SocWeb: Efficient Monitoring of Social Network Activities*

Fotis Psallidas[1], **Alexandros Ntoulas**[2,3] and **Alex Delis**[3]

[1] Columbia University, New York, NY 10027
[2] Zynga, San Francisco, CA 94103
[3] Univ. of Athens, Athens, 15784, Greece
fotis@cs.columbia.edu, {antoulas,ad}@di.uoa.gr

Abstract. Although the extraction of facts and aggregated information from individual *Online Social Networks* (*OSN*s) has been extensively studied in the last few years, cross–social media–content examination has received limited attention. Such content examination involving multiple *OSN*s gains significance as a way to either help us verify unconfirmed-thus-far evidence or expand our understanding about occurring events. Driven by the emerging requirement that future applications shall engage multiple sources, we present the architecture of a distributed crawler which harnesses information from multiple *OSN*s. We demonstrate that contemporary *OSN*s feature similar, if not identical, baseline structures. To this end, we propose an extensible model termed *SocWeb* that articulates the essential structural elements of *OSN*s in wide use today. To accurately capture features required for cross-social media analyses, *SocWeb* exploits intra-connections and forms an "*amalgamated*" *OSN*. We introduce a flexible *API* that enables applications to effectively communicate with designated *OSN* providers and discuss key design choices for our distributed crawler. Our approach helps attain diverse qualitative and quantitative performance criteria including freshness of facts, scalability, quality of fetched data and robustness. We report on a cross-social media analysis compiled using our extensible *SocWeb*-based crawler in the presence of *Facebook* and *Youtube*.

1 Introduction

The unprecedented growth rate of *Online Social Networks* (*OSN*s) both in terms of size and quality poses multiple research challenges. As individuals flock, the respective *OSN* volume is constantly increasing. Regarding quality, users often discuss about aspects of their daily life, thus making *OSN*s a source of information that is valuable in many different areas of interest. Among those, detection of events [4, 24, 11], identification of trends [5, 1, 11], announcement of news, detection of communities [2, 17], sentiment analysis, and location tracking [23] have been in the epicenter of attention. All of the above point into an ever–increasing need to better understand both the exhibited behavior and its development by either individuals or groups of users. In doing so, numerous forms of social awareness are being developed [19].

The typical process to unveil and further analyze underlying patterns is that given a single stream of social data, a *Complex Event Processing (CEP)* mechanism [12, 10, 26] is deployed to identify trends and formations of interest in an on-line fashion. Although, a number of studies have been conducted mining data streams emanating from a single social source, there is great interest in attaining cross-social media analyses involving multiple streams from different *OSN*s. Meaningful such aggregation of information can certainly lead to improved fact verification and enhance trend establishment [4]. Consequently, data originated from multiple *OSN*s should not be considered disjointly but rather should be co-developed and co-referenced. In turn, *CEP*-engines should blend social streams from multiple *OSN*s to benefit from their inter-connections. For instance, let an individual A be a user in 2 of the most widely-used *OSN*s: *Facebook* and *Twitter*. A well-known challenge where social media content can assist a great deal is the tracking of the movement and the identification of the current location of A. In [23], it is argued that the current location of A can be inferred using information about her friends. If we limit ourselves by extracting information from a a specific social network say *Twitter*, we are unable to correlate data potentially available from both accounts regarding

* This work was supported by PIRG06-GA-2009-256603.

the location of A. In this respect, we lose vital information for the location tracking task. Furthermore, information about an event detected in an online stream can be extended or cross-validated using other *OSN*s through query formulation strategies [4]. In this context, there is a pressing need to re-consider and benefit from intra-*OSN* relations and produce novel types of analysis and applications in numerous fields including news, events, polls, ads, marketing, games, information tracking, and intra-social awareness.

Obtaining meaningful data simultaneously from multiple *OSN*s is however not a trivial path to follow. Conventional crawling approaches for the "open" Web proposed in the last decade fall short in fetching effectively from multiple *OSN*s; such methods include (1) BFS crawling [18, 16], (2) contiguous crawling, (3) focused crawling [22], and (4) random walking [3, 14]. Furthermore, content found on database systems rather than on web servers, also known as "deep/hidden", is often crawled (or searched) by filling forms using appropriate keywords [20, 15]. Given the steadily increasing volume of data and the inherent physical-network limitations, the distribution and/or the parallelization of crawling have been proposed as an effective means to realize crawling [25, 9].

*OSN*s raise different crawling challenges that cannot be captured by state-of-the-art web crawling techniques. By nature, traditional Web data have two salient features that facilitate access: they are available freely via web-servers or supporting databases and they can be fetched in a straightforward manner. Social network providers allow only "subscribed" applications to fetch their data using exclusively provided *API*s. However, the fetched data is considered private with high sensitivity and heterogeneity. The popular *Facebook* and *Twitter* impose strict limitations and regulations on use of their data.[1] Conventional web crawling techniques that strive to obtain data from *OSN*s are very likely to deviate from the legal limitations and provider regulations imposed. Moreover, using the *API*s provided by individual *OSN*s comes at a high productivity cost. Such *API* calls can be parameterized in a multitude of ways requiring so user sophistication. Matters are also inherently more complex when multiple social networks using diverse structures and building elements are involved in the crawling as there is always need to properly disambiguate the returned results. *SN* providers also frequently impose constraints in terms of response time. Provider resources available for responding to application queries remain limited making those applications prone to network and processing bottlenecks. Indeed, several approaches have applied traditional crawling techniques to fetch data from specific *OSN*s encountering some of the above problems [6, 13, 7]. For instance, [6] uses BFS and uniform sampling crawling to gather data for *Facebook*'s analysis purposes; the study reports resource limitations, privacy restrictions and *API* misbehavior by both the provider and its users.

To alleviate the aforementioned problems and driven by the motivation to describe and seamlessly access multiple *OSN*s in an *amalgamated* form, we propose an extensible model *SocWeb*. *SocWeb* maps the structures of the underline *OSN*s, helps capture their relationships and ultimately offers the basis to develop a versatile distributed crawler/fetcher. *SocWeb* leverages two fundamental concepts that we introduce: the model and its requisite *g*eneric social network *API* or *SNAPI*. *SocWeb*'s programmatic interfaces intend on addressing issues encountered by standard web crawling techniques. The design choices of *SNAPI* adhere to the principle of appropriately abstracting the procedure of connecting one or more applications to desired social network providers by minimizing the effort required and complying with the norms and regulations imposed by the providers.

