

Airline flight delay check and delay cause

Background

In the U.S., airplanes have been one of the most popular transportation vehicles for people to travel domestically and internationally. Every day, FAA's Air Traffic Organization (ATO) provides service to more than 45,000 flights and 2.9 million airline passengers across more than 29 million square miles of airspace. However, flight delays often happen unexpectedly due to various reasons, causing a lot of inconvenience for passengers' travel plans. Our project aims to help passengers check if there is any delay in their flights so that they can make other arrangements and make this process very straightforward. They can also check the common causes of flight delays in their frequently travel airports.

Dataset

	Contents	link
Dataset 1 Source: Kaggle	Flight Delay	https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018/code
Dataset 2 Source: Kaggle	Flight and Airport Delay Cause	https://www.kaggle.com/datasets/ryanjt/airline-delay-cause
Dataset 3 Source: Kaggle	List of US Airports	https://www.kaggle.com/datasets/aravindram11/list-of-us-airports

Task1 implementation details

For the EDFS requirement, we chose the MySQL and firebase based implementation. The MySQL implementation is our main focus for the back-end server as well as the overall workflow of our project. We built on top of this MySQL implementation to finish task 2, and also connected our front-end implementation in task 3 to the MySQL EDFS. For the MySQL implementation backend, we chose to use an Object Oriented programming approach. First, to keep the code concise, we are reusing a lot of the class methods, such as `_check_dir_path()`, `_dataList()`, `_execute`, `_create_file_tab`, `_insert_file_data()`, `_insert_file_data()`, `_del_file_pathId_name`, `_create_db()`, etc. Throughout the EDFS class inside the python file, they are being reused to handle the user input that's redirected from the HTML form input, processing information such as commands, file path, name of the file to be operated, etc.

More Details on MySQL implementations:

- To build an emulated distributed file system, we applied MySQL KEY Partitioning. According to MySQL Documentation, partitioning by key is supplied by the MySQL server. The KEY function takes only a list of zero or more column names. MySQL will automatically use the primary key or a unique key as the partitioning column. If no unique keys are available, the statement will fail.
 - In our implementation, users are able to define one or more columns as partitioning keys and the number of partitions.
 - Code example:

```
def _create_file_tab(self, tableName, keys, partition:EDFS_PARTITION):
    sql = "CREATE TABLE `{table}` (`edfs_table_id` int(0) NOT NULL AUTO_INCREMENT".format(table = tableName)
    for key in keys:
        if partition != None and partition.partition_key == key:
            sql += ", `{key}` varchar(255) NOT NULL".format(key=key)
        else:
            sql += ", `{key}` varchar(500)".format(key=key)
    if partition != None:
        sql += ", PRIMARY KEY (`edfs_table_id`, `{key}`)".format(key = partition.partition_key)
        sql += "PARTITION BY KEY(`{name}`)".format(name = partition.partition_key)
        if partition.partition_count != None:
            sql += "PARTITIONS {count}".format(count = partition.partition_count)
    else:
        sql += ", PRIMARY KEY (`edfs_table_id`)"
    sql += ";"
    self._execute(sql)
```

- There are two major tables in our EDFS which are EDFS_PATH and EDFS_FILE. The “EDFS_PATH” is used to save path information. There are 3 columns in this table, there are “id”, “parent_id”, and “name”. The “parent_id” equals -1 standing for the root directory. The “EDFS_FILE” is used to save file information. There are also 3 columns in the table, which are “id”, “path_id” and “name”.
 - After inputting two commands “mkdir("/user/path")” and “mkdir("/user/path2")”, the EDFS_PATH looks below:

id	parent_id	name
1	-1	user
2	1	path
3	1	path2
NULL	NULL	NULL

The parent_id for user is -1 since it is the root directory and the parent_id for “path” and “path2” are both 1 because their parent directory is “user”.

