Twaler: A Crawler for Twitter

Alexander Liu
203304719

UCLA Computer Science Department
Masters Comprehensive Exam

Monday, July 12

Table of Contents

Abstract

Social networks present a new area of research interest. However, crawling and storing large amounts data from social networks remains a non-trivial, time-consuming task. In this paper, we present Twaler, a crawler/database system for the micro-blogging social network Twitter. We describe our methods for crawling a social network, and discuss issues with performance, scaling and updating data. Our system is designed to be easily extendable to accommodate various types of usages. The result is a self-sustaining crawler and database system that provides fresh and large samples of Twitter data for social network related research.

1 Introduction

From forums, blogs, to emerging social networks, the web's shift towards user generated content is fundamentally changing the way we interact with the web. Amongst these services, micro-blogging has recently emerged as an important form of communication. Micro-blogging can be accurately described as a form of blogging in which the message forms are usually brief updates (typically less than 200 characters) that could be sent to friends and/or interested observers. The simplistic nature of micro-blogs has prompted many to ponder the reason behind its success: Some point to its real-time nature and ease of consumption, others have provided more in-depth analysis based on its usages and demographics (1). Regardless of the how and why, it is undisputable that micro-blogs are quickly emerging as an integral part of web communication.

The most notable and currently largest micro-blogging service is *Twitter*[1]. Twitter employs a social-networking model where users may post updates, or *tweets*, under 140 characters on their account. At the same time, users may choose to *follow* other users, for which they will receive update messages each time the targeted followee posts a new update. The incredible growth of Twitter has been well documented. According to a report published by Nielsen Wire[2], Twitter is the top relative-growth website in 2009, and its current ranking by traffic is 11th on the internet[3]. Beyond the remarkable statistics, Twitter has also established its place as a cultural and social phenomenon. Usages of the service have ranged from the president of United States using it as an official platform[4], to being employed as an important emergency message system in times of crisis[5].

For researches in the academia, Twitter further presents a unique opportunity in social network related research. The sheer numbers of the users, the real-time nature of micro-blogs, the openness of its social graph, combined with the simplicity and structure of its messages suggest that Twitter data can be of great interest to researchers of all fields. On top of that, Twitter has strongly supported its public API since its inception, allowing developers and researches alike easy access to its data. A number of research has already been published based on Twitter, including using Twitter for real-time detection of earthquake centers (2) or attempting to quantify and rank the influence of Twitter users through ranking algorithms (3). Yet, even with the friendly API, acquiring and managing a large set of data sample remains a time-consuming chore. To begin with, engineering a

[1] Other major services include: *Plurk*, *Jaiku, Tumblr*
[2] http://blog.nielsen.com/nielsenwire/online_mobile/twitters-tweet-smell-of-success/
[3] http://www.alexa.com/siteinfo/twitter.com
[4] http://twitter.com/BarackObama
[5] http://www.cnn.com/2010/TECH/01/12/haiti.social.media/index.html

crawler for a social network is, on its own, a difficult problem. Generating a structured dataset for easy of investigation and inquiries is also a non-trivial task. On top of that, there are challenges such as the API rate-limit imposed by Twitter[6], its lack of support of community induced structure such as *mentions* and *retweets*[7], and many more.

In this project, we introduce Twaler, a system built with the goal of simplifying the data collection process for researchers. It provides: 1) a large-scale, up-to-date, and growing database for Twitter related research. 2) Software tools for crawling data from the Twitter API. 3) A solid framework that can be extended and modified for different research purposes. We believe that by providing easy access to a large set of up-to-date, structured Twitter data, researcher could save a significant amount of time and effort on data collection and focus purely on the investigations.

[6] The default limit rate is 150 API calls per hour, but developers can sign up for 20,000 API calls with Twitter's approval

[7] Since its inception, the Twitter community has developed unique jargons and usages that have become somewhat of a standard in micro-blogging. For example, (RT @ *user_name*) refers to a *retweet*, a reply to a tweet from another user; (#*hashtag*) refers to a *hashtag*, which is used for categorizing the topics of tweets. Twitter has recently introduced an "official" *retweet* button that is extensively supported by the API. However, historical tweets where *retweets* are simply text headers are not recognized as retweets by Twitter.

2 Twitter

2.1 Introduction to Twitter

Twitter is a micro-blogging service which allows its users to publish text messages limited to 140 characters, also termed *tweets.* The messages can be read publicly (or, although quite rare, privately to only the writer's choice of friends) by any reader on the web. If a reader is interested in receiving the tweets of a particular writer, he can simply choose to *follow,* or *friend* (a misnomer since in Twitter, a friend relationship is not bi-directional and requires no consensus) the particular user. After that, the reader will receive real-time tweet feeds from the writer. A typical Twitter user has the dual role of both publishing updates, and following multiple accounts. The user typically observes in a stream of real-time updates, sorted by their publish dates, on their Twitter feed wall (see figure 1 for a screenshot)[8]. Experts and technologists have long been curious about the factors contributing to the widespread success of Twitter. Some of the speculated factors include:

1. Real-time Information: Twitter feeds are updated instantaneously, making it a better source for real-time news than the actual web and/or traditional news sources.
2. Message Limit: The imposed limit encourages people to post short, stream of consciousness updates instead drawn-out blog entries. Information is compact and can be digested instantly.
3. Mobile Integration: The restricted format of tweets matches the modern user's lifestyle of digesting information on the go. Users also feel at ease to publish short musing tweet posts on the go.
4. Link Sharing: With link-shortener services such as *bit.ly*[9], Twitter has become a popular medium to share articles on the web. Users typically write a short synopsis of the content along with the link.