The contributions of our work are:

- We present and describe an "amalgamated" *OSN*, based on the observation that the underlying structures of *OSN*s remains similar, if not identical.
- We present the *SNAPI* that enables simultaneous connection to one or more generic applications with specific social network providers.
- We outline the design choices of our distributed crawler for automated content extraction from multiple *OSN*s.
- We evaluate qualitatively and quantitatively our system and provide characteristics of the data retrieved as a proof of concept that our system can efficiently monitor data from different *OSN*s.

[1] See for example the discussion on rate limits here: `https://dev.twitter.com/docs/faq`.

2 Representing *OSN*s in SocWeb

At a high level, an *OSN* s is typically represented by a directed graph $G_s(V_s, E_s)$ where vertices V_s correspond to objects (e.g. users, photos, comments) in the social network and edges E_s (which can be potentially labeled) correspond to relationships between those objects (e.g. user A posted photo p_1). Since our goal is to efficiently monitor a variety of *OSN*s, we need to extend this definition to include a set of *OSN*s.

To this end, we employ the following definitions for the building blocks (vertices, edges) of an *OSN*.

Object Types (OT). In our *SocWeb Model* we define two different basic types of objects (vertices), *p*rimitive and *c*omposite. More specifically a vertex $v \in V_s$ of *OSN* s is:

- $Primitive$, *iff* $d_{out}(v) = 0$ and $d_{in}(v) \geq 1$, or
- $Composite$, *iff* $d_{out}(v) \geq 1$

where $d_{in}(v)$, $d_{out}(v)$ are the in- and out-degree of v respectively. Intuitively, the *primitive* vertices define the boundary of a given *OSN*, while composite vertices may link to either primitive or composite vertices thus playing the role of both forming and populating a social network.[2]

In some cases, a given *OSN* may specialize the types of its objects. For example, Facebook has object types such as user, post, album and photo. Note that these types correspond to composite vertices. Every social network has to also define types for primitive nodes. Types that correspond to primitive nodes are, for example, integers, strings and timestamps. Furthermore, each vertex has a unique object type. We denote the object type of a vertex v as $V_{OT}(v)$.

Link Types (LT). Following the OT definition, we also define two kinds of basic link types between two vertices v_1, v_2:

- *P*-link, *iff* $(v_1, v_2) \in E_s$ and $Composite(v_1)$ and $Primitive(v_2)$
- *C*-link, *iff* $(v_1, v_2) \in E_s$ and $Composite(v_1)$ and $Composite(v_2)$
  Similar to the object types, some social networks may further specialize their link types. For example, a Youtube user may be linked to a post using a 'posted', 'liked' or 'disliked' link. We denote the link type of an edge (v_1, v_2) as $E_{LT}(v_1, v_2)$. We should note here that not all objects can be linked with any link type. For example, it may not make sense to connect two users with a link denoted as 'uploaded'.
- *Collections and Colinks.* One characteristic of *OSN*s is that an object can link to *c*ollections of objects of the same object and association type. For instance, a Facebook user can be associated with several photo albums using an 'uploaded' link type. Instead of referring to each of the albums as a different connection we encapsulate the photo albums to form an album collection and use a single link from the objects. In the *SocWeb Model*, we call such types of links *C*o-links and we formally define them as: $Colink(v, \mathbf{v}) = \{v_1, v_2, \ldots, v_n\}$, where $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is an object representing the collection of objects $v_1, \ldots, v_n$ with the same type, and v is the object linking to the collection. Note that we create collections of objects based on the link type rather that the object type. An object may be connected to objects of the same type but the connection between them has a different meaning. For instance, a Facebook user can be connected to posts either because she liked or posted or commented on them.
- *Intra-OSN edges and S-links.* By using the definitions above, we can describe a graph that represents a single *OSN* s. Since our goal is to monitor a set of *OSN*s, we need a way to represent the interconnections among them. One straightforward approach would be to consider the union of the set of *OSN*s, assuming that their graphs are disconnected. By following this approach, however, we are not able to take advantage of the fact that a user may have accounts in different *OSN*s and attribute her objects and links to the same person. Being able to identify the same person across *OSN*s is of great importance for several different scenarios such as opinion mining, personalization and ad targeting.
  To this end, we also define a special kind of cross-*OSN* links. More specifically, for two different social networks s_1, and s_2, we consider the additional set of intra-*OSN* edges E_{s_1,s_2}, which is the set of directed edges (v_k, v_l) where $v_k \in V_{s_1}$ and $v_l \in V_{s_2}$. For these intra-*OSN* edges, we define the following types:

[2] Note that if $d_{out}(v) = 0$ and $d_{in}(v) = 0$ then v is an isolated vertex. Such vertices are difficult to discover as there are no links to them and are very unlikely to appear in a real *OSN*. For simplicity, in our model, we assume that we do not have such nodes.

- $SP\text{-}link(v_1, v_2)$ iff $(v_1, v_2) \in E_{s_1,s_2} \wedge Primitive(v_2) \wedge Composite(v_1)$
- $SC\text{-}link(v_1, v_2)$ iff $(v_1, v_2) \in E_{s_1,s_2} \wedge Composite(v_2) \wedge Composite(v_1)$
- $SCo\text{-}link(v_1, \mathbf{v})$ iff $v_1 \in V_{s_1} \wedge v_2 \in V_{s_2} \wedge s_1 \neq s_2 \wedge Colink(v_1, \mathbf{v})$

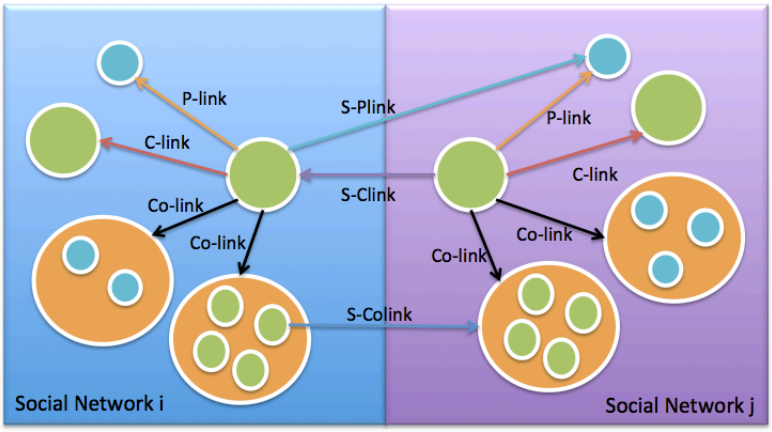


Fig. 1: Multi-social graph example with all possible basic object and link types of *SocWeb Model*.Nodes in blue and green refer to primitive and composite objects respectively.

