Kseniia Romanova, Keaton Mangone, Jackson Lundberg

CS/DS 4433 - Big Data Management & Analytics
Project 2 - Report

Data Set Preparation

The team used data from Project 1 to test Apache Pig solutions. We created a project folder and uploaded generated data into the Hadoop system using the following commands:

hdfs dfs -mkdir /project2

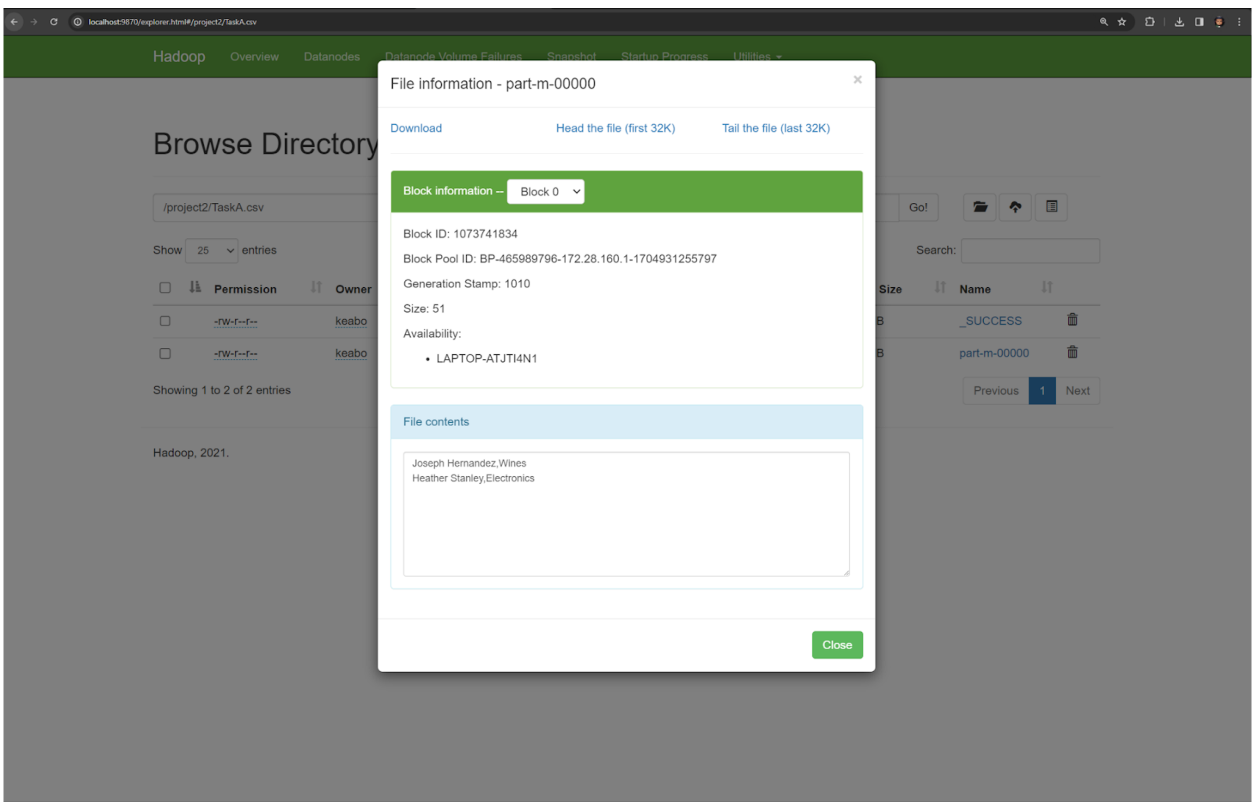
hdfs dfs -put access_logs.csv friends.csv pages.csv /project2/

Our team tested the solutions locally and then switched to HDFS paths using IntelliJ. Please make sure to put the files such that they are accessible on localhost:9000 on the hdfs.

Analytics Queries Logistics

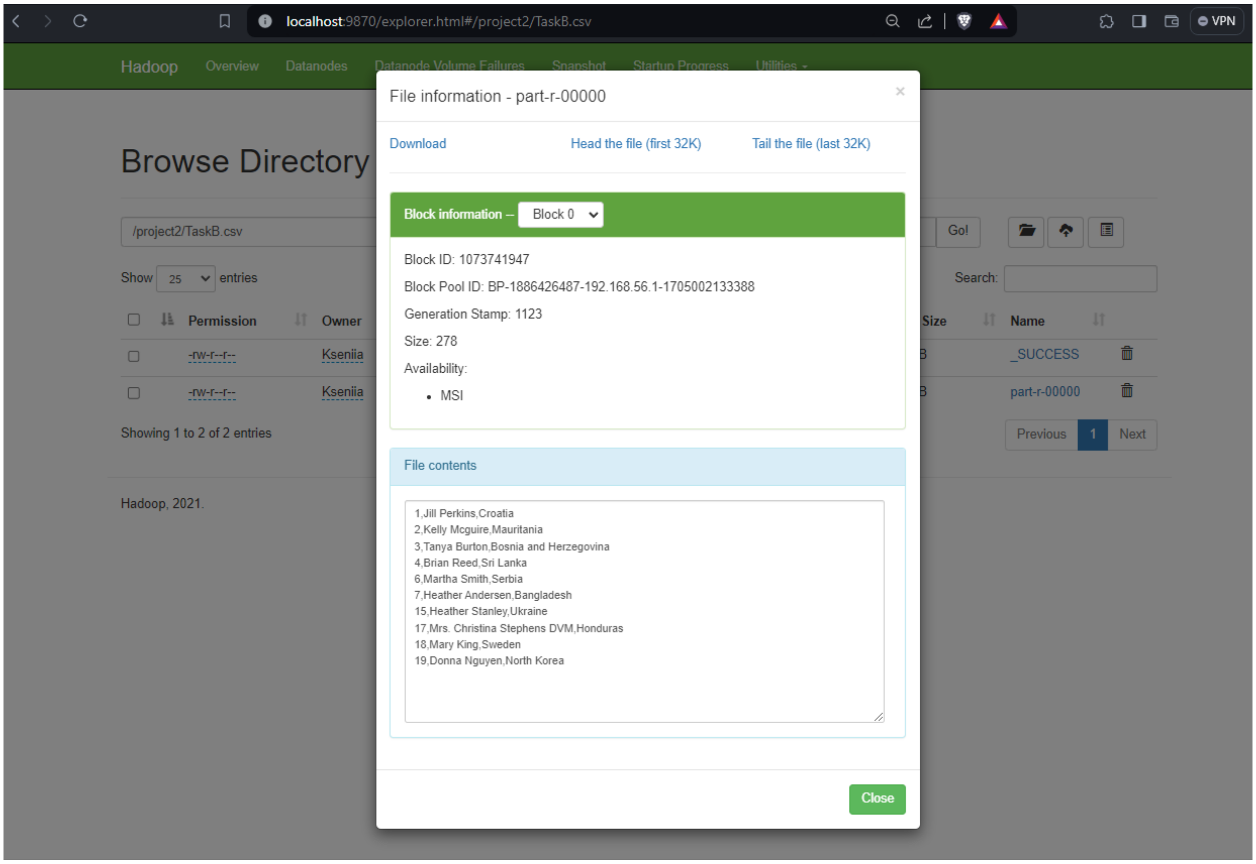
1. **Task A**

   Task A was accomplished in Apache Pig by first loading the pages.csv file from HDFS. This data is then filtered so that only data loaded into the attribute “nationality” with the value “Ukraine” are returned. From this new filtered data set, we want to select (using foreach + generate) a new dataset that is the Name and Hobbies of each tuple in the nationality dataset. This is then stored back in HDFS, labeled TaskA.csv. Using pig -x taskA.pig, the file takes 3.070 to complete. Compared to the 2.372 seconds to run the Java map function, this is significantly slower.



