Twitter Watch – Smart Collection of Twitter Data

Miguel Sozinho Ramalho
University of Porto
Porto, Portugal
github.com/msramalho
orcid.org/0000-0002-1350-2910

(other authors will be added to final version)

***Abstract*—This document describes Twitter Watch, an open-source tool that can be used to collect Twitter data through the Twitter API. We describe the architecture, implementation, and practical results of using Twitter Watch for collecting data of the Portuguese Twittersphere, gathering over 160 million tweets and over 7 million accounts. Twitter Watch is fully Dockerized, meaning it can run *virtually* in all major operating system, and is also capable of of saving regular snapshots of the collected dataset in Google Drive, and can notify of execution errors through Pushbullet API.**

***Index Terms*—twitter, twitter api, data collection**

I. Introduction

The first requirement for analyzing Twitter data is, well, having data. Although it is not uncommon for researchers to reuse already existing datasets for the validation of new approaches [1] [2], this option was not viable to us, as the most recent dataset focusing on the Portuguese Twitter context that we could find was from 2016. At the time of starting this work, we wanted to focus on the 2019 Portuguese legislative elections, occurring on October the 6th, 2019. Since no data was available for immediate use, we had to collect it. This section reveals how we achieved this goal and also how and why we developed a new Twitter data collection framework – Twitter Watch.

II. Tool Gap

Twitter has an Application Programming Interface (API) that can be used for a multitude of purposes, this work focuses solely on data collection for research purposes. Since our original goal when Twitter Watch was first envisioned was to focus on a specific context – the Portuguese Twittersphere– and on a particular political event – the 2019 legislative elections – we needed to find a tool capable of extracting a relevant dataset to conduct our study. We did not find such a tool. Indeed, we listed the requirements that such a tool should fulfill in order to generate a relevant dataset of both users and tweets:

With the above requirements in mind, we looked for tools that would either satisfy them or be flexible enough to accommodate them through easy changes.

Data Collection Requirements 1. Initial requirements

1) Capture a specified period of time;
2) Start from a list of relevant Twitter accounts and dynamically find new potentially relevant ones;
3) Restrict the collected data to the Portuguese content as much as possible;
4) Detect suspension of accounts as they occur;
5) Properly explore the API limits, as that is a potential bottleneck;
6) Save the data in a way that facilitates the subsequent analysis;
7) Ensure a coherent structure and avoid redundancy;
8) (Optionally) Allow regular backup of dataset snapshots;
9) (Optionally) Notify failures in real-time;
10) (Optionally) Ensure a logging mechanism to follow both progress and failures during the collection process;
11) (Optionally) Provide a visual interface that facilitates monitoring the collection process;
12) (Optionally) Be adaptable to different collection goals through an easy configuration;
13) (Optionally) Allow flexibility in how the data collection evolves.

On one side, we have commercial tools like Hootsuite [1], Sysomos [2], or Brandwatch [3] that are both commercial and abstract the access to the data, but focus on using the search endpoints by looking at hashtags or search terms, this means only a seven-day window of past data is available. Although this work had some initial efforts of data collection surrounding the election period (*cf.* Section III, p. 2), the usage of such approaches focusing on endpoints and a very narrow window for that collection discouraged the use of both that endpoint as the sole source of data and of tools that relied heavily on it. These observations mean these tools fail many of the mandatory requirements, the most limiting being 1, 4, 6.

On the other side, we have open-source tools like Social-

[1] https://hootsuite.com/
[2] https://sysomos.com/
[3] https://www.brandwatch.com/

bus [4], TCAT [5], sfm-twitter-harvester [6], or Twitter-ratings [7] that are more oriented towards research data collection. These tools, however, are limited. Socialbus and TCAT are configured to filter tweets by users or by topics/keywords, but these need to be specified beforehand, and any dynamic evolution is relying on restarting the system, therefore not meeting requirement 2. Sfm-twitter-harvester can be interacted with as either a REST or streaming API, this means it is more flexible but lacks the persistence desired when building a dataset, it can actually be seen as an abstraction layer above the API, and fails to meet requirements like 2, 6. These tools are still found to be too generic and don't add a lot of value for the current use case when compared to the API wrappers available like Tweepy [8] and python-twitter [9].

This initial desire to find a suitable tool was unmet, and the gap remained open. Before actually implementing a solution that would close it by fulfilling the above requirements, we hit some metaphorical walls that are described in the next section as a reference point for anyone interested in achieving a similar effort.

III. Failures

Initial approaches were *ad-hoc* and, unsurprisingly, faulty. Even before starting to collect data, we focused on understanding Twitter API, its different response formats and objects [10], endpoints [11], rate limits [12] and response codes [13]. In the end, we also developed a simple open-source scraper [14] that generates two JavaScript Object Notation (JSON) files containing error and response codes that can be used to better understand the interactions with the API.