- After inputting the command “put(“airports.csv”, “/user/path2”, k)”, the airports.csv file will be saved under path2. So the EDFS_FILE looks below:

id	path_id	name
3	3	airports.csv
NULL	NULL	NULL

The “id” stands for file id, because I have already put the airports.csv file before, so the id here is 3. The path_id is 3 because the id in EDFS_PATH for “path2” is 3. This is the method we use to query the address of the file.

- Supporting Commands - mkdir

Firstly, split the user input to find the path. Check whether the folder exists in the current directory. If it exists, we do nothing. If it does not exist, create a new folder.

- Code example:

```
def mkdir(self, path):
    path_list = path.split("/")
    parent_id = -1
    for dir in path_list:
        if dir == "":
            continue
        dataList = self._query_dir(parent_id, dir)
        if len(dataList) == 0:
            dataList = self._insert_dir(parent_id, dir)
            parent_id = dataList[0]["id"]
    print("mkdir success")
```

Input: `edfs.mkdir("/user/path2")`

Output: `mkdir success`

- Supporting Commands - put

First, checking whether the input file is a CSV file because our system only supports CSV files. Then, check whether the entered directory exists and whether the file with the same name exists in the current directory. An error will arise if the directory does not exist or if a file with the same name exists.

If there is no file with the same name in the directory, there are two steps to upload the file. The first step is to create a record of the file in the table “EDFS_FILE”, and save the id of the file, the id of the file directory, and the name of the file. The second step is to create a table in MySQL according to the key partition.

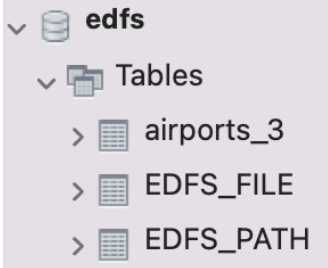
- Code example:

```
def put(self, file_name, path, k:EDFS_PARTITION=None):
    file_type = file_name.split(".")
    if (len(file_type) == 2 and (file_type[1] == "csv" or file_type[1] == "CSV")) == False:
        print("Error: Please upload a file in CSV format")
        return
    path_id = self._check_dir_path(path)
    if (path_id == False):
        print("Error: Path does not exist")
        return
    dataList = self._query_file_pathId_name(path_id, file_name)
    if len(dataList) != 0:
        print("Error: A file with the same name exists in this directory")
        return
    fileId = self._inster_file(path_id, file_name)
    fileId = fileId[0]["id"]
    with open(file_name, mode="r", encoding='utf-8') as read_obj:
        dict_reader = DictReader(read_obj)
        list_of_dict = list(dict_reader)
        table_name = file_type[0] + "_" + str(fileId)
        self._create_file_tab(table_name, list_of_dict[0].keys(), k)
        for dic in list_of_dict:
            self._insert_file_data(table_name, dic)
    print("put file success")
```

```
class EDFS_PARTITION:
    def __init__(self, partition_key, partition_count):
        self.partition_key = partition_key
        self.partition_count = partition_count
```

Input: `k = EDFS_PARTITION("IATA",2)`
`edfs.put("airports.csv", "/user/path2", k)`

Output: `put file success`

Database: 

EDFS_FILE:

id	path_id	name
3	3	airports.csv
NULL	NULL	NULL

EDFS_PATH:

id	parent_id	name
1	-1	user
2	1	path
3	1	path2
NULL	NULL	NULL

- Supporting Commands - cat

Firstly, splitting the user input to find the path and check if the path and file exist.

If it exists, just read the data from the table that we created for the file.

