Feed Following: The Big Data Challenge in Social Applications

Adam Silberstein, Ashwin Machanavajjhala, Raghu Ramakrishnan

Yahoo! Research
Santa Clara, CA, USA
{silberst,mvnak,ramakris}@yahoo-inc.com

ABSTRACT

Internet users spend billions of minutes per month on sites like Facebook and Twitter. These sites support *feed following*, where users "follow" activity streams associated with other users and entities. Followers get personalized feeds that blend streams produced by those followed. The emphasis on recency and relevance, and the highly variable fan-out of the follows graph, make this feature difficult to implement at the scale seen in major social networks.

In this paper, we place feed following in the context of existing research areas and highlight the novel data management challenges that it poses, with the goal of stimulating research in this new direction. We discuss solutions based on pub/sub, caching, and materialized views, and argue that none of these existing approaches fully exploit the unique characteristics of feed following. The number of distinct queries and the query rate per second that a feed following system must support are huge, but queries have simple structure and overlap. The system must handle high throughput input streams, but results are heavily biased toward recent events. The number of users is large, but they exhibit diurnal behavior, and we can dynamically modify the system to optimize for currently active users. These characteristics offer many opportunities for optimization, and the potential gains are substantial.

Categories and Subject Descriptors

H.2.4 [**Systems**]: distributed databases

General Terms

Performance

1. INTRODUCTION

Facebook and Twitter let users *follow* a larger number of other users/entities, and then retrieve personalized *feeds* consisting of *events* produced by (or associated with the activities of) these users/entities. Facebook's main offering, on its front page, is called the *News Feed*. Other sites, such as Yahoo!, offer social content both through a dedicated feed aggregation product, Yahoo! Pulse, and integrated within other products such as Mail and Messenger.

These sites all face an extremely challenging data management problem. Each site manages a large user base, which we logically divide into event producers and event consumers, and has a *connection network* that encodes *follows* relationships between *consumers* and *producers*. In the Facebook "friend" model, an edge implies the consumer and producer are friends, and the relationship is symmetrical. On Twitter, a producer might be a celebrity the consumer is following, and the relationship is asymmetric. In general, producers can be people, news feeds associated with real-world topics, etc., and events can be status updates, news articles, and so on. A consumer may follow many event producers, and a producer may have multiple followers; the distributions are highly skewed.

Follows networks are massive in scale. Facebook, for example, boasts (as of May 2011) 500 million unique users, each with an average of 130 friends [1]. Yahoo! has over 650 million unique users, and in addition to the large numbers of connections a user has via Mail and Instant Messenger, Yahoo! also allows a user to import their Facebook connections, and to follow this extended collection of connections. Each site then also faces extremely high event production rates across its producers, and extremely high feed query rates across its consumers.

The *feed query* is conceptually simple, and by far the dominant query that social networking sites must support: when a consumer C visits the site we (conceptually) gather all events produced by each producer that C follows, and present a subset of the most personally relevant events, with a premium on recency. Feed delivery is by no means a simple problem due to the combination of dense connection networks, low-latency requirements (tens of milliseconds), and extremely high event and query rates. The high skew in producer fan-ins, consumer fan-outs, and consumer arrival rates further complicates the optimization problem, since these factors affect the choice of caching vs. re-computation strategies. (The most popular producer on Twitter as of May 2011 is the singer Lady Gaga, with over 10 million followers [4]. The average Facebook consumer follows 130 producers [1], and some consumers follow over 1000 producers; others follow very few. Some consumers visit the site constantly, triggering requests for their feed query each time, while others visit once a week or less often.)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DBSocial '11, June 12, Athens, Greece
Copyright 2011 ACM 978-1-4503-0650-8 ...$10.00.

