Mining Twitter in the Cloud: A Case Study

Pieter Noordhuis, Michiel Heijkoop
MSc student Computing Science
University of Groningen
The Netherlands
{pcnoordhuis,mheijkoop}@gmail.com

Alexander Lazovik
Distributed Systems Group
University of Groningen
The Netherlands
a.lazovik@rug.nl

Abstract

Mining and analyzing data from social networks can be difficult because of the large amounts of data involved. Such activities are usually very expensive, as they require a lot of computational resources. With the recent success of cloud computing, data analysis is going to be more accessible due to easier access to less expensive computational resources.

In this work we propose to use cloud computing services as a possible solution for analysis of large amounts of data. As a source for a large data set, we propose to use Twitter, yielding a graph with 50 million nodes and 1.8 billion edges. In this paper, we use computation of PageRank on Twitter’s social graph to investigate whether or not cloud computing, and Amazon cloud services in particular, can make these tasks more feasible and, as a side effect, whether or not PageRank provides a good ranking of Twitter users.

1. Introduction

Social networks (e.g. Facebook, MySpace, Twitter, etc) can be defined as services that provide users the possibility to establish a connection with their friends through the application and share information with them. These networks can be characterized as a graph in which every user is a node, and an edge exists between two nodes if they are friends. Users will typically receive information and/or messages from other users if they are connected in this graph. Depending on the network, this happens up to a certain degree. With networks like Facebook having a social graph with over 300 million nodes [11] one can imagine that there are large amounts of data involved in performing analysis.

Another example of a social network that can be seen as a graph is Twitter, which calls itself a micro-blogging service.

It allows users to post their thoughts or status (called tweets) and follow status updates of other users. Twitter started as an experiment in 2006 and was founded in 2007 [12]. The number of users on Twitter has grown since the service started and peaked in March 2009 at a rate of 13% [6] new users per month. The total number of users as of January 2010 is estimated to be around 75 million [9].

Twitter is different from most social networks as its social graph is a directed graph: one can choose to receive updates from an user but that does not mean the reverse is true by default. Twitter is less focused on friendship and more on interest: a user is more likely to be connected to another user because he is interested in what this user has to say and less because they happen to be friends in real life. It is also different in the fact that you can be in contact with someone you are not connected to in the graph, by searching all recent tweets for a certain keyword.

Twitter is gaining a lot of relevance with mainstream media like CNN showcasing tweets on air and posting its breaking news to the service. It is also becoming an increasingly important tool for public relations used by both large multinationals and start-up businesses. Because of Twitter’s explosive growth one can imagine it has become equally difficult to find relevant information amongst all sources broadcasting their information. In search results a tweet announcing breaking news by default has the same value as a teenage girl complaining about her school work. One of the possible solutions to reduce this negative effect is to rank users. For example, the number of people that have subscribed to user’s updates can be used. The number of subscribers is then called the number of followers of the user. Using this method, a tweet from user a is considered more relevant than the one from the user b if the user a has more followers than the user b.

Because of the large amounts of data involved in the mentioned social media platforms, traditional solutions do not scale well and are known to be very expensive. In this work, we propose to use cloud computing as a possible solution as it allows for large scalability with relatively low...
costs. More specific, we propose to evaluate one of the available cloud infrastructures by applying the PageRank algorithm to calculate a non-generic ranking for Twitter’s user base.

In this work we mainly concentrate on Amazon web services and its cloud infrastructure, as Amazon is one of the most advanced and flexible cloud platforms in the market.

This paper is organized as follows: First, we discuss related work in Section 2 and give a detailed description of the problem at hand in Section 3. We continue by discussing our realization in Section 4 and how different cloud services can assist us in doing so. Finally, we give an evaluation of our realization in Section 5 and draw conclusions in Section 6.

2 Related work

In 1998, Page and Brin introduced the PageRank algorithm [10] which is used as the basis for ordering Google’s search results. It uses the Web graph with individual web pages as vertices and hyperlinks as edges. The philosophy of PageRank is that if a web page A links to another web page B, the author of page A is likely to find page B of importance. This way, the number of in-edges is used to determine how important a certain page is. This would result in simply counting the number of in-links of any page to determine its importance. PageRank, however, defines the importance of a page as the sum of the ranks of the pages that link to it. This way, a page with few in-edges from high ranked pages can be ranked higher than a page with many in-edges from low ranked pages. This results in pages being ranked by the cumulative rank of the pages that link to it rather than the number of in-edges.

