HDFS Emulation Project Report

By Yufei Wang and Xinyi Zhang

1. Topic

The architecture of our system is shown below. We use Vue.js to implement frontend UI interface and we use Python and Flask to connect the frontend page with the backend database. Our system can also accept shell commands in the terminal and we use sockets to implement that.

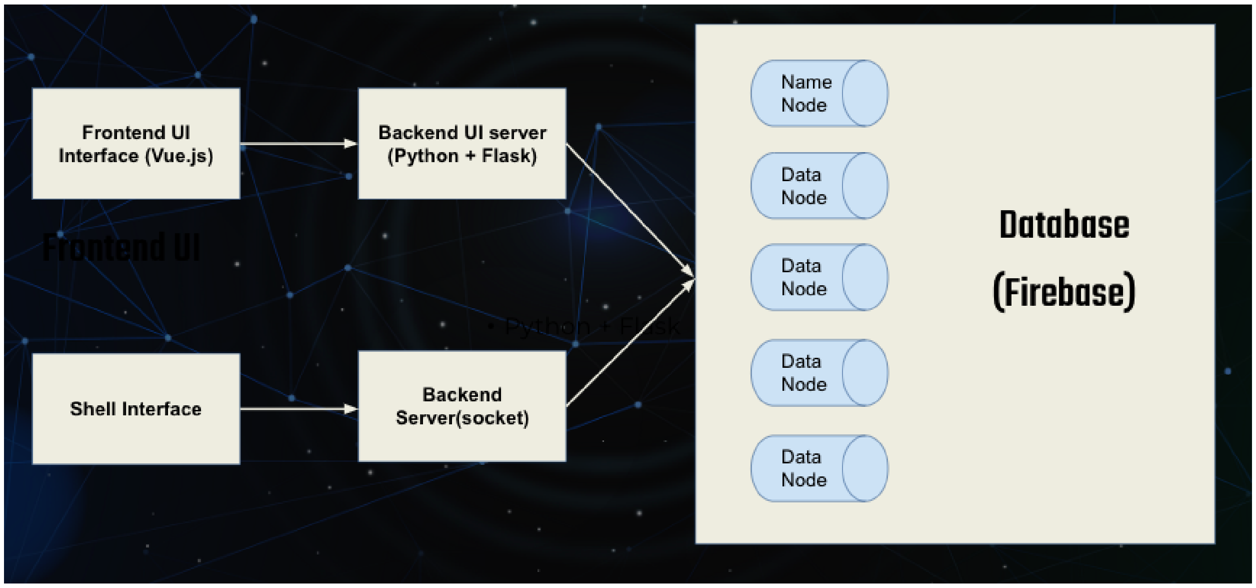


Fig1. System design of the project

2. Implementation

2.1 Database Design

We use Firebase for the database. And we design one NameNode to store the metadata and four DataNodes to store the data. In order to emulate the practical distributed file system where datanodes are more likely to distribute across different clusters all over the world, we store our DataNode in different databases of our group members' machines.

The structure of the NameNode is shown below. It has two main sections, dirSection and inodeSection. Inside the dirSection, the keys are the inumber of each directory and the values are the contents under that directory. Using the dirSection, we are able to explore the directory structure effectively.

