CS-6240 Final Project Report by Sahil Khandwala, Dung Nguyen

Introduction:

As frequent users of social media, we both wanted to focus on a project that we could relate to, understand, and learn from. As a result of this, we wanted to focus on a social media platform that was less "private" in the sense that a users' profile would rarely be hidden from the public. With some consideration, we thought Twitter was the most appropriate since many large companies use Twitter as a marketing platform and it was a platform that was meant to be public.

Using our dataset, we want to determine which users have their posts/tweets most retweeted. We hypothesize that the *"users with the most followers are most likely to get their posts retweeted regardless of their relationship with retweeters"*. We find this interesting because we believe that a more "popular" person will likely get their statuses more widely distributed.

To determine the granular details supporting our hypothesis, we intend to find:

1) Number of retweets overall for a user.
2) Number of retweets made for that user by a follower.
3) Number of retweets made for that user by someone not a follower.

The highlights of the project involves:

1. Challenges in parsing variable length and small text files as well as combining them into large chunks.
2. Using Hbase as an index to store the graph network consisting of more than 280 million entries and perform necessary operations.
3. Exploring the Hive framework, writing queries in HiveQL to perform MapReduce equivalent tasks, such as storing and transforming the data, to reach our conclusion.
4. Implementing a solution to above hypothesis using plain MapReduce design patterns to show a 2-way analysis between Hive and MapReduce.

Data:

We have selected this dataset from the below URL:
https://wiki.cites.illinois.edu/wiki/display/forward/Dataset-UDI-TwitterCrawl-Aug2012

Data Properties :
This dataset is a month's worth of Twitter crawling data from May 2011. This dataset was created using Twitter's APIs and randomly selecting 100,000 users as seeds to crawl for users' relationships. The creators of this dataset were able to crawl for 284 million following relationships among 20 million users. Among the 20 million, 3 million with at least 10 relationships was selected and further crawled.

We are working with 3 Dataset Files:

1. <u>Network File</u> - 200 million users following relationships, 5 GB file
It Consists of A->B relationships. Meaning A is a follower of B, and B is considered a friend of A, where A and B are User IDs.
eg -   2086647789
        20 18566342

2. <u>Users File</u> - 3 million users profiles, approx 250 MB file
Each line represents a user profile. It has the following fields -
User ID, User Name, Friend Count, Follower Count, Status Count, Favorite Count,Account Age, and User Location.
eg - 100024321 KatieStepek 64 93 503 0 28 Dec 2009 19:13:08 GMT Hamilton

3. <u>Tweets File</u> - 50 million tweets for 140 thousand users, approx 20 GB file There are 147909 files, 1 for each unique user. Each file contains at most 500 tweets. The most important part is Filename is consistent with UserId. Tweets are stored in the format -
Tweet1
Tweet2
Tweet..
TweetN

eg - Each tweet contains:

<u>Format -</u>	<u>Example -</u>
Type: status	Type:status
Origin: [ORIGINAL CONTENT]	Origin: @DLegacy5 damn no tweet from u #2day
Text: [PROCESSED CONTENT]	Text: damn no tweets from u 2day
URL: [URL TWEET]	URL:
ID: [TWEET ID]	ID: 96393692994211843
Time:[CREATION TIME]	Time: Wed Jul 27 20:37:01 CDT 2011
RetCount:[RETWEET COUNT]	RetCount: 0
Favorite: [FAVORITE]	Favorite: false
MentionedEntities: [MENTIONED USER ID]	MentionedEntities: 261918127
Hashtags: [HASHTAG]	Hashtags: 2day

Tasks Performed:

Pre-processing of Dataset-
Problem 1 - When writing the initial draft of our code to parse the dataset, we had attempted to use the standard parser. This was not problematic with our network or users file but with our primary tweets dataset. Since our dataset consisted of 20GB of small text files for each individual user, it was required that we be able to process the tweets of the user entirely within a single map task. It was also necessary that we be able to determine who the retweeter was which is based on the name of the text file. The standard parser only gave us the location within the file and the line within the file.

