Feeding Frenzy: Selectively Materializing Users' Event Feeds

Adam Silberstein[1], Jeff Terrace[2], Brian F. Cooper[1], Raghu Ramakrishnan[1]

[1]Yahoo! Research
Santa Clara, CA, USA
{silberst,cooperb,ramakris}@yahoo-inc.com

[2]Princeton University
Princeton, NJ, USA
jterrace@cs.princeton.edu

ABSTRACT

Near real-time event streams are becoming a key feature of many popular web applications. Many web sites allow users to create a personalized *feed* by selecting one or more event streams they wish to *follow*. Examples include Twitter and Facebook, which allow a user to follow other users' activity, and iGoogle and My Yahoo, which allow users to follow selected RSS streams. How can we efficiently construct a web page showing the latest events from a user's feed? Constructing such a feed must be fast so the page loads quickly, yet reflects recent updates to the underlying event streams. The wide fanout of popular streams (those with many followers) and high skew (fanout and update rates vary widely) make it difficult to scale such applications.

We associate feeds with *consumers* and event streams with *producers*. We demonstrate that the best performance results from selectively materializing each consumer's feed: events from high-rate producers are retrieved at query time, while events from lower-rate producers are materialized in advance. A formal analysis of the problem shows the surprising result that we can minimize global cost by making local decisions about each producer/consumer pair, based on the ratio between a given producer's update rate (how often an event is added to the stream) and a given consumer's view rate (how often the feed is viewed). Our experimental results, using Yahoo!'s web-scale database PNUTS, shows that this hybrid strategy results in the lowest system load (and hence improves scalability) under a variety of workloads.

Categories and Subject Descriptors: H.2.4 [Systems]: distributed databases

General Terms: Performance

Keywords: social networks, view maintenance

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Internet users like to be kept up-to-date with what is going on. Social networking sites like Twitter and Facebook provide a "feed" of status updates, posted photos, movie reviews, etc., generated by a user's friends. Content aggregators like My Yahoo and iGoogle let users create a customized feed aggregating multiple RSS sources. Similarly, news aggregators like Digg and Reddit provide a feed of the latest stories on topics like "entertainment" and "technology," while news sites like CNN.com provide the ability to follow fine-grained topics like "health care debate."

Each of these examples is a *follows* application: a user follows one or more interests, where an interest might be another user, content category or topic. In this paper, we formalize this important class of applications as a type of view materialization problem. We introduce the abstraction of a *producer*, which is an entity that generates a series of time-ordered, human-readable events for a particular followable interest. Thus, a producer might be a friend, a website, or aggregator of content on a particular topic collected from multiple sources. The goal of a follows application is to produce a "feed" for a user, which is a combined list of the latest events across all of the producers a user is following. For example, a feed might combine recent status updates from all of the user's friends on a social site, or recent stories on all of the user's topics on a content aggregation site. In some cases a user wants a combined feed, including both social and topic updates. An important point to keep in mind for optimization purposes is that we need only show the most recent events (specified in terms of a window of time or number of events) when a consumer checks his feed.

Follows applications are notoriously difficult to scale. The application must continually keep up with a high throughput of events. Twitter engineers have famously described re-architecting Twitter's back-end multiple times to keep up with rapid increases in throughput as the system became more popular (see for example [26]). At the same time, users expect their feed page to load fast, which means latency must be strictly bounded. This often means extensive materialization and caching, with associated high capital and operations expenditure. For example, Digg elected to denormalize and materialize a large amount of data to reduce latency for their "green badge application" (e.g., follow which stories my friends have dugg), resulting in a blow up of stored data from tens of GB to 3 TB [10].

There are several reasons why such applications are hard to scale. First, events fan out, resulting in a multiplicative effect on system load. Whenever Ashton Kutcher "tweets,"

his status update is propagated to over 4.6 million followers (as of March 2010). Even a considerably lower average fanout can cause severe scaling problems. Second, the fanouts and update rates have high skew across producers, making it difficult to choose an appropriate strategy. Facebook, for example, reportedly employs different feed materialization strategies for wide-fanout users like bands and politicians, compared to the majority of users who have much narrower fanout.

In this paper, we present a platform that we have built for supporting *follows* applications. Consider a *producer* of events (such as a user's friend, or news source) and a *consumer* of events (such as the user himself). There are two strategies for managing events:

- *push*—events are pushed to materialized per-consumer feeds
- *pull*—events are pulled from a per-producer event store at query time (e.g., when the consumer logs in)

If we regard each consumer's feed as a "most recent" window query, we can think of these options in traditional database terms as a "fully materialize" strategy versus a "query on demand" strategy. Sometimes, *push* is the best strategy, so that when users log in, their feed is pre-computed, reducing system load and latency. In contrast, if the consumer logs in infrequently compared to the rate at which the producer is producing events, the *pull* strategy is best. Since we only need to display the most recent N events, it is wasteful to materialize lots of events that will later be superseded by newer events before the consumer logs in to retrieve them.