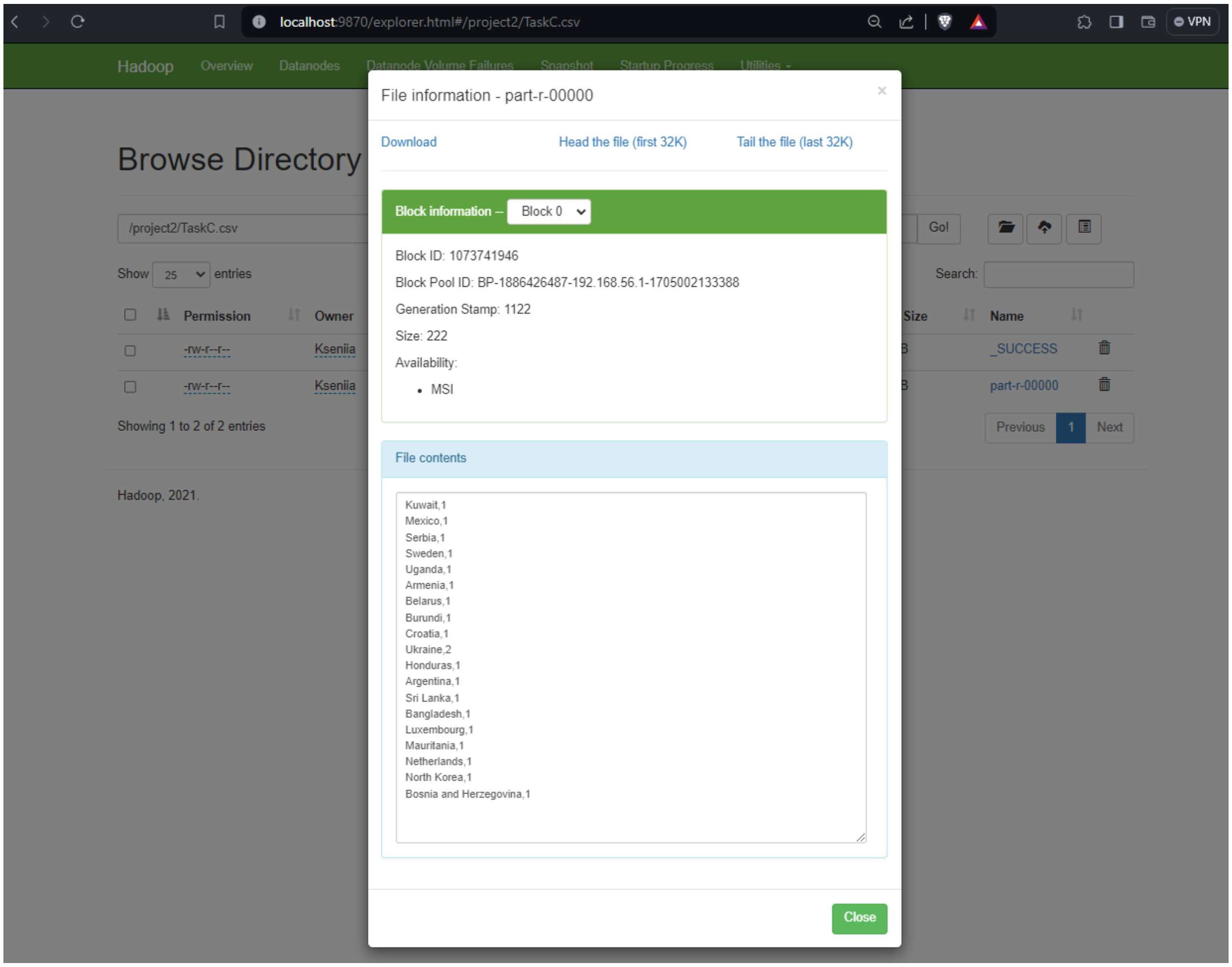
2. **Task B**

   Task B was accomplished by first loading in the access_logs.csv & pages.csv file from HDFS into variables. Pages are cleaned to only be the personid, nationality, and name of a page. The access logs are grouped on What Page was accessed. Then we count the amount of times a page is accessed. After that we order this list in descending order, limit the length to only top 10, and join this data on the page information to output the appropriate page information we need. This task takes 6.414 seconds to complete. The output is stored in HDFS. Compared to the 2.842 seconds to run the java mapreduce function, this is significantly slower.



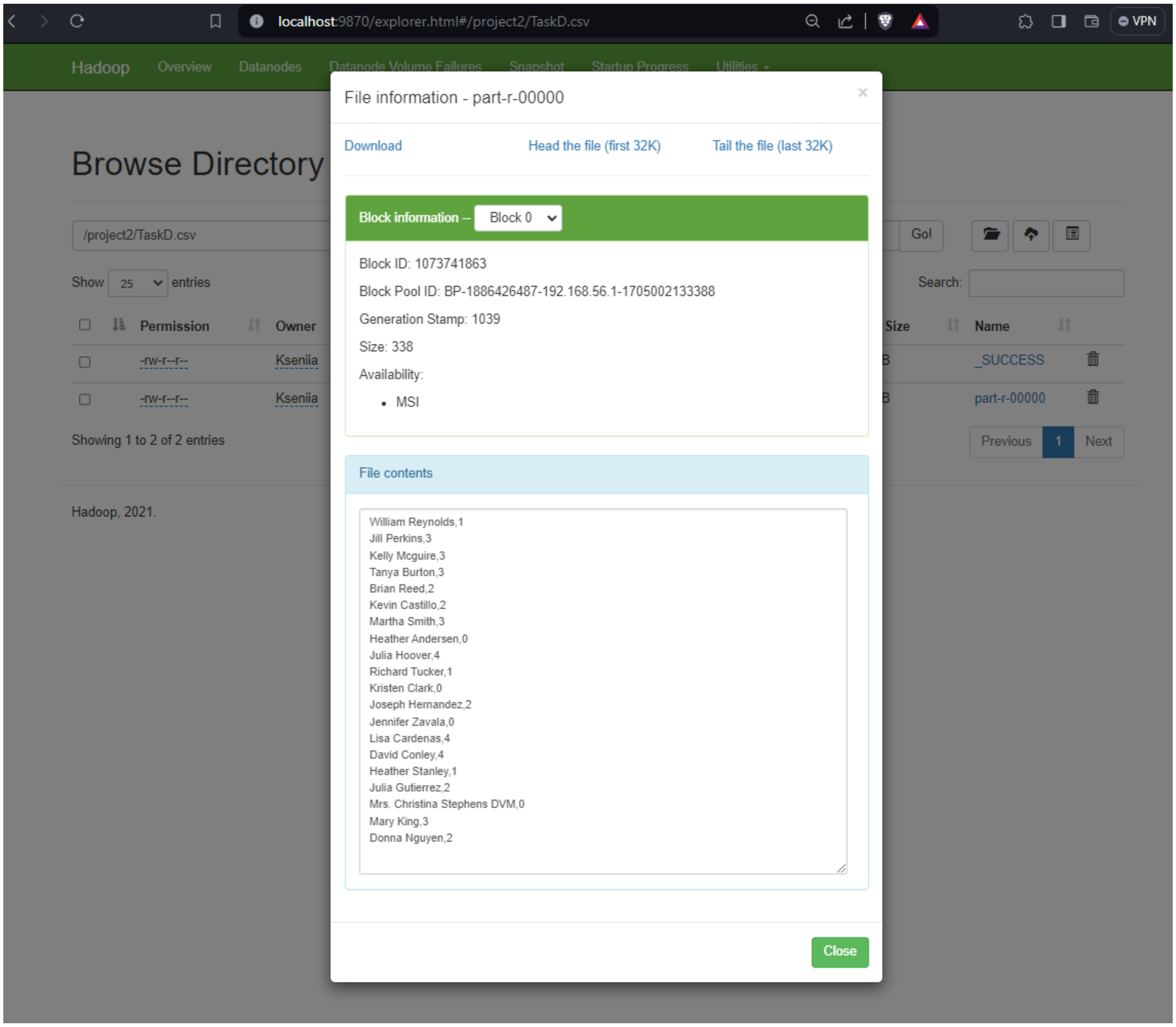
3. **Task C**

Task C was implemented by first loading in pages.csv from HDFS. This data is then shrunk to the personID and Nationality of each tuple. Then, the first “headers" line is omitted using a filter. This data is then grouped by Nationalities of tuples, and then for each tuple we generate the Nationality and count of pages that have this Nationality. This is then stored in HDFS. The job took 3.398 seconds to complete. Compared to the mapreduce java code execution time of 2.684 seconds, this pig execution time is significantly slower.