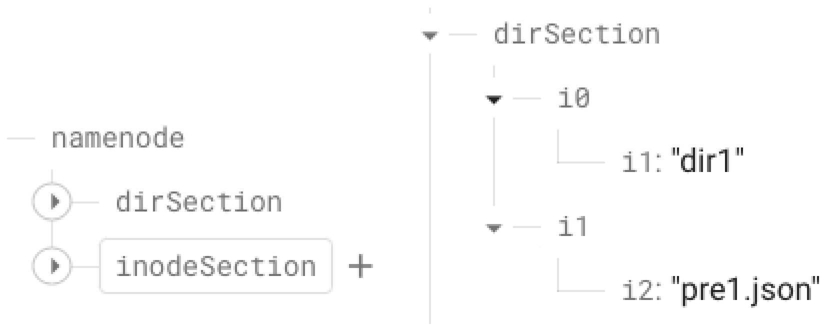
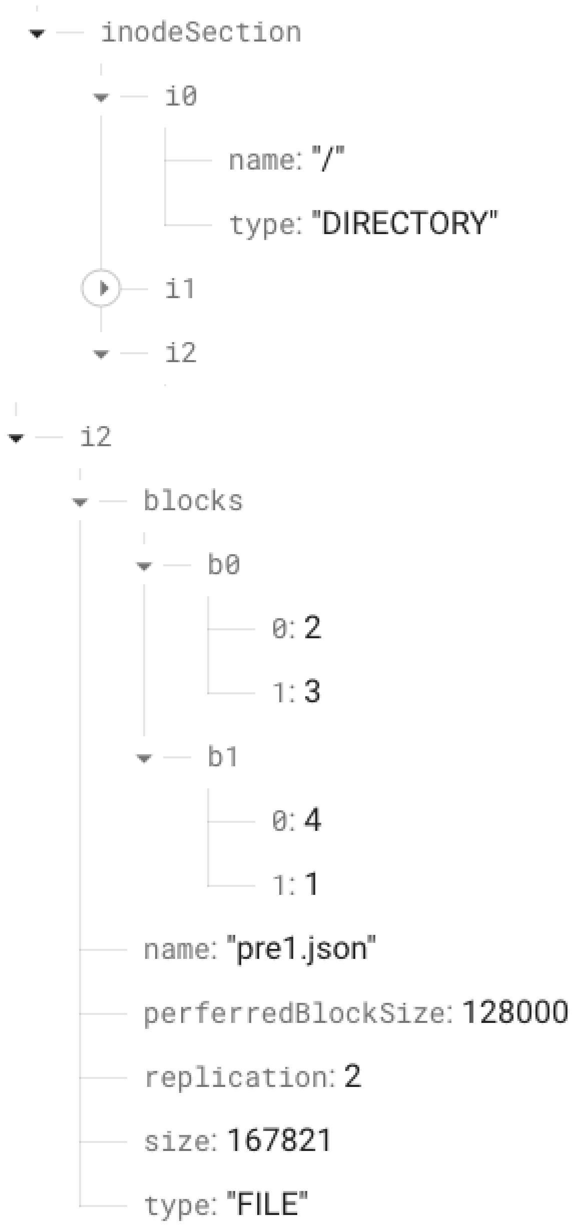


Fig2. Namenode directory section structure

The structure of inodeSection is shown below. The keys are the inumber of each file or directory. For a directory, it only records its name and type. For a file, it records its name, preferredBlockSize, replication factor, file size and type. For each file, it has a blocks section to store the location of the blocks, which are the datanode number of each block.



```
inodeSection
  i0
    name: "/"
    type: "DIRECTORY"
  i1
  i2

i2
  blocks
    b0
      0: 2
      1: 3
    b1
      0: 4
      1: 1
  name: "pre1.json"
  perferredBlockSize: 128000
  replication: 2
  size: 167821
  type: "FILE"
```

Fig3. Namenode inode section structure

We wrote a function called writeToNamenode() to update metadata in the NameNode. When a user wants to put a new file or a new directory into the system, the system will automatically call the function. It basically updates information in the two main sections, inodeSection and dirSection. Before adding new metadata, the function first will generate a new inumber for the new file or directory. The way we generate new inumber is we always find the last inumber of our system, and we plus 1 to generate the new inumber for the new file or directory. When writing to dirSection, if it is a directory, it first needs to add an new entry in the dirSection. Then the function will find the parent directory of the new file or directory and add certain metadata under the parent directory.

In order for our system to balance the block storage, we also write a function called allocateBlocks(). This function will allocate a datanode to the block to write. In order to return datanode which has the minimum number of blocks, we first record the number of blocks in each datanode and find the minimum number of blocks in each datanode. And if there is only one datanode with the minimum number of blocks, the function will directly return that datanode number. If there are more than one datanodes with the same minimum number of blocks, the function will randomly choose one datanode to return.