While the PageRank paper is among the most cited papers in its area, its concept is far from new. In [4], Franceschet lists a number of predecessors in areas of sociology and journal ranking based on citation count. Furthermore, the Hypertext Induced Topic Search (HITS) by Kleinberg [7] is also listed.

When the ranking algorithm of our choice is to be executed in the cloud, there are several factors that need to be taken in account. For example, the distributive nature of cloud infrastructure is important in deciding how the PageRank algorithm will be executed. Also, cost is a factor that is important in determining the optimal cloud platform and different services to be used. A survey of different cloud platforms and their costs is given in [3].

The similarity between Google using PageRank for ordering search results and Twitter can be found in the way to think about web pages or users. Where Google assumes that a link from one page to another implies that the author deems the linked page important, a Twitter user following another could imply the same. Twitter users tend to follow users who they think are interesting. There are of course exceptions as some users have their account set up so that users that start following them are automatically followed back. We have found that only a minority of Twitter users apply this technique and consider this to have negligible influence on the outcome. In short: where Google thinks of web pages as vertices and hyperlinks as edges, in the Twitter scenario users are vertices and follow-relations are edges. Where Google needs to crawl the internet to compute PageRank, we need to crawl the Twitter social graph to compute PageRank. By using PageRank, we consider relevant users a better judge of the relevance. This reduces the impact of things like offering new followers a chance to win a prize.

3 Data acquisition and analysis

To compute the PageRank of every Twitter user, there are two types of problems to consider. First, the entire Twitter social graph needs to be crawled. Second, the PageRank algorithm must be applied to this dataset. These steps are shown in Figure 1.

<table>
<thead>
<tr>
<th>Data acquisition stage</th>
<th>Processing stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue initial ID</td>
<td>Initialize matrix</td>
</tr>
<tr>
<td>Queue empty?</td>
<td>Iterate PageRank</td>
</tr>
<tr>
<td>no</td>
<td>50x</td>
</tr>
<tr>
<td>Retrieve followers</td>
<td>Store followers</td>
</tr>
<tr>
<td>Store followers</td>
<td>Queue IDs not yet in queue</td>
</tr>
<tr>
<td>yes</td>
<td>Store result</td>
</tr>
</tbody>
</table>

Figure 1. Applying PageRank to Twitter

3.1 Data acquisition

To gather the data that is needed to compute PageRank, the Twitter API [13] was used. This API allows developers to consume different types of data that Twitter exposes, such as user profiles, status updates and follower information. The follower information is used to crawl the social graph and acquire the dataset that is needed.

There are two main methods that Twitter exposes to gain knowledge of the social graph, being: followers/ids
and friends/ids. Both methods return a series of ID numbers of users that follow and are followed by a user respectively. As the number of users on Twitter is large, we decided to only use followers/ids to retrieve the social graph. As PageRank computation considers the rank of a vertex to be made up by the rank of the vertices with out-edges to it, retrieving only the sources of PageRank for every user would intuitively be the right thing to do. However, this does not take into account the possibility of users that do not follow anyone and thus are not listed in any user’s follower list.

The use of Twitter’s API is rate limited. This means that every user is limited to perform a number of API calls per hour. The rate limit defaults to 150 calls per hour, but lifted rate limits can be requested. For this project, Twitter allowed us to perform 100K calls per hour. As the number of users on Twitter is approximately 75 million, crawling all Twitter users will take at least 31 days in a best-case scenario.

Every user on Twitter is assigned a unique ID. The response to a Twitter API call on followers/ids is an array of IDs of users that follow the requested user. As some Twitter users are followed by a lot of other users, the response to this call may be paginated. This results in multiple API calls necessary to get the complete set of followers for a user.

Using the described method, it is possible to gather information on the followers of every user on Twitter. This data needs to be processable in a way that the computation of PageRank is possible. To get a clear notion of the format that is needed for PageRank computation, we now explain the different steps involved in applying the PageRank algorithm.