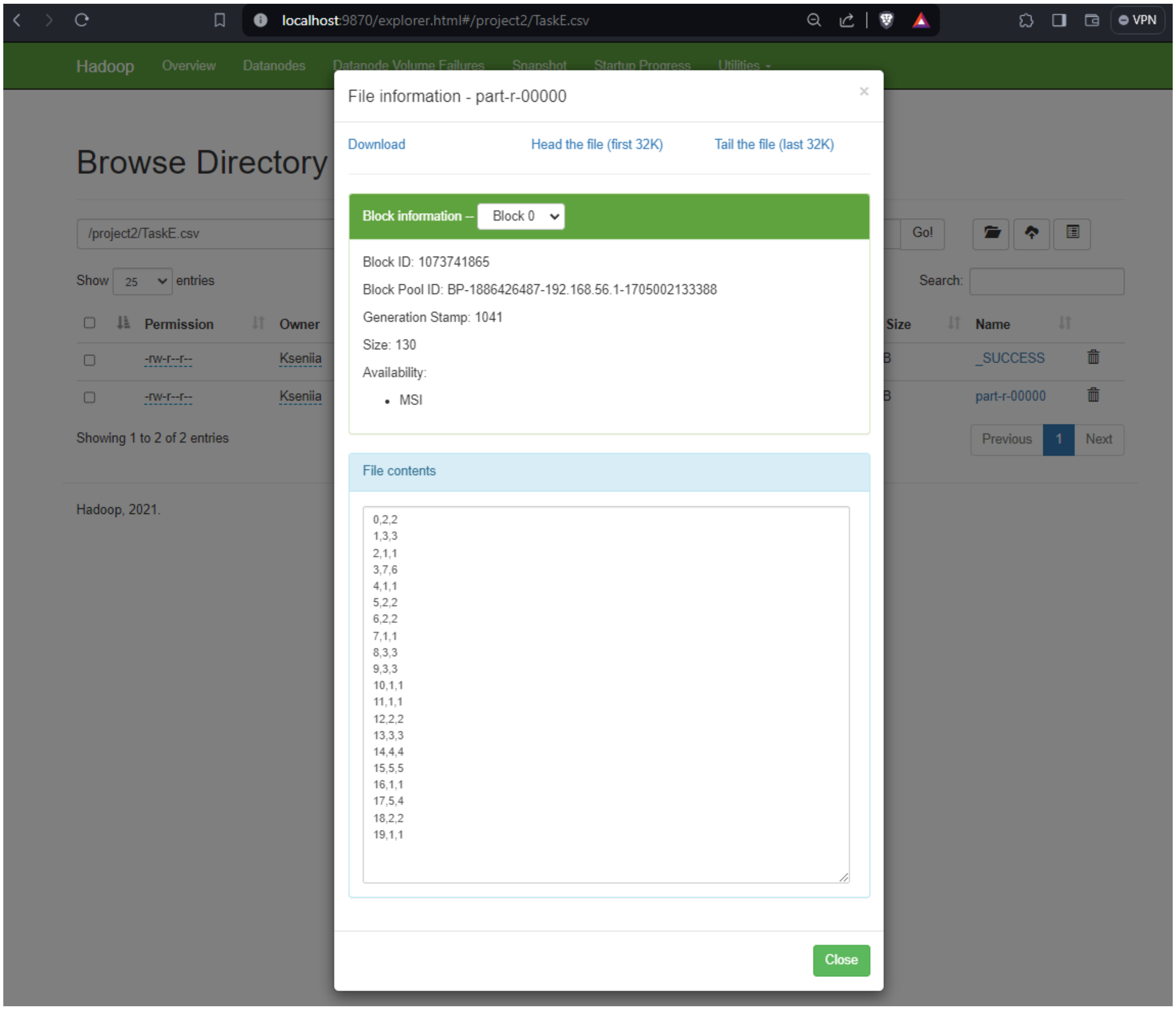
4. Task D

For task D, both friends and pages data sets were loaded into the system. We then cleaned the unnecessary columns and removed header tuples. After that friends data was grouped by MyFriend column and then a count was calculated for how many followers each person has. Then a left outer join between pages and counts was made, so every page would have a count, even if it's 0. The final output tuples are in the form: person's id, count of followers. The entire solution took 4.481 seconds, which is longer than Java's map-reduce solution (3.064 seconds).



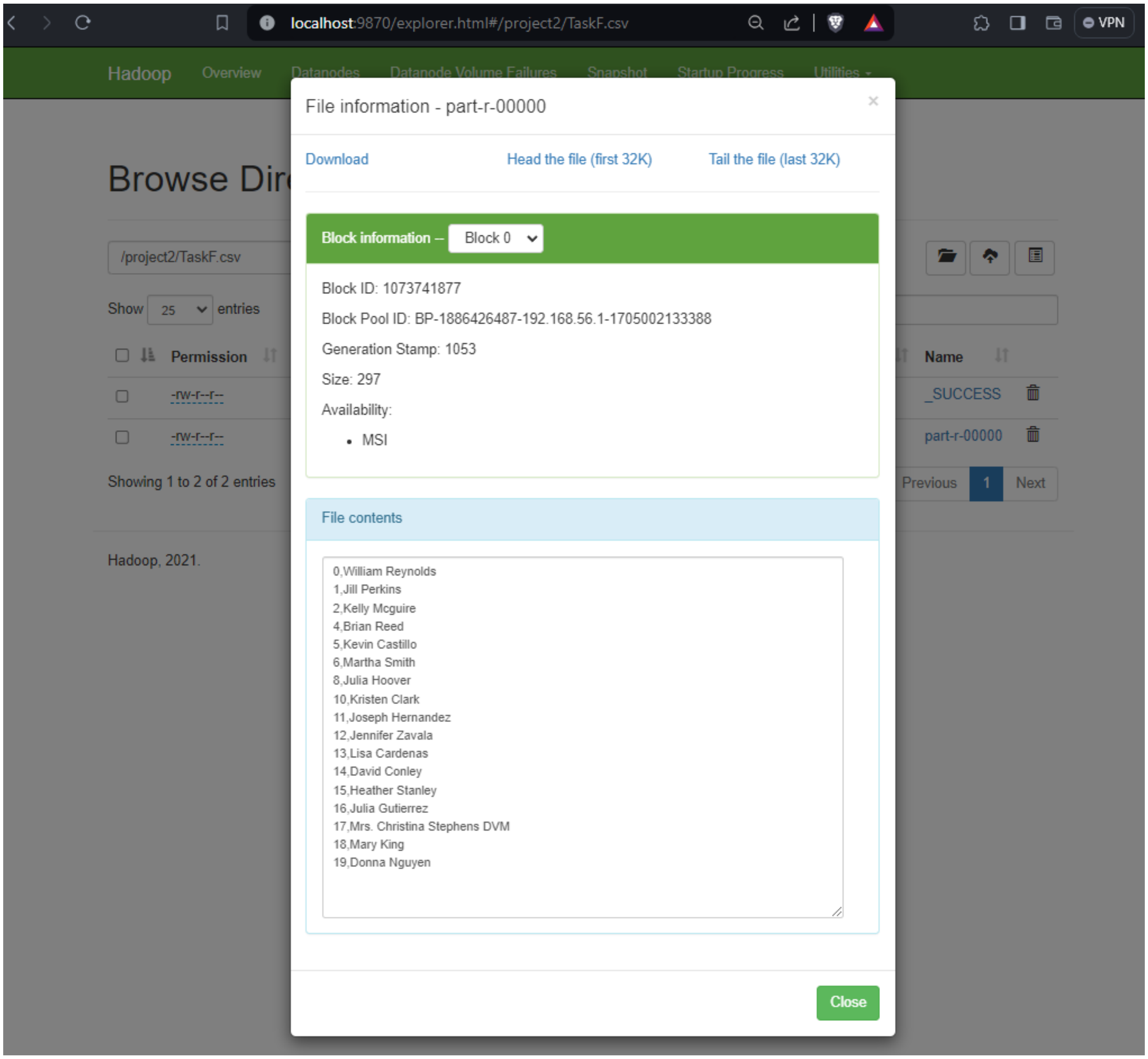
5. **Task E**

For task E we assume that Facebook page owners who are involved in this case should have at least one visit to a page. In this case, we do not need to take care of pages who never accessed anyone's page. Access logs data set was loaded into the system and unnecessary columns and header tuples were removed. Access logs were then grouped by ByWho column and the result was used to compute the final output. We calculated the total count of access by using COUNT, and the distinct number of access by using a combination of DISTINCT and COUNT. The final output is in the form: person's id, total access, unique accesses. The solution took 3.686 seconds, which is longer than Java's map-reduce solution (2.704 seconds).