There are some important characteristics of feed queries that present optimization opportunities. Given that many events (even from close social connections) are inherently uninteresting to a typical user, a good feed following solution must select judiciously from the incoming streams of followed producers. Even in variants of feed following where all followed events are presumed interesting to the consumer (e.g., Twitter), a consumer who visits infrequently will only see recent events; older events are potentially discarded in between visits. Facebook's *multifeed* system, built on memcache, caches only the latest events per producer [14]. This has a fortunate side-effect that users don't expect to see *all* events for the followed producers, and we have the ability to drop some events if we fall behind in our delivery. While we must return feed query results with low latency, the results need not include the latest events; some (bounded) staleness is acceptable. Another degree of slack is that we can delay delivering events produced by a given producer (typically, heavily followed and therefore causing processing delays). There are also some stable usage patterns. Consumers that query infrequently are not likely to suddenly become frequent, and vice-versa; the same holds true for producers. Moreover, we may be able to predict at what times producers and consumers are active, and when they are idle.

How do we build a large-scale, cost-effective system to support feed queries, leveraging the characteristics noted above? This question gives rise to a number of significant research problems. In the rest of this paper, we first formalize the feed follow problem (Section 2). We describe three standard approaches–publish/subscribe systems, caching and materialized views–for describing and solving the feed follow problem, and argue that none of these existing approaches fully exploit the unique characteristics of feed following (Sections 3 and 4). We conclude in Section 5.

2. EXAMPLE AND MODEL

Figure 1 illustrates an example instance of the feed following problem. The *Connections* table is a directed graph $G(V, F)$, where vertices $v_i \in V$ are either consumers or producer, and edges $(v_i, v_j) \in F$ represents the fact that consumer v_i follows producer p_j. The *Connections* table is stored on disk for persistence (in a PNUTS table [7] at Yahoo!), and perhaps also cached in memory (as in Facebook [3]). Each producer v_j generates an event stream PE_j (e.g., articles, status updates, etc.) which is stored on disk in an *Events* table.

Each visit by a consumer v_i results in the corresponding *feed query* for v_i. The query result blends the recent event streams of producers that v_i follows, and a simple version that returns the most recent event is shown below:

Feed Query. $$\sigma_{(k \text{ most recent events})} \bigcup_{\forall v_j : (v_i, v_j) \in F} PE_j$$

Feed following imposes performance requirements on latency and staleness, since users must be shown results as soon as they (implicitly) request their feed:

- **Latency SLA**: A percentage p_l of all queries must return to the consumer in less than or equal to time t_l.
- **Freshness SLA**: A percentage p_f of all queries must return a feed that was up-to-date within the last t_f.

Our formalization reflects the practical compromise that we tolerate a small fraction of queries seeing longer delays or stale results. In our application, we might set $p_l = 95\%$, $t_l = 100ms$, $p_f = 90\%$, and $t_f = 1hr$.

Our problem now is to design a system for answering feed queries while satisfying both the Latency and Freshness SLAs with minimal cost, where cost of a solution is determined by one or more the following bottleneck resources – total memory footprint (needed to store and answer queries), total computation time (for constructing personalized feeds), or the total network traffic in a distributed system (for routing events from producers to consumers).

There are a number of ways we can generalize the problem. First, producers can be any topic of interest, e.g., the music band U2. The associated event stream can come from news or user activity relevant to U2. Second, the *Connections* table can be unidirectional, as in Twitter: Lady Gaga has millions of followers, but does not follow them all. Third, a feed following application can let consumers explicitly choose which producers to follow, or infer followers from sources such as address books or activity logs. Applications also vary in how much control producers have over who can follow different kinds of activity. For example, Flickr lets producers limit consumers of posted photos to be friends or family [2]. A fourth variation is that the Connections graph can be multi-level, e.g., we can enable users to create a page around U2 by consuming various feeds (news, Youtube uploads of U2 videos, Flickr uploads of U2 photos), and other users could then benefit by consuming this aggregation of U2 events. Fifth, the feed query itself need not be limited to returning only the k latest events across all producers. The application may in principle consider a larger candidate window of recent events and choose the k most "interesting" events (using an application-specific criterion) to return in the result. For example, Facebook offers a *Most Recent* feed with the latest events, but also a *News Feed* that contains an alternate set. Feeding Frenzy [20] proposes an alternative that ensures diversity among the producers represented, preventing one extremely active producer from dominating the feed. In general, the notion of interesting can blend considerations such as click-through rates of events, match to the user's interests, etc. Regardless of the specific notion of interestingness, systems that don't promise to show *all* recent events can afford to omit some events from feeds for performance without compromising the user experience.

