UE19CS322 Big Data Project 1

YAH – Yet Another Hadoop

YAH, or **Yet Another Hadoop** is one of the project titles that can be taken up as a part of the UE19CS322 Big Data course at PES University. YAH is meant to be a mini-HDFS setup on your system, complete with the architectural structure consisting of Data Nodes and Name Nodes and replication of data across nodes.

With YAH, you will possess finer control over your data blocks and their replication, and allow you to create a distributed file system as per your needs. YAH also allows running Hadoop like jobs to break down computationally heavy tasks into smaller distributed tasks.

The files required for the project can be found here. **You must fill up this form and submit the link to your project's repository on GitHub** for your project's evaluation.

Project Objectives and Outcomes

1. This project will help students obtain an in-depth understanding of Hadoop and its underlying architecture and allow them to tweak it as per their needs and observe its inner working
2. At the end of this project, the student will be able to simulate a miniature HDFS capable of performing some of the important tasks a distributed file system performs, running HDFS commands as well as scheduling Hadoop jobs.

Ethical practices

Please submit original code only. You can discuss your approach with your friends but you must write original code. All code must be submitted through the portal. **We will perform a plagiarism check on the code and you will be penalised if your code is found to be plagiarised**

Software/Languages to be used:

Python `3.8.x`

You are free to use any external APIs and libraries as desired.

Marks

15 marks

Submission Link

Portal for Big Data Assignment Submissions

Submission Deadline

3rd December 11:59 PM, 2021

Project Guidelines

1. Your code will **not be executed on the portal**. However, it will be used to **perform a plagiarism check later** on.
2. You are free to use any libraries and APIs to implement the project
3. **There is no fixed guideline on how you may implement the project**. The design and architecture is completely upto you. There are no constraints on the implementation, however your code must demonstrate all the required working functionalities.
4. You will be required to **create a GitHub repository and add the Teaching Assistants as collaborators** to it. You have time until your team's presentation to add us to the repository failing which your team will **not be evaluated**. You can find the usernames to add below:
    - aditeyabaral
    - anshsarkar
    - Visheeeee
    - pvn-leo
5. **You must fill up this form and submit the link to your project's repository on GitHub**. Failing to fill up this form will lead to your project not getting evaluated.
6. **You are required to make regular commits to your repository**. Your progress on the project is tracked periodically. Failing to make regular commits will lead to you being penalised.
7. You are highly suggested to structure the project well by using multiple files (but not too many) structured in different folders. You can visit this link to learn about how to structure your repository well.
8. You will be required to **present your implementation to the faculty as a part of the viva**
9. **Do note that any member of your team may be required to demonstrate any feature, or make changes to the configuration and re-run your project**. Ensure that the project runs on every single team member's system.

10. Convert line breaks in DOS format to Unix format (**this is necessary if you are coding on Windows -** your code will not run on our portal otherwise)

```
dos2unix filename.py
```

Task Specifications

Do ensure that during implementation, each functionality is bundled into a separate process that can be executed on the shell. You may have as many helper modules as necessary.

Setting up the Distributed File System

In this section, you will be required to implement the core of the distributed file system. This section will require you to implement the s as well as the Name Nodes, and ensure efficient communication between the two.

A single `config.json` will be provided to you which will contain the configuration details of the DFS. Attached below is the schema for the config file. **You may add more attributes to the ones already listed. However note that the listed attributes are mandatory and need to be in every config file**.

```
{
    "block_size": int, indicates block size to split files,
    "path_to_datanodes": str, absolute path to create Data Nodes,
    "path_to_namenodes": str, absolute path to create Name Node and secondary Name Nod
e,
    "replication_factor": int, replication factor of each block of a file,
    "num_datanodes": int, number of Data Nodes to create,
    "datanode_size": int, number of blocks a single Data Node may contains,
    "sync_period": int, interval in seconds for the secondary Name Node to sync with t
he primary Name Node,
    "datanode_log_path": str, absolute path to a directory create a log file to store
information about Data Nodes,
    "namenode_log_path": str, absolute path to create a log file to store information
about Data Nodes,
    "namenode_checkpoints": str, abolute path to a directory to store Name Node checkp
oints,
    "fs_path": str, absolute path to file system where a user may create files,
```

```
    "dfs_setup_config": str, absolute path to DFS setup information
}
```

The setup process is expected to read in this config file and create the DFS based on the configuration provided. If no such configuration is provided, it must resort to using a default configuration (which is up to you). For each unique `config.json` provided as an argument, it must create a new DFS based on the values of the configuration variables.