Input: `edfs.cat("/user/path2/airports.csv")`

Output:

	edfs_table_id	IATA	...	LATITUDE	LONGITUDE
0	1	ABQ	...	35.04022222	-106.6091944
1	2	ANC	...	61.17432028	-149.9961856
2	5	BDL	...	41.93887417	-72.68322833
3	6	BHM	...	33.56294306	-86.75354972
4	7	BNA	...	36.12447667	-86.67818222
...
336	333	TYS	...	35.81248722	-83.99285583
337	335	VLD	...	30.7825	-83.27672222
338	338	WYS	...	44.68839917	-111.1176375
339	339	XNA	...	36.28186944	-94.30681111
340	340	YAK	...	59.50336056	-139.6602261

- Code example:

```
def cat(self, file_path):
    path_list = file_path.split("/")
    file_name = path_list[-1]
    file_type = file_name.split(".")
    dir_path = "/".join(path_list[:-1])
    path_id = self._check_dir_path(dir_path)
    if (path_id == False):
        print("Error: file does not exist")
        return
    dataList = self._query_file_pathId_name(path_id, file_name)
    if len(dataList) == 0:
        print("Error: file does not exist")
        return
    table_name = file_type[0]+"_"+str(dataList[0]["id"])
    dataList = self._query_file_data(table_name)
    df = pd.DataFrame(dataList)
    print(df)
```

- Supporting Commands - ls

Checking the file name and file id based on path information in EDFs_FILE table and checking folder and file information based on directory in EDFs_PATH table.

- Code example:

```
def ls(self, path):
    path_id = self._check_dir_path(path)
    if path_id == False:
        print("Error: Path does not exist")
        return
    file_list = self._query_file_pathId_name(path_id)
    dir_list = self._query_dir(path_id)
    for data in file_list:
        print(data["name"], end=" ")
    for data in dir_list:
        print(data["name"], end=" ")
    print("")
```

Input: `edfs.ls("/user/path2")`

Output: `airports.csv`

- Supporting Commands - rm

Firstly, splitting the user input to find the path and checking if the path and file exist. If it exists, we delete the file information in the table EDFS_FILE first, then corresponding drop the table from MySQL.

- Code example:

```
def rm(self, file_path):
    path_list = file_path.split("/")
    file_name = path_list[-1]
    dir_path = "/".join(path_list[:-1])
    path_id = self._check_dir_path(dir_path)
    if (path_id == False):
        print("Error: file does not exist")
        return
    dataList = self._query_file_pathId_name(path_id, file_name)
    if len(dataList) == 0:
        print("Error: file does not exist")
        return
    self._del_file_pathId_name(path_id, file_name)
    file_type = file_name.split(".")
    table_name = file_type[0] + "_" + str(dataList[0]["id"])
    self._del_file_data_table(table_name)
    print("remove file success")
```

- Supporting Commands - getPartitionLocations(file)

Firstly, split the user input to find the path and check if the path and file exist. If it exists, we use the name of the table to query the partition information.

- Input: `edfs.getPartitionLocations("/user/path2/airports.csv")`

- Output:

	TABLE_SCHEMA	TABLE_NAME	PARTITION_NAME
0	edfs	airports_3	p0
1	edfs	airports_3	p1

- Code example:

```
def getPartitionLocations(self, file):
    path_list = file.split("/")
    file_name = path_list[-1]
    dir_path = "/".join(path_list[:-1])
    path_id = self._check_dir_path(dir_path)
    if (path_id == False):
        print("Error: file does not exist")
        return
    dataList = self._query_file_pathId_name(path_id, file_name)
    if len(dataList) == 0:
        print("Error: file does not exist")
        return
    file_type = file_name.split(".")
    table_name = file_type[0] + "_" + str(dataList[0]["id"])
    dataList = self._query_file_data_partition_info(table_name)
    df = pd.DataFrame(dataList)
    print(df)
```

- Supporting Commands - readPartition(file, partition#)

Firstly, splitting the user input to find the path and check if the path and file exist.

If it exists, we can access data from a certain partition name under a table and return a list.