To illustrate the abstraction that *SocWeb Model* offers, we show 2 *OSN*s in Figure 1 with all the possible basic object and link types. These types can be further specialized based on the description of each social network.

3 Accessing *OSN*s through a Generic API

Most of the social networks today provide an Application Programming Interface (API) to their data in order to allow developers to build applications. In almost all cases, the API provides a way for authorizing an application or a user to access data on the *OSN*. There are typically two levels of authorization: (a) acquiring the minimum credentials that each user/application needs to connect to the *OSN*, and (b) potentially selecting additional credentials that are useful to a given application (e.g. to access the birth dates of a user's friends).

Since our goal is to access multiple *OSN*s at the same time, we need to be able to create and maintain such multiple credentials and interact with a variety of APIs. This, however, is a challenging problem for the following reasons:

– Given the number of available *OSN*s there is also a large number of available APIs with multiple versions available. For instance, Twitter supports a REST API and a Streaming API and the REST API has multiple versions.
– Each API supports a large number of calls, parametrizable in a multitude of ways.
– Each *OSN* provides a diversity of encoding formats that can differ from one *OSN* to another. For example, language encodings are typically one of UTF-8, IS0-8859-1 or 1-3byte Unicode sequences(e.g "\u00ed").
– There is a large number of different ways to exchange information with an OSN, for example JSON, XML, KML, RSS (2.0) or ATOM.
– Different *OSN*s use different authorization procedures. Most of them rely on the standard OAuth protocol, which has a multitude of versions. Additionally, different platforms (Web, desktop, mobile) require different authorization procedures within the same *OSN*.

– Each *OSN* enforces its own limitations to the amount of requests that are allowed. Such limitations can be enforced by IP, by application, by user or by authorized/non-authorized calls within a given time period (typically per hour or per day). This implies that we need to be aware that an application limit has been reached when accessing information in an *OSN* and back off if necessary.
– Each *OSN* has its own set of error codes. Hence, we need to take appropriate action which may be different per *OSN*.

To alleviate these problems we propose a generic Social Network API (SNAPI) for arbitrary *OSN*s that is based on the *SocWeb Model* discussed in Section 2. We proceed by introducing the `socWebObject` data structure that is central to our system and we then define the SNAPI, whose purpose is to interact with arbitrary *OSN*s using the corresponding APIs through the use of `socWebObject`s.

3.1 The `socWebObject` Data Structure

Based on our discussion in Section 2, each node of an *OSN* graph has a set of p-links, c-links, co-links, sp-links, sc-links and sco-links associated with it. Additionally, each node has a unique identifier, a unique object type and belongs to a single *OSN* and may have one or more nodes pointing to it. To this end, in our system we use the following data structure to represent a `SocWebObject`:

```
socWebObject(id, type, plinks, clinks, colinks, splinks, sclinks, scolinks)
```

where p-links, c-links, sp-links, sc-links are maps to other `socWebObject`s based on the association type and co-links and sco-links are maps to an array of `socWebObject`s. For instance, let `obj` be a `socWebObject` representing a Facebook user, then we can reference the user name as `obj.plinks["username"]`, the hometown as `obj.clinks["hometown"]` and the i-th post of the user as `obj.colinks["posts"][i]`. Additionally, the hometown is another `socWebObject` and we can reference its longitude as `obj.clinks["hometown"].plinks["longitude"]`.

Call
initialize($\bigcup_{i=1}^{N} Def(s_i)$, app_id[])
apply_definition($Def(s)$)
apply_credentials(user_id, app_id, credentials)
apply_constraints(constraint[])
get_object(object_id, type, linktype[])
get_links(object_id, type, linktype[])

Table 1: SNAPI Calls. $Def(s)$ is the definition of *OSN* s, i.e. the set of nodes and different types of links of s. `linktype[]` represents a subset of the different kinds of links p-links, c-links, etc.

3.2 A Generic Social Network API (SNAPI)

In our system, we provide access to the different *OSN*s by implementing wrappers for the different API requests together with the most popular parametrization options. Overall, our generic API consists of the set of calls as shown in 1, which we discuss in more detail.

SNAPI Initialization. SNAPI is initialized by providing the definition (i.e. set of nodes and links with their types) of the *OSN*s that we are interested in following to the `initialize()` call together with the application ids that are authorized to access information in the *OSN*s. In the cases where we discover a new *OSN*, or we decide to change accessing data through a different application, we can do so on-the-fly through the `apply_definition()` call.

SNAPI Authorization. For authorization purposes SNAPI provides the `apply_credentials()` call which uses the login and password provided to connect and acquire the necessary credentials. In most cases, the credentials are access tokens returned by the OAuth protocol.

SNAPI Constraints. In order to make our system's monitoring capabilities more flexible, we have provided SNAPI with the capability to specify a set of constraints that can be enforced during the monitoring process. More specifically, for each object type we can define rules for each one of its connected object types (i.e. p-,c-,co-,sp-,sc-,sco-links). For instance, suppose we want to retrieve the location of a tweet only if its longitude is within a specific range. We can express this rule as:

`tweet.clinks[''location''].plinks[''longitude''] ≥ minimum_longitude &&`
`tweet.clinks[''location''].plinks[''longitude''] ≤ maximum_longitude`

Providing this capability allows us to only monitor parts of an *OSN* that we are interested in thus saving resources. In our current implementation, we only allow simple boolean constraints to be provided as input to SNAPI. We plan to implement these constraints in the form of a full-scale Complex Event Processing system [12, 10, 26] in the future.

SNAPI Fetching of Data. We provide two different calls for fetching within *SocWeb*, fetching of objects and fetching of links, through the `get_object()` and `get_links()` calls respectively. For both calls, we need to provide an object type together with its id which uniquely identifies an object within a given *OSN*. Such an id is typically created by the *OSN* and found by following the different kinds of links during monitoring.[3] We can also specify the kinds of links that we are interested in (c-links, p-links, etc.) through the `linktype[]` parameter. In the case of `get_object()`, the system proceeds iteratively by fetching the object's links and applying the provided constraints. If, when retrieving an object, we cannot immediately decide whether it satisfies a constraint (e.g. we have not discovered a property such as longitude that was specified in a constraint), we place it in a queue which we periodically clean in order to keep only objects that satisfy the given constraints.