[8] It must be noted that Twitter users are not restricted to using the twitter.com homepage for sending/receiving tweets. In fact, 68% of Twitter users use 3rd party clients either on the PC or on mobile phones to access Twitter.

[9] Twitter has plans of implementing their own link-shortener, supposedly for security reasons.

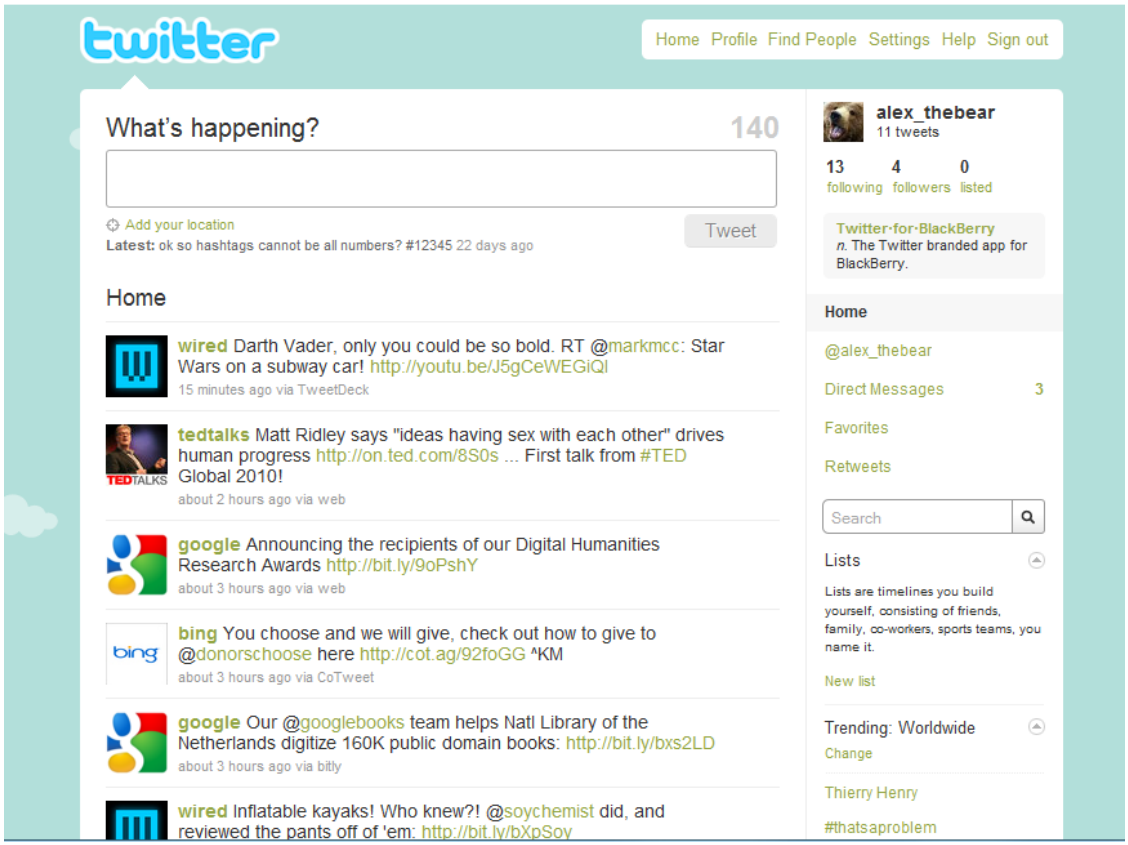

Figure 1: Typical Twitter User Experience: Users may observe tweets from various sources, as well as post their own update under the *What's happening* box

Despite its original simplistic model, the Twitter user community has developed several conventions of posting tweets that have expanded the structure of Twitter.

- *Mention:* When publishing a post which references another user, the user is referred to by their username prefixed with the '@' character.
- *Retweet:* A special case of mention, when a user repeats another user's post. In this case, the tweet is prefixed by 'RT @' followed by the user name of the original poster.
- *Hash-tags:* If the poster intends to categorize a post for easy of search, at the end of the tweet, the poster puts a '#' character followed by the assumed category of the post. For example, *'#badminton'* will tell the readers that this post is about the sport badminton. Users can input in the search box '*#badminton*' to get all posts that are self-categorized as related to badminton.

Recently, Twitter has added the feature of *lists.* A *list* is a user-generated list of other users. These lists help users focus on the posts of a subset of their friends. In general, these lists are meant to be used as a method of categorizing sources of Twitter

feeds. For example, a user may generate a list *'photography'* that is populated with writers that frequently tweet on topics related to photography.

2.2 Twitter API

One feature that has been crucial to the growth of Twitter is their well support API. As a result, there has been a wealth of Twitter applications developed around the Twitter platform. Some examples include: *Twitpic*, which streamlines the process of posting links to pictures as tweets; *Tweetdeck*, a Twitter client that provides more functionality compared to the native website; *Tweeter News,* a website that aggregates news results seen on Twitter.

Twitter offers an extensively supported REST API. Developers can gain access to publicly available data such as tweets or friends of a particular user in structured XML/json format. Other than the REST API, Twitter has been pushing for a new paradigm: the Streaming API. Streaming API starts with the client side initiating a connection to the server. Once the connection is established, the server pushes data onto the client whenever there is an update until the client side terminates the connection. This lessens the server load by preventing the client-side from having to constantly fire pull requests to the server. It is also a step towards what Twitter terms the *real-time web*, in which information is received and updated in real-time.

The openness and connection that Twitter has with its developer community further signifies a trend for social networks to transition to platform providers (*see Facebook).* Data is becoming more open and much more easily accessible through these platforms. We are grateful for this as our efforts to build a crawler were greatly simplified by the REST API.

