Social Piggybacking: Leveraging Common Friends to Generate Event Streams

Aristides Gionis, Flavio Junqueira, Vincent Leroy, Marco Serafini and Ingmar Weber

Yahoo! Research Barcelona

{gionis,fpj,leroy,serafini,ingmar}@yahoo-inc.com

Abstract

Social networking systems present users with online event streams that include recent activities of their friends. Materialized per-user views are a common way to generate such event streams on-line and with low latency. We propose improving throughput by using *social piggybacking*: process the requests of two friends by just querying and updating the view of a third common friend. By using one such *hub* view, the system can serve requests of the first friend without querying nor updating the view of the second. This reduces the overall load on views, and can be implemented with minimal adaptations by existing social networking architectures. We show that, given a social graph, there exist hub-based schedules that minimize the overall number of view requests, but computing them is NP-hard. We propose a heuristic to solve the problem and evaluate it using samples from the Twitter and Flickr social graphs. The evaluation shows that existing solutions generate up to 2.4 times the amount of view requests induced by our heuristic.

Categories and Subject Descriptors H.3.5 [*INFORMATION STORAGE AND RETRIEVAL*]: Online Information Services – data sharing

General Terms Algorithms, Performance

Keywords Social Dissemination, Social Piggybacking

1. Introduction

Communicating through prominent social networking applications like Facebook, Twitter, Tumblr or Yahoo! News Activity became highly popular in the past few years. With such applications, users connect to other users and share *events*: short text messages, URLs, pictures, articles they read, songs they listened to etc. Users can see online event streams containing recent relevant events shared by connected users. Users and their connections form a *social graph*.

In this paper, we focus on the problem of generating event streams, the predominant workload of many social networking web applications (e.g. 70% of the page views of Tumblr [1]). Social networking systems must generate event streams online, in order to include the latest relevant events shared by users, and with very low latency, for users expect web pages to load in fractions of seconds. Storing events in materialized views is a common optimization technique to handle a large load of queries (generating event streams) and updates (storing new shared events). Views are formed on a per-user basis, for each user sees a different event stream based on his social graph, and are stored by back-end data stores. Views can contain events from a user's friends and from the user itself. Our discussion is independent from the implementation of the data stores, which are typically in-memory key-value stores.

We consider standard social networking systems similar to the one depicted in Figure 1. Users send requests to the social network, for example through their browsers. Internal application logic of the social network translates these requests into updates and queries to the back-end data stores. The data store clients learn the set of friends of a user, and thus the views that need to be accessed, by reading the social graph. They then forward the requests to the data stores keeping the proper views. For queries, data stores and clients can select the most relevant events and return them to the user. The notion of "relevant" is application-specific.

The throughput of the system is proportional to the amount of data transferred to and from views. In this paper, we propose a new approach to improve the throughput of social networking applications by reducing the number queries and updates sent to data stores for a given workload. This enables serving the workload using fewer data store nodes.

Existing social networking systems already use strategies to reduce the number of requests sent to the data stores [1]. The views of the friends of a user, whom we will call John, can be included in two user-specific sets: the *push set*, con-

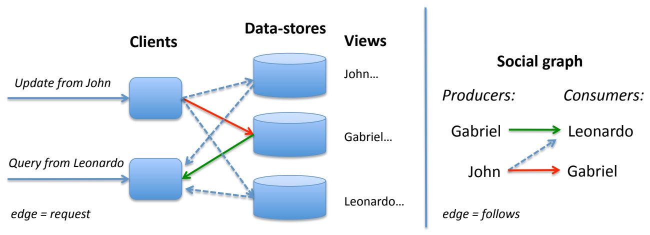

Figure 1. Standard architecture and request flow. The push set of John (red edges) and the pull set of Leonardo (green edges) include Gabriel's view, which acts as a hub. Direct communication over dashed edges is not necessary.

taining friend views that need to be updated when John shares a new event, and the *pull set*, containing friend views that must be queried to generate John's event stream. The composition of update and pull sets has a strong impact on performance. Two standard *request schedules* used to determine these sets are push-all and pull-all. In push-all schedules, the push set contains all John's friends, while the pull set contains only John's view. Such schedules are particularly efficient in read-dominated workloads because each query generates only one view request. Pull-all schedules are specular, and are better suited for write-dominated workloads. A hybrid between pull- and push-all schedules is proposed by Silberstein *et al.* [9] to make John's events visible to his friends: data store clients push John's updates to the views of the friends whose query frequency, is higher than John's update frequency; the clients pull from John's view when the remaining friends query the system.

