Project Mid-Term Report

“SMART CHOICE”

Emulation-based file system for Distributed File Storage & Parallel Computation

DSCI551-20223-Group- 128

Group Members:

Name	USC ID	Email id
Ankit Tripathi	4612 6769 99	ankittri@usc.edu
Lakshita Shetty	2162 0892 11	lshetty@usc.edu
Shreya Nayak	8592 8104 56	nayakshr@usc.edu

Implementation :

Our goal according to the timeline is to complete working on the HDFS commands and implement it on one EDFS which is MySQL-based emulation in our case. The other EDFS is Firebase which has similar implementation. Created PMR queries to perform search and analytics on the dataset. Creating the main EDFS (with a simple and user friendly HTML front end) and decide the flow of the application. Created few Visualizations to understand the data better.

- We have implemented a total of 7 commands in order to emulate HDFS.

- We also created a table called namenode to store the meta data of the directories created and files uploaded. This table is an emulation of namenode in HDFS.

Our application 'Smart Choice' aims to cater to the varying needs of diverse users and to help them shortlist a movie or a show that they can relax or unwind to after a long tiring day without having to spend a lot of thought on which of the many options they should consider. The user needs to simply choose from the available options whether to check if a particular movie is on Netflix or not.

Task 1 :

A) MySQL-based emulation:

- Built an emulated distributed file system (EDFS)

Similar to HDFS, which stores metadata in namenodes and actual file data in datanodes, our EDFS would store metadata (i.e., file system structure, file attributes, position of file partition, etc.) in a database and partition data of our file in a second database.

Here Python is used with SQL. For the SQL implementation, a total of 4 tables are made:

1. File system structure

- *Primary Key —> parent, child*
- *Foreign Key —> parent and child*
- *Attributes —> parent, child*

In this there will be 2 columns parent and child and combined form the key.

2. Namenode

- *Primary Key —> Id*
- *Attributes —> Id, Path, Type*

In this there will be 3 columns:

- Id – Each item of meta data is given an id. For the table, it is set to AUTO INCREMENT and PRIMARY KEY.
- Path – Path of all the Directories and Files created in the EDFS
- Type – Type of the newly created file/directory.

3. File partition

- *Primary Key —> Id, p_ Id*
- *Foreign Key —> Id*
- *Attributes —> Id, p_ Id*

In this there will be 2 columns Id and p_Id and combined form the key.

4. File partition Table

In this there will be tables with respect to their partition.

EDFS will use the following command:

1. **mkdir** – This function does two tasks with the user-supplied argument name. It does this by first making a new folder or directory. It will then incorporate meta information produced in the namenode.(a file system directory should be created)

```
def makeDir():
    name = input("Enter name of the directory: ")
    os.makedirs(name)
    cursor.execute("use dsci551")
    cursor.execute("insert into namenode2 (path, fileType) values('root/{0}', 'dir')".format(name ))
    db.commit()
```

2. **ls** – This function merely displays all of the current folders and files. It outputs a list of paths to show the list of current files and folders and requires no input from the user.

```
def ls():
    cursor.execute("use dsci551")
    cursor.execute("select path from namenode")
    for x in cursor:
        print(x)
```

3. **cat** – This function accepts a user-supplied file name and outputs the file's content. CSV files will display a data frame in the output, whereas txt files will just display all of the material, much like a nano.

   For example, if the user gives input as ‘Netflix-tites.csv’ as input it will show the data in a pandas data frame.

```
def cat(filename):
    cursor.execute("use dsci551")
    typeFile = cursor.execute("select fileType from namenode2 where name = '{}'".format(filename))
    if typeFile =='csv':
        df= pd.read_csv('{}'.format(filename))
        print(df)
    else:
        if typeFile=='txt':
            with open("{0}".format(filename), encoding = 'utf-8') as f:
                text=f.read()
                print(text)
        else:
            print(f'{sys.argv[0]}:{filename}:The system cannot find the file specified.')
```

4. **rm** – The directory or file that the user specifies is simply deleted by this procedure. It requests the file name from the user as an argument and deletes the file from the namenode and datanode tables.

```
# rm: remove a file from the file system, e.g., rm /user/john/hello.txt
def remove():
        name = input("Enter name of the directory or file you want to delete: ")
        os.chmod(name, 0o777)
        os.rmdir(name)
        cursor.execute("use dsci551")
        cursor.execute("Delete from namenode2 where path='root/{0}'".format(name))
        db.commit()
```

5. **put** – This function will upload a file to file system, will upload a file csv file to the directory in EDFS. It will then update 2 tables. Firstly, it will create an entry in the namenode and then it will store the data in the datanode table.The file will be storing in k partitions, and the file system should remember where the partitions are stored.