Efficiently fetching the data from a set of *OSN*s is a challenging task which we discuss next.

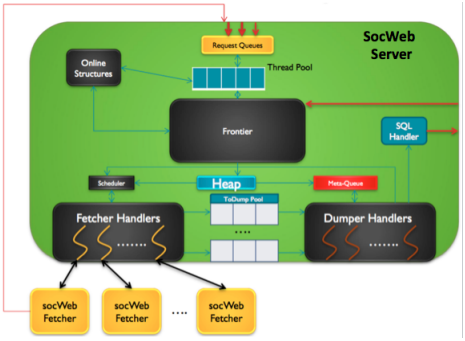


Fig. 2: SocWebFS Architecture

4 Efficient Monitoring of *OSN*s: The SocWeb Fetcher System

We have so far discussed how we represent a set of *OSN*s within SocWeb as well as its generic API that enables us to access the *OSN*s. Given the enormous size and update rates of information in the *OSN*s we need to find ways of monitoring this information in an efficient way. Not all users post information at the same time, or at the same rate. If we blindly start downloading everything our system comes across, we may end up with redundant information. In addition, since most of the *OSN*s impose limitations to the amount

[3] Of course, there could be a conflict where a given object id may correspond to objects in more than one *OSN*s. We handle this case internally in our system by having an additional *OSN* id attached to the object id.

of objects that we can download within a given period of time, making smart decisions of what to download and how often is of paramount importance.

In this section we discuss our design around a fetcher system (the SocWebFS) which is built with these constraints in mind and can make decisions on-the-fly on what to download at a given time. Our system consists of a central master server that coordinates a set of fetchers. The server keeps track of statistics on previously downloaded objects and makes estimations on what to download next. We present an overview of the whole system in Figure 2 and we describe each of the components in the following subsections.

4.1 SocWebFS Server

This is essentially the brain of the system which decides which object to fetch, when to fetch and how often. The SocWebFS Server comprises seven modules: (a) the Request Queue-Thread Pool, (b) the Frontier Queue, (c) the Scheduler, (d) the Fetcher Handlers, (e) the ToDump Pool, (g) the Dumper Handlers. As is typical in these kinds of architectures, these modules operate in a pipeline fashion, sharing global online data structures in order to coordinate.

Online Structures. This module is responsible for maintaining statistics regarding the online activity of the users. These statistics are useful in deciding which users to prioritize when downloading objects. The intuition is that a user who has been posting information at a high rate in the past will continue to do so in the future. To this end, for each user we maintain her effective online interaction rate:

$$EIR(u) = \frac{1}{7} \cdot \#Requests(u) \; in \; the \; last \; 7 \; days$$

Request Queue-Thread Pool. To handle the requests for downloading within our system we maintain a thread pool. Each request is handled by the first available thread, while the number of threads is tuned dynamically based on the amount of incoming requests. Depending on the incoming request, a thread, might change the online structures, update socWebObjects or statistics related to socWebObjects or spawn/delete fetcher/dumper handlers.

Frontier Queue. This is the most important part in the system as it decides on the prioritization of which objects to download at any given moment. In its simplest form [9], the first item from the queue is fetched and then it is placed back in queue to be refreshed again later. This operation can be performed in a variety of ways in order to optimize for freshness or age of the objects [8], optimize for bandwidth or cost [20], or adhere to politeness policies.

In our implementation, the frontier module consists of 2 sub-modules: a set of F front FIFO queues, that guarantee prioritization of `SocWebObject`s to download, and a set of Q back queues that guarantee polite behavior of the fetcher (i.e. ensuring that we are not downloading too fast from a given *OSN*) by monitoring download rates from the *OSN*s. Within the front queues we prioritize the objects based on a set of metrics:

- **User-based.** In this case, we take into account the user's interaction rates with the *OSN*. The higher the EIR for a given user, the higher priority her objects are given in the queue.
- **Change-Rate-based.** Since the objects change periodically, we need a way to keep track of which ones are more likely to change in order to prioritize them first. We consider an object to have changed if any of its c-linked objects has changed. To this end, we use a change rate metric for each object:

$$CR(obj, link) = \frac{\#Changed \; c\text{-}links}{t_l - t_s}$$

  where t_l, t_s are the timestamps of the last and first changed c-link objects respectively. This definition of average change rate works well in most cases, but our system is flexible to employ different and more sophisticated change rate approaches (e.g. fitting a probabilistic distribution to the changes to estimate the rate).
- **Importance-based.** Depending on the social network schema, it is sometimes the case that some objects are more important than others. For example, when a user updates their location may be more important than a new comment that she just posted.

To capture this difference in importance among the objects, we employ an importance metric $I(obj)$ for each object. $I(obj)$ can be static per object type (e.g. location is 10 times more important than comment) or can be dynamically adjusted. In our implementation, we followed a dynamic approach and we define the importance of each object as:

$$I(obj) = \frac{1}{n} \sum_{u \in Users} EIR(u) \cdot App(obj, u)$$

This is essentially the average EIR of all the users associated with the given object weighted by an application-specific weight App. For example, if we were using SocWeb to implement a search engine $App(obj, u)$ could be the number of times that obj was returned as a result to the user u. In this case, the more times the user sees obj the higher the weight.

Based on these metrics, all the objects of the frontier are given a priority and are placed in a queue to be scheduled for fetching.

Scheduler. The goal of the scheduler component is distribute the prioritized objects from the frontier across several fetcher handlers. The scheduler has three main goals:

- To balance the total fetching workload across the several SocWebFS Fetcher nodes by deciding when and where to send an object for fetching. To this end, the Scheduler maintains histograms per machine to estimate the amount of time needed for each machine to perform each request. Given a `socWebObject` and its change rate per link as computed in the change rate level, we estimate the amount of requests that have to be performed to fetch an object and we pick the machine with the smallest load to handle the fetching.
- To ensure that fetching of an object will not exceed the limitations (e.g. IP, time, API) posed by the *OSN*s. If the scheduler estimates that fetching of an object may potentially exceed one or more limitations imposed by the *OSN*, it postpones its fetching for later and periodically repeats this estimation.
- To ensure fault tolerance by guaranteeing that fetches that received an error or time out will be considered for fetching again in the near future. To this end, each object gets a unique session id (sid) that uniquely identifies the transaction. This sid together with the initial request timestamp t_s are used to detect whether we have waited sufficient time before we consider the fetching of the given object as timed-out. If the object request has timed out, we place the object back to the frontier to be fetched again later. The process of resending an object back to the frontier is performed only a fixed number of times per object (set to 3 in our system).