Solution - To resolve this issue, we had to create a custom FileInputFormat class object that would allow us to read the tweet file as a whole with the key as the name of the text file and the value being all the contents. Since the maximum size of each tweet was less than 600KB, we believed that this would be okay as long as the 20GB of text files were distributed across the mappers. Since tweets occupied a variable amount of lines within our text files, it would be most efficient to simply read in the file as a whole and parse it as a whole.

Java Files:
WholeFileInputFormat.java
WholeFileRecordReader.java

Primary Resource:
http://www.ibm.com/developerworks/library/bd-hadoopcombine/

Problem 2 -
Testing our new file input format, on a small 1.5GB dataset on AWS took 1.5 hours . We found this odd but after some research recognized that MapReduce is most efficient when dealing with large blocks of data. Our custom file input format sent each map job a single text file of a few hundred kilobytes for processing. This meant that there would be a map job for every tweet file within our dataset. The startup times of these map jobs was the reason our AWS cluster had performed so poorly and it would be much more apparent when running it on the full 20GB dataset.

Solution - to Consolidate a number of the files into large blocks without splitting the individual text files using CombineFileInputFormat. This allowed us to consolidate text files together into 64MB blocks and perform the same MapReduce algorithm on our dataset. This worked on our local machines but did not run on AWS returning a FileNotFoundException despite recognizing that a group of files had to be read. Changing Hadoop versions because users who had the same problem simply switched to Hadoop 2.x but we ended with HTTP connection errors.

Java Files:
CFInputFormat.java
CFRecordReader.java
FileLineWritable.java
MultiFileReader.java

Primary Resource:
http://www.idryman.org/blog/2013/09/22/process-small-files-on-hadoop-using-combinefileinputformat-1/

Final Solution - As a final step while using the standard parser for text files with Hadoop, we wrote a Java program to consolidate a number of our users' tweet files into a block. Since we knew that AWS's EMR uses 64MB blocks as its default split size per mapper job, we created roughly 60MB blocks of tweets with a special line specifying who the retweeter was without relying on the file name. This solution worked.

Java Files:
CombineTweetFiles.java

Task Plain MapReduce:
Cleaning of dataset : Since retweeting was based on text in 2011, users would add comments to the retweets that required parsing. It is also possible to have a chain of retweets and additional parsing would have to be done to determine the original author. Filtering out users that may have been tagged as part of the tweet so their names would be in the list of Mentioned Entities even though they were not an author or the retweeter.As a result parsing through the data based on text gave us a small amount of false positives that had to be discarded. We all discarded self retweets, as these would not be found in the network

Purpose - Find conclusions about our hypothesis. We decided to perform a plain MapReduce program, which forms the basis with which we can evaluate our Hbase and Hive tasks.

Approach - We have 3 programs MapReduce programs, programs
GetRetweetersAndCountPerUser:

```
Mapper:
For a block of tweets for multiple users:
    For each line in the file,
        If line is a new Tweeter header:
            store Tweeter ID
        else If the line is part of a tweet:
            Determine if it is a retweet or chain of retweets
                If True:
                    Determine index of who is the original tweeter
        Else if line is a list of mentioned users and a retweet happened:
            Retrieve the user ID based on index above
        Else if end of tweet information:
            If this is a retweet, emit original author and retweeter
Reducer:
For each key:
    emit original author and retweeter
```

Doing this allows us to group the original authors together in our result files. Using the output of these files, we wrote a program, that ran at job completion time, that would parse through the results and increment the count of retweets for an original author. This program ran in a matter of seconds on our standalone systems and would be inefficient to run as MR jobs.

FindTweetersFriends:

```
Using the file from TweetCountPerUser
    Setup()
```

```
        Parse through the file (distributed cache) and add users and retweets into hash
    Map()
        For each line in users.txt:
            If user in hash:
                emit users and object containing count of retweets and friends
    Reducer()
        For each key:
            Add up all the retweets and friends objects correspondingly
            Emit key and retweets and friends
```

This program is used to join the file generated in TweetCountPerUser with the users.txt file on the number of followers that user has.

RetweetFriendCount:
This program takes the output of the RetweetersPerUser program, and we check against Networks.txt file to find the count of followers who have retweeted that particular user's post.

setup():
Parse through the file (distributed cache) and add original tweeter and retweeters (duplicates included) as a list into hash. The maximum size of this file is only a few megabytes.