We propose a more efficient approached called *social piggybacking*: process the requests of two friends by just querying and updating the view of a third common friend. Consider again the example of Figure 1. For generality, we model a social graph as a directed graph where a user may follow another user, but the follow relationship is not necessarily symmetric. In the example, John includes Gabriel in its push set, so it updates Gabriel's view every time it shares a new event. When Leonardo sends queries, clients can pull John's updates from Gabriel's view. In this case, Gabriel's view becomes a *hub*. Using hubs requires fewer view requests than push-all, pull-all, or hybrid schedules: John does not need to update Leonardo's view, and Leonardo does not need to query John's view. The high clustering coefficient of social network indicates the presence of many hubs, making hub-based schedules very efficient [7].

Using hubs in existing social networking architectures is very simple, for it just requires a careful configuration of push and pull sets. In this paper, we describe algorithms to find, for a given social network with a given workload, a request schedule that enables users to receive recent updates from all their friends, while minimizing the overall rate of requests sent to views. We call this problem the *social dissemination* problem.

Our contribution is a comprehensive study of the social dissemination problem. We first show that solutions of the social dissemination problem either use hubs as Gabriel in Figure 1 or, when efficient hubs are not available, make pairs of users exchange events by sending requests to their view directly. This result reduces significantly the space of solutions that needs to be explored, simplifying the analysis. We show that computing optimal request schedules using hubs is **NP**-hard, and we propose a heuristic, which we call NOSY. Initial evaluation on samples from real Twitter and Flickr graphs show that existing request schedules generate up to 2.4 times the amount of view requests induced by our heuristic.

2. Social dissemination problem

We formalize the social dissemination problem as a problem of propagating events on a social graph. For the purposes of our analysis, we do not distinguish between vertices, the corresponding users, and their views. When a vertex u *pushes* an event to a vertex v in the model, this corresponds, in an actual system like the one of Figure 1, to the user u sending an update request containing the event to the view of the user v. Similarly, if a vertex v *pulls* events from a vertex u, this corresponds to the user v sending a query request to the view of the user u. Users always access their own view with updates and queries. According to the formal model, the actual system should store events pulled by a user v into the view of v; however, we will show that implementations of optimal schedules do not require that.

We model the social graph as a directed graph $G = (V, E)$. The presence of an edge $u \rightarrow v$ in the social graph indicates that the user v *subscribes* to the events produced by u. We will call u a *producer* and v a *consumer*. Symmetric social relationships, such as being friends on Facebook, can be modeled with two directed edges $u \rightarrow v$ and $v \rightarrow u$.

A fundamental requirement for any feasible solution is that the events included in the stream have *bounded staleness*, i.e., the system delivers any event posted by a user u to all other users v who subscribe to u within a time bound Θ. The specific value of Θ may depend on various system parameters, such as speed of networks, CPUs, and external-memories, but it may also be a function of the current load of the system. The underlying motivation of bounded staleness is that typical social applications must present nearly real-time event streams, but delays may be acceptable.

Definition 1 (Bounded staleness) *There exists some finite time bound Θ such that, for each edge $u \rightarrow v \in E$, any query action of v issued at any time t returns every event posted by u at time $t - \Theta$ or before.*

In practice, queries return only a finite list of events, filtered according to application-specific relevance criteria. For example, they may return the latest events. Different choices can be easily adapted in our framework. For generality purposes, we do not consider filtering criteria but instead assume that all past events are returned by all queries.

A trivial, albeit practically unacceptable, way to guarantee bounded staleness could be to have all producers push their events to a single centralized view, and to have all consumers pull from that view. We rule out this option by requiring that each user view contains only events from the user itself or its producers.