Fetcher Handlers. The fetcher handlers are responsible for the communication of SocWebFS Server with the SocWebFS Fetchers. Each SocWebFS Fetcher corresponds to a single Fetcher Handler that serves as middleware for the communication of the SocWebFS Fetcher with Server's resources. Upon retrieving a set of requests the handler serializes them and sends them to the fetchers for processing. The handler also updates the initial request timestamps t_s in the frontier.

SocWebFS Fetchers. The goal of a SocWebFS fetcher is to retrieve a single object that is assigned to it by its corresponding handler. Each fetcher uses the SNAPI that we described in Section 3 which is initialized at the startup of the system. After authentication, the fetchers operate in four states: (a) connect, which initializes the connection to the *OSN*, (b) wait for requests to be assigned, where the fetcher blocks and awaits yet-to-be-fetched objects from the server, (c) fetching, which requests objects from the *OSN*, and (d) upload, which uploads the fetched objects in a bulk mode back to the SocWebFS server. When a SocWebFS fetcher has finished fetching of objects it issues an upload request that returns a subset or all of the fetched objects along with statistics(amount of requests, time per request, limitations reached). Next, Fetcher handler has to send the statistics to the Scheduler, check whether the fetching of each object was valid or resulted in an error, and append the fetched object to the Dump Pool.

Dumper Handlers and Dump Pool. Each Fetcher Handler has a corresponding Dumper Handler which takes on the task of saving the retrieved objects to disk and appending newly found objects to the frontier. To enable the communication of those components, we use a Dump Pool which is essentially implemented as a set of queues, with each queue corresponding to a pair of Fetcher-Dumper Handlers. In this case, the Fetcher acts as a producer and the Dumper as a consumer. We handle updates of objects by maintaining multiple versions of objects, but keeping only those versions that are linked by other objects.

4.2 Privacy Considerations

We have discussed the overall architecture of our SocWeb system which enables us to download and store objects from a set of *OSN*s locally. Our system is flexible enough to handle different *OSN*s with different APIs and limitations on the data that we can store.

However, given the sensitive nature of some of the data in the *OSN*s we may need to enforce additional limitations in certain cases. For example, certain *OSN*s pose limitations on whether the data collected can be at all stored on disk or can only be used on-the-fly. To this end, SocWeb provides subscriptions of external applications and Dumper Handlers instead of storing the data locally. The only indirect requirement that we have for the external handlers is that they are capable of consuming the data at the rate that the fetcher retrieves them from the *OSN*s. If the external rate is slower, SocWeb drops some of the objects to match the external consuming rate.

In addition to storing the data, there are also challenges in enforcing access-level constraints to the data. For example, consider the scenario when users A and B are both friends with user C but user A can see C's birthdate but B cannot because C specified so. In this case, users A and B have different access permissions to user C's p-link objects. This scenario may happen for all different kinds of links that we defined in Section 2. To solve this problem, we additionally employ a per-user storage space where we keep, for a given user, the conflicting parts of an object that are different from the global version of the object in the system. In this way, we can enable applications using the data that SocWeb collected to adhere to the privacy of the data because of the different user access levels.

5 Evaluation

Our main objective in evaluating our architecture and the *SNAPI* interface is to establish the utility of our approach in terms of a number of key characteristics.

- *Robustness:* to avoid traps, *SNAPI* can be parameterized with constraints for cycle detection during the fetching process. Furthermore, Dumper Handlers determine whether an object may be eligible for (re-)fetching. Application dependent traps are also detected through parameterization of these handlers.
- *Politeness:* each *OSN* poses its own limitations in terms of calls per-application, per-*IP* and/or per-user. Our *Scheduler* and *SocWebFS*-fetchers offer compliance with imposed retrieval rates as designated by *OSN*s. The above can also dynamically re-set limitations that change on the fly. *SNAPI* also determines whether heavy workload crawling activities are in place and makes use the respective *Facebook/Youtube* streaming *API* to better facilitate fetching of objects.
- *Distributeness:* our proposed architecture is based on the single-*SocWebFS*–server and multiple-concurrent-clients model, all operating in star-like fashion. The design warrants for fault tolerance (*Frontier*), workload balance (*Scheduler*), elasticity (*Scheduler*) and redundancy manipulation (*Dumper Handlers*). Even in the case of a software crash, the *SocWebFS*–server loses no data as the crawlers will await for the main server to become alive anew. The state of the server is maintained as expressed by the *Frontier* and *MetaQueue* structures is maintained by the back-end database.
- *Quality:* the relevance of retrieved data is of paramount importance to users and their applications. Our policies that differentiate between data of "interest" and "no interest" and so assist in achieving per-user extraction quality characteristics.
- *Scalability and High Performance: SocWebFS* is able to scale up (or down), in the presence of more (less) machines and/or bandwidth changes. To examine the scalability of our system we conducted corresponding experiments.
- *Extensibility:* The modularity introduced in the design of *SocWebFS* allows several levels of extensibility. Application dependent policies are introduced in the form of constraints to parameterize *SocWebFS* components (*SNAPI*, *Scheduler*, *Dumper Handlers*). These policies help the proposed model to render a simple yet powerful abstraction for multi-social networks description.

By and large, the above characteristics are those that have been used over time to ascertain effectiveness in Web crawling [18, 21, 25]. We also treat carefully the trade-off between performance and maintenance

of up to date information (i.e., freshness). As an object may receive repetitive requests for either edited or deleted distributed content, this will inevitably lead to performance degradation. *Scheduler*'s design uses the object change-rate as an estimator for changes expected in the future and offers accurate quantitative information regarding this issues helping attain a good balance in the trade-off at hand.

We should also point out that we have designed our systems so as to be able to handle diverse types of retrieved objects, fetching protocols, arbitrary social networks as well as multiple data formats such as *XML*, *JSON*, UTF8-encoding, etc.