- Code example:

```
def readPartition(self, file, partition):
    path_list = file.split("/")
    file_name = path_list[-1]
    dir_path = "/".join(path_list[:-1])
    path_id = self._check_dir_path(dir_path)
    if (path_id == False):
        print("Error: file does not exist")
        return
    dataList = self._query_file_pathId_name(path_id, file_name)
    if len(dataList) == 0:
        print("Error: file does not exist")
        return
    file_type = file_name.split(".")
    table_name = file_type[0] + "_" + str(dataList[0]["id"])
    dataList = self._query_file_data_partition_data(table_name, partition)
    return dataList
```

Input:

```
data_list = edfs.readPartition("/user/path2/airports.csv", "p0")
print(data_list)
```

Output:

```
{'edfs_table_id': 1, 'IATA': 'ABQ', 'AIRPORT': 'Albuquerque International', 'CITY': 'Albuquerque', 'STATE': 'NM', 'COUNTRY': 'USA', 'LATITUDE': '35.04022222', 'LONGITUDE': '-106.6091944'}, {'edfs_table_id': 2, 'IATA': 'ANC', 'AIRPORT': 'Ted Stevens Anchorage International', 'CITY': 'Anchorage', 'STATE': 'AK', 'COUNTRY': 'USA', 'LATITUDE': '61.17432028', 'LONGITUDE': '-149.9961856'}, {'edfs_table_id': 5, 'IATA': 'BDL', 'AIRPORT': 'Bradley International', 'CITY': 'Windsor Locks', 'STATE': 'CT', 'COUNTRY': 'USA', 'LATITUDE': '41.93887417', 'LONGITUDE': '-72.68322833'}, {'edfs_table_id': 6, 'IATA': 'BHM', 'AIRPORT': 'Birmingham International', 'CITY': 'Birmingham', 'STATE': 'AL', 'COUNTRY': 'USA', 'LATITUDE': '33.56294306', 'LONGITUDE': '-86.75354972'}, {'edfs_table_id': 7, 'IATA': 'BNA', 'AIRPORT': 'Nashville International', 'CITY': 'Nashville', 'STATE': 'TN', 'COUNTRY': 'USA', 'LATITUDE': '36.12447667', 'LONGITUDE': '-86.67818222'}, {'edfs_table_id': 8, 'IATA': 'BOS', 'AIRPORT': 'Gen Edw L Logan Intl', 'CITY': 'Boston', 'STATE': 'MA', 'COUNTRY': 'USA', 'LATITUDE': '42.3643475', 'LONGITUDE': '-71.00517917'}, {'edfs_table_id': 9, 'IATA': 'BUF', 'AIRPORT': 'Buffalo Niagara Intl', 'CITY': 'Buffalo', 'STATE': 'NY', 'COUNTRY': 'USA', 'LATITUDE': '42.94052472', 'LONGITUDE': '-78.73216667'}, {'edfs_table_id': 10, 'IATA': 'BUR', 'AIRPORT': 'Burbank-Glendale-Pasadena', 'CITY': 'Burbank', 'STATE': 'CA', 'COUNTRY': 'USA', 'LATITUDE': '34.20061917', 'LONGITUDE': '-118.3584969'}, {'edfs_table_id': 14, 'IATA': 'CLT', 'AIRPORT': 'Charlotte Douglas International', 'CITY': 'Charlotte', 'STATE': 'NC', 'COUNTRY': 'USA', 'LATITUDE': '35.21401111', 'LONGITUDE': '-80.94312583'},
```

- Other Useful Methods:
 - query_file_pathId_name(path_id, name)

This method is used to query the file information, like file id and file name, in the table EDIFS_FILE based on the path id.

```
def _query_file_pathId_name(self, path_id, name = None):
    userSql = "SELECT * FROM {table} WHERE path_id={path_id} AND name='{name}'".format(table=db_file_table_name, path_id=path_id, name=name)
    if name == None:
        userSql = "SELECT * FROM {table} WHERE path_id={path_id}".format(table=db_file_table_name, path_id=path_id)
    self._execute(userSql)
    dataList = self._dataList()
    return dataList
```

- query_dir(parent_id, name)