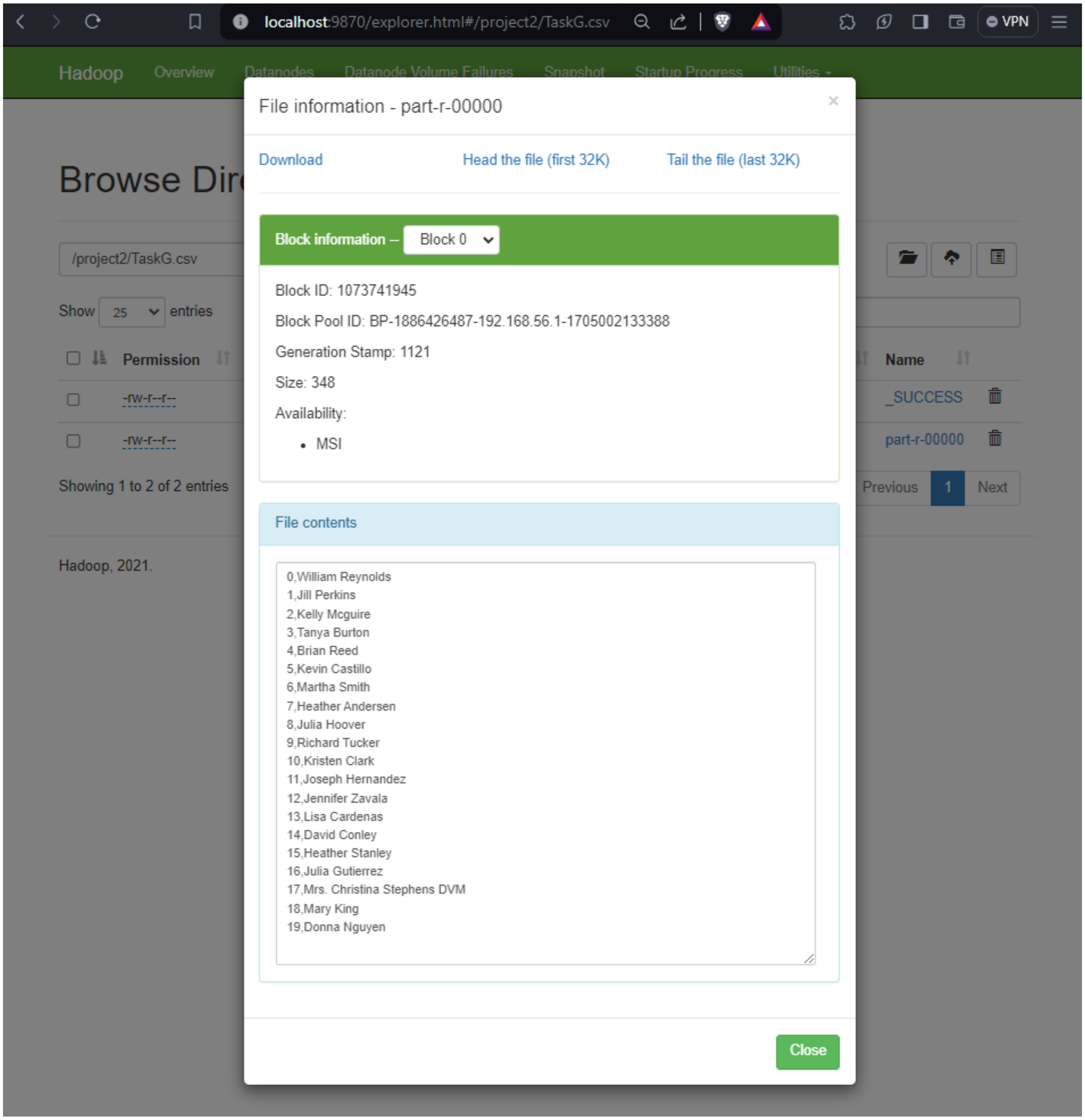
6. Task F

For task F, we loaded all three data sets into the system: access_logs, friends, and pages. We filter out all header tuples as well as unnecessary columns. After that we made a left outer join on friends and access logs data set to find people who declared someone a friend but never accessed their page, meaning that PersonID,MyFriend combination is in friends data set but not in access logs data set (ByWho,WhatPage). We then used filters to remove not null ids and grouped our results by person's id. Then we performed a join on the previous result using pages, so we could get the name of each person. The final output is in the form: person's id, name. The solution took 5.340 seconds to complete, which is longer than Java's map-reduce solution (3.428 seconds).