In our approach, a consumer's feed is computed using a combination of push and pull, to handle skew in the event update rate across producers: a particular user that logs in once an hour may be logging in more frequently than one producer's rate (so push is best) and less frequently than another producer's rate (so pull is best.) A key contribution of this paper is the theoretical result that making *local* push/pull decisions on a producer/consumer basis minimizes total *global* cost. This surprising result has great practical implications, because it makes it easier to minimize overall cost, and straightforward to adapt the system when rates or fanouts change. Our experiments and our experience with a live follows application show that this approach scales better than a purely push or purely pull system across a wide range of workloads. Furthermore, this local optimization approach effectively allows us to cope with flash loads and other sudden workload changes simply by changing affected producer/consumer pairs from push to pull. Thus, our techniques can serve as the basis for a general-purpose follows platform that supports several instances of the follows problem with widely varying characteristics.

The follows problem is similar to some well-studied problems in database systems. For example, the "materialize or not" question is frequently explored in the context of index and view selection [13, 4]. In our context, the question is not which views are helpful, but which of a large number of consumer feed views to materialize. There has been work on partially materialized views [16] and indexes [27, 24, 25] and we borrow some of those concepts (e.g., materializing frequently accessed data). In contrast to this previous work, we show that it is not possible to make a single "materialize or not" decision for a given base tuple (producer event) in the follows problem setting; instead, we need to make that decision for each producer/consumer pair based on their relative event and query rates. Other work in view maintenance and query answering using views targets complex query workloads and aggregation, while our scenario is specialized to (a very large number of) most recent window queries over (a very large number of) streams. We review related work more thoroughly in Section 6.

In this paper, we describe an architecture and techniques for large scale follows applications. We have implemented these techniques on top of PNUTS [9], a distributed, web-scale database system used in production at Yahoo!. In particular, we make the following contributions:

- A formal definition of the follows problem as a partial view materialization problem, identifying properties that must be preserved in follows feeds (Section 2).
- An analysis of the optimization problem of determining which events to push and which events to pull, in order to minimize system load (equivalently, maximize scalability) while providing latency guarantees. We establish the key result that making push/pull decisions on a *local* basis provides *globally* good performance (Section 3).
- Algorithms for selectively pushing events to a consumer feed to optimize the system's performance. Our algorithms leverage our theoretical result by making decisions on a per-producer/consumer basis. Further, these algorithms allow our system to effectively adapt to sudden surges in load (Section 4).
- An experimental study showing our techniques perform well in practice. The system chooses an appropriate strategy across a wide range of workloads (Section 5).

Additionally, we review related work in Section 6, and conclude in Section 7.

2. PROBLEM DEFINITION

We now define the follows problem formally. First, we present the underlying data model, including consumers, producers, and the follows concept. Next, we examine user expectations about the resulting customized feed. Then, we define the core optimization problem we are addressing.

2.1 Follows data and query model

A follows application consists of a set of *consumers* (usually, users) who are following the event streams generated by a set of *producers*. Each consumer chooses the producers they wish to follow. In this paper, a *producer* generates a named sequence of human-readable, timestamped events. Examples of producers are "Ashton Kutcher's tweets" or "news stories about global warming." In general, a producer may be another user, a website (such as a news site or blog) or an aggregator of events from multiple sources. We treat each followable "topic" (such as "global warming" or "health care debate") as a separate producer in our discussion, even if the content for different topics comes from the same website or data source. Events are usually ordered by the time they were created (although other orderings are possible.)

We define the *connection network* as a directed graph $G(V, F)$, where each vertex $v_i \in V$ is either a consumer or a producer, and there is a follows edge $f_{ij} \in F$ from a consumer vertex c_i to a producer vertex p_j if c_i follows p_j (i.e., c_i consumes p_j's events.) Social networks are one example of a type of connection network. Other examples

include people following each other on Twitter, status and location updates for friends, customized news feeds, and so on. While these instances share a common problem formulation, the specifics of scale, update rates, skew, etc. vary widely, and a good optimization framework is required to build a robust platform.

We can think of the network as a relation `ConnectionNetwork (Producer,Consumer)`. Typically, the *connection network* is stored explicitly in a form that supports efficient lookup by producer, consumer or both. For example, to push an event for producer p_j to interested consumers, we must look up p_j in the producer network and retrieve the set of consumers following that event, which is $\{c_i : f_{ij} \in F\}$. In contrast, to pull events for a consumer c_i, we need to look up c_i in the network and retrieve the set of producers for that consumer, which is $\{p_j : f_{ij} \in F\}$. In this latter case, we may actually define the relation as `ConnectionNetwork(Consumer,Producer)` to support clustering by Consumer. If we want to support both access paths (via producer and via consumer), we must build an index in addition to the `ConnectionNetwork` relation.