3 Twaler Design

Twaler is designed with three major goals in mind:

1. To create an efficient crawler that interfaces with Twitter's REST API and manages problems such as rate-limiting and connection errors… etc.
2. To create an up-to-date database with a snapshot sample of Twitter data where information can be efficiently retrieved and updated.
3. To create a software framework that not only maintains the process of crawling and populating the database, but can also be easily modified and extended for more specialized data collection.

In the following section, we discuss some of the design choices and musing/challenges for reaching each of these goals. In the next section, *Twaler Architecture*, we go over the implementation of the design in detail.

3.1 Crawling Twitter

3.11 Crawling a Social Network

A social network is structured very differently from the ordinary web. Whereas the ordinary web is centered on web pages, the social network is centered on its users. When crawling the ordinary web, we typically extract hyperlinks to other web pages and expand the crawl link by link. In a social network, the key is to find links between users and follow them to discovery more users.

In Twitter's case, the most obvious link is the *friend/followee* link. But there are some other, less obvious links. For example, we can also parse through a user's tweets to discover the names that the user has *mentioned,* or *retweeted.* With the introduction of lists, we can easily obtain new users by observing the list constituents. All of these sources are crawled in Twaler. Through traversing these links, we can slowly build-up and expand the social graph.

3.12 Expanding the Social Graph

Suppose that starting from a seed, we have obtained a new list of users from the friend's list, mentions… etc., and we want to crawl these users. Unfortunately, Twitter limits its API access rate to 20000 calls per hour[10], how do we pick and choose the "important" users to crawl, so that our crawl efficacy can be maximized?

[10] As mentioned previously, this is after obtaining a developer license from Twitter

We believe the solution to this problem differs greatly depending on the target sample data of the user. We could traverse these links in a breadth-first method typical to most simple crawlers. This would allow us to obtain a complete picture of all the out-going links to each and every user, one user at a time. The drawback of this method, however, is that our social graph would expand at a snail's pace. And since the method does not discriminate between users, we may waste time on a lot of *non-active* users[11].

Another method that we could use is to use some sort of ranking mechanism on the list of users to be crawled. A very simple mechanism we chose is to rank the un-crawled based on their number *followers* amongst the data that we already have. Intuitively, a user that has more followers probably has more useful content compared to a user with one follower. This is comparable to ranking the next node to crawl based on the number of incoming links, or an un-weighted Pagerank. The benefit of this method is that it would allow us to prioritize users with more useful tweets over the non-active users with little to no useful content. Ranking the users this way is also fast and efficient, and easy to implement. The primary drawback, however, is that one could argue that we are choosing only the *popular* users to crawl, and thus we aren't obtaining an accurate sample of Twitter, which includes spam accounts, non-active accounts… etc. Our reply is that, for our current research purposes, we would prefer obtaining data from the more active accounts than the plethora of non-active ones out there.

A number of other ranking mechanisms could also be used. Since we mentioned using an un-weighted, single iteration Pagerank, it seems obvious that the real Pagerank can also be used. A version of Pagerank for Twitter has been implemented by our lab, but it was not used due to performance considerations. Given the modular design of Twaler, one could easily change the method in which to generate new seeds.

3.13 Updating the Social Graph

Another important goal in Twaler is to maintain up-to-date information for its constituents. An active Twitter user could update his/her account several times a day, while inactive users may never update their accounts. Between crawling new users and crawling new lists, the requirement to update users further puts additional pressure onto the already limited API calls.

[11] A study shows that 73% of Twitter accounts are inactive.
http://www.afterdawn.com/news/article.cfm/2010/03/10/73_percent_of_twitter_accounts_are_inactive

A common method for web crawls to deal with updates is to measure the *freshness* of current data, and use exponential back-off methods to gauge the frequency of page updates in order to maintain the freshest database (4). Regrettably, we did not have time to implement a method of re-crawling users. Our database, however, does support manual updates according to timestamps, so it maybe an area worth investigating in the future.

3.14 Parallel Crawling

Twitter limits the developer's access to the API with 20,000 requests per hour based on the IP address. Our experiments have shown that it is difficult for a single continuous crawl process to even reach that cap. The direct solution to this problem would be to run parallel crawlers.

Cho and Garcia-Molina (5) have proposed parallel crawler architectures for the web. They listed three architectures to manage the crawl process amongst parallel crawlers: the *independent* architecture in which no coordination exists amongst crawlers, the *dynamic assignment* architecture, where a central coordinator acts a task delegator, and the *static assignment* architecture, where the graph is partitioned and assigned to each crawler. For our case of crawling a social network, it turns out that the dynamic assignment architecture works quite well (Similar work has already been done by Chau et al (6)).

For our system, we separate the process of crawling the targets and generating new targets. The centralized database spits out new seeds using the methods described in the previous section *Expanding the Social Graph*, and assigns the seeds to the crawlers. The crawlers work on these seeds independently, and when all the seeds are completed, new seeds get generated again. The crawlers are multi-threaded, thus are capable of opening up multiple connections in parallel. Besides improving general crawl performance, it also circumvents the potential problem of a halt to the entire crawl processes due to a single connection hang up. Our system also supports using multiple machines as crawlers, although currently this part is not developed fully. In the next section, we give a generalized view of our current crawl architecture.

3.15 Twaler Architecture

As depicted in figure 2, the architecture is a straightforward web information crawler/database system. We began by feeding a series of *seed files* into the *crawl* component. The seeds consist of users or lists for which we want to obtain information on. The *crawl* component downloads the data through the REST API and stores the raw XML files on disk. The *process crawl* component parses through the raw XML files and

generates intermediate files for fast and efficient data loading into the backend database. The files are then loaded into the database through the *load crawl* component. Finally, once we have crawled through all the users and lists specified in the seed files, *generate seeds* accesses the database and generates a new set of seeds (with a mixture of both new seeds and old ones for keeping up to date information). With the set of new seeds, the cycle starts over again.