6. **getPartitionLoc** – This function will return the locations of partitions of the file that is created by the EDFS.

7. **readPartition** – This function will return the content of partition # of the specified file. The portioned data will be needed in the second task for parallel processing.

Task 2 :
Implement partition-based MapReduce (PMR) for search/analytics

We will implement our mapPartition function as described in the guidelines (Map() function would process every partition and return the desired list. Reduce() function would return an aggregate of outputs from Map())

Examples of search and analytics queries:

We will implement Search queries and Analytics queries on the data sets mentioned above -

for example, searching the movies within a given rating or from a particular genre, comparative study across the 3 platforms etc.

PMR:

The SQL queries that will be used for PMR are listed below. On either the entire dataset or a partition of it, all queries can run and execute. The user will indicate whether they want to run the task on the entire dataset or only a certain segment.

- Searching and returning a list of titles of movies/show who have IMDB Rating greater than user input

  for example:
  user input = 8
  SQL query running at the backend:

  select title
  from Netflix-titles Where tmdb_score>8

- Searching and returning a list of titles of movies/show who have genres by the user

  for example:
  user input = ['drama', 'crime']
  SQL query running at the backend:

  select title
  from Netflix-titles contains genres like ‘drama' and 'crime'

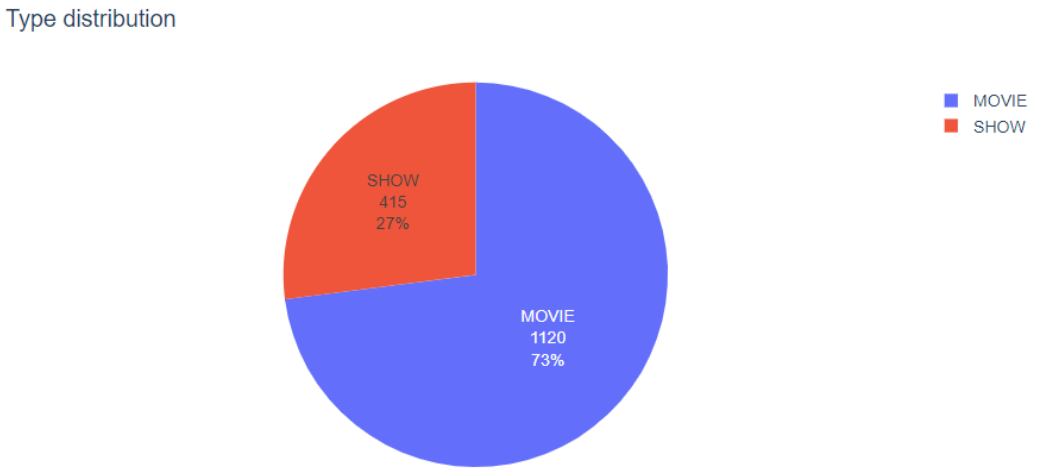
- Searching and returning a list of titles of movies from who have IMDB Rating 8 and genres by the user.

  for example:
  user input = ['drama', 'crime']
  SQL query running at the backend:

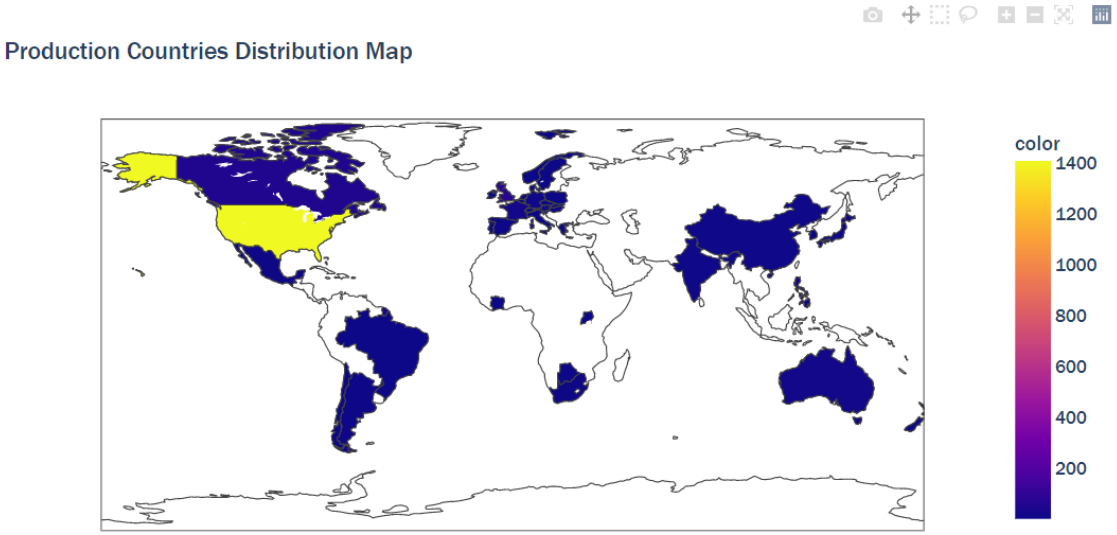
  select title
  from Netflix-titles\movies Where mdb_score>8 Group by genres