Each producer p_j generates a stream of events, which we can model as a producer events relation PE_j`(EventID, Timestamp, Payload)` (i.e., there is one relation per producer). When we want to show a user his feed, we must execute a *feed query* over the PE_j relations. There are two possibilities for the feed query. The first is that the consumer wants to see the most recent k events across all of the producers he follows. We call this option **global coherency** and define the feed query as:

$$\textbf{Q1. } \sigma_{(k \text{ most recent events})} \bigcup_{\forall j: f_{ij} \in F} PE_j$$

A second possibility is that we want to retrieve k events per-producer, to help ensure diversity of producers in the consumer's view. We call this option **per-producer coherency** and define the feed query as:

$$\textbf{Q2. } \bigcup_{\forall j: f_{ij} \in F} \sigma_{(k \text{ most recent events})} PE_j$$

Further processing is needed to then narrow down the per-producer result to a set of k events, as described in the next section. We next examine the question of when we might prefer global- or per-producer coherency.

2.2 Consumer feeds

We now consider the properties of consumer *feeds*. A feed query is executed whenever the consumer logs on or refreshes their page. We may also automatically retrieve a consumer's updated feed, perhaps using Ajax, Flash or some other technology. The feed itself is a display of an ordered collection of events from one or more of the producers followed by the user. A feed typically shows only the N most recent events, although a user can usually request more previous events (e.g., by clicking "next"). We identify several properties which capture users' expectations for their feed:

- **Time-ordered:** Events in the feed are displayed in timestamp order, such that for any two events e_1 and e_2, if Timestamp(e_1) < Timestamp(e_2), then e_1 precedes e_2 in the feed[1].
- **Gapless:** Events from a particular producer are displayed without gaps, i.e., if there are two events e_1 and e_2 from producer P, e_1 precedes e_2 in the feed, and there is no event from P in the feed which succeeds e_1 but precedes e_2, then there is no event in PE_j with a timestamp greater than e_1 but less than e_2.
- **No duplicates:** No event e_i appears twice in the feed.

When a user retrieves their feed twice, they have expectations about how the feed changes between the first and second retrieval. In particular, if they have seen some events in a particular order, they usually expect to see those events again. Consider for example a feed that contains $N = 5$ events and includes these events when retrieved at 2:00 pm:

	Event	*Time*	*Producer*	*Text*
Feed 1	e_4	1:59	Alice	Alice had lunch
	e_3	1:58	Chad	Chad is tired
	e_2	1:57	Alice	Alice is hungry
	e_1	1:56	Bob	Bob is at work
	e_0	1:55	Alice	Alice is awake

At 2:02 pm, the user might refresh their feed page, causing a new version of the feed to be retrieved. Imagine in this time that two new events have been generated from Alice:

	Event	*Time*	*Producer*	*Text*
Feed 2	e_6	2:01	Alice	Alice is at work
	e_5	2:00	Alice	Alice is driving
	e_4	1:59	Alice	Alice had lunch
	e_3	1:58	Chad	Chad is tired
	e_2	1:57	Alice	Alice is hungry

In this example, the two new Alice events resulted in the two oldest events (e_0 and e_1) disappearing, and the global ordering of all events across the user's producers are preserved. This is the **global coherency** property: the sequence of events in the feed matches the underlying timestamp order of all events from the user's producers, and event orders are not shuffled from one view of the feed to the next. This model is familiar from email readers that show emails in time order, and is used in follows applications like Twitter.

In some cases, however, global coherency is not desirable. Consider the previous example: in Feed 2, there are many Alice events and no Bob events. This lack of diversity results when some producers temporarily or persistently have higher event rates than other producers. To preserve diversity, we may prefer **per-producer coherency**: the ordering of events from a given producer is preserved, but no guarantees are made about the relative ordering of events between producers. Consider the above example again. When viewing the feed at 2:02 pm, the user might see:

	Event	*Time*	*Producer*	*Text*
Feed 2'	e_6	2:01	Alice	Alice is at work
	e_5	2:00	Alice	Alice is driving
	e_4	1:59	Alice	Alice had lunch
	e_3	1:58	Chad	Chad is tired
	e_1	1:56	Bob	Bob is at work

This feed preserves diversity, because the additional Alice events did not result in the Bob events disappearing. How-

[1]Note that many sites show recent events at the top of the page, so "preceded" in the feed means "below" when the feed is actually displayed.

ever, whereas before there was an event (e_2) between the Bob and Chad event, now there is not. This event "disappearance" is not possible under global coherency, but is allowed under per-producer coherency to preserve diversity.