This method is used to query the data, like file and folder under a certain directory in the table EDIFS_PATH.

```
def _query_dir(self, parent_id, name = None):
    userSql = "SELECT * FROM {table} WHERE parent_id={parent_id} AND name='{name}'".format(table=db_path_table_name, parent_id=parent_id, name=name)
    if name == None:
        userSql = "SELECT * FROM {table} WHERE parent_id={parent_id}".format(table=db_path_table_name, parent_id=parent_id)
    self._execute(userSql)
    dataList = self._dataList()
    return dataList
```

- query_file_data(table_name ,where_dic)

This method is used to query the data according to a table name.

```
def _query_file_data(self, table_name ,where_dic = None):
    userSql = "SELECT * FROM {table}".format(table=table_name)
    self._execute(userSql)
    dataList = self._dataList()
    return dataList
```

- query_file_data_partition_info(table_name)

This method is used to query the partitioning information according to a table name.

- def _query_file_data_partition_data(table_name,partition_name)

This method is used to query the data of a partition in a file table based on the file table name and partition name.

- create_file_tab(tableName,keys,partition:EDFS_PARTITION)

This method is used to create a table for the uploaded file and also create partitioning.

```
def _create_file_tab(self, tableName, keys, partition:EDFS_PARTITION):
    sql = "CREATE TABLE `{table}` (`edfs_table_id` int(0) NOT NULL AUTO_INCREMENT".format(table = tableName)
    for key in keys:
        if partition != None and partition.partition_key == key:
            sql += ", `{key}` varchar(255) NOT NULL".format(key=key)
        else:
            sql += ", `{key}` varchar(500)".format(key=key)
    if partition != None:
        sql += ", PRIMARY KEY (`edfs_table_id`, `{key}`)".format(key = partition.partition_key)
        sql += "PARTITION BY KEY(`{name}`)".format(name = partition.partition_key)
        if partition.partition_count != None:
            sql += "PARTITIONS {count}".format(count = partition.partition_count)
    else:
        sql += ", PRIMARY KEY (`edfs_table_id`))"
    sql += ";"
    self._execute(sql)
```

We also have a firebase implementation, it is not as complicated as the MySQL implementation because we chose to write python functions to perform the CRUD commands to manipulate the JSON store in firebase. First, we configured our app using our firebase configuration information.

```
6
7 Config = {
8     'apiKey': "AIzaSyDkW3a0Rjj0DW5mB4b9VSYjSQPJTTi4YFs",
9     'authDomain': "dsci551-group47-project.firebaseio.com",
10    'databaseURL': "https://dsci551-group47-project-default-rtdb.firebaseio.com",
11    'projectId': "dsci551-group47-project",
12    'storageBucket': "dsci551-group47-project.appspot.com",
13    'messagingSenderId': "732356400242",
14    'appId': "1:732356400242:web:7c538b48d914d213751eeb",
15    'measurementId': "G-EX56TCKHE7"
16 }

23
24 firebase = pyrebase.initialize_app(Config)
25 database = firebase.database()
26 ## load the csv file to firebase
27 baseURL = 'https://dsci551-group47-project-default-rtdb.firebaseio.com/'
28
29
```

Now we can access the firebase using the variable database with the commands. All the user input is directly from the terminal since we chose not to create a user interface for the firebase implementation, and focus on the MySQL version instead. These functions are mimicking the behaviors of the ls(), mkdir(), put(), cat() commands, etc. For example, to simulate a ls() command, we simply take the user input and pass it into the child() function to get a reference for the location at this specified relative path, access the data through the get().val() function call, and display only the name of the nodes by using keys(). We also have an exception handler for attribute error, which occurs when the user enters an invalid file path.