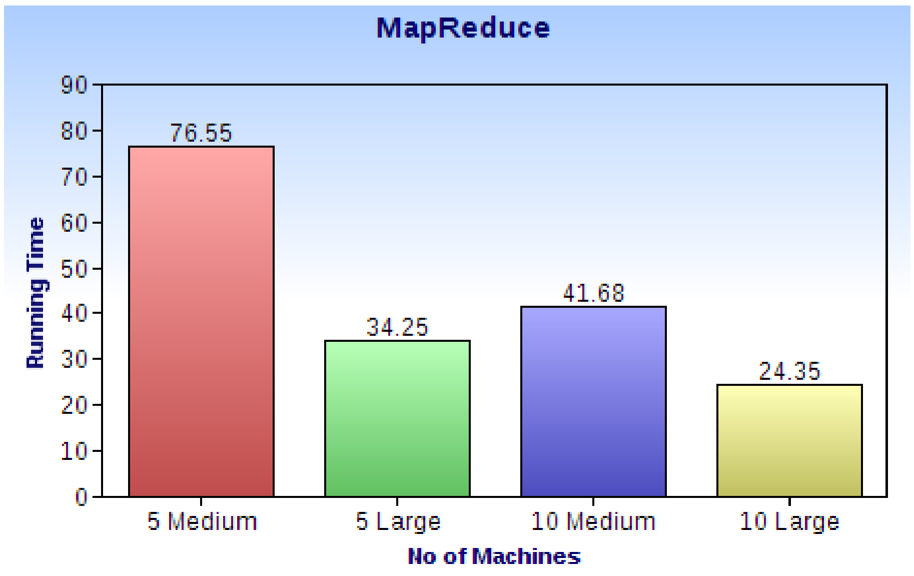
```
map():
Create HashMap, friendlyRetweets, with a user and number of retweets by friends as an integer
For each line in networks:
    If user being followed is in distributed cache hash keys:
        If the follower is in the retweeter's list
            add user with a count of 1 or count++ (if user exists)
For each key in friendlyRetweets:
    emit the key and count of friendly retweets

reduce():
For each key:
    Add up all the values of retweet counts
    Emit key and and total count of retweets by friends
```

To aggregate all of our results so that we could have the original tweeter, his/her number of followers, total retweets within our dataset, total follower retweets, and finally the number of retweets not made by followers, we wrote a program that looks at both files produced by RetweetFriendCount and FindTweetersFriends, match the original author IDs, and combine the two files to achieve the above mentioned. We implemented simple equi-join program.

Java Program:
CombineAllFiles.java

We have used replicated join design pattern and used in-mapper combiner design pattern where applicable for efficient processing. We have also changed the number of reducers depending on the number of machines we are running on.

Total Run Time Analysis - (For Log files refer MR folder inside Log.zip)



Task Hbase:

Purpose -

1. Make use of the fast associative access to small amounts hidden in big data.
2. Use the lookup facility provided by Hbase, joining the relevant useful data matching the row-key of the hbase tables.

Main Approach - We created 2 Hbase tables (NetworkData and UsersAndFollowers), one to store the 280+ million network of friend and follower relationship, and another using user dataset storing only user and follower count of 3 million users.

Source Code- (HPopulateNetwork.java and UserAndFollowersHPopulate.java)

HPopulateNetwork took the Network dataset, the key was as designed the concatenation of column2+column1 of the input file.

For UserAndFollowersHPopulate took the users dataset, key is column 1 of the input file.

Pseudo Code -

```
//  HPopulateNetwork
map(LongWritable key, Text value, Context context)
{
rowKey = column2+column1;
Put p = new Put(rowKey);
htable.put(Column Family, Column2)
}
// UserAndFollowersHPopulate
```

```
map(LongWritable key, Text value, Context context)
{
rowKey = column1;
Put p = new Put(rowKey);
htable.put(Column Family, Column4)
}
```

While populating the hbase tables we are keeping the connection alive for every map task by initializing the table in the setup() and cleanup() phase.
Later we would have many get() requests, we have split the region servers while populating the tables depending on the start character of the key. This helps in reducing query time as only a small single region server is accessed and row can be quickly found using binary search. While the Hbase would split into different region servers only after data exceeds 10GB.