2.2.1 Feed diversity

We now look closer at diversity and offer a formal definition. Because of the skew in event rates that are inherent in many follows applications, we may need to take explicit steps to preserve diversity. Consider for example a user, David, who logs in once a day. His father, Bob, may only generate an event once a day, while his sister, Alice, generates an event once an hour. When David logs in, he would like to see his father Bob's latest event, even though there might be many more recent events from Alice.

A simplistic way to define diversity is to specify that the feed must contain at least one (or at least k) events from each of the consumer's producers. However, a consumer may have more producers than there are events shown in the feed, making it impossible to show one event from every producer. Moreover, we do not necessarily want to show extremely old events just to ensure diversity. If Bob's latest event is a year old, we may not want to include it, even if it means showing zero Bob events.

Therefore, we define the notion of *k,t-diversity*. Informally, k,t-diversity specifies that if there is an event from Bob in the last t time units, then we should not show more than k Alice events unless we are also showing the Bob event. More formally, consider two producers p_i and p_j being followed by consumer C. Define Candidate(P,t) as the number of events from producer P that are no older than t seconds, and Count(P) as the number of events from producer P that are shown in C's feed.

- k,t**-diversity**: if Candidate$(p_i,t) > 0$, and Count$(p_i) = 0$, then $Count(p_j) \leq k$.

Consider again the feed examples in the previous section. Imagine we specify $t = 600$ sec and $k = 1$. Then Feed 2 is not permitted, since $Candidate(Bob, 600sec) = 1$ and $Count(Bob) = 0$, but $Count(Alice) > 1$. Feed 2', however, is permitted.

Note that we might prefer a stronger notion of diversity in some cases. For example, if there are actually many Bob events, we should not show ten Alice events and only one Bob event. Therefore, applications may wish to maximize some notion of "entropy" in the events that they show. For our purposes, however, k,t-diversity captures a minimal notion of diversity, and illustrates that even a minimal notion of diversity conflicts with the global coherency guarantee. For more flexibility an application can go further and acquire a *pool* of events, and then use a custom algorithm to decide which to show to the end user.

2.3 Optimization problem

To generate a feed, we must execute query Q1 or Q2. The main question considered in this paper is, for a given consumer c_i, whether to pull events from the PE_j relations for the producers c_i follows, or to pre-materialize (push) the events that would result from the selections in the query. In either case, system processing is required: work is done at query time for pull, or event generation time for push. The type of load placed on the system by this processing depends on the architecture. If we store data on disk, the main cost is likely to be I/O. However, in many follows applications, data is stored in RAM for performance reasons. In that case, the main cost is likely to be CPU cost.

Formally, let us define $Cost()$ as the total usage of the bottleneck resource (e.g., I/O or CPU). This usage includes both processing events at generation time, as well as generating feeds at query time. Then, the *general follows optimization problem* is:

- Minimize $Cost()$, while:
  - Providing feeds that are *time-ordered*, *gapless* and *no-duplicates*, and
  - Respecting the chosen level of coherency (global or per-producer), and
  - If using per-producer coherency, then also ensuring k,t-diversity.

Because a follows application is usually user-facing, we may additionally be concerned about latency, and may be willing to trade some extra system cost for reduced latency. We define the *latency-constrained follows optimization problem* as:

- Minimize $Cost()$, while:
  - Satisfying the conditions of the general follows optimization problem, and
  - Ensuring the Nth percentile latency is less than M time units (e.g., milliseconds).

For example, we might specify that the 99th percentile latency be no more than 50 milliseconds. This might lead us to make different decisions about pushing or pulling events (in particular, we might push more events so that query time latency meets the constraint.)

2.4 Aggregating producers

A producer may be an atomic entity (e.g., a user or news website) or may be an aggregator of multiple other sites. Aggregators are important to some follows applications because they can produce a single stream of events on a given topic from multiple sources. Moreover, aggregators can extract information from sources that would not normally push new events. In this paper, we treat atomic and aggregating producers the same, and consider only their event rate and fan out. However, a more general problem is to examine the decisions that an aggregator must make: should the aggregator use push or pull for a given upstream source? Although this decision is similar to the decisions we make when deciding to push or pull for a producer/consumer pair, there are additional complexities (like how frequently to pull, and how to deal with upstream sources with different push or pull costs). While this more general problem is outside the scope of this paper, we discuss some of the additional challenges in Section 3.4.