Then, having chosen python-twitter [15] as the API wrapper to use for the collection process, we identified all the accounts from the Portuguese political context that fell into one of the following categories:

- Political party account;
- President of a political party.

The final number of accounts found was 21, and this process was conducted in October 2019.

Starting from this seed of accounts we designed a single page Jupyter notebook that underwent the following phases of:

1) (A) Get updated profile information on the seed accounts;
2) (B) Get all the followers of (A);
3) (C) Get all the followees of (A);
4) (D) Get the 25 most recent tweets from (A);
5) (E) Get the retweeters of (D);
6) (F) Get the retweets of (D);
7) (G) Get the 25 most recent tweets from (B);
8) (H) Get the retweeters of (G);
9) (I) Get the retweets of (G);
10) (J) Get the 25 most recent tweets from (C);
11) (K) Get 10 tweets for every user in the database that did not have collected tweets;
12) (L) Calculate the 10,000 most common hashtags;
13) (M) Use the search API to get 200 tweets for each hashtag in (L).

All the data was saved to a MongoDB instance, which has the out-of-the-box benefit of ensuring no duplicate documents exist, combining this with Twitter object model's 14 `_id` field, means that the redundancy requirement (7 in the requirements list) was easily achieved. We executed this script in a time window that encapsulated the October 6th elections.

Although this dataset is created coherently, there are a few subtle inherent limitations to the way the collection steps are designed. Firstly, by not having the full tweet history, we cannot conduct any analysis on how Twitter's overall usage varied through time (on the Portuguese context). The same goes for analysis of each user's usage patterns, and other analysis that require complete temporal data. Then, there is a hidden assumption that malicious activity will necessarily be within accounts that are either followers or followees of the seed accounts, or of the retweeters identified in (E) and (H). Then, (L) and (M) enrich and add variety, but their benefit is not exploited since no new accounts are expanded from those collected tweets. Also, and in line with a limitation common to all the open-source tools mentioned above, they did not allow for requirement 4 to be met, as the suspension of accounts was not easy to monitor or record. This approach was far from ideal to what we required.

As the initial research focus was on combining textual content with structural information, we also went down another insidious path – attempting to save followers and followees of every account in the MongoDB instance. This proved hard due to quickly reaching MongoDB's maximum document size of 16MB [16]. Working around this was not advisable [17]. A few options were considered, but we ended up going for something outside our comfort zone – Neo4J – a graph database designed specifically to save relationships between entities.

After setting up a Neo4J Docker instance, we started using the old collection process with a few code changes that would ensure the follower/followee relationships would be saved. We had some success, as is visible in Figure 1 (p. 3), we were able to capture accounts, and their follow relationships. However, this proved to be yet another dead end when we observed

[4] https://github.com/LIAAD/socialbus
[5] https://github.com/digitalmethodsinitiative/dmi-tcat
[6] https://github.com/gwu-libraries/sfm-twitter-harvester
[7] https://github.com/sidooms/Twitter-ratings
[8] https://github.com/tweepy/tweepy
[9] https://github.com/bear/python-twitter
[10] https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json
[11] https://developer.twitter.com/en/docs/tweets/search/api-reference
[12] https://developer.twitter.com/en/docs/basics/rate-limits
[13] https://developer.twitter.com/en/docs/basics/response-codes
[14] https://github.com/msramalho/twitter-response-codes
[15] https://github.com/bear/python-twitter
[16] https://docs.mongodb.com/manual/reference/limits/#bson-documents
[17] https://jira.mongodb.org/browse/SERVER-23482

that the database-writes became the bottleneck of the process, and not Twitter's API, no tuning or batch writing solved this problem.

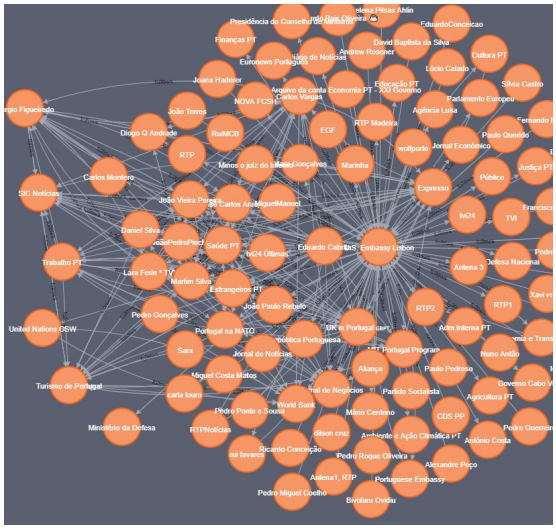

Fig. 1. Example visualization of Twitter "follow relationships" saved in Neo4j

After having spent a significant amount of time on struggles in the collection process, we took a step back and decided to focus on designing a more deliberate system, even if at a greater time cost, which could answer the requirements above, as well as ensure two new ones:

Data Collection Requirements 2. Additional requirements

1) Separate watched from non-watched accounts, the first type consisting of accounts with content posted in Portuguese and ideally within Portugal's Twittersphere;
2) For every account that was marked as being watched, all their tweet history should be collected.

This new system was dubbed **Twitter Watch**. The next section introduces the architecture designed to answer Requirements 1 and 2.

IV. Architecture

Twitter Watch's high-level architecture can be split into User Interface (UI), backend, and external services. Figure 2 contains a visual representation of this architecture, its components, and their interaction.

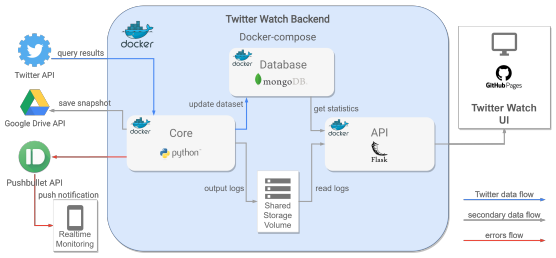

Fig. 2. Twitter Watch Architecture

A. Backend

The main functionality of Twitter Watch is its backend. This backend is organized in Docker containers. Docker Compose is used to orchestrate these containers and their interactions. The architecture requires three containers, as described below:

Database container: This is a standalone MongoDB instance where all the data collected is saved. This container is exposed to the outside of the Docker Compose network mainly so it can be interacted with from the outside even while data is being collected.

Core container: The core container contains all the logic behind the collection process, and that will be further detailed in Section V (p. 4). This container runs a Python-ready environment and includes the vast majority of code developed for the backend.

API container: This container is designed to provide information on the data collection process as requested by the UI. It shares a Docker storage volume with the core container so that it can serve the execution logs generated during the collection process. It can also interact with the database container in order to query statistics on the collected data, such as the number of accounts or tweets saved. Flask [18] is the chosen Python framework for implementing this service.

B. User Interface

The UI is a pragmatic effort to streamline the management of the data collection process through an easier and faster diagnosis of the system's state. The UI itself is agnostic to the which Twitter Watch backend it is connecting to and accepts as input the Internet Protocol (IP) address of any Twitter Watch API instance. The most updated version of the UI in production is available at msramalho.github.io/twitter-watch. The UI is hosted on GitHub Pages [19], it was developed with Nuxt.js [20] and Vuetify [21]. In terms of content, the current version has two main pages: statistics and logs.

The statistics page, as seen in Figure 3, contains plots of how the number of users and tweets recorded in the database evolve over time, as well as a plot of the database size evolution through time. Additionally, it contains some overall statistics like the current number of users and tweets, and can easily be expanded to accommodate more information.

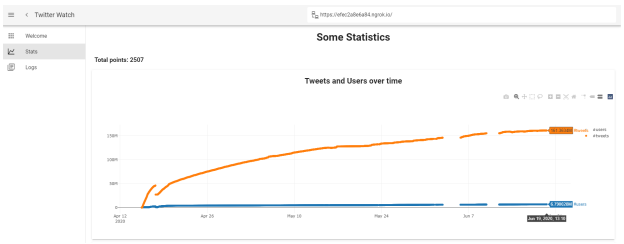

Fig. 3. Twitter Watch UI statistics page

[18] https://flask.palletsprojects.com/en/1.1.x/
[19] https://pages.github.com/
[20] https://nuxtjs.org/
[21] https://vuetifyjs.com/en/

The logs page, as seen in Figure 4, has two main sections. On the left, there is a panel to explore all the different scripts that are run by the backend, as specified in Section V. Currently running scripts are marked with a green dot that, when hovered, displays information on how long they have been executing. Furthermore, expanding any of the scripts reveals a list of the individual logs for the date and time their execution started. When one of these logs is clicked by the user, its output is fetched and displayed on the right panel (*cf.* Figure 4).

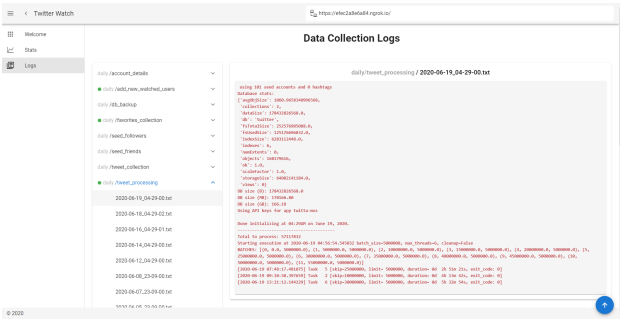

Fig. 4. Twitter Watch UI logs page

C. External Services

Twitter Watch interacts with external services each with a particular goal. The most obvious one is the Twitter API, a quintessential part of Twitter Watch, where all the data comes from. On top of that, Twitter Watch uses Google Drive API [22] for storing regular snapshots of the database in case there is a problem with the deployment. In practice, these snapshots are used for local data analysis as they constitute an easy way to download the compacted data and rebuild the database in any local deployment of MongoDB. Finally, out of necessity, we found that it is useful to have a way to receive notifications about unexpected collection errors. To achieve this, we used Pushbullet [23], a free push notification service that has a mobile client that can receive real-time push notifications. Push notifications were mostly used to detect errors in the most fragile processes like interaction with the Twitter API, building the database compressed snapshot, or uploading it to Google Drive. The logic behind this push notifications mechanism is isolated in the core container code and can easily be invoked anywhere else in the application, where any future user of Twitter Watch needs.

V. Implementation

Twitter Watch's implementation rests on a scheduling system that is capable of launching parallel processes, each with its logic, and the combination of individual tasks interacts indirectly, as these get their input from the database and write their output on it too – a holistic system. Although the system is designed to be both flexible and customizable, the out-of-the-box version is already capable of meeting all the desired requirements. Part of the flexibility comes from a configuration file used to dictate how the dataset should evolve over time, this configuration file is further explained in Section V-A (p. 5). The rest of the flexibility stems from the creation of a semantic folder structure along with the ability to add new features without having to change any core code. This last point is achieved by isolating all the collection process logic into Jupyter notebooks, which are automatically interpreted according to their location in the aforementioned semantic folder structure.

Figure 6 (p. 5) contains a high-level view of the backbone of Twitter Watch. It shows four different execution steps: launch, setup, run-once, and scheduled tasks. Each task is written in its own Jupyter notebook. The key to differentiating them is their location in the folder structure, namely the one visible in Figure 5. Each folder in the 14`collection` folder has a specific execution routine that maps directly into the implementation (*cf.* Figure 6, p. 5), as is described below.

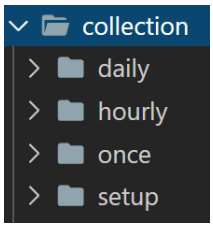

Fig. 5. Jupyter notebooks semantic folder structure

Initially, when the application is launched, there is a single line of execution. First, the configuration file is parsed and validated; then, the output folder structure is created if it does not exist; lastly, the Jupyter notebooks, where the collection logic is written, are converted into Python (*.py*) script files, as these can then be easily executed in separate processes.

Afterward, the now converted Python script files inside the 14`setup` folder are executed. In our implementation, this step has tasks related to inserting an initial set of accounts into the database for subsequent exploration (these seed accounts come from the configuration file), and executing database migrations (these can be anything from creating indexes on the database to restructuring the database and can vary through time). These setup tasks are executed as parallel threads. Once all the setup tasks have completed, both the run-once tasks and the scheduled tasks launcher loop are executed in parallel.

The run-once tasks differ from setup tasks by being able to co-exist with the recurrent tasks launched by the scheduled launcher. These tasks are often tasks that only need to be executed once, or that will be running non-stop until the application is manually closed. One-time data migrations fall into this category. In our case, we had a task running non-stop that simply logs the number of documents in each database collection at a custom interval.

Finally, the scheduled launcher is the piece of the puzzle responsible for parsing the filenames inside the daily and

[22] https://developers.google.com/drive/
[23] https://www.pushbullet.com/

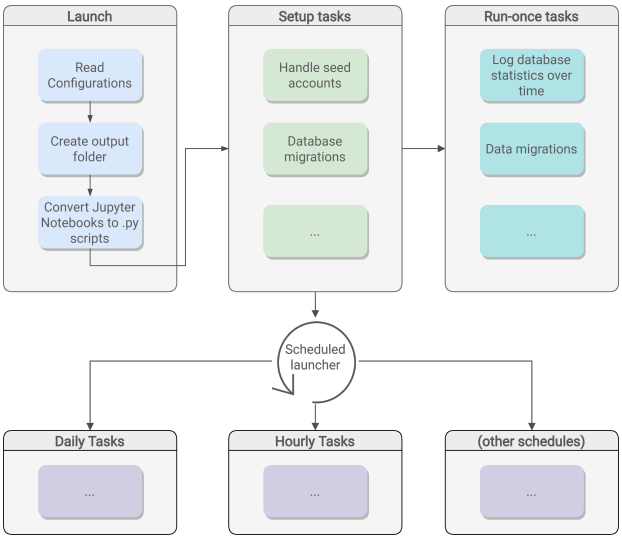

Fig. 6. Twitter Watch Implementation

hourly tasks and running them at the specified time. Note that other schedules can be added in the future, like weekly executions, with little effort due to the abstraction mechanisms implemented. Note also that the filenames of the files within the 14`daily` and 14`hourly` folders are used to infer the exact desired execution time. For instance, the filename 14`daily/03_00_tweet_collection.py` is instructing the launcher that it should be run precisely three hours and zero minutes after the application is launched, on a daily basis. This simple hack proved quite useful in practice, as it facilitates easy re-arrangement of the order in which scripts should be executed, even by playing with overlapping or distancing scripts based on their resource usage.

A. *Configuration File*

The configuration file works as a contract of settings that are unanimously used by all tasks, it is written in JSON and is automatically loaded into a global variable that becomes accessible to each process with a single import statement. Code Listing 1 contains a simplified example of the version used for the duration of this project.

Below is an explanation of the most relevant fields in the configuration file:

- The 14`seed` field contains usernames of the seed accounts that serve as a starting point for the collection process;
- The 14`collection` field specifies what the behavior of the collection process should be like, for instance:
  - 14`limits` is a set of restrictions on the total size of the dataset and how it can grow in size on each day;
  - 14`oldest_tweet` and 14`newest_tweet` restrict the time span during which Tweets are to be collected;
  - 14`search_languages` is a list of language codes [24] used to restrict tweet collection on accounts with more than 14`min_tweets_before_restricting_by_language` tweets, while not having a majority of them in one of the provided languages;
  - 14`max_threads` is a global restriction on the number of threads each process uses when is parallelized through the class described in Section VI (p. 7).
- The 14`mongodb` field and its inner fields indicate the database access credentials, the default database name to use, and whether to perform this database backup to Google Drive (*cf.* Section IV-C, p. 4) or not;
- The 14`notifications` field is used to provide the API credentials for Pushbullet (*cf.* Section IV-C, p. 4);

[24] https://developer.twitter.com/en/docs/developer-utilities/supported-languages/api-reference/get-help-languages

```
14 {                                                                    "
       "usernames": []
   },
   "collection": {
       "limits": {
           "max_watched_users":
            ↪  100000000,
           "max_daily_increase": 25000,
           "max_daily_increase_ratio":
            ↪  0.1,

            ↪  "min_appearances_before_watched":
            ↪  10
       },
       "ignore_tweet_media": false,
       "oldest_tweet": "Sun Aug 1
       00:00:00 +0000 2019",
       "newest_tweet": "Sun Aug 1
       00:00:00 +0000 2030",
       "search_languages": ["pt",
        ↪  "und"],
       "max_threads": 8,
       "min_tweets_before_
       restricting_by_language": 10
   },
   "mongodb": {
       "address": "mongodb://USERNAME:
       PASSWORD@mongo:27017/",
       "database": "twitter",
       "drive_api_backup_enabled": true
   },
   "notifications": {
       "pushbullet_token": "API TOKEN"
   },
   "database_stats_file":
    ↪  "out/db_logs.csv",
   "seconds_between_db_stats_log": 10,
   "api_keys": "TWITTER API KEYS FILE"
}
```

Listing 1: Simplified example of the JSON configuration file

- 14database_stats_file and 14seconds_between_db_stats_log specify the location and period of the database statistics collection process that is always running in the background;
- 14api_keys is simply the filepath of a JSON file where a list of our Twitter API credentials are stored.

This configuration file can quickly grow to meet the developer's end-goals since adding a field in the JSON file will make it immediately available in any of the Jupyter notebooks.

B. Collection Logic

With the above knowledge, we can now delve into explaining how the different collection tasks work. A first remark has to do with making the most out of Twitter API keys, namely by isolating each used endpoint in its task, since the rate limits apply at the endpoint level. Knowing this, it should also be noted that the functionality was designed to be as modular as possible. The next subsections reveal the way the current implementation extracts Twitter data by going into detail on the most relevant implemented Jupyter notebooks and respective tasks.

Seed followers (daily): This task runs daily and iterates over all the seed accounts specified in the configuration file updating the database with any new followers of those accounts by querying the GET followers/ids endpoint. All these accounts are marked as watched accounts.

Seed friends (daily): This task is in all aspects similar to the previous one, differing only by focusing on friends (*a.k.a.* followees) instead of followers, through the use of the GET friends/ids endpoint.

Account details (daily): This task runs daily and iterates over all the accounts without a screen name property. This happens because accounts are added to the database by other tasks whenever a new account is found, but they typically contain a single 14_id property and not the complete profile information. It uses the GET users/lookup endpoint to hydrate the account objects.

Tweet collection (daily): This task runs daily and iterates over all the accounts that are marked as watched, or accounts that have not yet been excluded due to the restriction imposed by the parameter 14search_languages (*cf.* Section V-A, p. 5)). The logic behind filtering out accounts is focused on producing results, *i.e.* accounts and tweets that are relevant for the subsequent study's goals. In this case, the focus is on restricting by proximity to the seed accounts and also to a set of specified languages. So, all the aforementioned accounts are then iterated and all their tweets are collected between the 14oldest_tweet and 14newest_tweet values, using the GET statuses/user_timeline endpoint, up to a limit of 3,200 tweets imposed by Twitter (*cf.* documentation [25]). In fact, this task is optimized as every time an account's tweets are collected, the 14_id of the last collected tweet is saved to the database and use in future calls as the 14since_id param. This is in accordance with the official optimization guidelines to minimize redundant API calls [26]. Finally, this task also updates the 14most_common_language of each account by pre-processing its collected tweets, this property is used to restrict further iterations of the task from processing users whose 14most_common_language is not in 14search_languages, note that 14most_common_language is only set if an account has at least 14min_tweets_before_restricting_by_language tweets.

Favorites collection (daily): This task is similar in behavior to the previous one, with two differences. First, it collects liked tweets instead of posted ones, which is achieved through the GET favorites/list endpoint. Second, it restricts the iterated accounts to watched users only.

Tweet processing (daily): This task runs daily. It does not rely on making API calls. It iterates over all the unprocessed tweets in the database, so that each tweet is only processed once. For each tweet it isolates the ids of all the accounts that are related to that tweet:

- all mentioned accounts;
- the author of the tweet;
- the author of the original tweet, if this is a retweet;
- the author of the original tweet, if this is a quoted tweet;
- the author of the original tweet, if this is a reply tweet.

Each of these account ids is inserted in the database if it does not exist yet and the appearances counter for each account is incremented by one. This counter is used as a minimum threshold to start including the respective account into the watched users set; this behavior is configured by the 14min_appearances_before_watched field in the configuration file. Larger values of that parameter will lead to a slower yet potentially more relevant expansion of the watched users set.

Seed tweet processing (daily): This task is similar to the previous one. However, it is focused only on tweets by the seed accounts defined in the configuration file and the accounts identified are directly marked as watched, due to their proximity to the seed account and expected relevance to the dataset.

Add new watched users (daily): The goal of this task is to iterate all users that are yet to be marked as watched or non-watched and insert the ones that meet the minimum number of appearances count as defined by the configuration field 14min_appearances_before_watched. Furthermore, the configuration field 14limits and its inner values will restrict the maximum number of newly watched users per day and, *in extremis*, will prevent the addition of any more watched users if the 14max_watched_users value has been reached. These configurations are meant to coerce

[25] https://developer.twitter.com/en/docs/tweets/timelines/api-reference/get-statuses-user_timeline

[26] https://developer.twitter.com/en/docs/tweets/timelines/guides/working-with-timelines

the system to evolve more slowly to avoid resource overload, when that is necessary.

Hashtag tweet collection (hourly): This task is executed every hour. First, it collects all the tweets in the database from the past 24h originating from the seed accounts. Then, the hashtags of those tweets are gathered and merged, and are then used to perform a language-restricted search (according to 14`search_languages`), for each unique hashtag. We use the Standard search API, noting that this is a unique endpoint where the results are restricted to a 1% sample of all tweets in the Twittersphere of the past seven days. The end-goal of this task is to introduce some relevant variety to the collection process since these tweets will later be processed and influence the expansion process.

Google Drive backup (daily): This task has been mainly explained in Section IV-C (p. 4) and it takes care of calling the 14`mongodump` command [27] from MongoDB and then uploading the resulting database snapshot to Google Drive.

Note on suspensions: It is important to highlight that on every API call that operates on account ids, namely looking up followers, friends, account information, tweet timeline, among others, Twitter Watch has a mechanism that wraps the errors returned by the API and, if a received error is related to the given account having been suspended (deletions, and private accounts are also recorded, for that matter), the database is updated to register this occurrence and its timestamp. This is useful for enriching the dataset with suspensions information.

Note on long-running tasks: The current version of Twitter Watch ensures that any scheduled task set to start its execution at a given moment is only launched if its previously launched instance (1 hour before for hourly tasks, and 24 hours before for daily tasks) has finished, in order to avoid excessive resource consumption.

VI. Parallelism

After having Twitter Watch collecting data for a while, we noticed that the size of the database was large, and some of the tasks, such as the daily tweet collection or tweet processing, were taking more than the ideal 24 hours threshold. Inspired by the Map-Reduce algorithm [3], we developed our own implementation of a parallel processing mechanism that each task could benefit from. This mechanism requires only isolating the logic code in each notebook to a 14`def task(skip, limit):` method that performs the same query on the MongoDB database but appends 14`.skip(skip).limit(limit)` filters to the query. Once this change is ready, all a developer has to do is invoke the 14`.run()` method on the 14`DynamicParallelism(..., batch_size, max_threads)` class. The 14`batch_size` and 14`max_threads` properties can be used to adapt the behavior of the parallel execution both in terms of the size of each batch of database documents to be processed as well as on the number of threads to use, respectively. The number of threads will default to the configuration field 14`max_threads`. A 14`.reduce()` method exists to merge all the outputs, but in practice, it has only been used for data analysis tasks and not actually in the collection process. In the end, this effort managed to significantly reduce the long-running tasks' length to under the desired 24h limit.

[27] https://docs.mongodb.com/manual/reference/program/mongodump/

VII. Results

First of all, the framework resulting from this effort was able to meet both the mandatory as well as the optional requirements specified in Requirements 1 and 2.

Second, the development of Twitter Watch was incremental, since such a complex and byzantine system was impossible to effectively tune in advance while assuring few wasted resources. This led us to give priority to things as they came up. In any case, we believe it served its purpose for our use case, and has led us to be hopeful about its potential to be adapted or used as-is for other research or industry efforts. In terms of results, we can look at its overall success as a consequence of both data quantity and quality it managed to collect. With that in mind, this section reports only the quantity results for the system where it is deployed, and the configurations used.

Deployment server

The current deployment server has an x86_64 architecture running Ubuntu 18.04.3 LTS; having 6 Central Processing Units (CPUs), with 8GB of Random Access Memory (RAM), 236GB of Solid State Drive (SSD) disk space, and running Docker version 19.03.11. The initial setup included only 2 CPUs and 4GB of RAM; this was considered a bit limited, especially when interacting with the machine through Secure Shell (SSH) while the collection process was under execution. The final setup proved to be sufficient, but no doubt that increasing processing power and available RAM could significantly speed up some operations such as database read/writes that use indexes, which can require a lot of memory and also benefit from available processing power. In fact, special care was taken to spread out the tasks that could compete heavily for the same resources, but this was done through trial and error.

Configurations

Although our initial approach described in Section III (p. 2) used 21 accounts as seed for the data collection process, we decided to enlarge it by taking into account all the accounts that fall under one of the categories in the following list that expands the original one:

- Political party account;
- Government ministry account;
- President of a political party;
- Minister in office;
- Secretary of state in office;
- Parliament deputy.

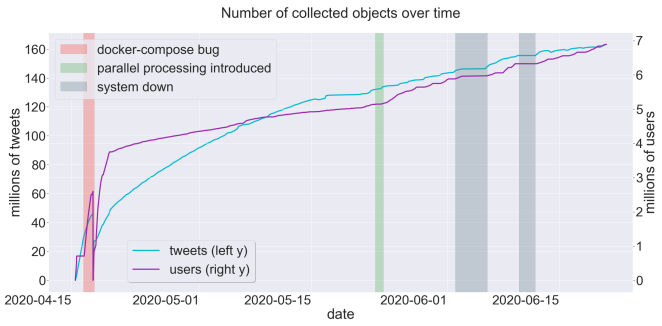

Fig. 7. Twitter Watch total accounts and tweets collected over two months and five days (double y-scaled)

The final number of accounts found is 101. These accounts match with the 14`seed.usernames` parameter in the configuration file. This seed identification process was conducted in February 2020. Other configurations match the values presented in Code Listing 1 (p. 5) excluding, of course, the placeholder values which do not influence the logic of the collection process, like the database credentials or the API tokens.

Output

At the time of writing this report, the system is still up, and will remain so for some time. For a period of little over two months, from April 15th 2020 to June 20th 2020, we have collected almost seven million accounts (over 6,890,000) and more than one hundred and sixty million tweets (over 163,280,000), corresponding to more than 170GB of data. Luckily, compressing this data with 14`mongodump` leads to a file with around 30GB. Figure 7 shows the evolution of the number of collected accounts and tweets over this time period. Four colored areas are highlighted, from left to right:

- The first area, "docker-compose bug", corresponds to a period where the system was operational but a line in the Docker Compose configuration had been commented with the consequence of not exporting the collected data outside the Docker container, once this was fixed, the system automatically recognized the database in the real file system and started the process again, where it had left off;
- The second, "parallel processing introduced", marks the approximate time of deployment of the parallel processing mechanism described in Section VI (p. 7). It is visible that before this moment, the data collection had almost flattened and quickly regained a steady increase;
- The last two areas, "system down", correspond to two different periods during which the system was offline – this was actually due to human error.

This figure also reflects a predicted phenomenon, when the system was designed: a fast initial growth, followed by a flattening of the curve.

The initial growth is mostly associated with a quick identification of the accounts that interact very intensely with the seed accounts, with the remark that their daily growth is limited by the inner fields in the 14`limits` configuration field. A similar effect happens with the number of tweets, which grows very rapidly. This growth is easily explained by the fact that the first time an account's tweets are collected, they are collected for the entire period from 14`oldest_tweet` until the moment of collection. In contrast, the following iterations for those accounts will only yield the most recent tweets, since the day of the previous collection.

The flattening of the curve, although related, at the same time, to the bottleneck that was fixed by introducing parallel processing, was not surprising. Since the collection process focuses on quickly including all the accounts that are close to the seed accounts (and restricted by the 14`search_languages` field too), and then it will only increase when new accounts are detected in the tweet processing task. These tweets come from accounts that interact with the accounts already in the database but also from the hashtag tweet collection task. Hypothetically, we could have defined a threshold for the total amount of collected accounts, and the system would keep on collecting tweets for that fixed number of accounts – this could be a strategy to fight an explosion of the number of accounts.

Experienced limitations

A final note on Figure 7 to explain some slight decreases in both curves. For our use case, we defined a few run-once tasks that would remove some accounts and their tweets from the database, and that explains some slight decreases in the curve. This is due to the first limitation of Twitter Watch, the fact that it relies highly on the language of the tweets to find relevant content. This is undesirable because, for instance, the English language is widely used, and it is hard to capture realities in English speaking countries without incurring the risk of having a lot of noise in the dataset since it will eventually expand to capture tweets in the same language but from other countries. For our disgruntlement, Portuguese is the official language of Portugal and several other countries, most relevant in practice, of Brazil. Actually, the official language is Brazilian Portuguese, but Twitter marks both types as 14`"pt"`. Having seen a non-negligible number of Brazilian Portuguese tweets in our data, we devised a task focused on identifying accounts with a location in Brazil and removing them. This was partially effective, but not entirely as many accounts do not specify their location and, therefore, remain in the dataset. Other Portuguese speaking countries hardly ever came up throughout the collection and following exploration processes.

VIII. Summary

The above sections have laid out the current standpoint of Twitter Watch, and how it is already in a version that allows for massive and structured data collection that satisfies the desired requirements. However, we envision several improvements in terms of usability and the collection process.

In terms of usability, we have thought of adding control functionality to Twitter Watch's interface like the ability to launch or kill a given task, or even edit the configurations file on the fly.

In terms of the collection process, we can highlight that the system's scalability is limited to a vertical growth of resources, and we believe that larger collection processes might benefit from a horizontal approach, for instance, by using MongoDB's sharding mechanisms [28] to benefit from a multi-cluster system.

In terms of already verified limitations, the language limitation could be mitigated by adding a filter for the watched accounts based on the location of the account. In fact, if the location is not explicitly provided, one can even rely on existing research to infer an account's location from its posted content [4].

Finally, we believe that Twitter Watch's approach can be adapted to other data collection processes and APIs.

Overall, we have taken a top-down approach to explain the inner workings of Twitter Watch. However, we believe that anyone using it from scratch will still need to spend a couple of days getting familiar with its overall structure if they intend to develop custom tasks. Even so, we have seen it working and are satisfied with the obtained results that enabled our analyses that would otherwise be limited to, and based on, an incoherent and incomplete dataset.

REFERENCES

[1] S. A. Chun, R. Holowczak, K. N. Dharan, R. Wang, S. Basu, and J. Geller, "Detecting political bias trolls in Twitter data," *WEBIST 2019 - Proceedings of the 15th International Conference on Web Information Systems and Technologies*, pp. 334–342, 2019.

[2] H. Fani, E. Jiang, E. Bagheri, F. Al-Obeidat, W. Du, and M. Kargar, "User community detection via embedding of social network structure and temporal content," *Information Processing and Management*, vol. 57, no. 2, p. 102056, 2019.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] J. Mahmud, J. Nichols, and C. Drews, "Where is this tweet from? Inferring home locations of Twitter users," *ICWSM 2012 - Proceedings of the 6th International AAAI Conference on Weblogs and Social Media*, pp. 511–514, 2012.

[28] https://docs.mongodb.com/manual/sharding/