clients with this code to manipulate different HDFS and Spark.

There are three supported functions in this application: <read>, <writeData>, <wirteScript>, <trigger>. Read helps users to get the analysis result from HDFS. WriteData helps users to upload data to HDFS. WriteScript helps users to upload analysis task code to HDFS. While uploading directory, the application automatically flats the directory and sends only the valid files in it. Trigger helps users to trigger the analysis dag at once.

Graphical user interface, text, chat or text message

Description automatically generated

# Data processing and analysis center

This processing and analysis center is implemented with Spark and HDFS, and customized analysis scripts. Basically I set up three folders Data, Result and Script on HDFS to store different data. Data folder contains csv files to be analyzed. WriteData operation will directly write csv files into this folder. Result folder contains csv files which are the analysis result. Script folder contains python scripts which define the tasks to be executed. These scripts must be put in the HDFS for two reasons. The first reason is, All host-related variables inside the scripts can be simply set as localhost. the second reason is, the python scripts should be triggered by Airflow once in a while, since sshOperator is in airflow core package and sparkSubmitOperator is in provider package, using ssh method to trigger the python script makes it easier to deploy.

# Task Manager

This part is implemented with Airflow platform and customized dags. The task manager enables us to view the task execution history and triggers tasks in a customized interval. The time interval set for our sample task is 1 day, which means that every day the analysis result will be updated. Also, the task manager can define the dependencies between tasks. For example, suppose we have a transactional database as our data source in future, we can set up two tasks to complete one target. Let’s say that the first task is to do ETL on the data and the second task is to run analysis on the data, obviously the first task should happen before the second task. If we define this dependency in Airflow, then the first task will be executed once the whole job is triggered. The second task will not execute until the first one is successful.

Diagram

Description automatically generatedIt’s worth mentioning that the uploading of dags are tricky. By default, a dag must be sent to airflow-webserver, airflow-scheduler and all airflow-workers. This has generated many duplicated operations. A workaround is to utilize Kubernetes-<git-sync> to synchronize the local folders with a remote file on github. With the help of git-sync, every time we push a new dag, it will automatically be pulled by the local repositories on airflow pods. In that case, deploying new dag becomes super easy.

Figure 1: Writing dag architecture

# Experimental tasks

## Dataset

The experimental task is based on LE dataset

## Destination

we want to study two relationships: The relationship between temperature and battery health reduction rate, and the relationship between m\_vs and battery health reduction rate.

## Algorithm

The core algorithm is simple, go through the whole dataset, compute battery capacity with battery\_state\_of\_charge\_pct and battery\_available\_energy\_wh , then divide delta battery capacities by delta time. The result should be the battery health reduction rate. Our result is to group the battery health reduction rates by either m\_vs or temperature, then take the average.

## Challenges

Although the core algorithm is instant and simple, there are some problems while implementing the algorithm. A big problem is that different dataset may have different headers. In that case, we cannot let spark to read them all with one schema, which will result in error. Even after we truncate the headers so that different dataFrames have same header, it still doesn’t work because order matters. For example, in dataFrame a, the header is <m\_vs, unitID> and in dataFrame b, the header is <unitID, m\_vs>, then when we union these two dataFrames, the unitID column in dataFrame b comes under m\_vs column in dataFrame a, and vice versa. So we decided to read datasets one by one, transform them into RDDs and arrange the header sequence with map function. After unioning all the RDDs, we change the RDD back to Dataframe and do operations on this standardized dataFrame. However, this method has resulted in serious performance problem. Compared with reading all datasets together, the union method consumes 10x longer time. It will not be so difficult to find a more efficient way with advanced knowledge of Spark.

## Result

Sadly, for now we haven’t found any meaningful relationship between m\_vs/temperature and battery health reduction rate. Our assumption was that as m\_vs/temperature goes up, the battery health reduction rate goes up as well. However, the data fluctuates and some times even show that battery health reduction rate is negative(which means the battery has larger capacity than at previous timestamp). I can’t explain this phenomenon and we may need a data scientist to design a more practical analysis process.

# Deployment

We will focus on Client deployment but also mention deployment of other parts.

## Client

Our client deployment is based on jar file. We don’t use docker for the deployment of this project for two reasons. The first one is that there is only one class for this client and the only dependency is JRE/JDK. Another reason is that this client manipulates local disk. Manipulating local disk is much easier than manipulating docker disk. For these reasons, although docker has very handy runtime environment management, we choose to use jar file instead.

## Data processing and analysis center

To find an easy way to deploy your HDFS, Spark and Hive, please refer to this resource: <https://bambrow.com/20210625-docker-hadoop-1/>

## Task Manager

Deployment of Airflow is super easy with the help of official helm charts and GKE(Google Kubernetes Engine). Please use LoadBalancer on GKE to expose the airflow-webserver.