3. OPTIMIZATION

We now solve the optimization problems introduced in Section 2. We first solve the *general follows optimization problem*. We call this solution `MinCost`. A particularly useful result is that locally assigning each producer/consumer pair to push or pull produces the globally optimal solution. We then address the *latency constrained follows optimization problem* by shifting edges from pull to push. This solution, called `LatencyConstrainedMinCost`, allows us to incrementally adjust the system to effectively satisfy latency constraints despite workload changes. Finally, we briefly

Notation	*Description*
p_j	Producer
c_i	Consumer
f_{ij}	"Follows:" consumer c_i follows producer p_j
F_i	The set of all producers that consumer c_i follows
H	The cost to push an event to c_i's feed
L_j	The cost to pull events from p_j
$e_{j,k}$	The kth event produced by p_j
ϕ_{p_j}	p_j's event frequency
ϕ_{c_i}	c_i's query frequency
k_g	Events to show in a feed (global coherency)
k_p	Max events to show per producer in a feed (per-producer coherency)
λ_m	Latency achieved in `MinCost`
λ_l	Target latency of `LatencyConstrainedMinCost`
σ_{ij}	Decrease in latency by shifting e_{ij} from pull to push
ϵ_{ij}	Extra cost from shifting e_{ij} from pull to push

Table 1: Notation

examine the optimization problems posed by aggregating producers.

3.1 Model and Assumptions

Recall the connection network $G(V, F)$ from Section 2.1. This graph is a bi-partite graph from producers to consumers. When consumer c_i performs a query, we must provide a number of events for c_i's feed. We consider both of the coherency cases introduced in Section 2. In the global case, we supply the most recent k_g events across *all* producers in F_i. In the per-producer case, we supply the most recent k_p events for *each* producer in F_i. Table 1 summarizes the notation we use in this section.

The decisions we make for storing producer events and materialized consumer feeds determine the cost to push or pull events. We denote H as the cost to push an event to c_i's feed, and L_j as the cost to pull a small constant number of events from producer p_j. L_j might vary between producers if we obtain their streams from different systems. We assume k_p and k_g are both small enough that L_j will be the same for each. Note that we assume that multiple events can be retrieved using a pull from a single producer. This is true if we have stored the events from producer streams, clustered by producer; a single remote procedure call can retrieve multiple events; and the disk cost (if any) for retrieving events is constant for a small number of events (e.g. since any disk cost is dominated by a disk seek). We do not consider the cost to ingest and store producer events; this cost would be the same regardless of strategy.

Furthermore, we assume that the H and L_j costs are constant, even as load increases in the system. If the system bottleneck changes with increasing event rate or producer fanout, this assumption may not be true. For example, we may materialize data in memory, but if the number of consumers or producer fan-out changes, the data may grow and spill over to disk. This would change the push and pull costs to include disk access. Usually we can provision enough resources (e.g. memory, bandwidth, CPU etc.) to avoid these bottleneck changes and keep H and L_j constant. Our analysis does not assume data is wholly in memory, but does assume the costs are constant as the system scales.

3.2 Cost Optimization

We now demonstrate that the appropriate granularity for decision making is to decide whether to push or pull individually for each producer/consumer pair.

3.2.1 Per-Producer Coherency

We begin with the per-producer coherency case. For c_i, p_j and event $e_{j,k}$, we derive the lifetime cost to deliver $e_{j,k}$ to c_i. The lifetime of $e_{j,k}$ is the time from its creation to the time when p_j has produced k_p subsequent events.

CLAIM 1. *Assume that we have provisioned the system with sufficient resources so that the costs to push and pull an individual event are constant as the system scales. Further, assume pull cost is amortized over all events acquired with one pull. Then, the push and pull lifetime costs for an event $e_{j,k}$ by p_j for c_i under per-producer coherency are:*

$$\begin{aligned} \textit{Push cost over } e_{j,k}\textit{'s lifetime} &= H \\ \textit{Pull cost over } e_{j,k}\textit{'s lifetime} &= L_j(\phi_{c_i}/\phi_{p_j}) \end{aligned}$$

Proof: Push cost is oblivious to event lifetime; we pay H once for the event. Pull cost does depend on lifetime. Event $e_{j,k}$ has lifetime k_p/ϕ_{p_j}. Over this lifetime, the number of times c_i sees $e_{j,k}$ is $\phi_{c_i}(k_p/\phi_{p_j})$. The cost to pull $e_{j,k}$ is (L_j/k_p) (L_j amortized over each pulled event). Thus, lifetime cost for $e_{j,k}$ is $(L_j/k_p)\phi_{c_i}(k_p/\phi_{p_j}) = L_j(\phi_{c_i}/\phi_{p_j})$. □

Then, we can conclude that for a given event, we should push to a given consumer if the push cost is lower than pull; and pull otherwise. Since the push and pull cost depend only on producer and consumer rates, we can actually make one decision for all the events on a particular producer/consumer pair. We can derive the optimal decision rule:

LEMMA 1. *Under per-producer coherency, the policy which minimizes cost for handling events from p_j for c_i is:*

$$\begin{aligned} \textit{If } (\phi_{c_i}/\phi_{p_j}) &>= H/L_j, \textit{ push for all events by } p_j \\ \textit{If } (\phi_{c_i}/\phi_{p_j}) &< H/L_j, \textit{ pull for all events by } p_j \end{aligned}$$

Proof: The push vs. pull decision draws directly from the costs in Claim 1. □

3.2.2 Global Coherency

Under global coherency, c_i's feed contains the k_g most recent events across all producers in F_i.