### 3.2 Data processing

The formal definition of the PageRank algorithm needs the definition of several variables. Let \( u \) be a Twitter user. Let \( I_u \) be the set of users that users that follow user \( u \) and \( O_u \) the set of users that user \( u \) follows.

Now, the rank of user \( u \), \( R(u) \) can be defined as:

\[
R(u) = \sum_{v \in I_u} \frac{R(v)}{|O_v|}
\]

From this formula, we can see that the rank of a user is equal to the sum of the ranks of its followers, divided by their number of users they follow. An example is given in Figure 2. Here, \( |O_u| = 2 \), \( |O_v| = 3 \) and \( I_u = \{ u, v \} \).

It is clear that this model does not compensate for loops and sinks in the graph. When a user has a number of in-edges and therefore receives rank, but has no out-edges, this user is a rank sink. To counter this effect, Page et al. [10] introduced the Random Surfer Model. This model describes an addition component in the PageRank equation, that can be interpreted as a random surfer in the original context of PageRank. The component adds a probability factor that accounts for users not always following the link structure of the graph, but sometimes jumping randomly in the graph. To capture this behavior, an additional vector \( E \) (or bookmark-vector) is introduced. This vector holds a probability for every node, that it is the target of a random jump. While these probabilities can be chosen arbitrarily, we have chosen to let every value in \( E \) is equal to \( 1/N \), where \( N \) is the number of nodes in the graph. Finally, this model is combined with the previously defined formula for \( R(u) \) by making a weighted combination. To do so, a factor \( \alpha \) is introduced and the formula for \( R(u) \) is changed to match:

\[
R(u) = \alpha \sum_{v \in I_u} \frac{R(v)}{|O_v|} + (1 - \alpha)E
\]

The formula for \( R(u) \) is recursive, but may be computed by assigning any set of ranks initially and iterating the computation until it converges. We choose to use our bookmark vector \( E \) as the initial set of ranks. This makes sure that every user has the same rank to start with.

The set of relations can be represented as a squared matrix \( A \) where the rows and columns are the different users. Let \( A_{u,v} = 1/|O_u| \) if user \( u \) follows user \( v \) and 0 otherwise. When user \( u \) does not follow any user, the entire row \( A_{u,v} = 1/N \) for all \( v \).

Now, to incorporate the random surfer model, matrix \( A \) is combined with matrix \( T \), where \( T \) is a matrix where each row is equal to the bookmark vector \( E \). Both matrices are combined using \( \alpha \). The resulting matrix is often called the Google-matrix.

\[
G = \alpha A + (1 - \alpha)T
\]

For larger \( \alpha \), reality is better modeled while for lower \( \alpha \) the algorithm converges faster. Experiments have shown that \( \alpha = 0.9 \) renders good results.

![Figure 2. Example where \( R(w) \) receives its rank from \( R(u) \) and \( R(v) \)](image-url)
Let $R$ be a row-vector containing the ranks of all users. To compute PageRank, the following algorithm needs to be executed until it converges.

$$ R_0 \leftarrow S $$

loop:

$$ R_i \leftarrow R_{i-1} \times G $$

Convergence can be measured by computing the difference between the vectors $R_i$ and $R_{i-1}$.

When the number of users is large, matrix $A$ becomes very large. Luckily, this matrix is sparse as the number of relations between users is generally low. To both store and perform computation on this matrix, we need a method that stores the relationships efficiently.

4. Realization details

We have evaluated two cloud platforms for the problem at hand, being Amazon’s AWS [2] and Google’s AppEngine [5]. We need to perform extensive crawling and execute the PageRank algorithm against a large dataset. This requires a lot of freedom in choice of application stack. Where AWS provided virtualized computing instances, AppEngine provides a pre-built application stack. This stack sets a number of requirements to the application that can be deployed. For instance, all code must be written in Java or Python. AppEngine’s platform facilitates to run web applications, so all tasks that may need to be executed must be invoked using a web interface. Also, when tasks need to be executed, they need to be placed in task queues, which incur significant overhead to simply executing tasks on a virtualized instance.