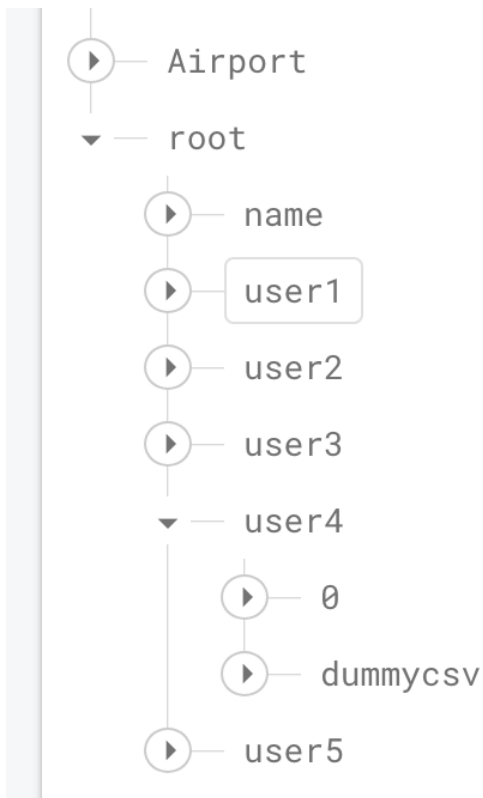
```

40
41 def ls():
42     userfilePath = sys.argv[1]
43     try:
44         nodes = database.child(userfilePath).get().val().keys()
45         print(list(nodes))
46     except AttributeError:
47         print('The path you enter is wrong, cannot perform ls on a file')
48

```

The file structure of the firebase:

The name, user1, user2, user3, and user4 nodes are under the root node, they are created to show proof of concept.



Call the ls() function to list the files under root/user4, the results are displayed as a list of directories/file names, just as it displays in firebase nodes.

```

• (base) kaitlynli@Kaitlyns-MBP firebase demo % python3 firebase.py root/user4
['0', 'dummycsv']

```

Task 2 implementation details

For task 2, we implement some more additional methods to finish our goal.

First of all, we build six additional methods to support the partition mapping and combine them to construct the total output.

To start with the whole process, we first figure out how to query data from each table with conditions. And the first step is to find whether the data we are searching is separated by SQL. To find out we can call the `_query_file_data_partition_info()` function to check. If the table is not partitioned, then we can directly return the output by adding the corresponding search result to the output array. If the table is partitioned, then we first get the data from each partition and concat them into the output['reduce']. We also have a column for output['partition'] in order to show exactly each part of the data from each partition in order to support our output['reduce'].

```
def query_data_from_table(self, table_name, where_dic=None):
    info_list = self._query_file_data_partition_info(table_name)
    output = {
        'partitions': [],
        'reduce': []
    }

    # if table has no partition
    if not info_list or not info_list[0]['PARTITION_NAME']:
        output['reduce'] = self._query_file_data(table_name, where_dic)
    else:
        # table has partitions
        for data in info_list:
            map_data = self.mapPartititon(table_name, data['PARTITION_NAME'], where_dic)
            output['partitions'].append({
                'name': data['PARTITION_NAME'],
                'data': map_data
            })
            # combine data
            output['reduce'] += map_data
    return output
```

To benefit our search and analytic function we later implemented in task3, the `add_where_dic` method takes the search conditions and add them to the normal SQL command: for example, the where part of the SQL commands(select * from table where ...)The part after 'where' is stored in instance 'where_dic'. If there is no additional search condition, then return the normal sql sentence. If there are elements, then add the condition to the normal sql sentence and return the final sql sentence.

```
def _add_where_dic(self, userSQL, where_dic=None):
    if not where_dic:
        return userSQL
    where = []
    for key in where_dic.keys():
        where.append(str.format('{ } { }', key, where_dic[key]))
    if len(where) > 1:
        userSQL += ' where ' + ' and '.join(where)
    else:
        userSQL += ' where ' + where[0]
    return userSQL
```

This method takes the modified sql command from method `_add_where_dic` and outputs the result.

```
def _query_file_data_partition_data(self, table_name, partition_name, where_dic=None):
    userSql = "SELECT * FROM {table} PARTITION({partition_name})".format(table=table_name,
                                                                           partition_name=partition_name)

    userSql = self._add_where_dic(userSql, where_dic)

    self._execute(userSql)
    dataList = self._dataList()
    return dataList
```