After the setup has been completed, you must create a file which stores information about the DFS in `dfs_setup_config`. This file should store the configuration settings and must be used to load the DFS for later use.

Initial Setup

The `fs_path` attribute stores the absolute path to a **virtual** directory where a user can create folders and add files. However, do remember that this is virtual, and **no data is stored here** (since this is simply the HDFS root directory). Data is only stored in the Data Nodes in blocks. The Name Node is incharge of mapping file system operations performed in this directory to files present in the Data Nodes. This path is used as a root directory to manipulate the distributed file system.

Data Nodes

1. The number of Data Nodes to create is given by `num_datanodes`. You are free to implement a Data Node based on your implementation.
2. All Data Nodes must be created inside the `path_to_datanodes` directory.
3. Each Data Node must store only `datanode_size` blocks. If there is no more available space in a datanode, it must be **removed from the block replication process**.
4. The block size for each file is provided by `block_size`. Each file must be split into the specified block size, and each block must be replicated `replication_factor` number of times. Each replicated block must now be stored in a Data Node.
5. Choosing a Data Node for a block must be based on hashing (or any other algorithm). **If there are collisions, or a Data Node is full, suitable measures must be taken**. The allocation of blocks to Data Nodes must be efficient.
6. **All information about the position of each replicated block in a Data Node must be stored on the Name Node**.

7. Data Nodes can store logging information (such actions performed on a block, creation of new files and folders, etc) in `datanode_log_path`. Each Data Node must create its own logging file inside this directory to track only its blocks.
8. **Optional**: It is possible for Data Nodes to crash or get deleted. If this happens, attempt to recreate the Data Node from the information stored in the Name Node about the deleted Data Node. You will be required to simulate the deletion of a Data Node to demonstrate this feature.

Name Nodes

1. The primary Name Node must keep track of all changes being made in `path_to_fs`. This includes creation or deletion of files and folders.
2. Each file in `fs_path` and its associated blocks must be tracked by the Name Node, which should store the mapping of each block of a file to a particular Data Node.
3. The Name Node should also track details about each file such as the **number of blocks it consists of, the location of the Data Nodes where its blocks are located** and so on.
4. The Name Node also logs information about Data Nodes. Meta data about Data Nodes (can be any information you wish to track, such as size of each Data Node, number of files, number of blocks occupied, etc) must be logged to `namenode_log_path`.
5. Each Name Node must periodically (`sync_period`) send a heartbeat signal to every Data Node. During this process, it must **ensure all Data Nodes are running, and update the status of each Data Node. If a Data Node is full, or if it cannot find a Data Node, it must take suitable actions** to ensure there isn't a failure while accessing the DFS.
6. Each Name Node should create a checkpoint of its data every `sync_period` seconds in `namenode_checkpoints`.
7. A secondary Name Node must be setup. It should back up all the data stored by the primary Name Node every `sync_period` seconds. **If the primary Name Node ever fails, the secondary Name Node must take over as the primary Name Node** and must retrieve all information from the last created checkpoint. It must also create a new secondary Name Node and restore information upto the last checkpoint.
8. **Optional**: Implement an Edit Log. This log must be stored inside the Name Node and must be updated with all the operations that take place on the Data Node. The contents of the log must be merged with the Name Node's meta data content about the file system mappings and block details at periodic intervals.