Definition 2 (Non-triviality) *If the view of a user v contains an event produced by a user $u \neq v$ then $u \rightarrow v \in E$.*

The request schedule determines which edges are included in the push and pull sets of any user of the social graph. In our formal model, push and pull sets determine the way clients propagate events over an edge $u \rightarrow v \in E$. If v is in the push set of u, we say that $u \rightarrow v$ is a push edge: every time u shares a new event, the client processing the request adds it to the view of v. If u is in the pull set of v, we say that $u \rightarrow v$ is a pull edge: every time v requests for his event stream, the client processing the request pulls events from the view of u.

Definition 3 (Request schedule) *A* request schedule *is a pair (H, L) with $H \subseteq E$ and $L \subseteq E$. The sets H and L specify the edges that are served by a push and pull operation, respectively.*

It is important to note that all state-of-the-art push-all, pull-all, and hybrid schedules described in Section 1 are subclasses of the request schedule class defined above.

The goal of this work is minimizing the throughput costs of processing a given social networking workload. A workload is defined by a *production rate* $r_p(u)$ and a *consumption rate* $r_c(u)$ associated with each user u: the first indicates how frequently u shares new events, the second how frequently u requests new event streams. Given an edge $u \rightarrow v$, the cost incurred by a push operation is $r_p(u)$, and the cost incurred by a pull operation is $r_c(v)$.

The cost of the request schedule (H, L) is thus:

$$c(H, L) = \sum_{u \rightarrow v \in H} r_p(u) + \sum_{u \rightarrow v \in L} r_c(v).$$

This expression does not explicitly consider differences in the cost of push and pull operations, modeling situations where the messages generated by updates and queries are very small and have similar cost. In order to model scenarios where the cost of a pull operation is c times the cost of a push, independent of the metric we want to minimize (*e.g.*, number of messages, number of bytes transferred), it is sufficient to multiply all consumption rates by a factor c. Note that the cost of updating and querying a user's own view is not explicitly represented in the cost metric. It is also possible to associate edges with weighs modeling different costs: for example, in systems where the views of two users are located in the same data store node, both views can be updated and queried using only one request.

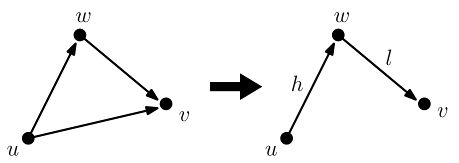

Figure 2. Hub optimization: by enforcing that the edge $u \rightarrow w$ is a pus$\underline{h}$ and the edge $w \rightarrow v$ is a pul$\underline{l}$, the edge $u \rightarrow v$ can be served with bounded-staleness guarantee, even if it is not served directly.

We are now ready to define the social dissemination problem that we address in this paper.

Problem 1 (DISSEMINATION) *Given a graph $G = (V, E)$, with production and consumption rates $r_p(u)$ and $r_c(u)$ for each node $u \in V$, find a request schedule (H, L) that guarantees bounded staleness and non-triviality, while minimizing the cost $c(H, L)$.*

In order to make bounded staleness possible, we assume that the underlying system has an upper bound Δ on the time it takes for an event to be pushed to or pulled from a view. Using the hub optimization described in the introduction and depicted in Figure 2 guarantees bounded staleness with $\Theta = 2\Delta$. In fact, this is the only optimization respecting bounded staleness and non-triviality.

Theorem 1 (Admissible request schedules) *Any request schedule (H, L) guaranteeing bounded staleness and non-triviality on a social graph $G = (V, E)$ is such that, for each edge $u \rightarrow v \in E$, it holds that $u \rightarrow v \in H$, or $u \rightarrow v \in L$, or there exists a vertex w such that $u \rightarrow w \in E$, $w \rightarrow v \in E$, $u \rightarrow w \in H$, and $w \rightarrow v \in L$.*

A consequence of this theorem is that when a user v pulls events from u, it is not necessary to copy these events into the view of v. In the example of Figure 1, events pulled by Leonardo from Gabriel's view do not need to be stored into Leonardo's view.