CLAIM 2. *Assume that we have provisioned the system with sufficient resources so that the costs to push and pull an individual event are constant as the system scales. Further, assume pull cost is amortized over all events acquired with one pull. Then, the push and pull lifetime costs for an event $e_{j,k}$ by p_j for c_i under global coherency are:*

$$\begin{aligned} \textit{Push cost over } e_{j,k}\textit{'s lifetime} &= H \\ \textit{Pull cost over } e_{j,k}\textit{'s lifetime} &= L_j\phi_{c_i}/\sum_{p_j \in F_i} \phi_{p_j} \end{aligned}$$

Proof: The proof for the global case is similar to the per-producer case, with a few key differences. Producer frequency is an aggregate over all of F_i: $\phi_{F_i} = \sum_{p_j \in F_i} \phi_{p_j}$. Event $e_{j,k}$ has lifetime k_g/ϕ_{F_i}, and its amortized pull cost is L/k_g. We substitute these terms in the per-producer analysis and reach the push and pull costs for global coherency. We omit the complete derivation. □

We can then derive the optimal decision rule for a (producer, consumer) pair in the global coherency case.

LEMMA 2. *Under global coherency, the policy which minimizes cost for handling events from p_j for c_i is:*

$$\text{If } \phi_{c_i} / \sum_{p_j \in F_i} \phi_{p_j} >= H/L_j, \text{ push for all events by } p_j$$

$$\text{If } \phi_{c_i} / \sum_{p_j \in F_i} \phi_{p_j} < H/L_j, \text{ pull for all events by } p_j$$

Proof: The push vs. pull decision draws directly from the costs in Claim 2. □

We summarize our main findings from Lemmas 1 and 2. Under per-producer coherency, the lifetime cost for an event for a particular consumer is dependent on both consumer frequency and the event's producer frequency. Under global coherency, lifetime cost for an event is dependent on both consumer frequency and the *aggregate* event frequency of the producers that the consumer follows.

3.2.3 `MinCost`

We have shown how to minimize cost for individual producer/consumer edges. We now show that such local decision making is globally optimal.

THEOREM 1. *For per-producer coherency, the `MinCost` solution is derived by separately choosing push or pull for each producer/consumer pair, with push vs. pull per pair determined by Lemma 1.*

Proof: We minimize global cost by minimizing the cost paid for every event. Similarly, we minimize the cost for an event by minimizing the cost paid for that event for every consumer. Lemma 1 assigns each edge push or pull to minimize the cost paid for events between the edge's consumer and producer. Further, no assignment made on any one edge imposes any restrictions on the assignments we can make to any other edges. Therefore, minimizing cost for every consumer and event minimizes global cost. □

THEOREM 2. *For global coherency, `MinCost` is derived by separately choosing push or pull for each consumer, with push vs. pull per consumer determined by Lemma 2.*

Proof: The proof mirrors that of Theorem 1. In this case, Lemma 1 assigns all edges either push or pull. Again, no edge assignment restricts any other edge assignments. Therefore, minimizing cost for every consumer and event minimizes global cost. □

Theorems 1 and 2 have important theoretical and practical implications. The ability to separately optimize each edge makes minimizing overall cost simple. Moreover, as query and event rates change over time and new edges are added, we do not need to re-optimize the entire system. We simply optimize changed and new edges, while leaving unchanged edges alone, and find the new `MinCost` plan.

We also observe that any system is subject to different query and event rates, and skew controlling which consumers or producers contribute most to these rates. To find the `MinCost`, however, there is no need to extract or understand these patterns. Instead, we need only separately measure each consumer's query rate and compare it to the event rate of the producers that the consumer follows (either individually or in aggregate).

3.3 Optimizing Query Latency

To this point, our analysis has targeted `MinCost`. We now consider `LatencyConstrainedMinCost`, which respects an SLA (e.g. 95% of requests execute within 100 ms). `MinCost` may already meet the SLA. If not, we may be able to shift pull edges to push, raising cost, but reducing work at query time, and meet the SLA. We might also move to a push-only strategy and still not meet a very stringent SLA. In this section we analyze the case where pushing helps, but at a cost. We ultimately provide a practical adaptive algorithm that approximates `LatencyConstrainedMinCost`.