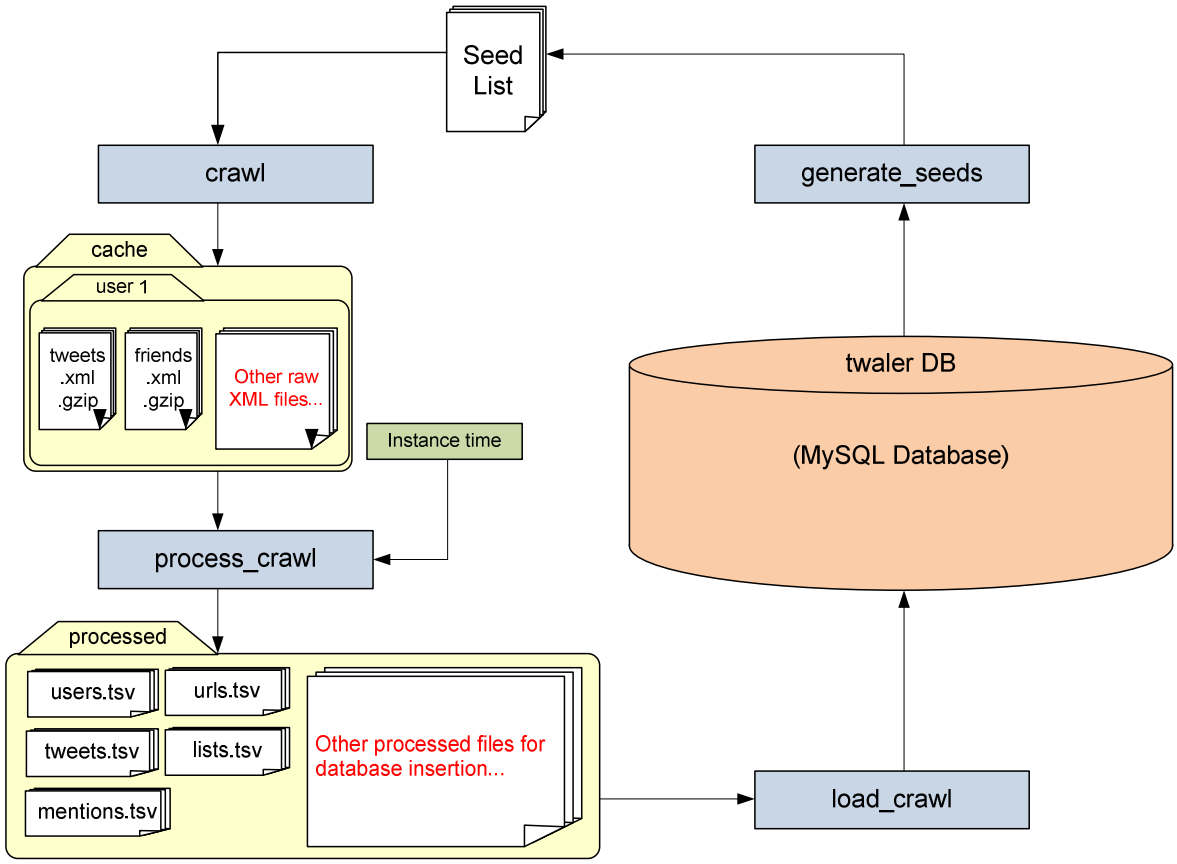

Figure 2: Overview of Crawl Process

3.2 Twaler Data

The Twitter API grants developers access to a large variety of data. For the scope of this project, the data crawled and stored is tailored towards the needs of academic research. Thus, we currently only crawling the user profile, tweets, friends, and list information.

Other interesting data that could be accessed include local trends, geo methods…etc. However, at this moment, these sets of data are not crawled in Twaler.

3.21 Twaler Database

Twaler uses MySQL as its backend database system and organizes the data into relational form. This gives us a free open source database that is familiar to most academic researchers. The schema for the database is shown in figure 3.