Even considering only this restricted solution space, Problem 1 is $\mathbf{NP}$-hard because it can be reduced to the SETCOVER problem.

Theorem 2 *The* DISSEMINATION *problem is* $\mathbf{NP}$*-hard.*

3. The NOSY heuristic

In this section, we introduce a greedy heuristic to solve the DISSEMINATION problem called NOSY. NOSY searches for *hub-graphs* that can reduce the cost of dissemination. A hub-graph $G(X, w, Y)$ centered on a vertex w is a generalization of the subgraph of Figure 2, depicted in Figure 3.

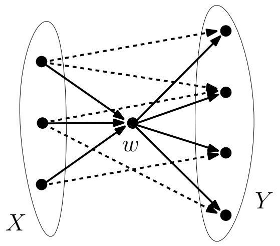

Figure 3. A hub-graph used by the NOSY heuristic.

It is a sub-graph of the social graph where the sets X and Y contain producers and consumers of w, respectively.

NOSY iteratively builds hub-graphs according to a pre-defined order, and considers only few pre-defined classes of hub-graphs centered at w. It maintains the set of edges, H, L, and C, which have currently been covered as push, pull, or cross-edges, respectively. Let $E(X, Y)$ denote the set of edges of the hub-graph between vertices of X and Y. For a hub-graph $G(X, w, Y)$, NOSY decides to cover the cross-edges in $E(X, Y)$ via the hub w only if using the hub w is cheaper than covering the cross-edges directly. We say in this case that G is *selected*.

Cost of selecting a hub-graph. The cost incurred by the baseline of [9] for covering an edge $u \rightarrow v$ is $c_*(x \rightarrow y) = \min\{r_p(x), r_c(y)\}$. We call $p(X, w, Y)$ the overall baseline cost of covering all edges of $G(X, w, Y)$ that have not been covered yet.

Selecting a hub-graph $G(X, w, Y)$ saves the costs of covering all cross-edges between vertices in X and Y, which is $c_*(x \rightarrow y)$ for every cross-edge $x \rightarrow y$. The cost of selecting a hub-graph $G(X, w, Y)$ is computed by considering the edges $E(X, \{w\})$ and $E(\{w\}, Y)$ that need to be scheduled as push or pull edges, respectively. The cost $c_X(e)$ of an edge $e = x \rightarrow w$ in $E(X, \{w\})$ is $r_p(x)$ if $e \in L \setminus H$, $r_p(x) - c_*(e)$ if $e \notin (H \cup L)$, or 0 if $e \in H$. In the first case, if the edge e is in $L \setminus H$, NOSY has already decided that the edge is served by a pull, but not by a push. Selecting G mandates that the edge must be served by a push too, hence incurring an additional cost of $r_p(x)$. In the second case, if the edge e is not in the set $H \cup L$, NOSY has not scheduled the edge yet. The additional cost of pushing over e depends on the cost of covering e directly, which is $c_*(e)$. Finally, if e is already served by a push, there is no additional cost. The cost $c_Y(w \rightarrow y)$ of an edge $e = w \rightarrow y$ in $E(\{w\}, Y)$ is specular.

The overall cost $c(X, w, Y)$ of a hub-graph $G(X, w, Y)$ is thus the sum of $c_X(x \rightarrow w)$ for every edge $x \in X$ and $c_Y(w \rightarrow y$ for every edge $y \in Y$. The selection criteria is that the gain of G must be higher than its cost, i.e., it must be $p(X, w, Y) - c(X, w, Y) > 0$.

Selecting hub-graphs. The order in which hub-graphs are built is illustrated in Algorithm 1. NOSY first builds hub-graphs $G(X, w, \{y\})$ for each edge $(w \rightarrow y)$, where X includes all nodes x having an non-covered cross-edge pointing to y. Selecting large X sets leads to a large number of covered cross-edges, and thus to potentially high gains.