Suppose `MinCost` produces a latency of λ_m, while `LatencyConstrainedMinCost` requires a latency of λ_l, where $\lambda_l < \lambda_m$. Intuitively, we can shift edges that are pull in `MinCost` to push and reduce latency. Formally, we define σ_{ij} as the decrease in global latency gained by shifting e_{ij} to push. We define ϵ_{ij} as the penalty from doing this shift. Formally, $\epsilon_{ij} = \phi_{p_j}(H - L_j(\phi_{c_i}/\phi_{p_j}))$ for per-producer coherency; this is the extra cost paid per event for doing push instead of pull, multiplied by p_j's event rate.

Consider the set of edges pulled in `MinCost`, R. To find `LatencyConstrainedMinCost`, our problem is to choose a subset of these edges S to shift to push, such that $\sum_S \sigma_{ij} \geq (\lambda_m - \lambda_l)$ and $\sum_S \epsilon_{ij}$ is minimized. Solving for S is equivalent to the knapsack problem [19], and is therefore NP-hard. We sketch this equivalence.

LEMMA 3. *`LatencyConstrainedMinCost` is equivalent to knapsack.*

Proof sketch: We start by reducing knapsack to `LatencyConstrainedMinCost`. Consider the standard knapsack problem. There exists a weight constraint W and set of n items I, where the ith item in I has value v_i and weight w_i. The problem is to find a subset of items K such that $\sum_{i \in K} w_i \leq W$ and $\sum_{i \in K} v_i$ is maximized. We can convert this to an equivalent problem of finding the set of items K' *omitted* from the knapsack. Assume $\sum_{i \in I} w_i = T$. The problem is to find K' such that $\sum_{i \in K'} w_i \geq T - W$ and $\sum_{i \in K'} v_i$ is minimized. Through variable substitution (omitted), we show the omitted knapsack problem is `LatencyConstrainedMinCost`. We can similarly reduce `LatencyConstrainedMinCost` to knapsack. This verifies the two problems are equivalent. □

3.3.1 Adaptive Algorithm

We are left with two issues. First, knapsack problems are NP-hard [19]. Second, we have defined σ_{ij} as the reduction in latency from shifting f_{ij} to push; in practice, we find that we can not accurately predict the benefit gained by shifting an edge to push. We now show how we handle both issues with an adaptive algorithm.

Though we do not know the exact latency reduction from shifting an edge to push, choosing a higher rate consumer should result in a greater reduction, since this benefits more feed retrievals. Therefore, we set $\sigma_{ij} = \phi_{c_i}$, estimate the $\sum \phi_{c_i}$ that does reduce latency by $(\lambda_m - \lambda_l)$, and solve for S. We measure whether S exactly meets the SLA. If not, we adapt: if latency is too high, we solve for a larger $\sum \phi_{c_i}$: if latency is too low, we solve for a smaller $\sum \phi_{c_i}$.

We want a solution that incurs only incremental cost when we adjust the target $\sum \phi_{c_i}$. Moreover, the knapsack problem is NP-hard, so our solution must provide a suitable approximation of the optimal solution. To address both concerns,

we use an adaptive algorithm that produces a greedy approximation of the optimal solution; we expect a greedy approximation to be effective in practice. We simply sort the pull edges by $(\phi_{c_i}/\epsilon_{ij})$ descending, and shift some number of top-ranked edges to push. Then, if our latency is higher than the SLA, we incrementally shift the top-ranked pull edges to push. If our latency is lower than the SLA, we incrementally shift the lowest-ranked push edges back to pull (note that we never shift edges to pull if they are assigned to push in `MinCost`).

We run the adaptive algorithm from `MinCost` as a starting point. We must also periodically re-run the algorithm to ensure we have the `LatencyConstrainedMinCost` solution. Suppose, for example, a number of consumers add new interests (e.g. producer/consumer pairs), and the system, optimizing for `MinCost`, chooses the pull strategy for them. These new producer/consumer pairs may cause latency to increase, even though push/pull decisions have not changed for existing producer/consumer pairs. Thus, we run the adaptive algorithm and shift more edges to push, returning the system to `LatencyConstrainedMinCost`.

3.3.2 Moving consumers wholesale to push

An alternative strategy to moving producer/consumer pairs is to move consumers as blocks; when moving pairs to push, we select consumers with some pull pairs, and move all such pairs to push at once. The intuition is that if we have to pull even one producer for a consumer, that pull will dominate the consumer's latency. In this case, we compute ϵ for an entire consumer c_i as $\epsilon_i = \phi_{p_j}(H - L_j\phi_{c_i}/\sum_{p_j \in F_i}\phi_{p_j})$. The downside of moving entire consumers is that it reduces our flexibility, as we cannot convert highly beneficial producer/consumer pairs to push unless we convert the entire consumer. We compare the "per-pair" and "per-consumer" strategies experimentally in Section 5.