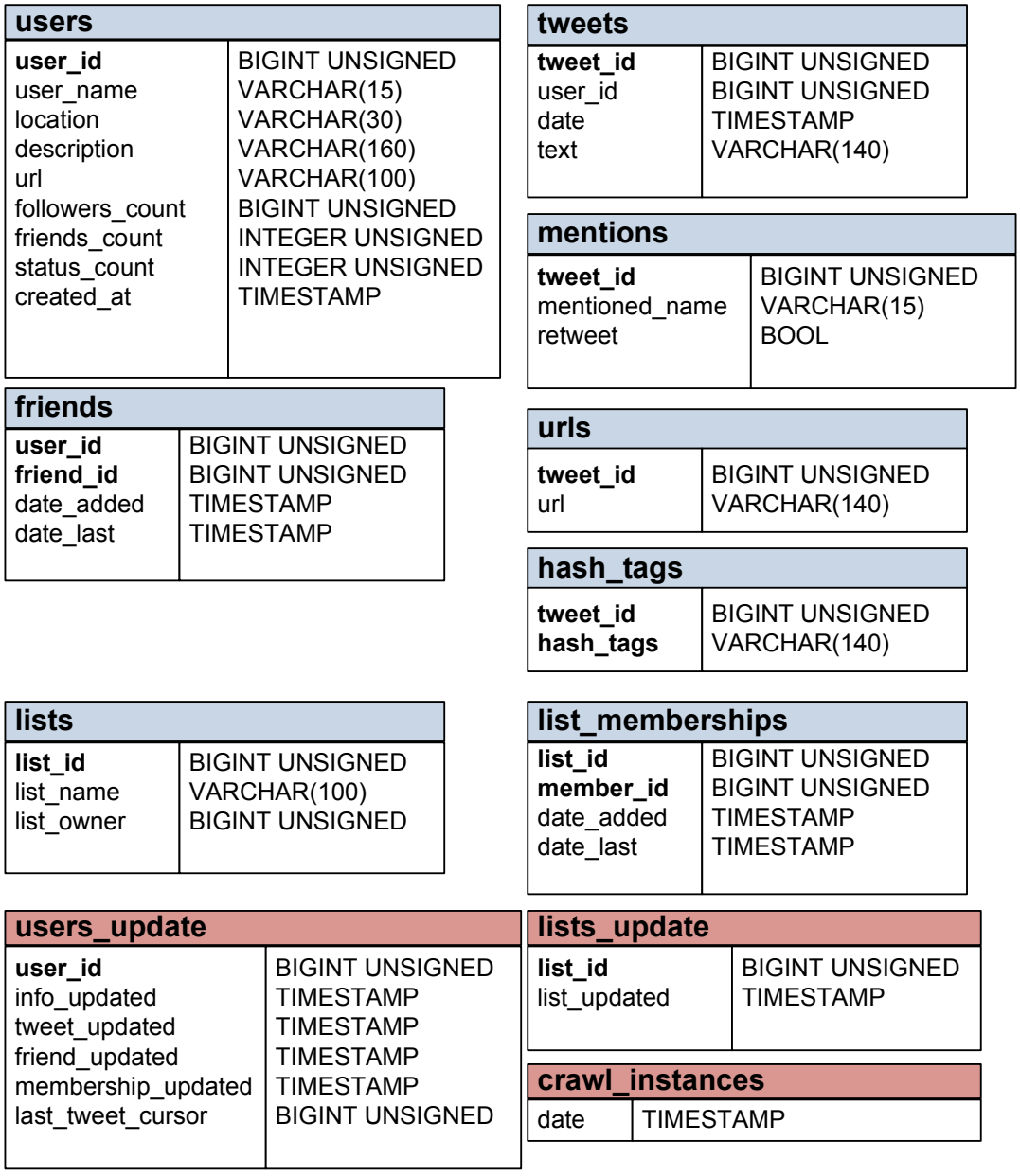

Figure 3: Twaler DB Schema

Couple items worthy of note:

1. Specialized symbols/usages developed as a result of user community conventions (as opposed to being inherent in Twitter design) are parsed from the tweets (i.e. @ mentions, # hash tags)
2. The url links are also parsed and stored from tweets. However, almost all the url links from tweets are processed through some sort of link-shortener service. As of now, we have not chosen to follow the shortened link to obtain the actual url.
3. Twitter users are specified by both a unique user_id and a unique user_name. The user_id is immutable, but the user_name can be changed. A problem occurs when a tweet "mentions" a user_name that has since been changed. We would not be able to trace the user which the tweet actually refers to. There are many ways of dealing with this problem, but due to time constraints and performance considerations, we have decided to ignore this problem for the time being.
4. An inherent problem in crawling is the trade-off between maintaining up to date data and expanding the crawl database. As seen in figure 3, meta-data tables *users_update, lists_update,* and *crawl_instances* are created to keep track of updates.

Figure 4 shows some of the simple queries that could be performed over the database.

- **Getting all tweets of a user, with known user_id:**

```
SELECT    tweet_id, date, text
FROM      tweets
WHERE     user_id = 12345
```

- **Getting all tweets of a user, with known user_name:**

```
SELECT    t.tweet_id, t.date, t.text
FROM      tweets t, users u
WHERE     u.user_name = user_name AND
                t.user_id = u.user_id
```

- **Getting all the usernames that has been mentioned by a specific user (@/ RT):**

```
SELECT    t.tweet_id, m.mentioned_name
FROM      tweets t, mentions m
WHERE     t.tweet_id = m.tweet_id AND
                t.user_id = 12345
```

Figure 4: Example SQL queries

3.22 Representing Updates and Temporal Information

Representing Updates

To deal with the problem of maintaining up to date data, we must first design a system to maintain the update records for each piece of data.

For each seed, several API calls to the Twitter server are made. The data returned from the API call would consist of the latest information on that specific user. Thus, we keep an update record for each user, entering the time when the last update was made. Since our crawler allows for specific crawls (i.e. only download data on friends or tweets… etc.), we have a column for each crawl type, denoting the last time a specific data type was updated for the given user.

There are two significant reasons for us to maintain this data:

1. The user of the database needs to know the freshness of the data. In fact, a user can specifically restrict the data used to the latest crawl, or even another earlier specific date.
2. Ultimately, the goal of Twaler is to maintain an up-to-date, accurate sample of Twitter. Update records is necessary for the system to pick and choose seeds to update.

In terms of how the timestamps are obtained and organized: The Twaler system organizes its crawls based on "instances". During each crawl instance, a certain number of seeds are crawled, processed, and loaded into the database, all timestamped based on the same time instance. This simplifies file organization as well as the engineering of the system.

Temporal Information

In social graph researches, it is sometimes useful to maintain historical records on the social graph in order to observe trends and changes. For Twitter, the social graph specifically refers to the *friend* relationship between users.

The API call for friends simply returns the list of user_id's for each user; it does not provide information on the additions/removals to the list. One method of capturing the historical changes is to store the friend lists for each and every crawl instance. Using the timestamps obtained from each crawl, we would know the state of each friend list at each crawl instance. However, this method scales horrendously, as the size of the friend list of a single power user can run up to thousands of friends.