3. EXISTING APPROACHES

This section describes three well known solutions that seem natural for describing and solving the feed following problem. We argue that none of these existing approaches fully exploit the unique characteristics of feed following.

3.1 Pub/Sub

Pub/sub systems [19, 6, 15, 8] have been heavily studied in the database and systems communities. The models assumes a large number of producers (aka publishers) and consumers (subscribers), and a set of follows edges (subscriptions). As in feed following, one of the key questions is to determine, for each new event, what subscribers should receive the event.

While it seems natural to borrow implementation ideas, the feed following problem crucially differs from the pub/-sub problem in delivery semantics. Each subscription (or consumer) in a pub/sub system is modeled as a continuous query that must be answered *every time a new event en-*

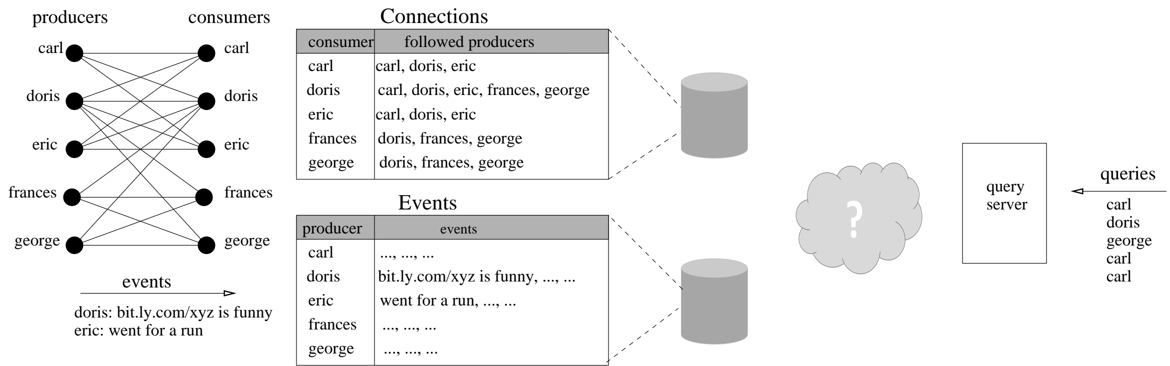

Figure 1: Example Feeding Following Application and System.

ters the system; i.e., the consumer is always listening to the stream of events. In a typical pub/sub system like Scribe [19], any event is disseminated *immediately* to all consumers interested via a peer-to-peer network, where the event is either consumed or buffered for future consumption. In feed following, on the other hand, even the most active consumers are almost always disconnected and not listening to the event stream (on Facebook, only half of active users get their feed at least once per day [1]). Buffering events like in Scribe is not right for a feed following application either, because when consumers do arrive, they are only interested in the latest events. System-wide, an overwhelming majority of events need not be delivered because they are considered too old to be interesting when the consumer (finally) visits. Suppose in our example from Section 2, Carl is not only a frequent feed consumer, but is also a frequent producer. Suppose Carl produces 100 events for each latest-10 feed query by Doris. For each Doris query, even if none of her other producers appear in the result, we will have propagated at least 90 events from Carl that she will not see.