Because of the limiting requirements of AppEngine and our need for both freedom in creating a custom application and performance in computing the ranking, we have chosen to use AWS in favor of AppEngine. Apart from the virtualized instances that AWS offers, it also provided cloud services for tasks such as persistent storage, queueing and distributed computation. With total control over the virtualized instances allowing us to choose our development language, we have chosen to use the Ruby programming language to gather data (because of its rapid development) and the C programming language for the PageRank computation (because of its speed and memory efficiency).

AWS offers a series of cloud services including EC2 (Elastic Compute Cloud), MapReduce (distributed processing), S3 (Simple Storage Service) and SQS (Simple Queue Service). We now describe the various problems we encountered when implementing the Twitter crawler and how Amazon’s services did or did not assist in this process.

4.1. Queueing

To crawl the entire Twitter social graph, we need to maintain a queue of user IDs that need to be crawled. Also, we need a way to check if IDs are already appended to the queue or not.

AWS offers a queueing service called SQS, which features unlimited queues with unlimited messages, variable message size and locking. However, messages in the queue are only retained for up to 4 days. The cost of using SQS is dependent on the number of messages that transfer through the system and the data traffic that is involved. For every 1 million messages, $1 is billed [2]. This means that if it were possible to check for duplicate messages and messages were retained in queue for at least one month, the cost of using SQS as our queue would be approximately $75, excluding data traffic.

Because the crawling process takes at least a full month, entries may be queued for a long time before being processed. As SQS retains messages only up to 4 days, it is not suitable for the problem at hand. Therefore, we have implemented our own queueing solution.

The queueing solution we have used consists of two parts. First, there is a lookup table which contains the IDs of users that have been crawled or are already queued for crawling. Second, there is a queue that holds all the IDs of users that need to be crawled.

After the user IDs of the followers of a certain user have been retrieved, every ID is checked against the lookup table. If an entry exists, this means this ID has already been crawled, or is queued for crawling. Every ID without an entry in the lookup table is both added to the queue and inserted in the lookup table. This ensures that these IDs will be crawled somewhere in the future, and will not be queued again. This process is visualized in Figure 3.

This FIFO queue holds all the IDs that need to be crawled and does not suffer from the restriction of IDs being deleted after a given amount of time. The lookup table we use provides an easy way of ensuring an ID is never queued twice.

4.2. Storage

To be able to query the followers of a user in a later stage, all responses from Twitter need to be stored. The number of IDs that need to be stored per user can vary from 0 to 4 million. Because IDs can be encoded as 32-bit integers (4 bytes), the maximum amount of data that needs to be stored for a single user is approximately 16 megabytes.

Amazon offers a service for storing structured data in their cloud, called SimpleDB. SimpleDB organizes data as items which contain name-value pairs, and therefore does not enforce a strict data schema. There are, however, limits...
to the data that can be stored in SimpleDB. Every item is limited to contain 256 name-value pairs, where every name and value is limited to 1024 bytes. In the situation where these pairs can be fully used, this would result in storing a maximum of 512 kilobytes per item.

The cost of SimpleDB usage is measured in processor hours, data storage, and data traffic. We assume that 100,000 queries cost a single machine hour and only one query is needed to store follower data for a user. With a price of $0.14 per machine hour, this results in $105 for storing information for 75 million users, excluding storage and traffic pricing. Because all data also needs to be retrieved at least once to perform the computation, the amount will double to $210.

To bypass the limit of storing 512 kilobytes per entry, a sharding technique could be applied that distributes data over multiple database entries. Clearly, this increases the number of queries and therefore makes usage of SimpleDB more costly.

Because the limits of SimpleDB would require us to build complex solutions to bypass these limits, we have chosen to use our custom storage format. With the PageRank computation in mind, the storage format should be streamlined to be easily traversed.

Using benchmarks, we have seen that it is unfeasible to store the response of every call in separate files. When retrieving the IDs from disk, this results in a lot of random access and high latencies. To optimize this, we have used an adaption of the Compressed Column Storage format. For every response we append the IDs of the users to a file (called the data-file). In another file (called the pointer-file), we append a triplet containing the user ID that belongs to the response, the pointer to the first ID in the data-file and the pointer to the last ID in the data-file. This structure is visualized in Figure 4.