An alternative method is to simply record two timestamps for each friend relationship. One which describes when the relation was first observed in the database and one which describes the last time the relation was observed. Using this method, we can get a rough estimation of when the relation was first established, and if/when the relation was terminated. Note that this method cannot capture the case where a relationship is established, then terminated, then re-established again… and so forth. However, we reason that, for the most part, the friendship timeline between two users should be a continuous timeline. The amount of extra space required for this method reduces from being dependent on the number of crawl instances to being only dependent on the number of users. Finally, for Twitter lists contributors, in which users are constantly added and removed, we also apply the same kind of temporal data structure.

Figure 5 shows some temporal queries that could be made on the *friends* table. Note that, when combined with the information in the update tables, we can restrict ourselves to only getting the freshest friends list of any particular user.

- **Getting the friends of a user (latest update):**

```
SELECT      f.friend_id,
FROM friends f, users_update u
WHERE       f.user_id = 12345 AND
            u.user_id = 12345 AND
      f.date_added = u.friend_updated
```

- **Getting the friends of a user on 2010-01-01**

```
SELECT      f.friend_id,
FROM        friends f
WHERE       f.user_id = 12345 AND
                  f.date_added < '2010-01-01 00:00:00' AND
                  f.date_last > '2010-01-01 00:00:00'
```

Figure 5: Example temporal SQL queries

4 Twaler Implementation

Twaler is implemented with separate components that can also be used independently apart from the system. The benefit, besides code modularity, extensibility and readability, is that researchers can easily modify and specific pieces depending on their research goals. This was especially important, since we cannot predict what type of target data would be required for each research project.

The entire system is implemented in Python 3.1 and designed to run on Linux or other Unix-like systems. We chose the programming language Python because of its renowned code-readability as well as easy of extension. One of the primary drawbacks of using a scripting language, the performance, is not an important factor here since the primary bottleneck in our architecture stems from the HTTP requests required to access the Twitter API.

Each component is a single python script file on its own. A *config.py* file acts as the "header" of some sorts, providing helper classes that are shared across the components. The file *twaler.py* acts as the main program and synchronizes between all the components. Using the program, or any of the components, requires the user to provide a configuration file as an argument. The configuration file should have all the arguments required to run the component. Below, we go through the design and use cases of each the components.

4.1 config.py

Config.py consists of the following helper classes that are used across various components:

- *logger:* a class that specifies the format and output stream to which to print the log to. All classes use this.
- *Timefunctions:* a group of static functions simply used to convert various time formats between each other
- *Cache_accessor:* a wrapper that manages the way crawled data are stored onto disk. During each crawl instance, we may go through up to 20,000 users and generate a larger number of files (in zipped XML format). The cache_accessor organizes files by generating a directory for each user id. To avoid the case where a folder has 20,000 directory under it, the cache_accessor also generates intermediate directories by hashing the last three digits of each user id. For the caller function, the caller can simply call to either insert or obtain all the files of a

particular user id. The cache_accessor also allows the caller to iterate through all the users under a specific directory.

- *Mysql_db:* a wrapper that allows the user to perform MySQL inserts/lookups to a specified MySQL databased and table.
- *File_db:* a wrapper that allows the user to perform inserts to a file using the same interface of Mysql_db. This is created primarily to provide an alternative to using the default MySQL database as the storage system. This class is also used in *process_crawl* to generate tsv files to load to the MySQL database.
- *Parse_arguments:* a function that parses both command-line arguments and arguments from the configuration file.

4.2 crawl.py

The task of *crawl.py* is to read a seed_file consisting of a list of user_ids or list_ids, perform the necessary API calls to crawl the seeds, and store the downloaded data on disk.

Seedfile Format

The user may specify what type of information is the target of each crawl. For example, if we pass a text file with the following contents as the seeds:

```
f       123
t       456
l       789
7654321
        …
m       123     my_list
        …
```

The first line with a prefix *'f'* tells the crawler to only download the friend list of user *123*, the next line, prefixed *'t',* only downloads the tweets of user *456*, and the prefix *'l'* only downloads the lists the user belongs to. In most usage cases, however, forgoing the prefix will download the friends, tweets, and list membership of the seed, in this case the user *7654321*.

For lists, it involves a little more. The user specifies that a list is to be crawled with the prefix 'm' (the **m**embers of the list are the data being crawled). The user then needs to provide the user id of the list owner, as well as the name of the list or the list id. This is necessitated because the API call for lists requires the list owner's identification.

Multi-Threading

The crawler parses the seed file and crawls each seed accordingly. To support concurrent crawls, we use the python module *threading* for thread generation. The overall pseudocode of the crawl process is shown in figure 6. Essentially, the main thread begins by creating the specified number of worker threads. The main thread then begins to parse the specified seed file, and puts each seed on a thread safe, shared queue (from the python module *queue*). The workers are constantly waiting (blocking wait) for items on the queue. When the worker takes an item from the queue, it parses and downloads the XML files accordingly. Finally, when the seed file completes, the master sends a terminating thread to each of the workers and exits.

```
Main Thread:
      Generates worker threads
      Read in seed file
      For each seed,
            Place the seed on a queue shared between threads
      Send terminating signal to threads when seed file is complete
      Exit

Worker Thread:
      Repeatedly, until terminating signal is received
            Gets seed from queue
                  Blocks if none exists
            Fetches data according to seed type of user id
      When terminating signal is received, exit

Parameters:
      -seed file
      -directory to store downloaded files
      -directory of logs
      -number of worker threads
```

Figure 6: Pseudocode of crawl.py

HTTP Connection Errors:

For every seed, HTTP requests are sent to the server and responses are received. The Twitter API website lists all the error responses possible, and what each implies[12]. We deal with each error accordingly. For example, 503 imply that the Twitter servers are overloaded with requests, in which case we simply back off for two seconds and try again. On the other hand, if the response implies that the rate-limit has been reached, we will pause everything until the required time has passed. Certain API's require authenticated users to access, in these cases we hardcode a username and password as part of the API call.