The ability to query for older events is important in case a consumer wants to go backward in time to see events they have missed. This query is rarer, however, and we want to avoid eagerly doing any work for it. Moreover, because feed following is typically web-based, consumers pose this query by scrolling downward on a webpage or clicking a "back" button. Both of these actions give us extra time to retrieve older events without the consumer perceiving a delay.

3.2 Caching

Caching keeps important events on hand (e.g., in memory), and relegates others to more archival storage. Caching has several attractive properties. First is the flexibility to keep the state we want and decide how to organize it. Hence, rather than buffering all events as in pub/sub, we can cache, say, just the per-consumer query result objects. Second, caching allows policies for controlling what state to keep and drop. Least-recently used (LRU) eviction and variations prefer to remove objects that are least recently read, and will be effective for deciding which consumer result objects we cache, especially if most consumers have diurnal patterns (e.g., active during the day and inactive at night). Third, cached events grow stale, and to meet the freshness SLA we must replace them with newer events. Caching strategies address this issue with time-to-live (TTL). Given a freshness constraint t_f of 1 hour, we set TTL for each cached per-consumer query result to 1 hour past creation time, ensuring entries are discarded before they become stale.

Caching strategies, like the baseline described above (caching per-consumer query result objects with LRU eviction and TTLs), work since most workloads satisfy the 80/20 rule: 80% of all queries can be satisfied by caching results for the 20% most popular unique queries [13]. Feed following workloads too satisfy this rule. We monitored 20 days of consumer feed requests to a Yahoo! social platform and randomly sampled 1 million consumers. Of requests made by the consumers, 74% of requests came from the 20% most frequent. The baseline cache strategy, then, looks promising.

There is a key difference, however, between the 80/20 property in typical web workloads and the 80/20 property that we observe in feed following. In typical web workloads, each consumer poses multiple unique queries; some of these queries are globally popular (e.g., in the top 20%), and others are globally obscure (e.g., in the bottom 80%). Caching the most popular 20% ensures each consumer gets fast results for their popular queries and slow results for their obscure ones. In contrast, in feed following, each consumer poses only a single query over and over again. If that query is in the popular set, as is Carl's feed, the consumer *always* gets fast results. If that query is not in the popular set, the consumer *never* gets fast results. This inequity is significant and detrimental to growing a user base. Moreover, even the frequent consumers might not be happy to always see the same cached results, but instead want to see different recent results every time they query the service.

A second crucial difference between caching and feed following is that while cache maintenance tends to focus on validation/invalidation [17], changes to the cached objects are incremental in feed following. For instance, the change to a query result due to a single new or modified event does not require invalidating the whole cache. Yet another example against cache-invalidation arises due to access control changes. Consider a system like Flickr where connections maybe tagged as "friends" or "family"; an event marked "friends" is seen only by connections tagged as "friends". Suppose Carl and Doris are connected as friends, but Carl and Eric are not. Carl creates a photo event, and we cache a pointer to it in feeds for Doris and Eric. Now Carl edits the event to be "friends" and in effect wants it removed from Eric's feed immediately. Validation/invalidation force us to discard the Eric cache entry, instead of only (incrementally) deleting the photo event.

To sum up, while caching strategies allow us to speed up frequent queries, they fail to capture two properties specific to feed following – each consumer asks the same query, and changes to query results are incremental.

3.3 Materialized Views

An alternative to caching query results is *view materialization*: where arbitrary views over the base tables are maintained with the goals of reducing overall processing costs, meeting the latency SLA, and/or meeting a memory footprint constraint. In the context of feed following applications, these views could be consumer-pivoted query results, producer-pivoted views capturing the recent events by single producers, or recent events generated by groups of producers. There is a great deal of view materialization literature on choosing the right set of views in the OLAP [10] and SQL [5] settings, answering queries given complex views [11], incrementally maintaining views [9], and making views self-maintainable (views are maintained without needing to consult the base tables) [21]. Feed following is mainly concerned with maintaining lists of most recent events, and so incremental and self maintenance come easily.