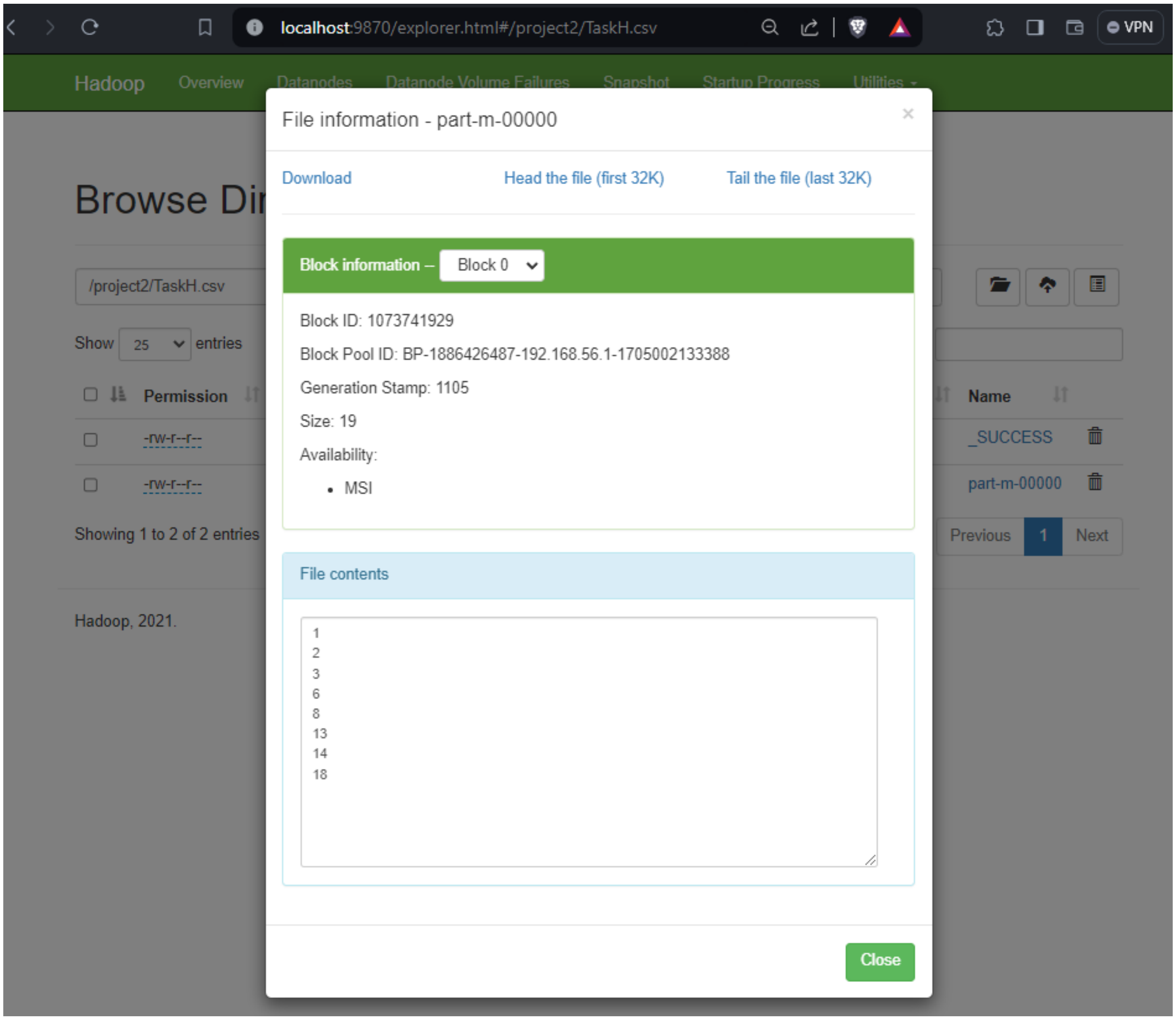
7. **Task G**

Task G starts with loading access_logs and pages into the system and removing unnecessary columns and header tuples. Then, we create a group of access logs for PersonID, so we can see all of the time logs for each person's id. After that, we generate new data to find the max date (the latest access date-time). This lets us filter tuples using days between function and current time function. Then we join the resulting data with pages to get people's names and output their id and name. The solution took 4.592 seconds, which is longer than java's map-reduce solution (2.868 seconds).



8. **Task H**

For task H we assume that every page has at least one follower, so we don't have to count pages which are not in the friends.csv. First, we load friends.csv data set into the system. Then we filter out the unnecessary columns and header. After that we group based on MyFriend, so we can count up all of the followers for this id. Then, we group by ALL to calculate the average. Finally, we filter out people who have below average follower count and output person's id. This solution took 4.963 seconds, which is longer than java's map-reduce solution which only took 2.716 seconds.



K-Means Clustering Algorithm

Assumptions:

- All coordinates are stored as int values (after centroid calculation double value is converted to int, example: 2065.032 = 2065).
- If the cluster doesn't have any points near it then it will be removed and not outputted.
- Centroids might have duplicates and are generated from data points using dataGenerator class, duplicate centroids will be removed since all points will be assigned to the first duplicate.

To run the code dataset.csv, friends.csv, access_logs.csv, pages.csv need to be in hadoop server. Otherwise, change the paths to use local versions of the files.

A. Single-Iteration K-Means

We pass k number in args[0] or we can initialize it in the beginning of the main() function. Then, we generate random centroids and store them on hadoop. After that our team wrote configuration for the map-reduce job and cached centroid file, since it is a small data set.

On the map side, we used the setup function to read cached centroids and store them in memory. In the map function, we read each coordinate stored in our big data set and calculate the distance from this coordinate point to all centroids and pick the shortest one. Then, we output the key as centroid coordinate and value as point coordinate.

On the reduce side, we iterate through all points for each centroid and calculate the mean for x-coordinate and y-coordinate. Then, we output a new centroid coordinate as a key and nullwritable as value.

B. Multi-Iteration K-Means

The logic for the multi-iteration k-means algorithm is pretty much the same. The only difference is that we set up a loop in the main() function that runs map-reduce jobs multiple times, in our case it's 6 iterations. After centroids.csv has been created and stored locally in the data folder, we cache this file to use during our first iteration. During consequent iterations we cache output from the previous job to update our centroids.

C. Convergent K-Means

This version of k-means builds on top of the multi-iteration k-means that we created before. It has a for loop that terminates after 20 iterations. It also has an if statement inside the loop that checks whether previous centroids are similar to the centroids calculated during current iteration. If they are similar, then the centroids converge and we can break out of the for loop and be done.

For this version we changed reducer's output value which was changed from NullWritable to Text. It now outputs the same key and the value is equal to "yes" if the centroid converged or "no" if it didn't converge. Meaning that if one centroid didn't converge, then we will need another iteration. The threshold for converging is set to 1 and can be changed.

D. Optimized K-Means

In terms of optimization, we can use combiner and singular reducer to optimize the solution in part C. We added combiner class which outputs key as key and values as sumX, sumY, and count, which is then passed down to reducer. We used a singular reducer since the amount of data passed to one reducer is small due to combiners. The logic is very similar to what we had previously, but with addition of the combiner that sums up x-coordinates, y-coordinates, and total count of points.

E. Different output for Part D

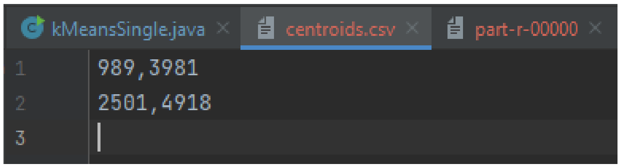
We already output cluster centers along with an indication if convergence has been reached for each center in Part D. We can do this directly in the reducer, which simplifies the process of identifying if another iteration is needed. Since we have our previous center available to us, we can compute the new center and see if they are equal or within threshold, in our case the threshold is equal to 1. If the new center is within threshold, we output the value of "yes", if it's not, then we output value of "no".

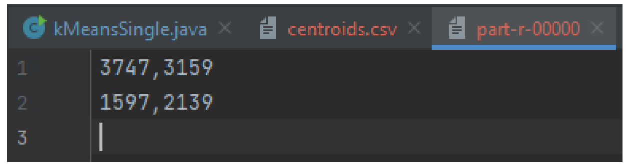
In order to output center as a key and value as points around this center, we needed to store all points in the combiner output value and then perform a split in the reducer using two delimiters. The first one is ";" to separate points and our calculations for the means. And then use "," to separate the calculations. The final output of the reducer will look something like this "1657,4018   627,3460 | 2687,4576". In this example 1657,4018 is the center and 627,3460 and 2687,4576 are points around the center.

F. Comparison on large file (3000 points)

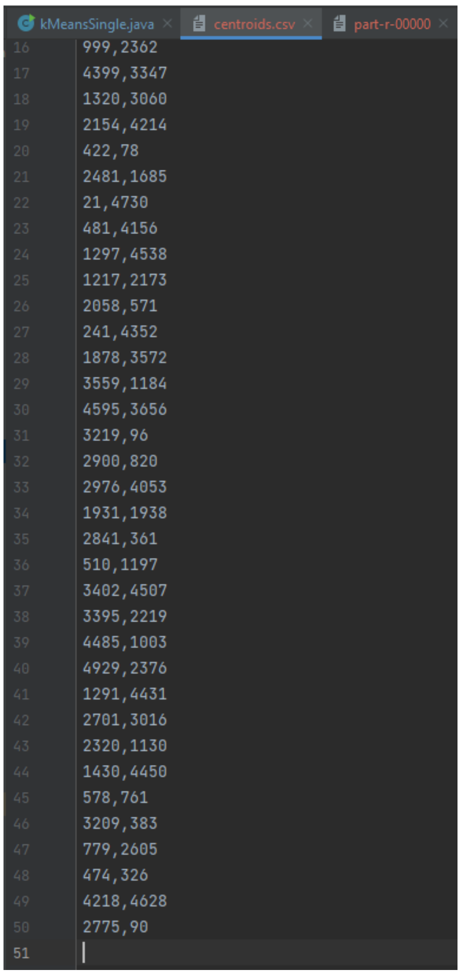
a. Single-Iteration

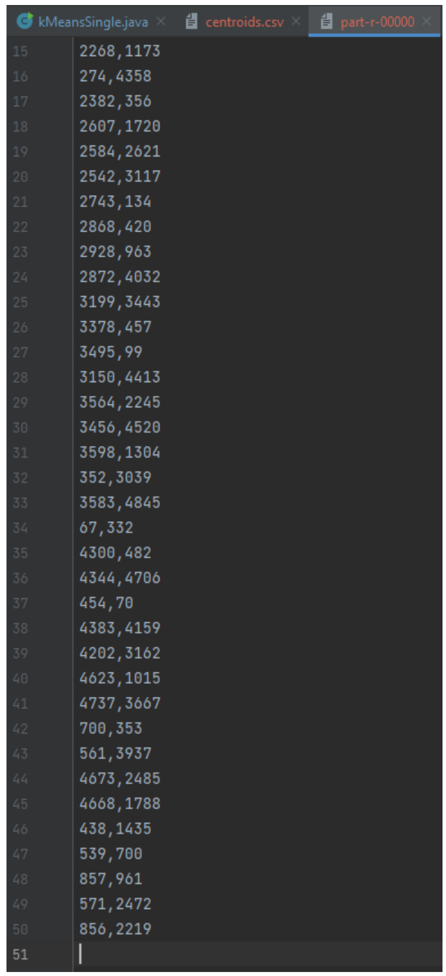
K = 2. The solution took 3.113 seconds.