5.1 Evaluation Approach and Settings

We experimented with *SocWebFS* using 2 popular *OSN*s: *Facebook* and *YouTube*. Our evaluation is weaved around 2 experimental use-cases that illustrate not only the use of our system but also its compliance of our prototype with the aforementioned behavioral and performance characteristics. In the first use-case, we exclusively use *Facebook* to demonstrate the applicability and value of *SocWebFS* in the presence of a single *OSN*. The second scenario uses both *Facebook* and *YouTube* networks and explores the inter-connections formed as soon as users deposit videos on one and proceed with respective posts on *Facebook*. We refer to the first use-case as *Facebook Spider* and to the second as *Aggregated Social World*.

For both experiments, we employed a private laboratory cloud made up of physical machines featuring Intel(R) Xeon(R) CPU X3220 processors at 2.40GHz, 8GB of RAM all connected through a 1GBps Ethernet switch. We used 7 virtual machines (VMs) from this cloud with each VM featuring a 2GB main memory running a client module (i.e., *SocWebCrawler*). One of these VM servers also undertook the role of the *gateway* as all *SocWebCrawlers* would issue requests to *OSN*s through this gateway using the *NAT* protocol.

5.2 Facebook Spider

In this use-case, we employ *SocWeb* to play the role of a social *spider* for *Facebook*: given an initial number of objects-nodes in the *OSN* graph, we intend to retrieve objects by exploiting adjacent links to already visited nodes. Social graphs are however inherently dynamic. Thus, we will have to continually monitor already visited nodes for new, deleted and/or updated adjacent links.

Before we start working with *SocWeb*, we are first required to formally define an abstraction of the candidate *OSN* to be crawled as Section 2 outlines. Most of the definition effort here is directed towards designating the specific objects and link type of interest than the entire network. In this use-case, we create an abstraction of *Facebook* that consists of users, albums and posts as our key-interest object types. Each user is linked with her posts, albums and friends (*co-links*), his hometown, school, work (*c-links*) as well as his first name, surname, birthday, last post's and album's creation time (*p-links*). Each post and album is connected to the number of likes and comments received (*p-links*). We also consider albums to be connected with their photos (*co-link*).

During the first time of crawling of a user we want to download the full list of her posts, albums and friends and her *c-* and *p-links*. We decide to re-introduce a user to the *Frontier* module, if and only if her amount of posts is more than a threshold [4]; this threshold can be perceived as a constraint of significance (i.e., user["posts"]["likes_cnt"] $>$ 1000). We also append to the *Frontier* her friends and corresponding albums if they were uploaded during the last week with respect to the time of the their crawling. For albums, we also require to be of significant interest (i.e., user["albums"]["comments_cnt"]$>$1000). Once an object has been fetched, its monitoring commences with regard to changes in her links (i.e., user["posts"]["created_time"] $>$ user["last_post_time"]). If we are required to crawl a friend of a user, we follow the same procedure but we don't further crawl the friends of friends. If *SocWeb* determines to re-crawl some specific object we consider only content created after the last time it was crawled.

[4] Manually set to 1,000 for this experiment.

The last action during *SocWeb* initiation is to acquire a set of nodes in order to start crawling. Here, this initial set of nodes consists of users whose profiles are publicly available. A large database containing such names found at `drupal.org/project/namedb`, contains 129,036 names listed in alphabetical order. We then use *Facebook–API* to obtain user–ids with similar names. We were able to retrieve 25,556,963 user–ids from which we considered only 881,549 users who maintain entirely public profiles.

We focused our crawling in the period of July 26th, 2011, to August 19th, 2011. Our assumption was that a good number of crawled users in this period were on vacation. In this regard, they were presumably uploading live content of their daily activities in a bursty manner throughout the day. Also, live content regarding vacation is invariably deemed attractive for followers (i.e., friends, subscribers etc.) to either "like" or comment on. Table 2 depicts the overall outcome of this specific crawling exercise that yielded a sizeable dataset of 2,437 albums and 75,370,629 posts. Apart from the live content, we also crawled their full history.

Table 2: *Facebook* Dataset

Names	Ids Found	Public Users	Albums	Posts
129,036	25,556,963	881,549	2,437	75,370,629

Figure 3 presents the distribution of the posts made by users in this use-case. This distribution has a mean of 85.98, standard deviation of 182.48, skewness of 4.10, kyrtosis of 23.79 and follows a power-law. Evidently, the rates at which users generate new content vary significantly. The overwhelming majority of users does not create much content and they infrequently use *Facebook*; some may even have deactivated their accounts. On the other hand, a small portion of users produce content at a high-rate. The *Frontier* component of *SocWeb* in conjunction with the *Scheduler* can effectively monitoring the change–rate metric in order quantify content differentiations. Responding to such observations, users with high content creation rates are placed to high–priority queues.

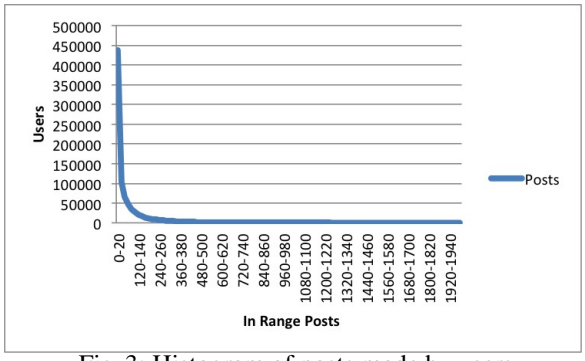
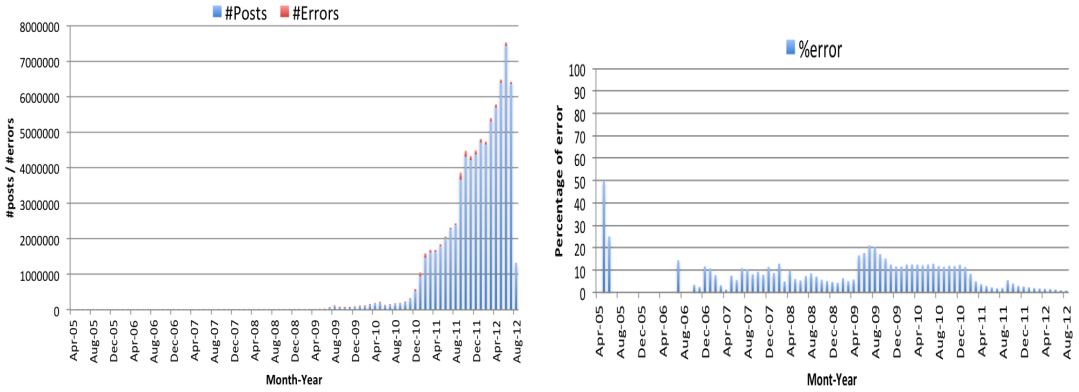