Algorithm 1 NOSY

```
Input: Directed graph $G = (V, E)$;
Output: Dissemination schedule $(H, L)$;
 1: $H \leftarrow \emptyset$;                                          {Push edges}
 2: $L \leftarrow \emptyset$;                                          {Pull edges}
 3: $C \leftarrow \emptyset$;                           {Edges covered via some hub}
    {Select hub-graphs}
 4: for (each $y \in V$) do
 5:     for (each $w \in V$ s.t. $(w \rightarrow y) \in E \setminus C$) do
 6:         $X \leftarrow \{x \mid (x \rightarrow w) \in E \setminus C \wedge (x \rightarrow y) \in (E \setminus (C \cup H \cup L))\}$;
 7:         if $p(X, Y) - c(X, w, Y) > 0$ then
 8:             $H \leftarrow H \cup E(X, w)$;
 9:             $L \leftarrow L \cup \{w \rightarrow y\}$;
10:             $C \leftarrow C \cup E(X, y)$;
    {Extend selected hub-graphs}
11: for (each $y \in V$) do
12:     for (each $w \in V$ s.t. $(w \rightarrow y) \in E$) do
13:         $X_y \leftarrow \{x \mid (x \rightarrow w) \in H \wedge (x \rightarrow y) \in (E \setminus (C \cup H \cup L))\}$;
14:         if $p(X_y, \{y\}) - c_Y(w \rightarrow y) > 0$ then
15:             $L \leftarrow L \cup \{w \rightarrow y\}$;
16:             $C \leftarrow C \cup E(X_y, y)$;
    {Cover directly the edges that are still uncovered}
17: for (each $(u \rightarrow v) \in E \setminus (C \cup H \cup L)$) do
18:     if $h(u \rightarrow v) < l(u \rightarrow v)$ then
19:         $H \leftarrow H \cup \{(u \rightarrow v)\}$
20:     else
21:         $L \leftarrow L \cup \{(u \rightarrow v)\}$
22: return $(H, L)$
```

After selecting new hub-graphs, NOSY tries to enlarge the singleton sets Y. The rationale is that, for each selected hub-graph $G(X, w, Y)$, the costs of pushing over the edges in $E(X, \{w\})$ has already been paid. Therefore, given a node y pointed by w and not already in the set Y, all non-covered cross-edges in $E(X, \{y\}) \cap E$ can be covered via w by paying only the cost $r_c(y)$. If $G(X, w, Y)$ is selected, the view of w is required to store events from users $x \in X$, which are all producers of w by definition. This is consistent with non-triviality.

4. Experimental evaluation

In this section we evaluate NOSY using datasets obtained from real networks in a variety of different settings. We consider different workloads, characterized by their distribution of production and consumption rates. For every workload, we calculate a schedule using NOSY and compare it with the schedule generated by the baseline of Silberstein *et al.* [9]. We define the *gain* of NOSY as the relative cost reduction compared to the baseline, that is, the reduction in the overall number of query and update requests to user views. This notion of gain translates to a lower load on the data stores. Lower load means that the same amount of data store servers is able to support a heavier workload using NOSY instead of the baseline. The expression of the gain given a workload W is $(c_{\mathrm{B}}(W)/c_{\mathrm{N}}(W)) - 1$ where $c_{\mathrm{B}}(W)$ is the baseline cost of [9] and c_{N} is the cost of NOSY. Note that the gain is negative if the baseline outperforms NOSY.

The experiments highlight that NOSY effectively exploits the high clustering coefficient of social graphs under a wide range of parameters and distributions.

Input data. We obtain our datasets by sampling two real-world social graphs: the `flickr` social graph as of April 2008, and the `twitter` social graph as of August 2009, which was made available by Cha et al. [2]. We employ a sampling method based on random walks, where we restrict the number of visited neighbors for each vertex. The sampled graphs for `flickr` have on average 4K nodes and 112K edges, while `twitter` graphs have on average 25K nodes and 158K edges. Each experiment is conducted on 10 samples and we present the average. In order to study the impact of graph connectivity, we add a post-sampling step where only a fraction s of sampled edges is kept. We set $s = 1$ to be the "reference" value, representing the full sample of the social graph.