[12] http://apiwiki.twitter.com/HTTP-Response-Codes-and-Errors

4.3 process_crawl.py

Process crawl goes through the target cache folder and parses through each XML file to get the data required to insert into the MySQL database.

Users can choose to either directly insert each row into the MySQL database (this proved to be very slow), or generate intermediate tsv files first, and then load them into the MySQL database using *load_crawl.py*.

4.4 load_crawl.py

Load crawl goes through the specified directory which contains all the processed tsv files and simply loads them into the MySQL database.

Bulk Updating Specific Columns

MySQL supports replacing data when a row with the same key already exists in the database. However, there is no way of only updating specific columns in bulk through using *LOAD DATA*. This scenario comes up when we want to update the *last_updated* column in the *friends* and *list_update* table. The most efficient way of bulk updating these tables seems to be: 1) load the file into a temporary table. 2) Perform a bulk *INSERT* from the temp table to the target table, with the option *ON DUPLICATE KEY UPDATE TABLE key=value*. 3) Remove the temporary table.

4.5 generate_seeds.py

Generate seeds performs the job of generate the next batch of user_ids, lists to crawl. Currently we generate user seeds using the ranking method described the previous section. In terms of choosing users to update, we currently use a simple method to choose the least updated users. For lists, we choose to update lists randomly. All of these tasks can be executed with MySQL queries.

The user may specify the number of seeds that fits into each file. There are two purposes for splitting seeds into multiple files: 1) in Twaler, we define each crawl instance by each seed file. An instance is completed when a single seed file is completed. Splitting the seeds into tinier instances makes error recovery easier and manageable. 2) For using multiple crawl machines, we can simply send separate seed files to each machine. This is a way of delegating seeds to ensure that no overlap occurs.

There is a lot left to be desired in the future for this component. Most notably, the algorithm for generate new seeds and updating old seed can be very non-trivial. Regrettably, due to time constraints, we had to make do with our current simplistic model.

4.6 twaler.py

Twaler executes the entire process, runs each component accordingly, and keeps the entire process synchronized. Twaler begins by detecting seed files under the specified directory. If a seed file exists, Twaler then generates a crawl instance, in which the seed file will be crawled, processed, and the results loaded into the database. This process continues until all the seed files in the seed directory have been crawled and processed. In this case, Twaler will call the *generate seed* component to generate new seeds and fill the seed directory, and restart the process all over again. Figure 7 shows the various states of the Twaler process.

There are several points worth noting. After completing the *crawl* portion of an instance, Twaler immediately generates a new instance and begins the crawl job on the new instance **while** processing and loading the instance that has just finished crawling. The reason for this slight optimization is that, when dealing large amounts of data, the process and load portion end of taking a significant amount of time and CPU processing. On the other hand, the crawl portion is still by far the biggest bottleneck, but only because of the network delays from performing API calls and the rate limit imposed, the CPU utilization is actually quite low. Thus it is sensible to be processing and loading while the next crawl process is going on to avoid wasted time in processing and crawling. There should never be a case where more than two instances exist at the same time, since crawling the data will always take longer than processing and loading the data.

On the other hand, in the generate seeds state, all processes are halted. This was a design choice. We will be performing heavy duty queries in the generate seeds phase. If during this time, the load data phase kicks in, then the database will almost come to a halt.

