MemoryWhole: A Distributed Deleted Tweet Discovery Service

Rob Gevorkyan, Michal Guzek, and Ai Enkoji

March 2020

1 Introduction

In the age of the internet, social media is a major player in mass communication. One of the largest platforms, Twitter, provides a platform for everyone from common citizens to heads of state to broadcast their thoughts in short "tweets" of 280 characters or less. Tweets are often mundane or self-promoting, but in some cases have significant political and social weight. According to Business Insider, Twitter's guidelines as of 2017 enable groups to use claims that tweets are "sensitive" or "potentially offensive" to delete tweets or even suspend whole accounts, often on ideological grounds. On an even grander scale, governments including but not limited to Turkey, Venezuela, and France have successfully censored tweets and deleted accounts entirely based on simple requests to the company. On the opposite extreme, there is great public interest, especially amongst news outlets, in the controversial tweets of well-known figures who try to scrub the site of such posts after receiving backlash.

In all cases described above, there is a strong interest by various groups to preserve what has been deleted. We introduce in what follows MemoryWhole, a service designed to archive tweets from large numbers of public figures, reporting when any are deleted, and what those tweets contained.

2 Architecture

Apache Hadoop is a software ecosystem that is mainly used to store and process big data. In particular, it allows storing of large sets of data in clusters, and its distributed file system allows for both fault tolerance and parallel processing. In particular the Hadoop framework, makes assumptions that hardware failure is inevitable and thus is designed for software-level handling of these failures. One of the main modules of the Hadoop ecosystem are the following:

- Hadoop Common: the collection of utilities and libraries utilized by the other Hadoop modules.
- Hadoop Distributed File System (HDFS): the distributed file system implemented by Hadoop that stores data in clusters with high throughput access
- Hadoop YARN: the layer that manages resources within the cluster as well as job scheduling
- Hadoop MapReduce: the data processing component of Hadoop that, like its name, splits and maps data and then reduces it

Since the HDFS and MapReduce play a visibly large role in this project, we will discuss these two in more detail below. The Hadoop File System implements a distributed file system that takes data and separates them it into chunks to be distributed to different nodes within a cluster. In

particular, the HDFS is designed to be fault-tolerant, and thus, it replicates the data several times so that it is stored in multiple nodes. This ensures that in the case that one of the nodes containing some set of data crashes, a copy can be recovered from one or more of the other nodes. Such design allows for both fault-tolerance and parallel processing. In terms of the nodes, the HDFS has a master/slave architecture. The main node, called the NameNode, acts as the master server and manages the file system and file access operations. The DataNodes, store some block(s) of a split data file and also have the ability to create, delete, and replicate data blocks. It should be noted, however, that a DataNode cannot contain more than one copy of the same block.

Hadoop MapReduce is a software framework that allows writing data processing applications to be used alongside the HDFS. Its main components include the ResourceManager that oversees all the nodes, the NodeManager for individual nodes, and MRAppMaster for each application. A MapReduce *job* and *job configuration* are specified by the application, and typically has inputs and outputs as well as map and reduce functions. A *job client* provides the *job* – which is some executable file (typically a jar file) – and *job configuration* to the ResourceManager, which in turn provides directives for performing the tasks scheduled in the *job* to its worker nodes. In terms of the inputs and outputs, they are set to be $< key, value >$ pairs. A MapReduce *job* often consists of first splitting the input data into blocks, mapping each individual split, and ultimately reducing them to some "reduced," or in other words, aggregated, output.

3 Tools and Techniques

The core of our application consists of MapReduce framework used from within Java application. Its purpose is to analyze consecutive batches of data and detect tweets that have been deleted. A tweet is considered deleted if it appears in some batch i and then is missing for all batches $k > j > i$, for some batch j. The tweet might appear again in batch k if, for instance, a user temporarily had disabled his account and then enabled it back again in batch k onward.

The input to our program is contained in a folder corresponding to the given batch, which name is specified as a command-line argument by the client. It contains all the JSON files to be processed by the program in a single run. The first part of MapReduce model in our application, the map, revolves around extracting a tweet ID, hashtags and text from each tweet in the input JSON file. By default, a map task is assigned an input split of size of one HDFS block (usually 128 MB). However, due to the fact that JSON files are virtually JSON arrays containing particular tweets as their elements (JSON objects), an input file could not be split arbitrarily so as not to violate the JSON syntax. Accordingly, the default file input format (consisting of individual lines in a text file) had to be substituted by our specialized input format class which assigns each map task a separate JSON file from the input folder. Thus, the input types of the $< key, value >$ pair in our map tasks are $NullWritable$ and $BytesWritable$, respectively. (These are specialized Hadoop data types for such purposes which are optimized for network serialization, as opposed to Java built-in types like *int* or *String*). That means we do not make use of *keys* (which normally would be the offset of the line from the beginning of the file) but only of the *value* which constitutes the content of the entire JSON file. Having processed each tweet, a map task emits it as a $< key, value >$ pair, with tweet ID being the *key*, and hashtags along with the tweet text acting as the combined *value*. Each emitted pair (of type $< Text, Text >$) goes to the reduce task.

Our MapReduce model consists of only one reduce task. Before the actual reduce part, a setup code

is executed. It consists of filling the $HashMap$ data structure with all of the previously processed data (with tweet IDs serving as key). For that purpose, a special file is maintained on HDFS that the program appends to whenever it finds a new tweet ID in the current batch. Afterwards, having received a pair of the type $< Text, Text >$ from a map task, the reducer performs a hash table lookup to determine if the given tweet ID has already been seen by the application in previous batches. If so, it deletes the corresponding entry from the hash map. Consequently, at the end of the reduce phase, the $HashMap$ will only contain the tweet IDs that are missing. On the other hand, if the reducer does not find the tweet ID in the $HashMap$, it appends it to the aforementioned HDFS file. Therefore, the actual crux of the reducer is implemented in the clean up method which is executed after all the tweet IDs have been processed by the reducer. Its purpose is to emit every $HashMap$ entry to the HDFS, constituting the ultimate program output - the missing tweet IDs (the reducer class allows the pairs to be emitted from any of its methods, not only from the *reduce* one). The emitted pairs are also of the type $< Text, Text >$.

4 Evaluation

To evaluate the correctness of MemoryWhole, we built a tweet simulator based on samples collected from the Twitter API. Using a Python script, we were able to generate sequences of historical tweet data with an arbitrary number of tweets. Some tweets were guaranteed to be preserved, and a specified parameterized number were guaranteed to be deleted at some point in the future. The structure of our simulated tweet data is a stripped down JSON object having the following fields:

- *created_at* : a timestamp of when the tweet was made
- *id* : a globally unique identifier for tweets used by the Twitter back end
- *text* : the text contents of the tweet
- *user_id* : a globally unique identifier for a twitter user
- *user_name* : a human readable name of the user
- *screen_name* : the twitter handle of the user

5 Conclusion and Future Work

There are a number of future directions we could take with the service, largely related to the velocity of data ingest.

In our proof of concept, we were limited in the capabilities of our Twitter developer accounts in terms of both request frequency and functionalities. With a paid Twitter developer account, we could in principle make many more requests within a short window of time and even tap into a real-time notification feed for new tweets. This would allow a finer-grained view of tweet activity and the potential to catch more tweets that are deleted shortly after they are posted.

An important next step would also involve monitoring a larger number of users and actually finding interesting deleted tweets. We had some difficulty in this regard because there are potentially a large number of users to target and many of them did not have anything interesting disappear from their feeds. More time and exploration could lead to some interesting discoveries.