K = 50. The solution took 3.156 seconds.

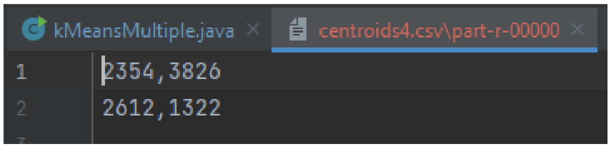


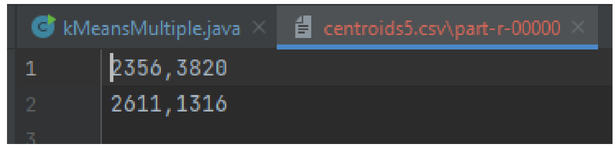


Overall, k doesn't change the run time of this solution by much. However, since it's a single iteration k-means algorithm, the means that we get at the end might not be the actual centers of the data points that we were given.

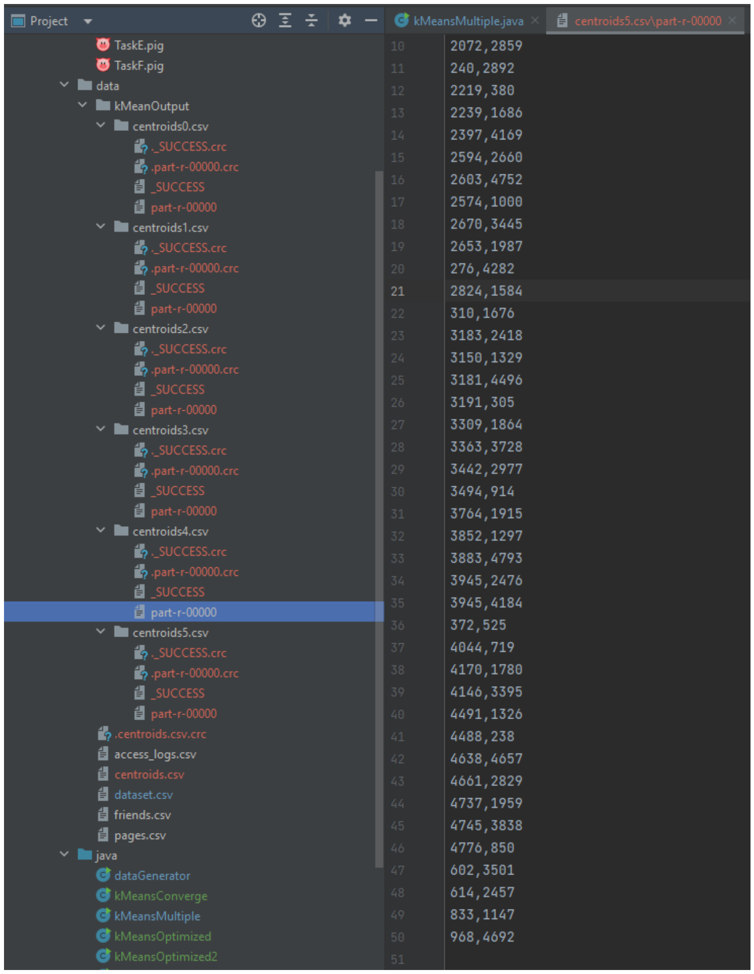
b. Multi-Iteration

K = 2 and R = 6. The solution took 10.729 seconds.





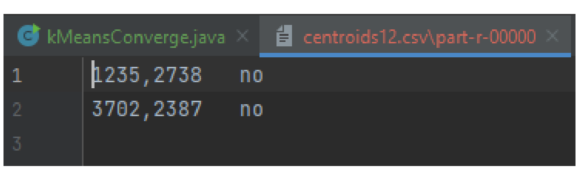
K = 50 and R = 6. The solution took 11.457 seconds.

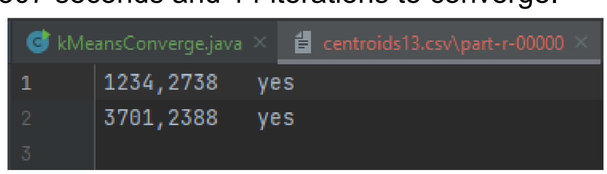


Since we have iterations set to 6, the solution takes longer (Larger R => Longer Time). Even if centers converge early, we will still have to run 6 full iterations. In the case of K = 2 and R = 6, the numbers almost converge, however, the difference between previous x,y and new x,y is still bigger than the acceptable threshold of 1 (can be changed).

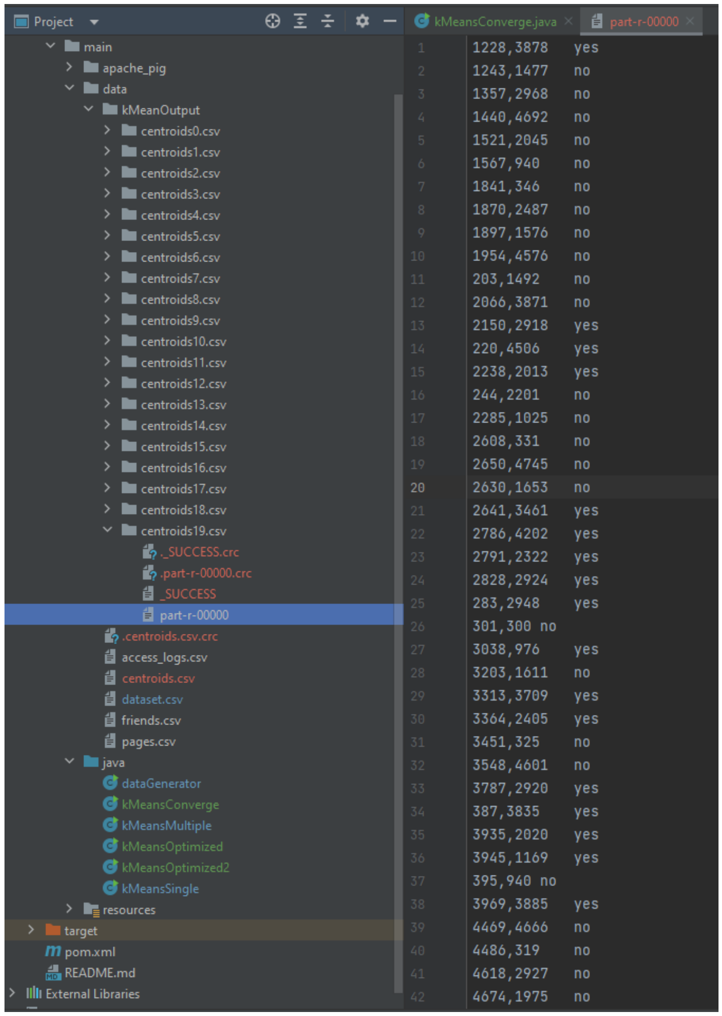
c. Convergent Version

K = 2 and R = 20. The solution took 24.307 seconds and 14 iterations to converge.





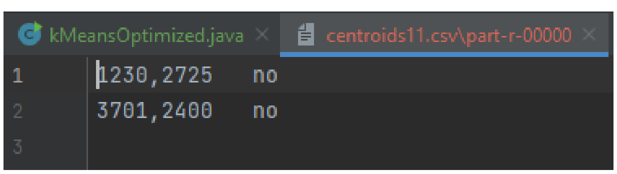
K = 50 and R = 20. The solution took 34.009 seconds and 20 iterations didn't converge.