We consider different models to generate workloads. First, we vary the average ratio r between the consumption and the production rates of each node. Following Silberstein *et al.* we use $r = 5$ as "reference" value. The production and consumption rate of each node are drawn from a number of different distributions. First, we use the Zipfian model proposed by Silberstein *et al.*, which is *graph-agnostic* in the sense that the sample rates do not take into account the graph structure. We then use two other models, in which the production and consumption rates of a node are drawn in accordance to its out-degree and in-degree, respectively. Thus, nodes with many followers tend to have a higher production rate, and nodes following many other nodes tend to have a higher consumption rate, as observed by Huberman et al. [5]. For this class of *graph-aware* models we use two policies: (i) setting the rates proportional to the node degrees, and (ii) proportional to the logarithm of the node degrees. Details of the sampling and generation processes are omitted due to lack of space.

Effect of the clustering coefficient. NOSY exploits *clustering coefficient* by using hubs to relay information. Thus one should expect little gains when there is low connectivity and when edges are isolated, rather than parts of triangles. In fact, in such a setting *no* algorithm will improve over the baseline. To demonstrate the clustering effects, Table 1 reports the gains, NOSY (N) versus the sampling parameter s and the clustering coefficient C of the graph.[1] As expected,

[1] Even though the clustering coefficient is defined for undirected graphs, here we use a notion for directed graphs, where we account only triangles as in Figure 2.

	flickr			twitter		
s	C	B+	N	C	B+	N
0.1	0.03	0.003	0.01	0.026	0.007	0.02
0.5	0.12	0.028	0.15	0.096	0.043	0.26
1.0	0.17	0.044	0.43	0.112	0.059	0.62

Table 1. Gains with different graph connectivity.

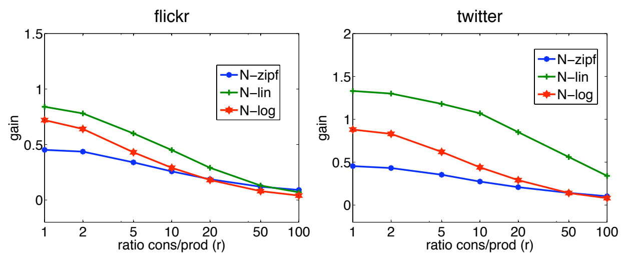

Figure 4. Gain of NOSY as a function of ratio of consumption/production rates (r).

the higher the connectivity the larger is the potential gain (as indicated by the lower bound) and the larger is the achievable gain of NOSY.

In order to show that exploiting high clustering coefficient requires non-trivial algorithms, we also compare with an extension of the baseline of [9], called B+. B+ first schedules all edges as the baseline, then locks all push edges, and then checks if it can prune pull edges that are not locked by other "hubbing" triangles by applying the elementary optimization of Figure 2. Table 1 shows that the gain of B+ also grows with the clustering coefficient but remains negligible.

Efficiency under varied workloads. Figure 4 shows the gain of NOSY for different distributions and different values of the ratio of consumption vs. production rates. NOSY performs well, reducing costs by up to 0.8 times for `flickr` and 1.4 times for `twitter`. For the reference parameters, NOSY has a gain of about 0.45 for `flickr` and 0.62 for `twitter`. The gain is sustained even at high values of r.

In order to understand the trends, we set r as high as 100, which is 20 times the reference value. Intuitively, if users consume information every second while producing information only once a day then a baseline that uses push edges to spread the (rare) events through the network will be nearly optimal. The experiments confirm this intuition.

Colocation. Our algorithms can be applied to arbitrary location functions. With colocation, push and pull operations among views that are mapped to the same location have no cost. In our experiment we use a hash function as done in standard partitioning algorithms such as consistent hashing [6]. The parameter n represents the number of locations, or data stores.