Data Visualization:

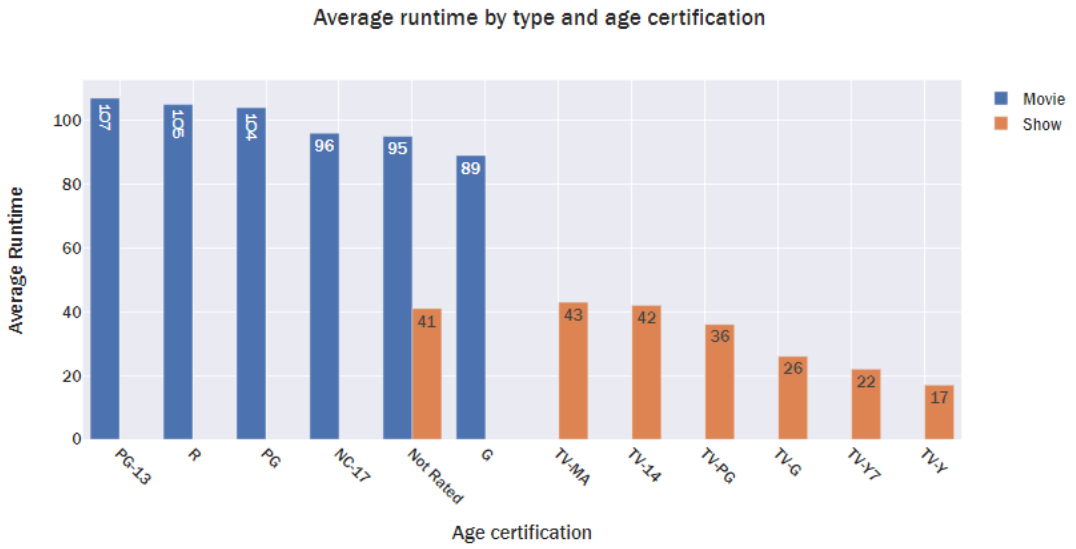
```
In [18]: type_count = df_disney["type"].value_counts()

type_fig = px.pie(values=type_count.values,names=type_count.index,template="plotly_white",title="Type distribution")
type_fig.update_traces(textinfo='label+percent+value')
type_fig.update_layout(font = dict(size= 15, family="Arial"))
type_fig.show()
```



From the piechart it can be seen that 36% data is Show and 64% data is Movie.



United States has the yellow color, which means, for this dataset, the United States produces the most of shows and movies.



For movies, the age certification of PG-13 has the highest average runtime of 107 mins, the age certification of G has the lowest average runtime of 89 mins.

For shows, the age certification of TV-MA has the highest average runtime of 43 mins, the age certification of TV-Y has the lowest average runtime of 17 mins.

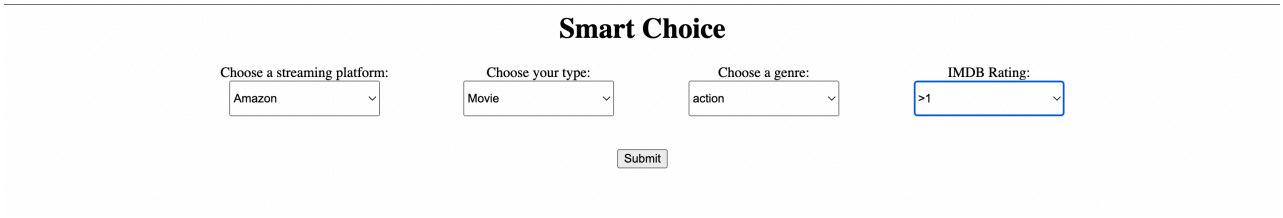
Task 3 :
Creating an app that uses the implemented PMR methods.

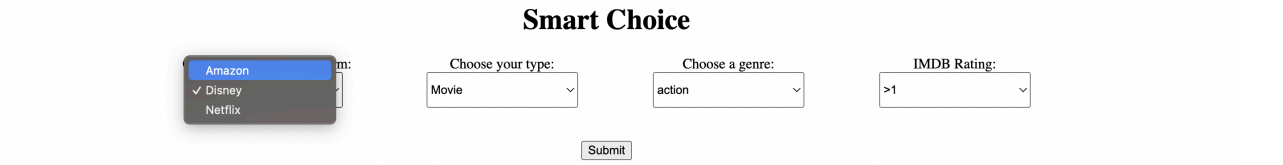
The web application is user-friendly interface that the user can use to achieve the above-mentioned goals. Currently we have achieved the basic frame of the website we will connect is to both the databased and integrate it using Flask framework of Python.