Persistent Storage

1. All files, folders and logs created in the Data Nodes and Name Nodes must **persist to disk**.

2. The configuration stored in `dfs_setup_config` should be used to load the DFS for later use to manipulate on the DFS.
3. It **must be possible to load the configuration of multiple DFS configurations**, since multiple `config` files may be used to setup different DFS on the same machine.

Manipulating the Distributed File System

In this section, you will be required to implement the file system operations which will allow a user to work with the DFS.

The `dfs_setup_config` created during the previous section must be used to load the configuration of the DFS. After loading, it must display the configuration of the DFS being loaded for the user and provide a CLI interface to access the file system.

Loading the Distributed File System

1. The absolute path to a `dfs_setup_config` file will be provided to load the DFS. If such an argument is not provided, it must load the most recently created DFS. If no such DFS exists, it must prompt the user to create a new one.
2. After reading the `dfs_setup_config` file, it must display the configuration it is trying to load for the user.
3. It should **verify if the DFS can be loaded, by verifing the paths of the Data Nodes and Name Nodes. It must ensure and perform basic checks** such as (but not limited to):
    - Metadata contents about the Data Nodes stored by the Name Node exist and are correct
    - Ensure that each Data Node contains all the blocks it should contain
    - Ensure that the Name Node contains the correct mapping of blocks to Data Nodes
    - Take suitable actions and attempt to restore the configuration in case of failures
4. **If the DFS has been loaded for the first time, it must prompt the user to format the Name Node**. The format operation must delete all data inside Data Nodes and Name Nodes and delete the contents of all the log files.

Accessing the Distributed File System

1. Implement a CLI interface that prompts the user for commands
2. The CLI must **mandatorily** implement the following commands. You are free to add more commands to your CLI as well.
    - put

    - cat
    - ls
    - rm
    - mkdir
    - rmdir
3. For each command, the corresponding HDFS action must be carried out. For example, if the `put` command is used, it must also receive two arguments (`source_file` and `destination_directory`). The `source_file` must be split into blocks based on the configuration and each block must be assigned to a Data Node. The Name Node must be updated with the details of the mapping between blocks and Data Nodes and must also contain the virtual path of the `source_file` inside the `fs_path` .
4. The **CLI should never crash** unless the process has been terminated.

Running Hadoop Jobs

In this section, you will be required to simulate the running of a Hadoop job.

The `dfs_setup_config` created during the previous section must again be used to load the configuration of the DFS.

The process must take in the following arguments via command line:

```
--input, -i :  str, absolute path to input file inside fs_path,
--output, -o: str, absolute path to directory to store output and job status,
--config, -c: str, absolute path to dfs_setup_config to load DFS configuration,
--mapper, -m: str, absolute path to mapper.py,
--reducer, -r: str, absolute path to reducer.py
```

1. From the `dfs_setup_config` , the configuration of the DFS must be loaded, which in turn will load the configuration of the Name Node and Data Nodes.
2. This should be used to obtain the number of blocks for the `--input` file. For each block, obtain its contents and combine them in the right order.
3. You need to pass this input through the `cat` command to your `--mapper` file, sort the output, feed that to your `--reducer` file and store the output in the `--output` directory.

Optional: Implement it using multiple subprocesses. Your main job process must be capable of distributing the task at hand into multiple parallel running processes.

1. From the `dfs_setup_config`, the configuration of the DFS must be loaded, which in turn will load the configuration of the Name Node and Data Nodes.
2. This should be used to obtain the number of blocks for the `--input` file. For each block, a new subprocess must be forked from the main process.
3. Each subprocess will receive a block of the input file, along with the `--mapper` file. Each subprocess will now in turn execute the mapper with the block
4. The output from all subprocesses must be aggregated together. After aggregation, another subprocess must be forked for the `--reducer` file, which will receive the aggreated output from the mapper subprocesses as input.
5. The output from the reducer subprocess must be written into a file in the `--output` directory.
6. All logs about the process (number of mappers, number of blocks, logs about the job) must be stored in a log file inside the `--output` directory.