The `mapPartition` takes `table_name`, `partition`, and search conditions and returns the resulting output.

```

7 def mapPartititon(self, table_name, partition, where_dic=None):
    partitionData = self._query_file_data_partition_data(table_name, partition, where_dic)
    return partitionData

```

Then the reduce function first checks whether the file exists by calling the method getDataList. Then call the query_data_from_table method to map all the results and finally return the result.

```

7 def reduce(self, file, where_dic=None):
    data_list, file_name, path_id = self.getDataList(file)
    if len(data_list) == 0:
        print("Error: file does not exist")
        return
    file_type = file_name.split(".")
    table_name = file_type[0] + "_" + str(data_list[0]["id"])
    table_data = self.query_data_from_table(table_name, where_dic)
    return table_data['reduce']

```

```

def getDataList(self, file):
    path_list = file.split("/")
    file_name = path_list[-1]
    dir_path = "/".join(path_list[:-1])
    path_id = self._check_dir_path(dir_path)
    if not path_id:
        print("Error: file does not exist")
        return [], file_name, path_id
    dataList = self._query_file_pathId_name(path_id, file_name)
    return dataList, file_name, path_id

```

The result from task 2 is displayed in task 3.

Task 3 implementation details

For task 3, we chose to use HTML and CSS to build the basic framework to show the stimulation of user navigation on our Emulated Distributed File system. To connect

the front end to the back end server written in python, we use the python Flask framework, which is a very efficient micro-framework. Since we need to take in user input, which is the commands (ls, cat, rm, put) from the HTML page, send it to the python backend server for processing and querying data, and send the result back to the front end.



Insert a command

ls /root/user

Enter

2.CSV CARS.CSV AIRPORTS.CSV COPY.CSV COPY1.CSV MYCAR11.CSV

On the front end, we have an HTML form input box where the user can enter a command as they would enter for exploring their file system in the Linux terminal. Once the user hit the enter button, the data will be captured by the form and rendered to the process_command() function in the python program via a post HTTP request.

```
48 <div class='mydiv'>
49   <form action="{{ url_for('process_command') }}" method="post"> Insert a command
50     <input type="text" name="projectFilepath">
51     <input type="submit" value="Enter">
52   </form>
53
54
```

```

# Flask constructor
app = Flask(__name__)
##command_list = []
## once the user hit the submit button, the process_command will get the form data
@app.route('/', methods = ['GET', 'POST'])
def process_command():

    edfs = EDFS("localhost", "root", "Qiqi140624..", 3306)
    if request.method == 'POST':
        k = EDFS_PARTITION("IATA", 2)
        userinput = request.form.get('projectFilepath')
        command = userinput.split()[0]
        path = userinput.split()[1]
        my_input = userinput

```

To display the partition-map-reduce from task 2, we have an HTML input form where the user can select a CSV file, and enter information in the three text boxes to query the corresponding data.

Select file:

condition 1:

Result:

The result will be displayed on the HTML page as follow.

partition p0:

edfs_table_id	IATA	AIRPORT	CITY	STATE	COUNTRY	LATITUDE	LONGITUDE
336	VPS	Eglin Air Force Base	Valparaiso	FL	USA	30.48325	-86.5254
337	WRG	Wrangell	Wrangell	AK	USA	56.48432583	-132.3698242
341	YUM	Yuma MCAS-Yuma International	Yuma	AZ	USA	32.65658333	-114.6059722

partition p1:

edfs_table_id	IATA	AIRPORT	CITY	STATE	COUNTRY	LATITUDE	LONGITUDE
335	VLD	Valdosta Regional	Valdosta	GA	USA	30.7825	-83.27672222
338	WYS	Yellowstone	West Yellowstone	MT	USA	44.68839917	-111.1176375
339	XNA	Northwest Arkansas Regional	Fayetteville Springdale Rogers	AR	USA	36.28186944	-94.30681111
340	YAK	Yakutat	Yakutat	AK	USA	59.50336056	-139.6602261