Fig. 3: Histogram of posts made by users

To further quantify the effectiveness of the change-rate metric, we measure its fit on the retrieved dataset of Table 2. In our setup, we sort the retrieved posts in ascending creation time. Then, for each user, we capture the change rate of his/her posts at any point in time while processing the sorted dataset in a streaming fashion. As soon as a new user post shows up in the stream, we use the current change–rate to measure whether it could accurately predict or not the creation time of the new post. This strategy is reflected in the decision making process of our *SocWebCrawlers*. In this context, we are only interested in establishing mis-predictions as yield of our under-estimation of the creation times of posts. Over-estimating the creation time of posts, while it is considered a misbehavior in general, in our setup is partially alleviated by deciding a time window of maximum expected change (i.e. the *Scheduler* decides to re-crawl an object if this time window has elapsed). In this use-case, we have empirically set this time window to 1 month. In Figure 4a, we show the distribution of posts and the fraction of mis-predictions yielded by the change-rate metric per month, beginning between April 2005 and some time in August 2012. An interesting property of this

distribution is the growth of *Facebook* in terms of posts per month. Thus, the choice of the scaling factor of the change-rate metric introduced in Section 4 can adequately fit this kind of increasing *OSN* behavior. Finally, our analysis shows that the amount of mis-predictions as a percentage of posts on a monthly basis remains significantly low as Figure 4b indicates.



(a) Posts and mispredicted posts(#posts that exceeded the estimated time of creation) per month

(b) Error rate percentage

Fig. 4: *Facebook*'s growth and *SocWeb*'s adaptivity

5.3 Aggregated Social World

Figure 5 shows how *Facebook* users create a post on which they upload content from *Youtube*. The objective of this use-case is to demonstrate that *SocWeb* readily facilitates fetching *Youtube* information pertinent to *Facebook* posts. Offering aggregate information has been successfully employed in event processing before [4]. When more multiple *OSN*s are present, information aggregation calls for communication across social networks, a function that *SocWeb* can readily offer. Here, we also report on *SocWeb* performance as the number of *SocWebCrawlers* increases and discuss the quality of retrieved data.

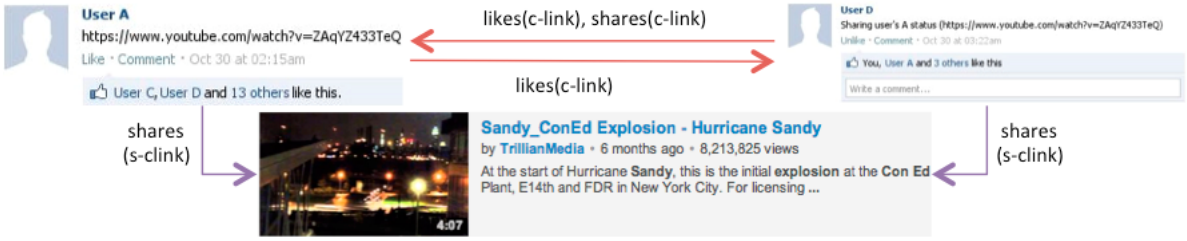


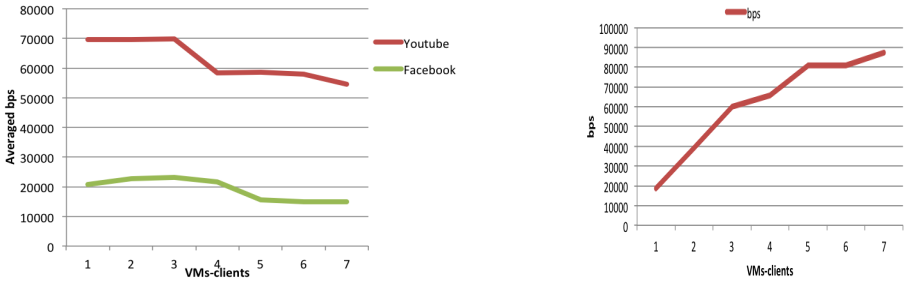
Fig. 5: An example of intra-social connections.

We initiate *SocWeb* by following the same steps as in the *Facebook Spider* case; object and link types of interest are shown in Figure 5. By exploiting *SocWeb Model*'s semantics, we formally define a *Facebook* post to be connected to *Youtube* video(s) as *sc-link(s)*. Further, we let *SNAPI* trigger requests via *Youtube*'s *API* depending on the content of *Facebook* post by also applying constraints (i.e. `post[''type''] = ''video" && is_youtube_video(post[''link''])`). As social networks inherently display power-

law distributions (i.e. Figure 1), we select 10,000 users from our first experiment, whose posts follow the same skewed distribution, as node-seeds to commence crawling.

Every object that undergoes crawling spends time passing through the various *SocWeb* components including the fetcher handlers, clients, dump pool and finally the dumpers. The expended time for such trips largely depends on the workload of *SocWeb* and the available clients to perform the crawling step. The major overhead though comes from the time required to fetch each object; in our experiments, we establish that an average 84% of the time goes towards fetching objects when all VMs are in use. Moreover, social network providers penalize the concurrent access of their graphs. Thus, incrementing the number of *SocWebCrawlers* employed doesn't necessarily reciprocate in terms of the volume of data retrieved. In this regard, we report on the scalability of *SocWeb* in terms of data returned to the *SocWeb* server from the *SocWebCrawlers* in the unit of time (bytes per second-*bps*) as well as the average *bps* retrieved from each social network provider as a function of the *SocWebCrawlers* employed.

Figure 6a shows the average *bps* rates obtained. *Youtube*'s response rate is uniformly higher than that of *Facebook* although no major optimization (i.e., batch requests) was included in our implementation to the *Facebook*'s *API*. Both providers show consistency to their response rates; however after 3 concurrent connections, throttling was encountered. Moreover, when the number of *SocWebCrawlers* increased, the *OSN*-provider limitations became apparently severe.