For fairness, we consider a trivial optimization of the baseline of [9] that exploits colocation. If a node pulls from a

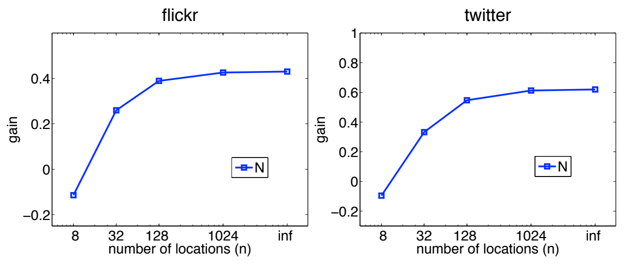

Figure 5. Gain of NOSY as a function of the number of locations (n).

data store, it can pull from all its neighbors in that data store at once. For each consumer v and data store l, we define $X_v(l)$ to be the set of producers of v located in l. In the input graph, we replace all producers in $X_v(l)$ with a unique abstract producer that has the sum of the production rates of the nodes in $X_v(l)$ as production rate.

Our results are shown in Figure 5. As one may expect, for a small number of locations (data stores) the gain of our algorithms is negligible. However, the gain converges very fast to the one we have without colocation. The reason is that as the number of data stores increases, the chances of having neighbors on the same location decreases.

5. Related Work

Most closely related to our paper are the works of Pujol et al. [8] and Silberstein et al. [9]. Pujol et al. describe SPAR, a new storage layer for social networking systems which uses a variant of the push-all strategy. SPAR reduces space consumption by co-locating the views of neighboring nodes in the same physical machine, and transferring views online across machines when the social graph is modified. This makes SPAR is significantly more complex than the storage layers used in production systems. Schedules produced by NOSY can be used at the client-side of standard, vanilla data stores, such as memcached.

Similar to the MIN-COST problem of Silberstein et al. [9], our problem takes as input the consumption and production rates of users. We generalize their problem definition as a graph propagation problem, encompassing a strictly larger set of propagation policies. This generalization enables taking advantage of the high clustering coefficient of social graphs and leads to substantial gains over the propagation policies of [9], the baseline of our evaluation.

User views are simple materialized views [4] for social networking applications. From a routing viewpoint, a social networking system can also be seen as a very simple publish-subscribe systems [3], although we consider subscribers (consumers views) as internal to our system and we use their storage to serve other consumers.

6. Conclusion and future work

Generating on-line social event streams with high throughput is an important problem in reducing the cost of social networking systems. We proposed social piggybacking, a technique that leverages the high clustering coefficient of social graphs to reduce rate of requests sent to back-end data stores. We formalized the DISSEMINATION problem: finding request schedules that guarantee bounded staleness and non-triviality. We then proposed a hub-based optimization that clearly delimits the solution space and leverages the high clustering coefficient of social networks. Since the problem is **NP**-hard, we introduced a heuristic, called NOSY, to solve it. Initial evaluation using samples of the Twitter and Flickr graphs shows that existing state-of-the-art request schedules generate up to 2.4 times the number of view requests induced by NOSY.

We are currently working on identifying an approximation algorithm to solve the DISSEMINATION problem. We also plan to scale the heuristic to full social graphs, if necessary by implementing a MapReduce variant of NOSY. Finally, we want to evaluate the throughput gain of our approach using a real implementation.

Acknowledgments

This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Community, and by the INNCORPORA - Torres Quevedo Program from the Spanish Ministry of Science and Innovation, co-funded by the European Social Fund.

References

[1] Tumblr architecture - 15 billion page views a month and harder to scale than twitter. http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html.

[2] M. Cha, H. Haddadi, et al. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, 2010.

[3] P. T. Eugster, P. A. Felber, et al. The many faces of publish/subscribe. *CSUR*, 35, 2003.

[4] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10, 2001.

[5] B. A. Huberman, D. M. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. *First Monday*, 14(1), 2009.

[6] D. Karger, E. Lehman, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

[7] M. E. J. Newman. The Structure and Function of Complex Networks. *SIREV*, 45(2), 2003.

[8] J. Pujol, V. Erramilli, et al. The little engine(s) that could: Scaling online social networks. *CCR*, 40(4), 2010.

[9] A. Silberstein, J. Terrace, et al. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, 2010.