![Figure 3. Queueing process](image)

**Figure 3. Queueing process**

To limit the risk of data being corrupted after being retrieved, we have chosen to make a split and start with a new pair of files once the data file reached a size of 32 megabytes.

As Amazon EC2 instances are advertised to be volatile, these intermediate pairs of files were persisted to Amazon S3.

### 4.3. Data acquisition

With lifted request limits, Twitter limits a single IP to 20K API calls per hour. For this reason we were required to separate the crawling process across 5 different computing instances, in order to consume all 100K API calls each hour. If Amazon AWS allowed for the possibility to assign multiple IPs to a single instance, this could have been done on a single instance as crawling will not consume a significant amount of resources. To simultaneously crawl different users on different instances, we created a web service exposing the central queue. Each crawler is able to dequeue user IDs and return the response of the API call. This way, the limits are fully consumed, while keeping the result centralized. The architecture is shown in Figure 5.

As the resource requirements for performing 20K API calls per hour are very low, we have chosen to deploy the queue on one of the crawlers to reduce complexity.

Because an estimate of 75 million users and a best-case scenario of crawling 100K users per hour results in a process that takes at least a full month, there was a need for monitoring this process. To have insight in the ongoing crawling process, we have created a monitoring interface that shows sparkline graphs of several properties for each of the crawlers, such as number of API calls per second, number of failures per second and number of API calls remaining in the current hour. Furthermore, we used sparklines to monitor the number of users processed and the size of the queue, to estimate when the process would finish. To
adequately monitor the status and progress, the interface is updated with real-time data every second. The dashboard of the monitor interface is shown in Figure 6.

4.4. Computation

The computation of PageRank on the data that was crawled consist of several passes. Initially, the data needs to be filtered for the number of users each user follows. Recall from the introduction on PageRank that \( |O_i| \) is needed for every user. While this information can be extracted from the Twitter API, we have used a reverse index to calculate this. As is clear from the section on storage (4.2), the available data consist of a number of large files containing user IDs. First, a vector \( O \) is set up to hold the number of out-edges per user. For every user ID that is seen in the data-files, we know that the corresponding user follows someone. Therefore, we can increment an integer in \( O \) on the position of the ID. When the pass is complete, we have a consistent projection of the number of users each user follows. This vector is required to be consistent with the dataset in order for the PageRank algorithm to execute properly.

To start iterating, we first need the vector \( R_0 \), which can be initialized to any set of values. As mentioned before, we choose to initialize this vector to equal the bookmark vector \( E \), where every element equals \( 1/N \). Every iteration requires vector \( R_t \) and the matrix \( A \). Where \( A \) is static, the input \( R_t \) is used to compute the output \( R_{t+1} \). While it is possible to distribute computation of every iteration across several instances using either Amazon MapReduce or a custom distribution, this is not optimal.

The vector \( R \) has as much elements as there are users. End every element is stored as a double floating point number (of 8 bytes). With 75 million elements, this vector would be 600 megabytes in size. In a distributed solution, this 600 megabyte vector needs to be transferred to each of the nodes performing the computation. After a subset of the computation is done, the different intermediate results need to be centrally aggregated before continuing with the next iteration. If the instances were to be connected with Gigabit Ethernet, this would cause an overhead of at least 6 seconds per instance in a best-case scenario. The aggregation of the results would incur a similar overhead. Furthermore, when the results are aggregated centrally, other computation instances would be idle.

In this case, the size of vector \( R \) and matrix \( A \) are small enough to perform computation on a single machine. Instead of distributing the computation, we have chosen to employ a high-memory instance in the Amazon cloud. This instance type offers an 8-core machine with 68.4GB of memory. The memory is large enough to hold both the vector \( R \) as well as the entire dataset (including overhead approximately 12GB in total).

Because all the data fits in memory on a single machine, and this instance type offers a large number of available processor cores, this is the fastest approach to compute PageRank on this dataset.