Analysis - Populating the Hbase tables (Refer Log Files in Hbase/HPopulate)

5 Large Machines	55.667 minutes
10 Large Machines	38.4 minutes

We have analyzed 2 different approaches for computing the results-
For both cases we a pre-processing step of running the GetRetweetersAndCountPerUser mapreduce program.
a) Where we performed an equi-join to get our final result -
RevisedNFT.java

```
map1():
    parse each line and perform a get() against the NetworkData table
    emit (original_tweeter, “N”,1)

map2():
    parse each line and perform a get() against the UserAndFollower table
    emit (original_tweeter, {“U”,countofFollowers,TotalRetweets })
reduce():
    Same key entries end inside the same reduce call.
    segregate values starting with N and U.
    sum() all values starting with N, to get total count of retweets by followers, we subtract this
    from the total count of Retweets to get retweets from non followers.
emit(key, {total Followers, total Retweets, retweets by friend, retweets by   NonFollowers})
```

and **b)** Used a sort-merge join approach-

Created 3 programs and 2 more Hbase tables -

HbaseRetweetFriendCount:-

```
map():
    split the value by tab
    emit(column1 , column2)

reduce():
    Add count of each individual value and insert into hashmap
    create a list of all entries in hashmap, and issue a multiple get() request against the NetworkData table
    for every match we add the local count to the finalcount of retweets by friends for the original_retweeter.
    put entries into new table "ResultTable2" having key as original_tweeter and totalcount of retweets by followers as column value
```

Instead of single get request, we used in-reduce combiner design pattern, and creating a batch get() request call.

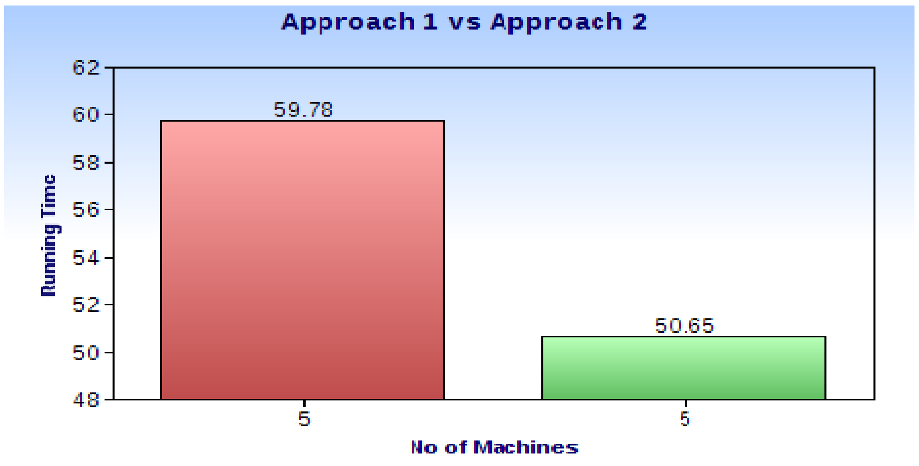
UsersFriends -

```
map():
    parse each line and perform a get() against the UserAndFollower table
    put each matching entry into new table "ResultTable" having key as original-tweeter  and countofFollowers,TotalRetweets as column values
```

HbaseFinalResult -

```
map():
    scan all entries from ResultTable
    Perform a get() request to ResultTable2 on matching original_tweeter key
emit(key, {total Followers, total Retweets, retweets by friend, retweets by NonFollowers})
```

<u>Run Time Analysis of Approach 1 vs Approach 2 (on 5 Large Machines) </u>-



Task HIVE + HIVEQL :

Purpose:

1. Leverage Hive's partitioning capabilities to decrease our job time.
2. Design table schema for each dataset and output involved
3. Write simple queries that were equivalent to MapReduce programs to achieve same results

Since Hive has the ability to partition data into bins so that not all rows are scanned, we decided to partition our network and user data based on the initial ID digit of all users. This allowed us to effectively create 9 bins of users and their respective information. This allowed us to exclude 8 other groups of users for any query.

Our Hive tables are defined as follows:

RetweetersFor(retweetee string, retweeter string)

This table is defined as the table to contain all users that have retweeted as the reweeter as well as the original author of the tweet. The files of this table are derived from RetweetFriendandCountPerUser MapReduce program. This table is partitioned into 9 bins using initial ID of the retweetee.