partition total:

edfs_table_id	IATA	AIRPORT	CITY	STATE	COUNTRY	LATITUDE	LONGITUDE
335	VLD	Valdosta Regional	Valdosta	GA	USA	30.7825	-83.27672222
338	WYS	Yellowstone	West Yellowstone	MT	USA	44.68839917	-111.1176375
339	XNA	Northwest Arkansas Regional	Fayetteville Springdale Rogers	AR	USA	36.28186944	-94.30681111
340	YAK	Yakutat	Yakutat	AK	USA	59.50336056	-139.6602261
336	VPS	Eglin Air Force Base	Valparaiso	FL	USA	30.48325	-86.5254
337	WRG	Wrangell	Wrangell	AK	USA	56.48432583	-132.3698242
341	YUM	Yuma MCAS-Yuma International	Yuma	AZ	USA	32.65658333	-114.6059722

The following code is the javascript code embedded in the HTML code to handle the data that is routed from the partition.

```

83 <script type="text/javascript">
84   var table_name='';
85   var table_columns = []
86   function add_data(partition, is_reduce) {
87     var table = '<div id="data_partition0" style="margin-top:20px;"><label>partition '+ partition
88     if(partition.data.length > 0) {
89       table += '<thead><tr>';
90       for(var i = 0; i < table_columns.length; i++) {
91         table += '<th>' + table_columns[i] + '</th>';
92       }
93       table += '</tr></thead>';
94     }
95     table += '<tbody>';
96     for(var i = 0; i < partition.data.length; i++) {
97       data = partition.data[i]
98       table += '<tr>'
99       for(var j = 0; j < table_columns.length; j++) {
100         table += '<td>' + data[table_columns[j]] + '</td>';
101       }
102       table += '</tr>';
103     }
104     table += '</tbody></table>';
105     if(is_reduce) {
106       $('#data_reduce').append(table);
107     }else{
108       $('#data_partition').append(table);
109     }
110   }
111   $(document).ready(function () {
112     $("#file_name").change(function(){

```

Learning experience

We all think this is a good learning experience, we got to dive deep into the data science concepts (EDFS, Partition Map Reduce) learned in class and built a project, which made our memory and understanding of the topic much deeper. By implementing two different EDFS using MySQL and firebase, we got a much better understanding of the underlying principles and logic of the SQL queries and the firebase CRUD commands. While implementing both of them, the MySQL implementation definitely took a lot more time and effort than the firebase implementation.

The challenge we faced when we implemented task 2 is that it is a little bit hard to understand how the map and reduce function works in the sample wordcount.java

and how to implement our own map and reduce based on the idea. Since I am not very familiar with python, it takes me a while to find out how to put the search conditions together and learn how to separate them in a good manner. I've also learned how SQL stores the data in partitions and the way to access them.

One of the challenges we faced was choosing the right framework for connecting the frontend and backend servers. At first, we ran into failure using the Flask framework because the flask app route cannot be applied to class methods in our python code and can only be used on static methods, so the routing always failed. Then we tried many different other methods, such as python tornado, cgi without a framework, Django, etc but they weren't working as expected. We then went back to using the flask framework and figured out that we could make an instance of the EDFS class and have the instances called the corresponding class methods (ls(),mkdir(), cat(), put(), etc) in the EDFS class, use these new static functions to perform the command tasks and render the result data back to the user interface. This way, the data transmission between the front end and back end is much more efficient and makes our code clear.

Google Drive link to our code:

https://drive.google.com/drive/u/0/folders/1gk7bwCBtqXdHk3woKM7D5_KrxN-mZQ6J

Presentation video link:

https://www.youtube.com/watch?v=PHTWv90Xz24&ab_channel=JiaqiLi