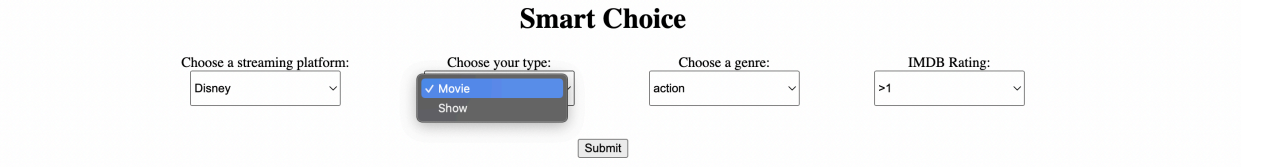
The application will allow users to select from any of the databases listed via the input/choose box. They can then choose their interest-specific options from a wide range of selection criteria, which can subsequently be analyzed/searched within the database of their choice.

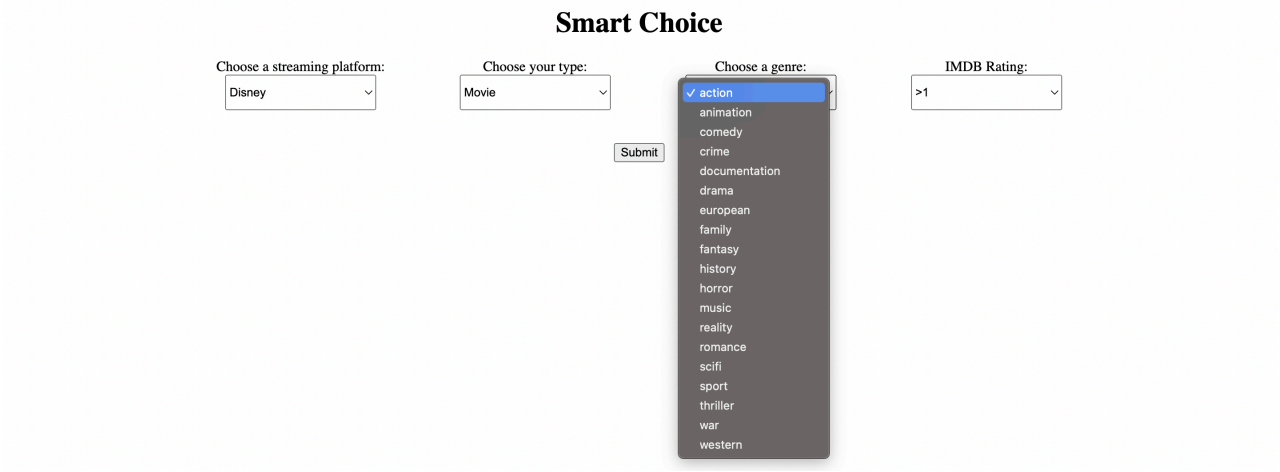
The user will first select the database to which they want to gain access, and then they will be able to execute the following functions:

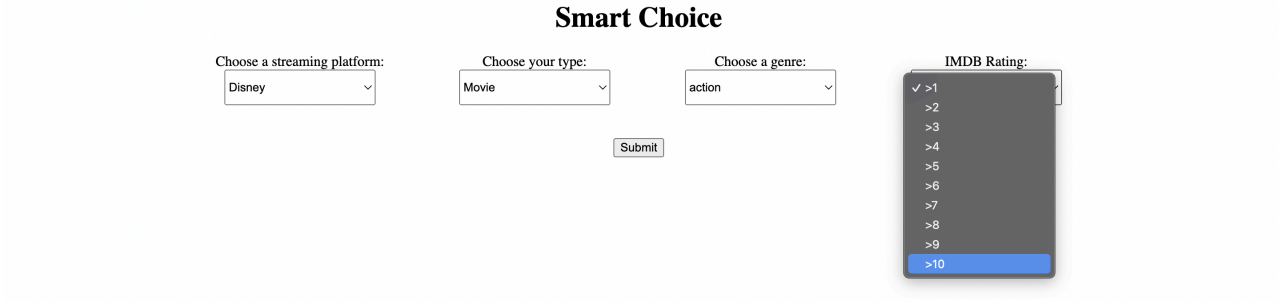
1. Filter movies and TV Shows based on IMDB ratings.
2. Filter the content by Genre (Action, Thriller, Comedy, Drama)
3. The OTT platform where the corresponding Tv show, Movie is available.











HTML Wireframe: