A Faceted Crawler for the Twitter Service

George Valkanas, Antonia Saravanou, and Dimitrios Gunopulos

Dept. of Informatics and Telecommunications
University of Athens, Athens, Greece
{gvalk,antoniasar,dg}@di.uoa.gr

Abstract. Researchers, nowadays, have at their disposal valuable data from social networking applications, of which Twitter and Facebook are the most prominent examples. To retrieve this content, the Twitter service provides 2 distinct *Application Programming Interfaces* (APIs): a probe-based and a streaming one, each of which imposes different limitations on the data collection process. In this paper, we present a general architecture to facilitate *faceted crawling* of the service, which simplifies retrieval. We give implementation details of our system, while providing a simple way to express the crawling process, i.e., the *crawl flow*. We experimentally evaluate it on a variety of faceted crawls, depicting its efficacy for the online medium.

Keywords: Faceted Crawling, Twitter, Architecture, Design, Applications

1 Introduction

The numerous prototypes built on top of social networks are proof of their importance [19, 16, 23, 22, 18, 15, 14, 6, 7]. These applications focus on different key properties of the underlying data. **Big data** problems require *voluminous* datasets. **Emergency management** requires access to *locational* information [5, 21, 12]. **News reporting** must access high-quality information from *high-quality* posters. **Social graphs** are another major asset.

In this paper we focus on Twitter because it is prevalent, and has an open-data policy. Building a system to retrieve the desired information is both time consuming and technically challenging. For example, the service provides two distinct *Application Programming Interfaces* (APIs), with certain limitations: Access is restricted to authenticated (i.e., registered) users, and to public tweets, i.e., tweets visible to anyone. The default *streaming* API returns only 1% of public tweets [1], while the *REST* API limits the number of requests issued within a specific timeframe, imitating the politeness principle [11].

In this paper, we consider the problem of an efficient crawler for the Twitter service, to fetch content with desired properties. Those properties refer to different *facets* of the data (tweets, users, graph, location, etc.), giving rise to the *faceted crawling problem*. We discuss our system's design and implementation details. In summary, we make the following contributions:

[1] Elevated access can be granted for a fee.

– We present the architecture and implementation of a *faceted crawler* for the Twitter service.
– We simplify the crawling process through a *crawl flow*, which can multiplex queries in an elegant, yet effective way. We have implemented default crawl flows, which can be used in numerous scenarios.
– We present results on our crawler's performance and discuss lessons learned from our interaction with the service's APIs.

2 Related Work

Harvesting web documents is as old a task as the web itself. Search engines rely on web crawlers [9, 10, 17] to fetch online documents, which they subsequently index and make available. These are built for the *surface* web, where webpages are reachable through hyperlinks. However, Twitter's multiple information *facets* do not allow a straight-forward modification of existing crawlers, which would also violate the *politeness policy*, given the *real-time* nature of the medium. Similar problems exist for crawlers of the *Hidden Web* [8, 13], where information can be accessed through query forms.

Several libraries exist [3], to access Twitter's APIs programmatically, one resource at a time. Therefore, these are not complete solutions, whereas we facilitate the *faceted* crawling process through the *crawl flow*.

The work in [4] also discusses *facets* on Twitter, but in a conceptually different way. More importantly, our research goals are different: [4] is interested in enriching tweets with "context", whereas we aim at the implementation of an efficient and robust crawler. Therefore, [4] can be thought of as an application, that one can build on top of our proposed infrastructure.

3 Twitter API Background

Twitter provides two main Application Programming Interfaces (APIs) to access publicly available data, i.e., data that anyone can see. The first API is a RESTful one, and requires probing the service with HTTP requests. The second API is a streaming one, and resembles a publish-subscribe mechanism. In both cases, the user can apply filters, to restrict the information they are looking for. In both cases the user needs to be authenticated through one of the available options. In the next paragraphs, we give a more detailed overview of these two APIs.

3.1 REST API

The REST API uses HTTP requests (i.e., `GET`, `POST`) to perform the communication between the end user and the Twitter service. This API supports multiple query types, each of which can be employed by contacting a carefully constructed URL, with all the necessary information.

From the REST API specification [1], we identify four types of restrictions, which we must take into account in our crawler. Table 1 gives additional details.

– **Rate restrictions**: The number of queries of a specific type that the developer can issue within the 15 minute window.
– **Maximum Result Size**: The upper bound on the results of a particular query. For instance, even if a user has posted 5000 tweets, we are only able to access the most recent 3200.
– **Probing Result Size**: The number of results that we can retrieve each time we probe the service with that particular query. For instance, a query for a user's timeline will return *at most* 200 tweets.
– **Maximum Query Size**: The number of objects that we can query simultaneously with a single probe to the service. Typically this is 1, (e.g., 1 tweet each time, using its id), but there are some exceptions (e.g., *lookup* at most 100 users).

3.2 Streaming API

Through this API, one can receive data as a *flow* of tweets. The API returns a 1% sample of all public posts, though not uniformly [**?**]. Consequently, data received through this API may reflect fluctuations of the actual stream, e.g., increase / decrease of posts, temporal patterns of user interactions, etc. A drawback of this API is that it can not be used for all information facets, e.g., the social graph.

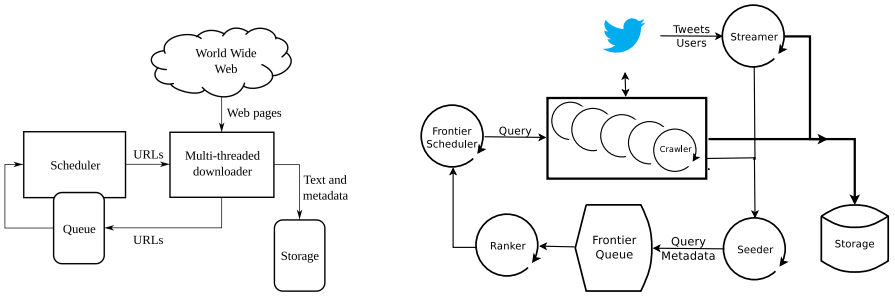
4 Faceted Crawler Architecture

Figure 1 shows the architecture of a classic web crawler [10] on the left, compared against our Faceted crawler architecture, on the right. The two designs appear to be similar for the most part. The contents of the frontier queue, however, in the two cases are different, because surface crawlers need only handle URLs of the next pages to fetch. On the contrary, our crawler needs to handle different query types, each of which takes different parameters.

The components Seeder, Ranker and Streamer are also different. The latter exists to harvest data using the *streaming* API. The Seeder exists to support various applications in a unified way. The Ranker is separate from the scheduler, because one application may combine multiple query types, and to simplify application development.

Table 1. Restrictions for some major query types.

Query	Rate	Max Result	Probe Result	API limit
User Lookup	180	∞	100	100
Tweet Show	180	1	1	1
Friends	15	∞	5000	1
Followers	15	∞	5000	1
Timeline	180	3200	200	1
Retweets	15	100	100	1



(a) Classic Web Crawler (b) Twitter Faceted Crawler

Fig. 1. Architectural designs of both classic web crawler and our Twitter faceted crawler

4.1 Streaming API

To obtain information from the `Streaming` API, we employ the STREAMER component, as shown in Figure 1. The component receives the stream, and may forward it for processing, storage and seeding.

4.2 REST-based crawling

Aside the STREAMER, the components in Figure 1 largely resemble a classic crawler architecture. However, the actual design is quite different. We have already pointed out the difference in the frontier queue. Moreover, each application may have a different crawling process, therefore we need to efficiently multiplex queries. For this reason, we have decoupled crawling (i.e., accessing the service) from seeding.

4.3 Scheduler

A major component of our system is the SCHEDULER, responsible for queueing crawl tasks. A crawl task contains information about the query type and all of the parameters that accompany it. The SCHEDULER is also responsible for enforcing the rate limits. To achieve this, the component operates in an event-driven manner, shown in Algorithm 1.

The component starts with the query types of the crawling process. It will enqueue these queries to a *timedQueue*, and will trigger the event, leading to the execution of the "EventTrigger()" method. The queries that triggered the alarm are dequeued and passed to the *queue* for crawling. We then reset the timer to trigger for the next query item.

Items in *queue* are processed one by one, by the main scheduler thread. For the current query type, we probe th "Frontier Queue" (Line 3), implemented as a database relation, and pass the result for crawling. The scheduler stores metadata in the database (e.g., statistics) and requeues the query for timely execution, computed through its rate limit.

Algorithm 1 `Scheduler Algorithm`

```
    Input: Database db, Ranker ranker
    Output: outQueue
    Shared Queue queue, timedQueue

    //Main Thread
 1: while !stopped do
 2:     qry ← queue.dequeue();
 3:     data ← ranker.getNext( qry );
 4:     outQueue.enqueue( qry, data );
 5:     db.store( qry.qryMeta );
 6:     timedQueue.enqueue( qry, NOW + qry.ival );

    EventTrigger()
 7: nextQuery ← timedQueue.dequeue();
 8: top ← timedQueue.top();
 9: queue.enqueue( nextQuery );
10: resetTimer( top.TIME - NOW );
```

Algorithm 2 `Seeder Algorithm`

```
    Input: Database db, ResultQueue RQ, CrawlFlow CF

 1: while !stopped do
 2:     (result, qry) ← RQ.dequeue();
 3:     storeResult( result );
 4:     update = CF.stepSeeding( qry, result );
 5:     if (update) then
 6:         nxtQrs ← CF.nextQueries( qryMeta );
 7:         for ( i = 0; i ¡ nxtQrs.size; i++ ) do
 8:             nxtQrs( i ).stepSeeding( qry, result );
 9:     db.store( qry.qryMeta );
```

4.4 Ranker

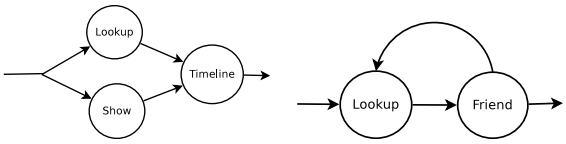
As seen in Algorithm 1, the scheduler relies on a `Ranker` object. A `Ranker` implements our `IRanker` interface, shown in Figure 2. The `init()` method is used to properly initialize resources (e.g., database relations). The *getNext()* method returns the next item to submit to a crawler as our next query. The *id* is decided by the scheduler to simplify the architecture. The *query* also contains the `RateLimit` information. This allows for a common interface across queries.

4.5 Seeder

As shown in Algorithm 2, the SEEDER operates in an endless loop, much like the SCHEDULER. It receives information from the result queue $\mathcal{RQ}$ (Line 2), where crawlers write the result of probing Twitter. Results are forwarded for storage (line 3). We then update the frontier in two steps. First, update the current query (line 4). The result is a binary variable (TRUE/FALSE), that

```
public interface IRanker{
  public void init();
  public List getNext( long qid, Query query );
}
```

Fig. 2. The IRanker interface



(a) Timeline (b) Sampling

Fig. 3. Schematic representation of Crawl Flow examples.

determines whether we should move to the second step, i.e., update subsequent frontiers (lines 5-8).

4.6 The QueryLog Relation

To restart after a (forceful) shutdown, and monitor our system's performance, we store appropriate information, in a relational table, called QuerLog. A partial view is shown in Table 2. Statistics of the system include rank time, seed time, etc (not shown here). Fields **qid** and **result** are straightforward. Values of the **result** field can be found in [2]. The next three fields ensure that the rate limits are enforced in cases of failure or restarts.

4.7 Crawl Flow

To further simplify the crawling process, we introduce the concept of a Crawl Flow. The idea is that on Twitter, a crawl is driven by the underlying application, which can be generally expressed as a sequence of faceted probes (with cycles). The Crawl Flow can be thought of as a state automaton, and defines the sequence of the queries to the service. Figure 3 shows a schematic representation of two Crawl Flows that we provide, a user's timeline, and sampling the social graph. Through the Crawl Flow, the user specifies:

- The general execution sequence of queries. The sequence may contain loops (including self-loops), depending on the goal.
- An object implementing the IRank interface.
- An object implementing the ISeed interface.

5 Use Cases

We have fully implemented our system in Java 1.6 and used the Twitter4j library [2] for method probes. We used PostgreSQL 8.4 with its default configura-

[2] http://twitter4j.org

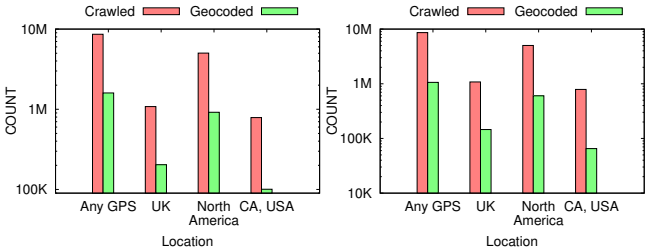
Table 2. Fields of the QueryLog relation

Field	Description
qid	Unique Identifier for the Query
result	Code Signifying how the query
toq	The type of query associated with this tuple
crawler	An identifier for a crawler
tssched	Timestamp when this query was scheduled

tion, but any SQL-compliant database will suffice. Each component runs on a separate thread. Nevertheless, our experiments were run on a single quad core machine @3.4GHz, with 16Gb of RAM, though half of it was set as Java's heap space, and Ubuntu Linux 64bit.

5.1 Crawling by Location

Location is a very important aspect of tweets. Tweets with location can be retrieved through a geographical filter, specified as 2D bounding box with GPS. Despite its accuracy, GPS is not the sole approach to geocode data. External geocoders [21] can be applied directly to the streaming API. Figure 4 shows the number of tweets (on the left) and users (on the right). Custom geocoding (GEOCODED) can extract an additional 10% to GPS-filtered information (CRAWLED).



(a) Tweet Counts (b) User Counts

Fig. 4. Comparing raw counts between crawled and geocoded locational information

Changing the bounding box of a crawl returns different results, even when the boxes overlap. Table 3 depicts the similarities in terms of received users (upper right, in red) and tweets (lower left, in blue), where each crawler is configured to monitor the corresponding location. Despite some commonly shared users and tweets, a lot of new content is being delivered by each stream.

5.2 Crawling User Timelines

User timelines are useful in several cases, e.g., behavior analysis. To efficiently crawl a user's timeline, it is best to know the number of expected results.

Crawling Basic User Information: We improve user information extraction by 1%, as seen in Figure 5(a), through the combination of two query types.

Table 3. Jaccard Similarity between the GPS enabled crawls using the Streaming API

	ANY GPS	UK	N AMER	CA, USA
ANY GPS	1.0	0.069	0.249	0.038
UK	0.057	1.0	6×10^{-4}	0.001
N AMER	0.218	0.0	1.0	0.138
CA, USA	0.042	0.0	0.145	1.0

Figure 5(b) depicts this improvement (blue line) as a percentage. The 1 less query out of every 4, shown in Figure 5(a), is due to our best-effort approach for crawling the service, which has tight time constraints.

Crawling the Timeline: Figure 6(a) shows how much time is spent on the *getNext* method by the RANKER component, which is below 10ms. Both the *getNext* and *stepSeeding* methods do not take, on average, more than 4ms (Figure 6(b)).

The average elapsed time between consecutive schedulings of Timeline queries is shown in Figure 6(c). The interval does not increase, but stabilizes over time. Even with multiple crawlers, our system maintains its performance. With 10 crawlers, we are at more than 98.5% of the optimal case for Timeline queries (Figure 7(a)) perform similarly for Lookup queries (Figure 7(b)). Evidently, more crawlers yield more results faster (Figure 7(c)).

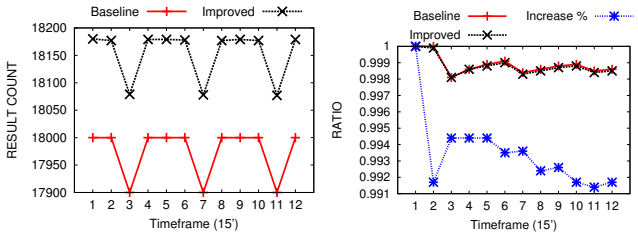
5.3 Sampling

For large networks, sampling is important. We have implemented the Metropolis-Hastings algorithm [20], through the IRANK and ISEED interfaces. Instead of *thinning*, we use reservoir sampling on the collected nodes, which has the same effect. Figure 8a) shows the time required for ranking and seeding. This is the only case where the timings are high, compared to other use cases, and eventually reach an average of ∼2.5 seconds. The reason is that sampling, synchronizes on shared resources. Regardless, our implementation is well within the timeframe between subsequent queries of this type (60 seconds).

5.4 Additional Use Cases

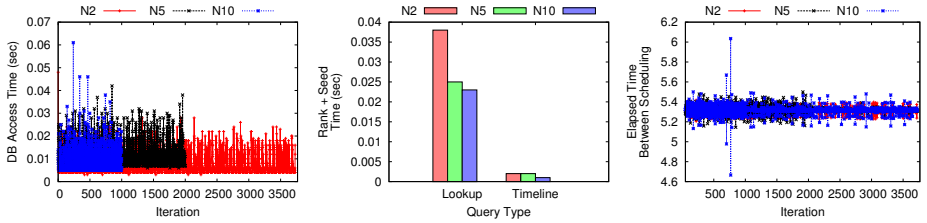
Retweet Graph Retweets are a key concept in Twitter, allowing users to re-post / endorse tweets of others. They can be used to identify cascades of information, leaving the actual reason as a latent feature to be explored. Crawling retweets has been implemented through appropriate seeders / rankers in our framework.

Retrieve tweets by ID Per Twitter policy, one may only disclose the tweet IDs of their dataset. Therefore, this use case becomes very important for reproducibility of results and fair comparison of techniques.



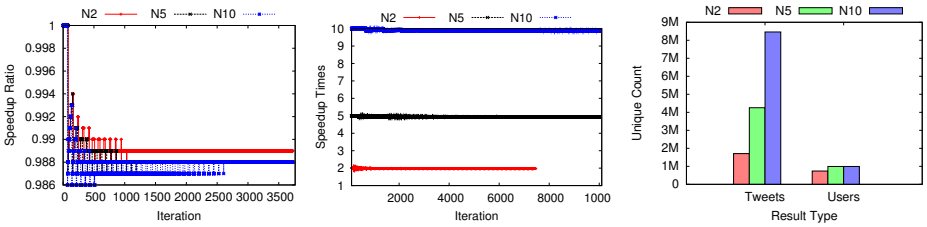
(a) Results Received (b) Improvement %

Fig. 5. Crawling comparison for basic user information



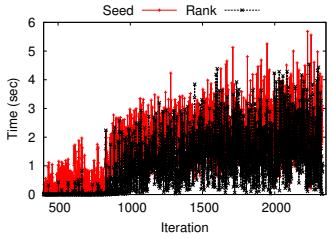
(a) Avg. getNext() for Lookup (b) Avg. Rank+Seed time (c) Avg. Elapsed Time for Scheduling Timeline

Fig. 6. Average crawler performance for harvesting (a) user information, (b)-(c)and user timelines.



(a) Avg. Timeline Query Optimal Ratio (b) Lookup Speedup (c) Volume of crawled Information

Fig. 7. Collective performance of the crawlers used for harvesting user timelines.



(a) Runtimes

Fig. 8. Sampling scenario statistics.

Crawl Social Graph We have also implemented a BFS traversal of the social graph of Twitter, through Seeders and Rankers.

6 Conclusion

In this paper we presented a framework for faceted crawling of the Twitter service. Our framework respects rate limits imposed by the service, and interleaves queries to boost performance. We simplify the crawling process through the *Crawl Flow* concept.

Acknowledgements: This work has been co-financed by a Heraclitus II fellowship and EU project INSIGHT.

References

1. https://dev.twitter.com/docs/api/1.1.
2. https://dev.twitter.com/docs/error-codes-responses.
3. https://dev.twitter.com/docs/twitter-libraries.
4. F. Abel, I. Celik, G.-J. Houben, and P. Siehndel. Leveraging the semantics of tweets for adaptive faceted search on twitter. In *ISCW*, pages 1–17, 2011.
5. A. Ahmed, L. Hong, and A. J. Smola. Hierarchical geographical modeling of user locations from social media posts. In *WWW*, 2013.
6. E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts. Everyone's an influencer: quantifying influence on twitter. In *WSDM*, pages 65–74, 2011.
7. N. Barbieri, F. Bonchi, , and G. Manco. Influence-based network-oblivious community detection. In *ICDM*, 2013.
8. M. Bergman. The deep web: Surfacing hidden value. Technical report, ,, 2001.
9. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.
10. C. Castillo. Effective web crawling. *SIGIR Forum*, 39(1):55–56, June 2005.
11. J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Trans. Database Syst.*, 28(4):390–426, dec 2003.
12. J. Eisenstein, B. O'Connor, N. A. Smith, and E. P. Xing. A latent variable model for geographic lexical variation. In *EMNLP*, 2010.
13. H. Garcia-Molina. Challenges in crawling the web. In *BNCOD*, page 3, 2003.
14. S. Ghosh, G. Korlam, and N. Ganguly. Spammers' networks within online social networks: a case-study on twitter. In *WWW*, pages 41–42, 2011.
15. M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *INFOCOM*, pages 2498–2506, 2010.
16. C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The underground on 140 characters or less. In *CCS*, pages 27–37, 2010.
17. A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, Apr. 1999.
18. K. Y. Kamath and J. Caverlee. Content-based crowd retrieval on the real-time web. In *CIKM*, pages 195–204, 2012.
19. H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.
20. D. Stutzbach, R. Rejaie, N. G. Duffield, S. Sen, and W. Willinger. On unbiased sampling for unstructured peer-to-peer networks. In *Internet Measurement Conference*, pages 27–40, 2006.
21. G. Valkanas and D. Gunopulos. Location extraction from social networks with commodity software and online data. In *ICDM Workshops (SSTDM)*, 2012.
22. G. Valkanas and D. Gunopulos. How the live web feels about events. In *CIKM*, pages 639–648, 2013.
23. J. Weng and B.-S. Lee. Event detection in twitter. In *ICWSM*, 2011.