(a) *Facebook* and *Youtube* response rate

(b) *SocWeb*'s scalability

Fig. 6: Quantitative analysis of scale

Figure 6b shows the average *bps*-rate extracted by *SocWebCrawlers* as the number of deployed *SocWebCrawlers* increases. The *OSN*s-imposed limitations for more than 3 concurrent connections affect the obtained *bps*-rates. In our experiment, no NAT-pipe saturation occurred and contention was sufficiently low; as far as the degradation of the stream rate (*bps*) achieved in the dumpers was only 2% of the average stream rate of *SocWebCrawlers*.

It is also worth pointing out some limitations we encountered as *OSN*s apparently impose constraints on (possibly) *IP*s, *Application ID*s and/or volume of data fetched. In the early stages of our experimentation, *SocWeb* faced an outage of more than 1 day due to the above limitations whose nature appeared to be dynamic. By taking into account the history log, we were able to guide our *Scheduler* to a more productive fetching cycle by following a more polite etiquette and building on our experience regarding the specific times of the day in which we could launch more voluminous crawling activities.

6 Conclusions

In this paper we present *SocWeb*, a distributed crawling system that helps monitor multiple *OSN*s simultaneously. We introduced the *SocWeb Model* to formally define not only every participating *OSN* bit also existing and developing intra-connections among them. We discuss problems that emerge when applications communicate with with social network providers in the presence of multiple *OSN*s and suggest a generic

API, *SNAPI*, to alleviate them. Using the semantics of *SocWeb Model* and the *SNAPI* we outline the key design choices for our monitoring system based on *SocWebCrawlers*. We demonstrate the utility of *SocWeb* while experimenting with *Facebook* and *Youtube* and working on two use-cases.

References

1. S. Asur, B.A. Huberman, G. Szabo, and C. Wang. Trends in Social Media. In *5th Int. AAAI Conf. on Weblogs and Social Media*, Barcelona, Spain, February 2011.
2. L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution . In *Proc. of the 12th ACM SIGKDD Conf.*, Philadelphia, PA, October 2006.
3. Z. Bar-Yossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz. Approximating Aggregate Queries about Web Pages via Random Walks. In *Proc. of 26th Int. VLDB Conf.*, pages 535–544, Seoul, Korea, September 2006.
4. H. Becker, D. Iter, M. Naaman, and L. Gravano. Identifying Content for Planned Events across Social Media Sites. In *Proc. of 5th ACM Int. Conf. on WSDM*, Seattle, WA, February 2012.
5. C. Budak, D. Agrawal, and A. El Abbadi. Structural Trend Analysis for Online Social Networks. *Proc. of the VLDB Edowment*, 4(10):646–656, July 2011.
6. S.A. Catanese, P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti. Crawling facebook for social network analysis purposes. In *Proc. of the Int. Conf. on Web Intelligence, Mining and Semantics (WIMS '11)*, Songdal, Norway, May 2011.
7. D. Horng Chau, S. Pandit, S. Wang, and C. Faloutsos. Parallel crawling for online social networks. In *Proc. of the 16th Int. Conf. on WWW*, pages 1283–1284, Banff, Canada, May 2007.
8. J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. of the 2000 ACM SIGMOD Conf.*, pages 117–128, Dallas, TX, May 2000.
9. J. Cho and H. Garcia-Molina. Parallel Crawlers. In *Proc. of the 11th Int. Conf. on WWW*, pages 124–135, Honolulu, HI, May 2002.
10. E.A. Rundensteiner D. Wang and R.T. Ellison. Active Complex Event Processing Over Event Streams. *Proc. of the VLDB Endow.*, 4(10):634–645, July 2011.
11. W. Dou, K. Wang, W. Ribarsky, and M. Zhou. Event Detection in Social Media Data. In *IEEE VisWeek Workshop on Interactive Visual Text Analytics*, Seattle, WA, October 2012.
12. M.H. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. *Proc. of the VLDB Endow.*, 2(2):1558–1561, August 2009.
13. M. Gjoka, M. Kurant, C.T. Butts, and A. Markopoulou. Walking in Facebook: a Case Study of Unbiased Sampling of OSNs. In *Proc. of the 29th INFOCOM Conf.*, San Diego, CA, March 2010.
14. M.R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. On Near-uniform URL Sampling. In *Proc. of the 9th Int WWW Conf.*, Amsterdam, The Netherlands, May 2000.
15. P.G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *Proc. of the ACM SIGMOD Cong.*, pages 265–276, Chicago, IL, June 2006.
16. B. Kahle. Preserving the Internet. In *Scientific American*. Nature Publishing Group, March 1997. `www.sciamdigital.com`.
17. J. Leskovec, K.J. Lang, and M. Mahoney. Empirical Comparison of Algorithms for Network Community Detection. In *Proc. of the 19th Int. Conf. on WWW*, pages 631–640, Raleigh, NC, April 2010.
18. C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, 2008.
19. M. Naaman, J. Boase, and C.-H. Lai. Is It Really About Me?: Message Content in Social Awareness Streams. In *Proc. of ACM Conf. on Computer Supported Cooperative Work (CSCW '10)*, pages 189–192, Savannah, GA, February 2010.
20. A. Ntoulas, P. Zerfos, and J. Cho. Downloading Textual Hidden Web Content Through Keyword Queries. In *Proc. of the 5th ACM/IEEE JCDL Conf.*, Denver, CO, June 2005.
21. M. Rabinovitch and O. Spatscheck. *Web Crawling and Replication*. Addison Wesley, 2001.
22. K. Punera S. Chakrabarti and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proc. of the 2002 ACM WWW Conf.*, pages 148–159, Honolulu, Hawaii, USA, 2002.
23. A. Sadilek, H. Kautz, and J.P. Bigham. Finding your Friends and Following Them to Where You Are. In *Proc. of the 5th ACM Int. Conf. on WSDM*, pages 723–732, Seattle, WA, February 2012.
24. T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors. In *Proc. of the 19th Int. Conf. on WWW*, pages 851–860, Raleigh, NC, April 2010.
25. V. Shkapenyuk and T. Suel. Design and Implementation of a High-performance Distributed Web Crawler. In *Proc. of the 18th IEEE ICDE Conf.*, pages 357–368, San Jose, CA, February 2002.
26. E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing Over Streams. In *Proc. of the 2006 ACM SIGMOD Conf.*, pages 407–418, Chicago, IL, June 2006.