2.2 Backend Server and Frontend UI

We used Javascript axios and python Flask to communicate between frontend and backend. Here is the in-depth implementation of our backend functions that manipulates the database:
ls: getting the names of all files and directories under the given path sent from frontend. We first checked the inode section of the namenode to find the inumber of the directory, and then found the corresponding directory in the directory section of namenode and returned all entries under it to the frontend.
rm: removing a file in our file system. We first checked the inode section of the namenode to find the inumber of the file, and then found the datanodes which stored the blocks of the file, and deleted data in datanode and the metadata in namenode. Note that we need to delete it file under the directory in which it is located, so we need to retrieve parent information as well in the inode section as well.
put: putting the content from frontend into our system. Predefined the default block size, we partitioned the file into several blocks and allocated datanode randomly to store the block and its replications. We created block number for each partition by searching for the maximum block number inside the inode section. Then we called writetoNamenode to write metadata to namenode, returning json "ok" string back to the frontend.
mkdir: creating a new directory.We simply called writetoNamenode using file name parsed from frontend, and return json "ok" string back to the frontend.
rmdir: The input argument is the path of the directory to be removed, and the function will report if the command was executed successfully or not. Firstly, we found the inumber of the directory and checked the number of entries under the directory in the dirSection of namenode to check if the directory is empty or not. Delete the directory only if it's an empty directory.
cat: getting the content of the file to the frontend. We implemented this by first finding the information of the file in the inodeSection of the namenode from the source path, and then found the datanode where blocks of the file were stored and joined data together, returning it to the frontend.

getPartitionLocation and readPartition: returned the partition file location and content to the frontend for better user experience.
File: receiving the file from frontend to backend using flask. This function does not actually interact with the database.

For User Interface, we used vue.js. The structure of pages and dialog elements are written in HTML. When the user clicks on a file icon, a dialog box will show up; and when the user clicks on a directory icon, the page would proceed into the directory. Also, when the user clicks on different buttons, vue would execute different functions in order to fulfill interaction needs of the user. (Fig 4) For detailed frontend design, please refer to our code inside UI/hdfs-frontend/src/Homepage.vue on Google drive.

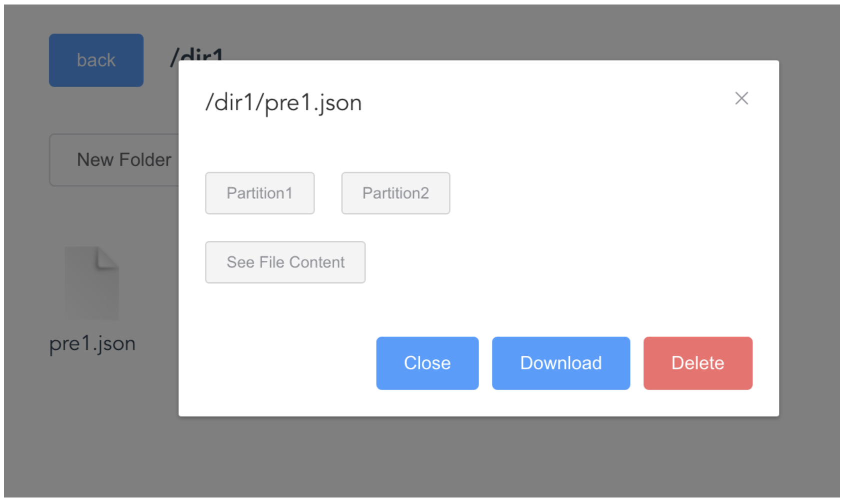


Fig4. Sample UI interface

2.3 Socket

In order for the client and server to communicate with each other, the server first needs to open up a connection and the client needs to connect to it using the same port number. Then the client can accept the shell commands typed by users in the terminal. When the client receives the shell commands, it needs to send them to servers. After the server receives the commands, the server can execute certain functions. After successfully executing the function, the server will write a report message back to the client to report whether the function is executed successfully or not. And the client will write the report message from the server in the terminal so that users can get feedback.

3. Learning experiences

As both of our team members had no experience in full-stack development, this project provided us with an excellent opportunity to build up our code from scratch, using knowledge learnt from the course. We designed our database using real-world HDFS format, learnt new skills including flask and vue.js when implementing backend and frontend interaction. We also learnt a lot during the debug process, such as how to download a file in frontend, how to send file content using socket. The project significantly improved our coding skills.

4. Additional Information:

Video link:
https://youtu.be/SLGoAuBmRKc
Code link:
https://drive.google.com/drive/folders/1vIXazFwnYm9iWOSYDN4zGBqhyOSozQgl?usp=share_link