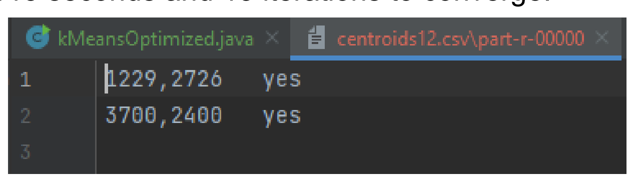


From the provided image it's clear that some centroids did converge, but others did not. In this case more iterations are needed for better results. This solution also took much longer than the previous one, meaning that the bigger K we have, the more time it will take.

d. Optimized Version

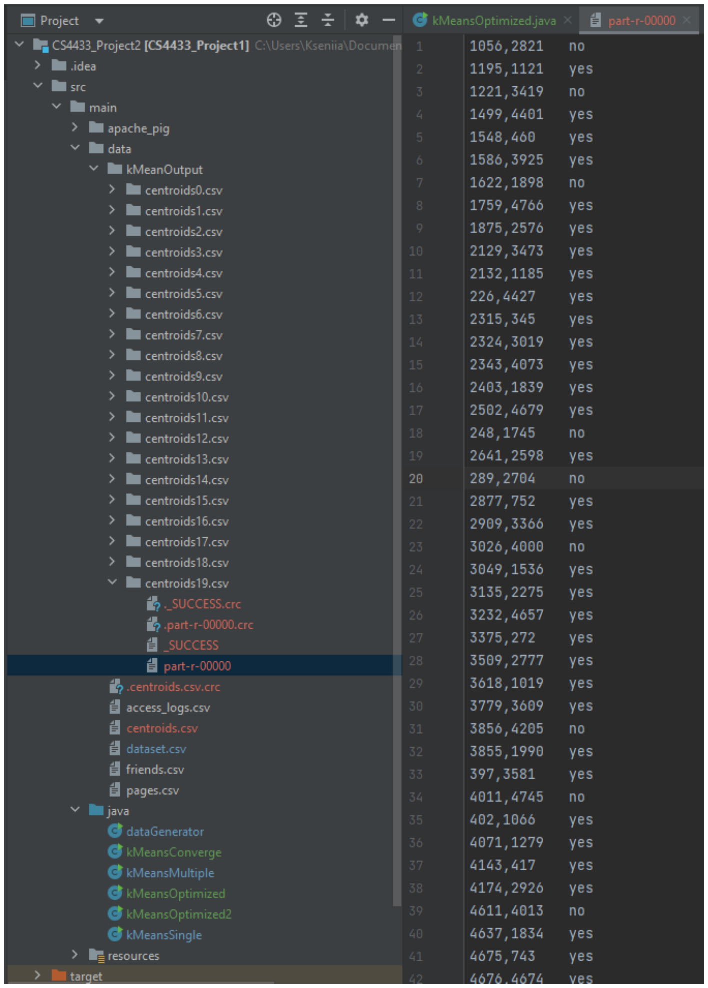
K = 2 and R = 20. The solution took 22.615 seconds and 13 iterations to converge.





Considering this solution took almost the same amount of iterations to converge, it is faster than a simple convergent solution. This means that optimizations that were made are working and making our solution faster.

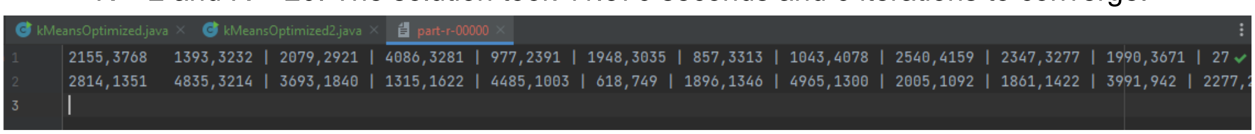
K = 50 and R = 20. The solution took 33.785 seconds and 20 iterations didn't converge.



This solution also took less time to execute than a simple convergent solution. In this case the centroids almost converged, however, there are still some centers that need more iterations to converge.

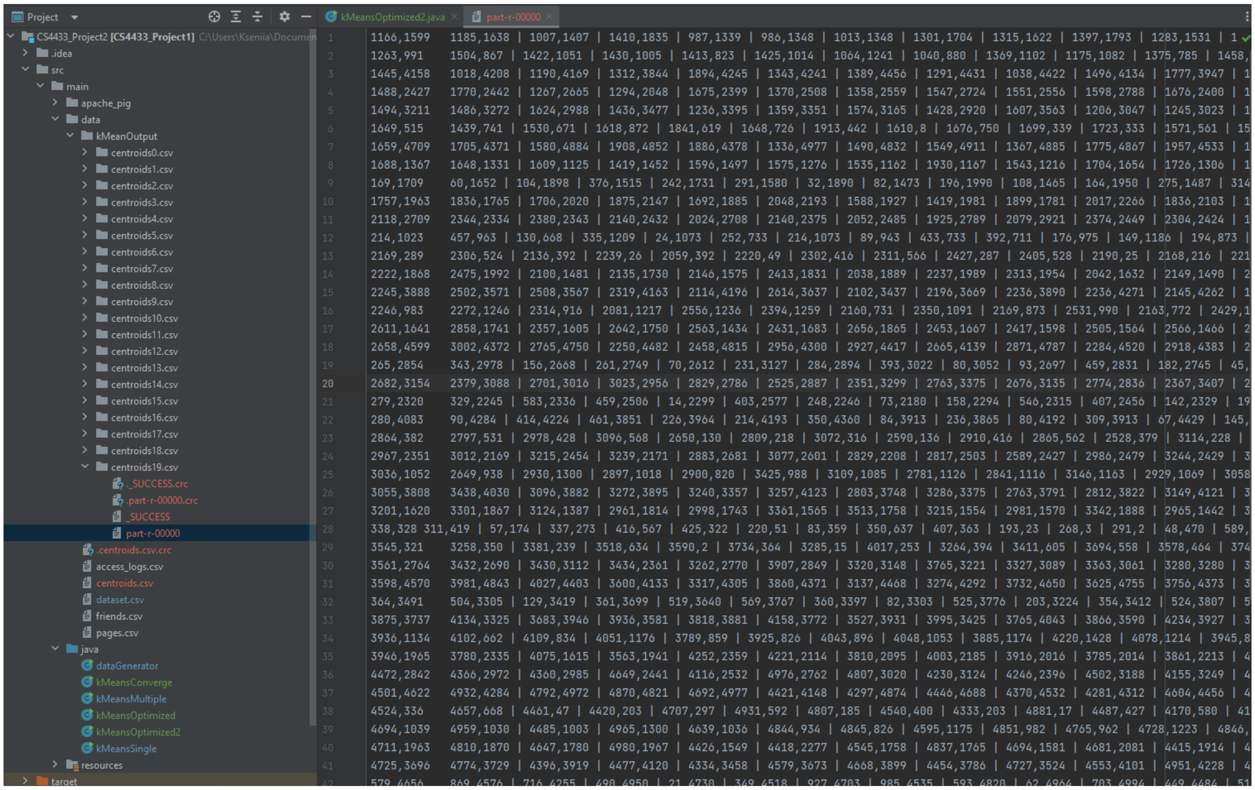
e. Different Output

K = 2 and R = 20. The solution took 11.376 seconds and 6 iterations to converge.



This solution has the same logic as the previous one, however, the output value is different. It lists all of the points which are around a particular centroid.

K = 50 and R = 20. This solution took 33.896 seconds and 20 iterations didn't converge.