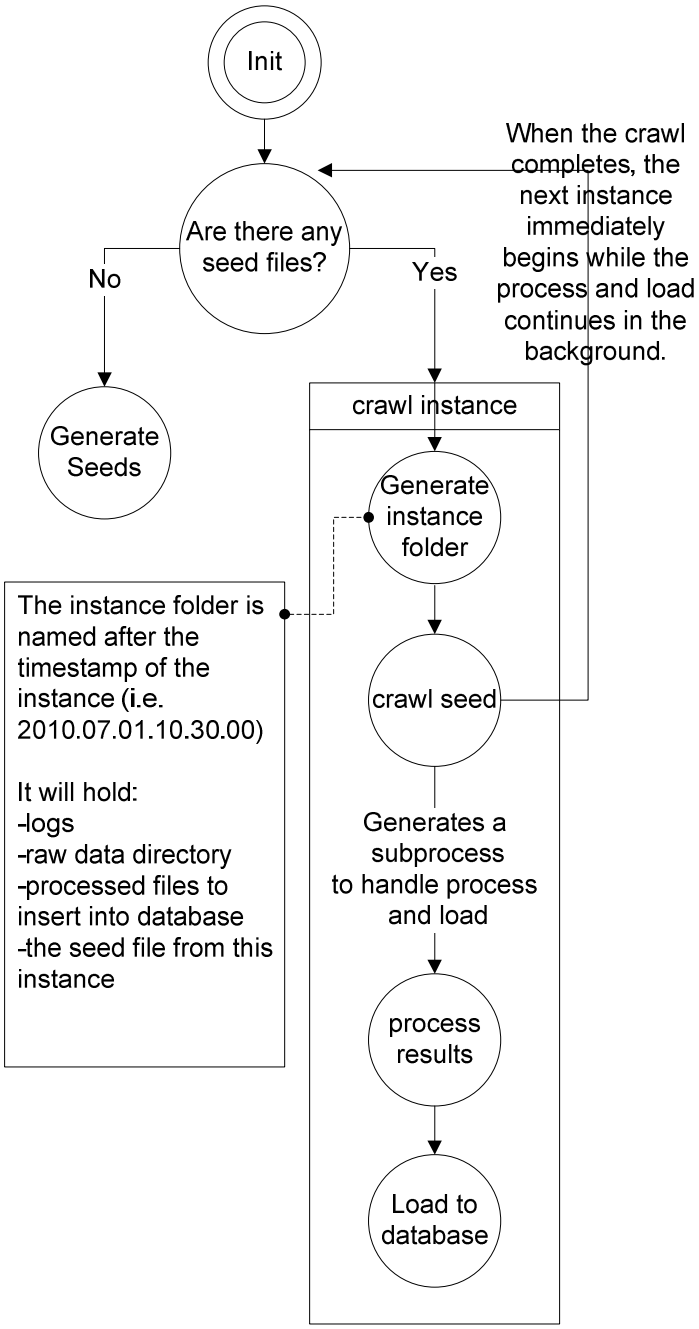

Figure 7: Twaler.py's process state diagram

5 Conclusion

In this project, we set out to build a crawler and database for Twitter with the goal of: 1) Investigating the unique problem of crawling a social network. 2) Create a solution for the difficulties in data collection for researching a social network. As a result, we developed Twaler, a system that has constantly updated data that could be used by researchers, and also tools that may easy the process of data collection for the research community. Although we have accomplished our goal of developing a working architecture and program, we have many problems that we hope to solve in the future.

5.1 Current State

Currently, we are in the process of importing a large amount of data into the Twaler DB. The Twaler DB currently has data from previously crawling ~1.7 million users, with a total number of 323 million tweets, 242 million friendship relationships, 2 million lists and their members, along with data on urls, mentions, and hashes. The data has been used by the SNORG (Social Network Online Research Group) in its investigation of the Twitter social graph (7), which looked at the friendship, mention, and retweet links between users in an attempt to identify influential Twitter users. The work on Twaler comes from close collaboration with the group's researchers. We are also crawling new data, albeit at a smaller scale. We hope to run the Twaler process on full-scale continuously once the importing process completes.

5.2 Future Work

As mentioned many times throughout the paper, there is a lot of work that can be done. The most significant area lies in the mechanisms of updating old profiles. As mentioned previously, there is established literature on the same problem for generalized web crawlers, but we simply did not have enough time to fully develop a specialized solution for social network crawlers.

Although Twaler can easily support multi-machine crawling, that part of the functionality has not been fully-implemented yet, and problems may always arise when dealing with parallel machines. Also, although we should never reach the scale where the MySQL database's scaling limitations become an issue, but one should beware that

scaling issues in MySQL has prompted Twitter to migrate to using Cassandra as their database management system.[13]

Currently the Twaler process is nom-interactive. Meaning that once the process starts, the only interaction we have with the process is to terminate it manually. It would be nice to implement interactive functionality where a user can more easily check the status of the crawler, and perhaps pause or modify the process of crawling.

5.3 Final Thoughts

Through this project, we experience the difficulties of building a full-fledged crawler. We realized that, even after restricting ourselves to a well-structured domain, there are still unique problems and situations to prove that crawlers are a complicated subject.

On the other hand, Twitter's open API has made this process a lot smoother. The openness of Twitter data stands in huge contrast to the closed policy of another popular social network site, *Facebook.* Even with their relatively strict privacy policy, Facebook is still consistently getting vilified for their perceived loose security on private data. There seems to be two opposing voices advocating different approaches regarding the future of web data. One voice seems to gravitate towards an open web, while the other voice advocates privacy and protection of personal information. Of course, it is not fair to compare blogs like Twitter, in which users post things with the intention of reaching a mass audience, to more private services Facebook, in which users may engage in more personal conversations with personal friends.

It was also interesting to have studied the rise of Twitter. What looked like a trivial concept on the paper has developed into a full-fledged, important means of communication for many of its users. Critics can debate the merits of micro-blogging and argue over the true reason behind Twitter's success. What intrigues me is that, reading the incessant updates from the public Twitter feed, I can't help but feel like I am reading the collective stream of consciousness of mankind.

[13] See: http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king

Acknowledgements

Uri Schoenfeld, Michael Welch, and Vince Chen have all contributed the code of Twaler.

Much credit must be given to Uri Schoenfeld, who has offered many advice and ideas in the design of the system. He is also the other half of the “we” that I have referred to throughout the paper.

References

1. **Akshay Java, Tim Finin, Xiaodan Song, Belle Tseng.** *Why We Twitter: Understanding Microblogging Usage and Communities.* San Jose : Procedings of the Joint 9th WEBKDD and 1st SNA-KDD Workshop, 2007.
2. **Takeshi Sakaki, Makoto Okazaki, Yutaka Matsuo.** *Earthquake Shakes Twitter Users: Real-time Even Detection by Social Sensors.* Raleigh, North Carolina : WWW2010, 2010.
3. **Jianshu Weng, Ee-Peng Lim, Jing Jiang, Qi He.** *TwitterRank: Finding Topic-sensitive Influential Twitterers.* New York City, New York : WSDM'10, 2010.
4. **Junghoo Cho, Hector Garcia-Molina.** *Effective Page Refresh Policies for Web Crawlers.* s.l. : ACM, 2003.
5. —. *Parallel Crawlers.* s.l. : In Proceedings of the Eleventh International World Wide Web Conference, 2002.
6. **Duen Horng Chau, Shashank Pandit, Samuel Wang, and Christos Faloutsos.** *Parallel Crawling for Online Social Networks* . s.l. : WWW 2007, 2007.
7. **Uri Schonfeld, Michael Welch, Dan He, and Junghoo Cho.** *Deep Inside Twitter.* 2010.