Despite these advantages, solutions for materialized views do not provide a complete solution for the feed following problem. One shortcoming of traditional materialized views is that the solution tends to be static, in that we analyze the query workload, choose a set of views, and maintain them. This is reasonable when the workload is static, and the number of views is small, but heavyweight and expensive to build from scratch. In feed following, however, the query workload changes quickly, the number of possible views is huge, and each is a small list of events. Each view may be useful only for a small number of queries. Doris's consumer-pivoted view is only useful for Doris feed queries. Doris only queries in the morning, so her view is useless the rest of the day, but consumes memory and requires maintenance. To verify consumers like Doris exist, we again looked at our sample of 1 million Yahoo! consumers from a 20 day period of queries, and then trimmed that sample to only those users making 20 or more queries (an average of one query a day or more). We measured for each user the maximum gap between accesses in the trace. 1% of consumers have a maximum gap of less than 1 hour, 5% less than 5 hours, 10% less than 10 hours, and more than 75% of consumers have a gap of greater than 1 day. It is easy to see that most views for individual consumers are useless most of the time. There are big gains to be had if we can more dynamically swap views in and out of memory, as we see in caching; this strategy is not part of the views literature.

A second shortcoming is that traditional view maintenance seeks to support the semantics that answers are equivalent to re-computing the query. As discussed, feed following offers more slack for serving stale results and omitting some events, and we must leverage this in a principled way.

4. NEW CHALLENGES

From the previous section, it is easy to see that a large-scale feed follows system must borrow ideas from caching and view maintenance, even though some of these techniques seem to directly contradict one another (e.g., staleness and incremental maintenance). Suppose we cache events for a consumer who is very active for four hours in the morning, but idle otherwise. We want to build incrementally maintainable caches to help answer this consumer's query during the active hours (i.e., incremental maintenance as with traditional views), but evict the views in the afternoon (analogous to evicting infrequently used objects from a cache). We describe in this section a selection of new research challenges opened by the conflicting requirements of feed following. We highlight three key challenges: (a) *view selection*, the problem of choosing which views to materialize (consumer-pivoted, producer-pivoted or other complex views), (b) *view scheduling*, determining when to build a view in memory, when to incrementally maintain and when to expire the view to free up memory, and (c) *view placement*, finding the optimal placements of these views in a distributed setting to minimize communication costs and response times to geographically diverse requests. While we introduce them as three standalone problems, they are naturally interlinked.

4.1 View Selection

The view selection problem has been extensively studied for relational and OLAP data. The question is familiar: what views do we materialize and maintain in order to minimize some cost function (based on throughput, latency, memory footprint, etc.). Feeding Frenzy [20] (FF) represents one foray into this space, using two types of views: *producer-pivoted views*, which maintain the latest k events from every producer, and *consumer-pivoted views*, which incrementally maintain the result of a consumer feed. FF selects either a producer-pivoted or a consumer-pivoted view for every edge in the connection network such that the sum of event and query processing costs is minimized. Facebook only maintains producer pivoted views. Ultimately, the choice of views greatly depends on the resource for which we are optimizing. In this section, we show that additional view types can help when optimizing for memory footprint.

Suppose in our Figure 1 example each user queries for the k latest events from their connections. We might materialize consumer-pivoted views for all the users for quick response times. First, note that both Carl and Eric follow the same set of producers; hence, their consumer pivoted views are duplicated, increasing both the memory footprint and update load; a better solution is to serve both Carl and Eric from the Carl view. Second, note that Doris follows the producer set {Carl, Doris, Eric, Frances, George}, Carl follows {Carl, Doris, Eric} and Frances follows {Doris, Frances, George}. Hence, any event needed to construct Doris' feed either appears in Carl's feed or Frances' feed. If we have consumer-pivoted views for both Carl and Frances, we can *compute* the feed for Doris from Carl and Frances' views, thereby eliminating the need to maintain Doris' view.

