# Common Friends - A Hadoop Program Report

By Santipab Tipparach

## Introduction

This program was written for the Hadoop environment, its main goal is to find friends that two users have in common. There are many approaches to this problem, but about a couple solutions are viable for Hadoop. The main idea in approaching this problem is that Hadoop is designed as a parallel programming architecture, this means that the software will be run on multiple nodes simultaneously, thus the program must be written with little knowledge of the current state of the data. For example, the data must be broken up into its smallest parts, use keys to define the expected result, and finally aggregate the values until one key-value pair remain.

## Run Instructions

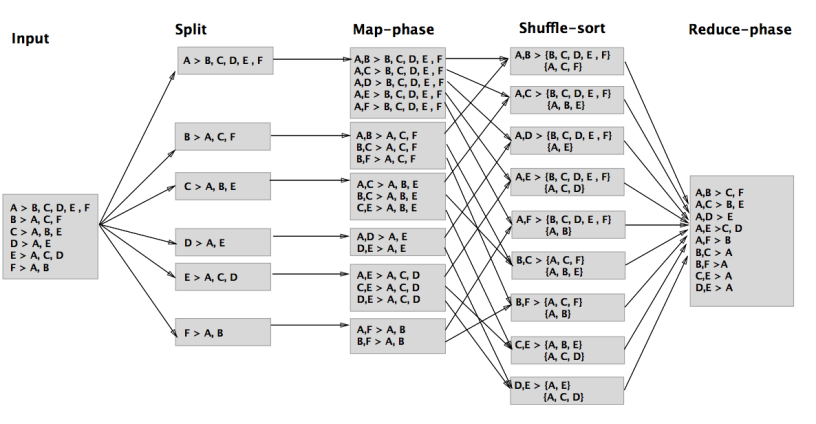
These can be found in the “HackerManual2.0.txt” file.

## Design & Implementation

To achieve the search for common friends, I made some assumptions to simplify the problem a little bit. I approached this problem as finding friends between every pair of friends, although the code could work for n^2 pairs, but I also made the assumption that the program was not aware of all possible combinations of pairs in the entire file set. Thus, two complete strangers were not compared.

In my initial attempt, I took two pairs of friends, and mapped them with the array (or line number in the file) they were associated with. This was a step in the right direction. For the reduction phase, I then merged this list with another similar list in the with two others with the same keys in the list.

Illustration from: <https://evantamle.wordpress.com/2016/03/14/implement-finding-common-friend-with-map-reduce/>



I later found an illustration of exactly what I was trying to do. Each pair of pairs was a small version of map-reduce in itself, using a hash map to combine the same values in two lists. The output were the friends that two friends had in common.

## Experimentation & Result

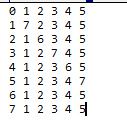
The following table is how much time it took to run for each configuration of nodes.

|  |  |
| --- | --- |
| Number of Nodes | Time |
| 2 | 14131 |
| 4 | 14222 |
| 6 | 15160 |
| 8 | 14180 |
| 16 | 19150 |

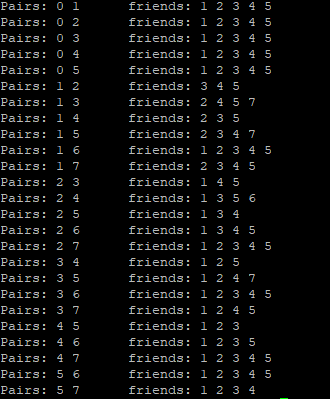
These results show that 2 nodes is the most efficient because of the communication between nodes causing overhead to slow down the speed of process. The first few seconds seem to be taken up to boot up the process, and the process of matching the data only takes a few seconds, possibly even less. I don’t have access to the web portal or the original log files that would allow deeper benchmarking, so performance observations are all based on the results above. More nodes usually means more time it takes to sync data across nodes, and at 8 nodes, the time seems to drop briefly, but raise drastically at 16 nodes.

The algorithm I wrote for parsing is not the best, since the integer parse would fail when it encountered an empty line. I ended up just skipping this portion by using a try catch block, for some reason the empty string check didn’t work on the map reduce server and without more in-depth logging, I couldn’t determine this issue.

Initially I used a simple list to test and debug the program. The cases for this list numbers was limited, but gave me an ideal testing bed for running the algorithm.



The result of this data is :



This result shows that the algorithm is working because of the friends that are uncommon between each of these pairs are eliminated.

## Conclusion

The final result can be found in the result.txt file. The results didn’t take long to obtain, I also found that when I ran it as 1 node, only the mapping portion was done. This was kind of an interesting result. I still feel that this program is not yet fully up to my personal standards in terms of unresolved issues and its lack of performance tuning and logging. The biggest thing that held me back from completing these goals were being able to see the server-side logs. Another feature I could also implement in the future is combining all files together and doing some scanning ahead to check for all pairs that exist, but I feel that this was not necessary for this project since it made more sense to assume that this would be an arbitrary list. I also didn’t want to read and store these values in any kind of list external to the map function just for simplicity sake. Most of the struggles came from trying to figure out why my values initially weren’t showing up after reduction took place. I eventually learned that the reduce node took 2 passes, the first pass to match each pair, and the second to check that there were no more matches. This last pair cause my matching algorithm to report an empty match list, thus the friends column was empty. I tried checking for the dnumber of lists checked, and that fixed the problem. After that it was just some small code cleanups to move from test data to actual data.