NETWORK(follower string, followee string)

This table is defined as the table that contains the network file. This table was partitioned into 9 bins using the initial ID character of the follower.

USERS(user string, name string, friend_count string, follower_count int, status_count string, favorite_count string, age string, location string)

This table stores all the users and the information of their profiles as given in the users file. The table is partitioned by using the first character of the user's ID, giving us a total of 9 bins.

RetweetCount(retweetee string, num_of_retweets decimal)

This table stores all the users that have had their tweets retweeted. The second field defines the total count of how many tweets have been retweeted.

To solve our tasks, we defined the following table to store our calculations:

FindTweetersFollowers(tweeter string, total_retweets decimal, total_followers int)

This table combines the RetweetCount and the Users table so that we can have a tweeter, his/her total number of retweets, and the total number of followers. This required a join on the Users table and RetweetCount table using the retweetee and user field as the common column value joining criteria.

Query:

SELECT retweetee, num_of_retweets, follower_ct FROM retweetCount JOIN users ON retweetee = user;

RetweetFriendCount(retweetee string, num_of_retweets decimal)
This table takes our RetweetersFor table and does a count of tweets made by the original authors made of a retweet. The generation of this table required us to join the two tables together based on two column values and then finally grouping it based on the retweetee (the person that had their post retweeted).

Query:

```
SELECT retweetee, count(*) FROM network, retweetersForFinal
WHERE SUBSTR(network.follower, 1, 1) = SUBSTR(retweetersFor.retweeter, 1, 1)
AND retweetersFor.retweetee = network.friend
AND retweetersFor.retweeter = network.follower
GROUP BY retweetee;
```

Combine(tweeter string, total_followers int, total_retweets decimal, num_of_retweets int, non-friends decimal)
This table joins the two tables from FindTweetersFollowers and RetweetFriendCount into the same table to give us a better conclusive view. It also performs the arithmetic to calculate the number of tweets made by non-friends and inserts it into a separate column.

Query:

```
SELECT tweeter, total_followers, total_retweets, num_of_retweets, (total_retweets -
num_of_retweets) AS non-friends FROM
FindTweetersFollowers, retweetFriendCount WHERE retweetFriendCount.retweetee =
FindTweetersFollowers.tweeter;
```

All tables mentioned above are stored as text files that are tab delimited.

Analysis of Hive vs Plain Map Reduce :
As can be seen in the logs for RetweetFriendCount, the number of tasks performed by the plain MapReduce jobs had a total of 82 map tasks and 7 reduce tasks. Hive had 24 map tasks and 7 reduce tasks. Given the number of tasks, the plain MapReduce program completed in 51 minutes. In contrast, Hive had taken 24 minutes. This was done on 5 medium machines. (Log files MR/5M/RetweetFriendCount and Hive/5M/)

Increasing the number of medium machines to 10 also decreased our time significantly. Our plain MapReduce program took 15 minutes with 10 large machines. However 10 medium machines running on Hive was able to achieve 12 minutes completion time despite the size differences. Even though the MapReduce job's machines were more powerful, the partitioning prevented us from doing a full table scan that the plain MapReduce job had to perform.

Setup Challenges while using Hive:

Preprocessing:

When partitioning our dataset into their respective tables, we had to first import the data as a whole, and then perform a (Select * FROM table Where condition) statement to retrieve the appropriate fields for insertion into the partition. This was effectively a full table scan for as many partitions as we needed. Due to this our setup time increased greatly for a simple task.
Please refer to our data_load.q script to see how we had created the tables and loaded the data.

<u>Local:</u>
1) Hive requires Hadoop and Java to execute on a local machine. So their paths are set to access them easily.

2) We used - https://cwiki.apache.org/confluence/display/Hive/GettingStarted has step-to-step
and https://www.youtube.com/watch?v=Dqo1ahdBK_A
configuration of hive and it helped us to install and configure hive locally on our systems.

<u>AWS:</u>
We had two options to run on AWS i.e., batch mode and interactive mode. We primarily used batch mode to run our scripts in an automated fashion and then finally use interactive mode to confirm our results.