By looking at this solution and previous ones for K = 50 and R = 20, it's clear that 20 iterations is usually not enough to get all convergent centroids. The run time of this solution is not much different from the one in part d.

f. Bonus K-means

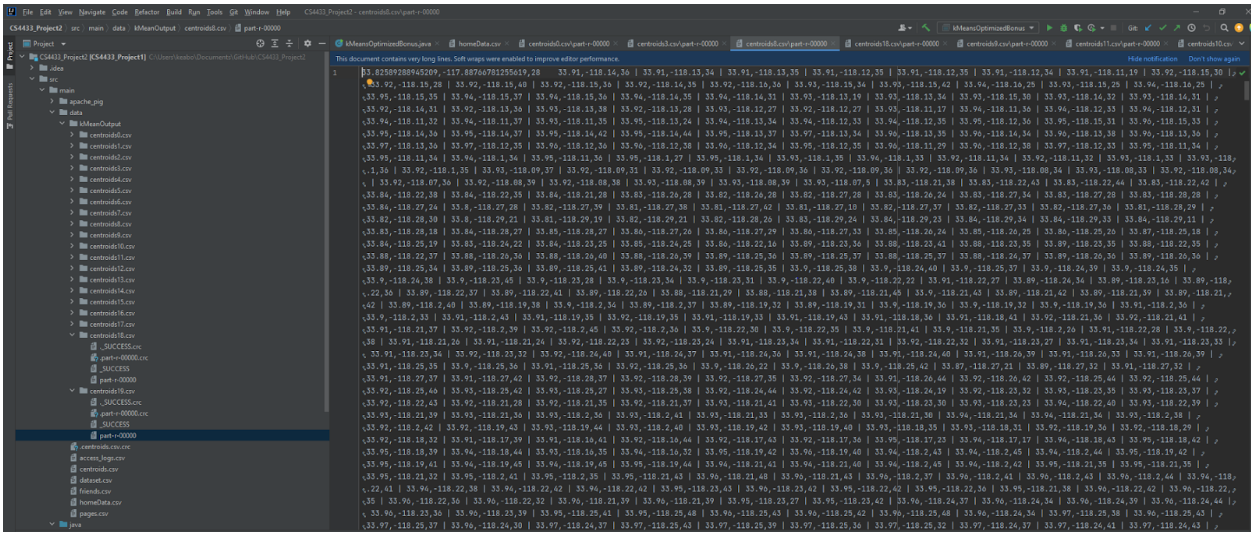
The dataset we chose to implement a new iteration of K-means clustering around was a very large dataset of about 20,000 tuples regarding housing information for houses in California. I found this dataset on Kaggle. Though there is a lot of information about the houses in the dataset, I wanted to take advantage of the house latitude (x), longitude (y), and the median age of the household – this file can be found as homeData.csv in our submission.

This data could be applied in the real world to analyze data based on clusters of housing in an area, calculating average ages, income, and more data based on these clusters for realtors to convey to future house buyers or other realty applications.

With the new implementation, I wanted centroids to also contain the average age of houses clustered and output this alongside the longitude and latitude averages of the house, though the age did not have any factor into how points are clustered, that solely based on Latitude and Longitude.

For this, we also created a new data generator to fit the latitude & latitude values (doubles), so that centroids work properly.

With this new implementation, running the clustering with R = 20 and K = 5, it takes 63.49 seconds, and converges by file centroids18.csv (19 iterations).



Contribution Statement

Skills

Kseniia Romanova:

Prior to working on this project, I had some knowledge about SQL queries from Database Systems I, as well as map-reduce jobs from project 1. After this project, I familiarized myself with the k-means algorithm and Apache Pig.

Keaton Mangone:

Jackson Lundberg:

Before working on this project, I was knowledgeable about SQL queries and getting familiar with working with hadoop and had just learned about the topics of apache pig and k-means algorithms that were covered in class. After this project, I am much more knowledgeable and comfortable using pig, hadoop, and working with k-means algorithms.

Contributions

Before starting to work on this project, our team decided on how to split work between all members equally. We came up with the solution that the tasks could be split between members of the team, and then the team assembled and checked the work that has been done.

- Kseniia Romanova:
  - Created report
  - Completed tasks D, E, F
  - Completed k-means algorithm
- Keaton Mangone:
  - Completed tasks A, B, C
  - Helped with k-means algorithm
  - Completed bonus k-means clustering adjustments
  - Assisted with tasks G, H
- Jackson Lundberg:

    - Completed tasks G, H and reviewed all tasks
    - Helped with k-means algorithm

Resource Usage Statement

Our team used resources provided by the professor and TAs on canvas, such as discussion boards, helpful links, and tutorials.

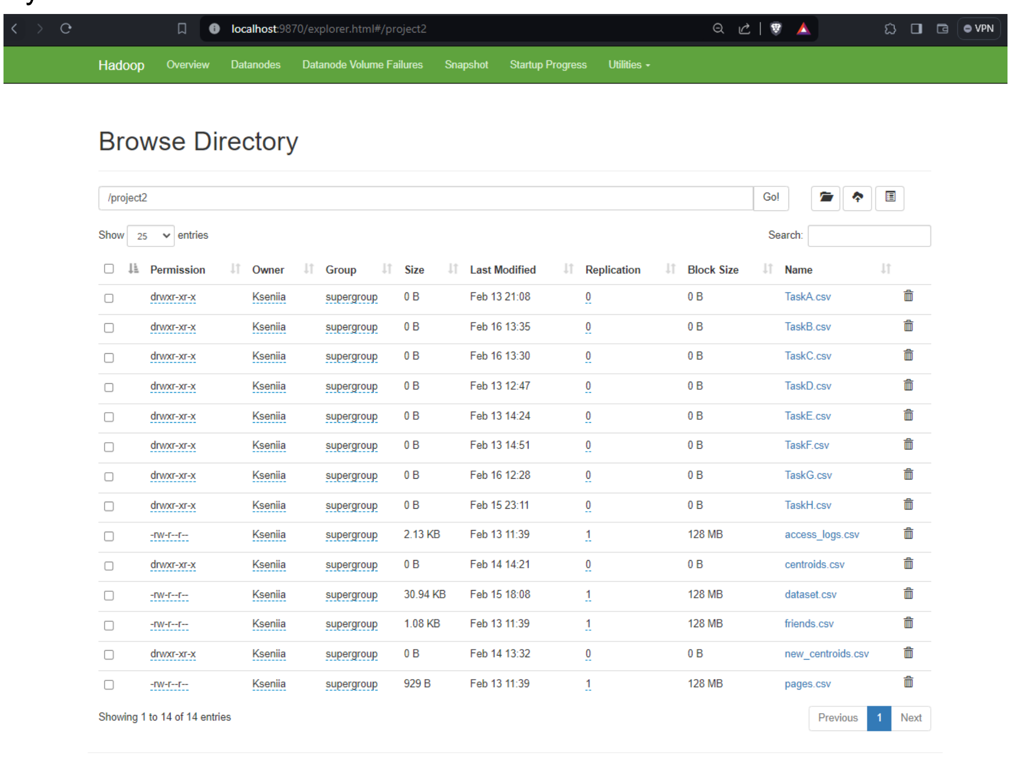
Remove file from hadoop: hdfs dfs -rm -R /project2/TaskC.csv

Run file locally for apache pig: pig -x local C:\Users\Kseniia\Documents\GitHub\CS4433_Project2\src\main\apache_pig\taskC.pig

- Kseniia Romanova:
    - I used resources provided in the discussion board.
- Keaton Mangone:
    - I used resources provided in the discussion board.
- Jackson Lundberg:
    - I used the resources provided by the professor and TAs on canvas, such as discussion boards, helpful links, and tutorials to help me with this assignment. I used the pig documentation to help me learn more about working with apache pig on my computer as well.
    - https://pig.apache.org/docs/latest/func.html
    - https://pig.apache.org/docs/latest/basic.html

Credits

We provided screenshots for each task as well as the screenshot of our hadoop directory containing all tasks. Each member of the team was able to run all the tasks successfully.



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 13 21:08	0	0 B	TaskA.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 16 13:35	0	0 B	TaskB.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 16 13:30	0	0 B	TaskC.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 13 12:47	0	0 B	TaskD.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 13 14:24	0	0 B	TaskE.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 13 14:51	0	0 B	TaskF.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 16 12:28	0	0 B	TaskG.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 15 23:11	0	0 B	TaskH.csv
-rw-r--r--	Kseniia	supergroup	2.13 KB	Feb 13 11:39	1	128 MB	access_logs.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 14 14:21	0	0 B	centroids.csv
-rw-r--r--	Kseniia	supergroup	30.94 KB	Feb 15 18:08	1	128 MB	dataset.csv
-rw-r--r--	Kseniia	supergroup	1.08 KB	Feb 13 11:39	1	128 MB	friends.csv
drwxr-xr-x	Kseniia	supergroup	0 B	Feb 14 13:32	0	0 B	new_centroids.csv
-rw-r--r--	Kseniia	supergroup	929 B	Feb 13 11:39	1	128 MB	pages.csv