4.5. Interface

For the computed ranking to be accessible, we have created a web interface. This interface exposes different types of data. First, the absolute ranking is displayed as a list of users, ordered by descending rank. A screen shot of this view is shown in Figure 7. Second, we expose the rank of every Twitter user, together with his 30 highest ranked followers and 30 users with the highest contributors to his rank. These lists of followers can be computed on the fly when the number of followers is small, but becomes more infeasible as the number of followers grows, as both lists
require a sort operation against the complete set of followers. Therefore, we have chosen to precompute this data. As discussed in Section 4.2, the crawling process results in a list of followers for every user. To compute the 30 highest ranked followers for a user, the user IDs of the followers are combined with their ranks, sorted and written to disk. To compute the contribution of a follower $u$ to the rank of user $v$, we use the following formula:

$$\text{contribution}(u, v) = \frac{R(u)}{|O_u|} \cdot \frac{1}{\alpha R(v)}$$

The list of followers is sorted by the outcome of this formula and the top 30 user IDs are written to disk, identical to the 30 highest ranked followers.

The total cost of both crawling Twitter and computing PageRank on the acquired dataset was $275. This amount consists of $255 for EC2 (~ 2500 instance hours), $15 for data traffic (~ 200GB) and $5 for S3 (~ 10GB/month, including overhead). If Amazon would have allowed to attach the 5 different IPs to a single instance, the cost for EC2 could have been 5 times lower with the total cost being approximately $75.

## 6 Conclusion

As performing analysis on large social networks such as Twitter requires a lot of resources due to the large amount of data that is involved, we propose to use cloud computing as a possible solution. In this work, we applied the PageRank algorithm on the Twitter user base to form a user ranking. This has been done in a two-phase process: in the crawling phase all data was retrieved from Twitter and in the processing phase the PageRank algorithm has been applied to the acquired data.

Amazon cloud infrastructure was used to host all the related computation. In particular, we have heavily relied on Amazon EC2 (virtualized instances) and S3 (persistent storage). We have also discovered that Amazon’s queueing service (SQS) is not suitable to be applied in this case.
SimpleDB has also proven not to be very suitable for the problem due to the high complexity solution that would be required to meet our needs. As so, we implemented our own solution for both queueing and data persistence.

During the crawling stage we managed to crawl a graph containing 50 million nodes and 1.8 billion edges, which is roughly two-thirds of Twitter’s estimated user base. Users that could not be crawled include private accounts and blocked accounts. Retrieving these 50 million users took 3 weeks using 5 EC2 instances. Due to Twitter’s dynamic nature, its social graph changes continuously. This means that changes in this graph cannot be detected without doing a full re-crawl, during which other changes may occur. However, we cannot incrementally update the gathered data, without completely re-crawling the social graph, as Twitter does not offer incremental updates through its API.

We have seen that resource usage on our 5 crawler instances was generally low. If Amazon would have allowed to attach multiple IPs to a single instance, a single computing instance would have sufficed, significantly lowering costs.

As a result, we have implemented a relatively cheap solution for data acquisition and analysis using the Amazon cloud infrastructure.

We have found that deploying an application in the cloud is easy, though not all Amazon cloud services are useful in this kind of problem. For example, if a lot of relatively short messages are needed or longer retention is required, it is more feasible to implement a dedicated queueing system than to use Amazon’s queuing service (SQS). Despite this, cloud services can be used to reduce costs and streamline the development process and execution, and also to make it possible to address problems that can be previously solved only on dedicated clusters of servers what made it very expensive for an average user, e.g., small and medium enterprises.

As mentioned before, the computation of PageRank over the acquired dataset can be done on a single machine. The distribution and aggregation of results between multiple nodes would take a significant amount of time, which cannot be justified as long as it is possible to fit the entire dataset in the main memory on a single machine, making the network bandwidth to be a bottleneck. When it is no longer possible to fit the dataset in memory, it is either necessary to let the virtualized disk or the network be the bottleneck. As a part of our future work, we plan to find out whether solutions based on distributed computing infrastructures such as Amazon’s Map/Reduce or a custom Hadoop cluster will further reduce the cost yet maintaining achieved performance in the same time.

While extensive analysis of the data that was gathered is beyond the scope of this paper, we plan to apply graph analysis algorithms, such as used in Mislove et al. [8] and Ahn et al. [1], in future work.

Improvements in the ranking algorithm can be made by attaching weights to relations between users. These weights may be based on the number of conversations between users or the number of times that the status update of a certain user is re-posted (re-tweeted) by other users.

References