Challenges in Hive -
1. <u>Problem</u> - Dealing with the import of data and keeping it persistent. When attempting to import our tab delimited file into Hive and making sure that our query to create our table delimited on tabs, we still faced problems. After some time debugging, we were able to recognize that the import query had imported the entire row of the file into a single column.

<u>Solution</u> - To resolve this, we had to write a Java program to take in our users data and strip it of all whitespaces between each intended column value, and then replace the pre-existing whitespaces with commas. This gave us a CSV file that imported properly into our Hive table

2 . <u>Problem</u> - When importing our data into our default tables, we ran into a problem of importing our files when using 'LOCAL' as part of our insert table partitions. The Hive cluster was unable to find the file and could not import the data.

<u>Solution</u> - To resolve this, we removed the 'LOCAL' portion of the statement and as a result were able to successfully import the files because of Amazon's S3 filesystem.

3. <u>Problem</u> - Using a identical create and insert query. Every time a script is ran to load a table that is using partitions, the data does not populate properly, and shows up with 0 rows.

<u>Solution</u> - When we researched, we found out that we had to first import the table and then alter it to recover the partitions using one of the following: 1) msck repair "tablename" 2) alter table "tablename" recover partitions. Running any of the two commands returned an error that hive was unable to parse it. After some research, AWS's documentation seems to be outdated and the first

command is actually “msck repair table <tablename>”. This allowed us to import table by using the create table query and then running the above command allowed us to recognize the partitions made. All queries against the partitioned table is successful thereafter.

Primary Resource:
http://stackoverflow.com/questions/20995489/unable-to-recover-partitions-in-shark-for-hive-table-with-s3-location

4. Problem - When running our queries on large machines regardless of number, the cluster crashes even though the logs display 100% map and reduce completion. A out of heap space error is returned. For this reason we were unable to complete the running of this application using 5 or 10 large machines. We had used the Bootstrap action for an intensive memory configuration without any change in results. We did however test it on 5 XL machines with results and as expected faster run times. We were unable to test 10 XL machines due to the limitations on our account. Without AWS support, it would be difficult to debug this issue. Our logs for 5 and 10 large machines displaying this behavior are included. (files in Hive/stderrFilesforLargeMachines)

Conclusion:

Prior to our project we had believed that the dataset was enough for us to complete our experimentation to prove or disprove our hypothesis. After writing our code and realizing a number of flaws within the code which we’ve fixed, we began to recognize that the dataset was incomplete as well. Since the origin of the data started with the crawling of a network, to users, and then finally to tweets, we recognized that the original author of retweets may not be in the network file. As a result, our final results are recognizing a number of retweets as one not by a follower. Without having the original author and the his/her relationships, we are unable to say with full confidence that our hypothesis is incorrect.

As the results currently stand, our hypothesis has been proven false by a majority of the instances within the final output for the reason above.

Original_Tweeter ID	Follower Count	Total Retweet Count	Retweet Count by Followers	Retweet Count by non-followers
125786481	801290	10677	1800	8877
14230524	11060753	5595	545	5050
27260086	10468722	5708	421	5287
813286	8737791	6107	536	5571
16409683	8262393	1026	122	904
25365536	7952110	1008	118	890
21447363	7940255	739	85	654
19058681	6992290	968	62	906
15846407	6902535	1691	466	1225

However when the network file has the original tweeter's relationships, we can see results which support our hypothesis.

Original_Tweeter ID	Follower Count	Total Retweet Count	Retweet Count by Followers	Retweet Count by non-follower
16082787	4248	126	103	23
4257381	1281	115	92	23
15784742	13704	59	36	23
29139288	5619	55	32	23
21410529	30518	47	24	23

We believe that our dataset has more room to become a better sample.

Given the size of this project, we believe the following would allow us to further explore the hypothesis and strengthen our statement:

- Using live streaming of Twitter data via Twitter API
- Start from crawling of User profiles and all followers and build network file
- Crawl for users' tweets and retrieve all retweets of those tweets

Performing the above steps would provide us solid data that would allow us to fully provide or disprove our hypothesis.

To conclude, most of the problems and challenges that we faced with this dataset and its intended calculations involve: data processing, incomplete sample data, data storage and reading optimization for all major tasks, and finally external resource configurations, primarily EMR clusters.