Formally, we say that a feed f can be computed with a set of other feeds $f_1, f_2, \ldots, f_i$, if there is an efficient algorithm F such that $f = F(f_1, f_2, \ldots, f_i)$. We define a set of cached feeds $\mathcal{F}$ to be *minimal* if (a) every consumer feed can be computed correctly using one or more feeds in $\mathcal{F}$, and (b) no feed in $\mathcal{F}$ can be computed using one or more other feeds in $\mathcal{F}$. Ultimately, we also must limit the number of views from which a feed is formed, or else we lose too much of the advantage gained from consumer-pivoted views. We leave that formalization and solution as future work.

To see how many consumer views can be removed in the same manner as Eric and Doris, we ran an experiment on a connection graph from Twitter to compare the number of consumers against the size of the minimal set of cached feeds. The graph, borrowed from [20], contains 400,000 consumers and 79,842 producers. We began with the set of all feeds, and removed every feed that can be answered as the union of other feeds. We found the minimal set contained

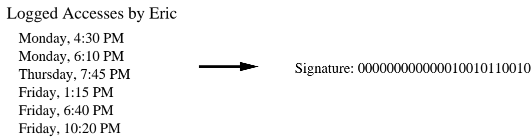

Figure 2: Eric's signature, evening-heavy.

only 70,926 feeds. Assuming each cache entry has equal size k, caching all consumers requires 5.6x the memory of the minimal set, and even caching all producers (like in Facebook) increases the memory footprint 12% over the minimal set. FF builds producer-pivoted cache entries for all producers in this Twitter graph, and additional consumer-pivoted entries that help lower overall processing cost. The number of consumer-pivoted entries is 13,372, for a total of 93,214 views, 31% larger than the minimal set.

We have proposed one optimization for reducing footprint, but already see that it competes with optimizations whose goal is to group together all events needed to produce a single feed. The general view selection problem must consider all system resources and performance constraints.

4.2 View Scheduling

While the view selection problem deals with choosing the set of views that minimize footprint and/or communication costs, as we pointed out in Section 3.3, in a feed following system all the views that are selected need not be built or incrementally updated at all times. Suppose we maintain a consumer-pivoted view for Doris, who only queries in the morning. At 11:00 AM, after she has been idle for 30 minutes, we might stop incrementally updating her feed, letting it grow stale; at 11:30 AM, we might evict her view. One of the key traits of feed following is that most queries and most events are unimportant most of the time. Hence, instead of maintaining a static set of views, we should maintain a set of views whose membership we change dynamically over time. We call this open problem *view scheduling* – determining when to build a view in memory, when to incrementally maintain it and when to expire the view to free up memory.

Let us consider the problem of determining when to bring an up-to-date view (say a consumer-pivoted view) into memory. One option is to build the view on-demand. That may be acceptable in web caching, where we can sacrifice one consumer having a slow response time, if it allows many subsequent consumers to have fast response times. However, in feed following, views are tailored to one or a few consumers. Therefore, when a consumer queries after some idle time, they will either get a stale result or a slow result a significant number of times; since our SLAs rarely let us serve stale or slow results, this solution does not work.

One strategy for avoiding the above problem is to predict when queries will arrive, and build/refresh views in advance. This strategy is feasible for two reasons. First, event production and consumer queries follow patterns that can be detected. For instance, there is a clear diurnal pattern in consumer queries. Second, feed following views are small but numerous, and contain a few recent events that are useful for a small number of users. Therefore, unlike in traditional view maintenance, we can tear down and rebuild views from scratch without significant cost. Further, the views we need to re-build are based on recent events, thus reducing the cost of re-building by caching recent events in memory.

To assess the feasibility of predictive view scheduling, let us consider a simple scheduling strategy: We take a query trace as input and generate consumer *signatures*; a signature encodes the time intervals in a day where we have seen the consumer make feed queries. Figure 2 shows a sample signature for Eric. The signature uses a time interval of 1 hour, with 24 total bits; a bit is set to 1 if Eric requested his feed in the corresponding hour. We hypothesize that if consumers have regular behavior, we can use the signatures to predict future queries.

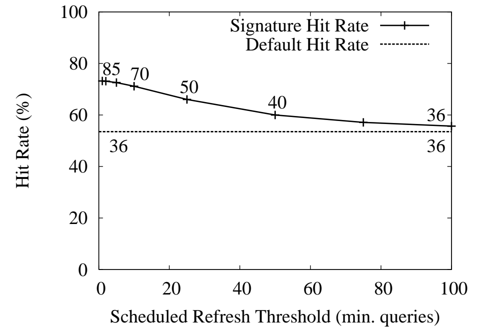

Figure 3: Consumer-driven refreshing versus consumer and signature-driven refreshing.

We used a query trace from Yahoo! spanning a period of 10 days to build signatures; a user's signature has its ith bit set to 1 if the user requested the feed during the ith hour on one of the days. On the subsequent days, 30 minutes prior to the ith hour, our scheduler starts rebuilding the consumer-pivoted views of all the users with the ith bit set to 1; to avoid a burst of activity, we spread that refreshing load over 60 minutes. If a consumer's feed view is less than 1 hour old at query time, the feed is fresh and accrues a "hit"; if the feed is stale we accrue a "miss." When a consumer gets a miss, we refresh the feed view.

We evaluated the effectiveness of this algorithm on a Yahoo! query trace from a subsequent 10 day period. For simplicity we assume unlimited memory footprint and hence we can always serve a stale feed. Also, we consider a freshness constraint of 1 hour. We compare the miss rate when refreshes are purely consumer-driven versus one where we additionally drive background refreshes from the trained signatures. For the signature strategy, we also experiment with varying the number of queries a consumer must do in training before we start building signatures for them (consumers without a signature have their views rebuilt on demand).

Figure 3 presents the results comparing the two strategies. The consumer-driven strategy gives a hit rate of about 54%; this is a function of the consumer query pattern and is completely out of our control. If 54% does not meet our freshness constraint (90% in Section 2), there is nothing we can do about it. On the other hand, as we lower the threshold for building consumer signatures, we improve the hit rate from 54% to 73%. The background refreshes are not free. We have annotated points in Figure 3 with the maximum query rate (queries/second) the policy imposes against the event store. The higher the rate, the higher the provisioning cost of the event store. We leave as open challenges the problem of how to best drive background refreshes, while limiting the read rate to the underlying event store and the write rate to the view set, the problem of incorporating incremental maintenance strategies, and the problem of improving the hit rate by better predicting user activity using data mining tools, such as calendric association rules [18].

4.3 View Placement

A third important question is *view placement*. A set of views that are optimally selected and scheduled may still incur high costs if the views are not carefully distributed. To understand this challenge, suppose we materialize producer-pivoted views for Carl, Doris, Eric, etc. and store them in a standard distributed memcache instance. Each producer will be hashed to a random server, requiring us to communicate with many servers in order to generate a user feed. Suppose we materialize consumer-pivoted views; again, these views are randomly distributed across servers, requiring us to duplicating events across the views on these servers. Both cases may result in significant communication overhead.

Existing work such as [12] shows that social networks tend to consist of communities of users that are highly connected among themselves, but loosely connected to other clusters. Based on this observation, we could organize views such that users that form a cluster have their events co-located on a server. In the producer-pivoted view case, we can acquire all events at a single or a few servers. For the standard latest k feed query, instead of collecting the latest k from each producer and paying the bandwidth cost to collect them at a central point, we can push that aggregation down to single storage server and only pay a bandwidth cost for k events. In the consumer-pivoted view case, recognize that there would be significant overlap of events in consumer views in the same cluster, and a strategy like maintaining pointers to single event store on the server might limit the duplication of events. There has already been some work on exploiting clusters to place event caches. *Little Engines* [16] ensures that all events needed to generate a feed can be found on one server. We believe the question of how to best trade-off the increased cost of ensuring a good clustering and the decreased cost from that clustering is still wide open. We conclude this discussion by noting that despite the community structure and clustering of users, some producers are followed by millions of users (like celebrities and news feeds), and clustering solutions may not apply for such high fanout producers. A combination of clever replication and clustering algorithms are required to handle the conflicting requirements of communities and celebrities.

4.4 Other Challenges

We have presented three fundamental challenges in building feed following systems. While these lay important groundwork, they are by no means the only open problems in feed following. Section 2 lists a number of ways feed following is more general than in our example application. Since feeds often present the most "interesting" events as decided by the application, they are very tolerant to missing events. How do we formalize this tolerance? Is it acceptable to miss an entire producer in a feed, miss the same events or producers repeatedly over multiple queries, are some producers more important than others, etc? A related open problem is that of *not* missing access level changes.

The Yahoo! social platform lets consumers link their accounts from 3rd party social applications like Facebook. When the event store is a 3rd party, we may not have the same level of access as when we own the event store, which may limit view strategies (e.g. if the 3rd party does not notify us of new events, we cannot do incremental maintenance). Another open issue prevalent in feed aggregators like Yahoo! is how to handle multi-level connection graphs (e.g. a user-generated page on U2), and how do they connect to view selection and view placement? These are just a taste of the problems that arise in feed following.

5. CONCLUSION

Feed following has become a ubiquitous application, supported by sites with huge user bases such as Facebook, Twitter, and Yahoo!. Building a cost-effective feed following system is a tremendous data management challenge. We have presented some of the unique characteristics of feed following that must be leveraged, such as its focus on recency, the skew in producer and consumer frequencies, and tolerance for missing events. We placed feed following in the context of related problems such as caching and view maintenance, and identified several open problems: what views should we build, when should they be built and dropped, and where should we place them in a distributed system. These challenges provide a foundation for designing feed following systems, but by no means cover all issues. As feed following systems proliferate in number and size, with more events, more queries, and denser connection networks, solutions to these and future challenges will be crucial.

6. REFERENCES

[1] Facebook Statistics. http://www.facebook.com/press/info.php?statistics.

[2] Flickr FAQ. http://www.flickr.com/help/privacy.

[3] Scaling memcached at Facebook. http://www.facebook.com/notes.php?id=9445547199.

[4] Twitter Counter. http://www.twittercounter.com.

[5] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *VLDB*, 2000.

[6] L. Brenna et al. Cayuga: a high-performance event processing engine. In *SIGMOD*, 2007.

[7] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[8] Y. Diao et al. Yfilter: Efficient and scalable filtering of xml documents. In *ICDE*, 2002.

[9] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[10] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[11] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[12] J. Leskovec et al. Statistical properties of community structure in large social and information networks. In *WWW*, 2008.

[13] A. Mahanti et al. Traffic analysis of a web proxy caching hierarchy. *IEEE Network*, 14:16–23, 2000.

[14] Mike Schroepfer. The Infrastructure of Facebook, 2009.

[15] J. Pereira et al. Publish/subscribe on the web at extreme speed. In *VLDB*, 2000.

[16] J. Pujol et al. The little engine(s) that could: Scaling online social networks. In *SIGCOMM*, 2010.

[17] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*, chapter Cache consistency. Addison-Wesley, 2002.

[18] S. Ramaswamy et al. On the discovery of interesting patterns in association rules. In *VLDB*, 1998.

[19] A. Rowstron et al. Scribe: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[20] A. Silberstein et al. Feeding frenzy: Selectively materializing users' event feeds. In *SIGMOD*, 2010.

[21] J. Yang and J. Widom. Temporal view self-maintenance. In *EDBT*, 2000.