3.4 Composite consumer, producer networks

In this paper, our focus is on simple connection networks, where each node is either a producer or consumer, and the network is a bipartite graph. Suppose we have a *composite network*, where some nodes might be both a consumer relative to upstream event sources and a producer relative to downstream consumers (as in Section 2.4). For example, we might have a node that aggregates "health care debate" events from several sources. The optimization problem is still to assign push or pull to edges to minimize system cost.

The problem, however, is that decisions we make about edges are no longer independent. If we decide to push for a given edge, we cannot decide to pull for any adjacent upstream edges, because we cannot push events downstream if upstream edges are also not pushing. Because this independence no longer holds, our proofs of Theorems 1 and 2 no longer hold. Therefore, new techniques are needed to globally optimize a composite network, and we are developing these techniques in ongoing work.

4. IMPLEMENTATION

We have implemented the hybrid push/pull technique as a view maintenance mechanism on top of our elastically scalable web serving database, PNUTS [9]. Although other architectures are possible, our approach cleanly encapsulates the materialization decisions in the view maintenance component, while using the web database to handle other tasks such as processing updates and feed queries. In this section, we describe our implementation and the rationale for the design decisions we have made.

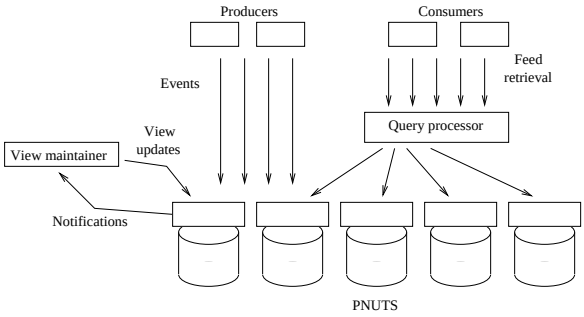

Figure 1: Architecture for follows application.

4.1 Architecture

Figure 1 depicts our implementation architecture. When a *producer* generates an event, it is stored in PNUTS. For simplicity, we have elected to create a single table `ProducerPivoted` to store all of the producer event streams; so that `ProducerPivoted`$= \bigcup_{\forall j} PE_j$. A producer in this case might be an application layer (for example, if users directly generate events), a crawler (for example, to retrieve events from external sources) or an ingest API (for example, to allow sources to push events.) When an event enters the system, PNUTS sends a *notification* to the *view maintainer*. The view maintainer responds to this notification by deciding whether to push the new event to one or more consumer feed records, using custom logic to implement the decision rules of Section 3. The view maintainer must read information from the `ConnectionNetwork` table to make this decision, as described in Section 4.2.

If an event is pushed, it is written back to PNUTS into a materialized view. Again for simplicity, we store all of the materialized queries for different consumers in a single table, `ConsumerPivoted`. The `ConsumerPivoted` table also stores information about which producers have not been materialized for the consumer, so that we know which producers to pull from at query time. When a consumer retrieves their view (e.g. when a user logs in or refreshes their page), this results in a query to the *query processor*. The query processor reads pushed events from `ConsumerPivoted` and also pulls events from `ProducerPivoted`. In our current implementation, `ProducerPivoted` and `ConsumerPivoted` are stored in the same PNUTS instance. However, it is possible to store these tables in separate infrastructures if desired.

Notifications are a native mechanism in PNUTS, similar to very simple triggers. If our techniques were implemented in a system without triggers, some other mechanism would be needed (such as updating the view as part of the event ingest transaction.) Notification allows view maintenance to be asynchronous, improving update latency. However, views may be slightly stale; this is acceptable in many web applications (including follows applications).

4.1.1 Storage

PNUTS is a massively-parallel, distributed web scale database, that provides low-latency reads and writes of individual records, and short range scans. Range scans are particularly important for efficient push and pull operations, as we describe in Section 4.2. We have elected to provision enough main-memory to store the data tables in our application to

ConnectionNetwork

Key	*Value*
bob_1.3_alice	...
bob_2.4_chad	...
bob_3.9_roger	...
roger_9.2_chad	...
roger_9.8_alice	...

ProducerPivoted

Key	*Content*
alice_t0	"Alice is awake"
alice_t1	"Alice is hungry"
alice_t2	"Alice had lunch"
bob_t0	"Bob is at work"
bob_t1	"Bob is tired"

Figure 2: Example tables

Key	*Events*	*Priority*
alice_bob	t0="Bob is at work", t1="Bob is tired"	2.9
alice_chad	t0="Chad is tired"	2.4
bob_alice	t0="Alice is awake", t1="Alice is hungry", ...	2.1
bob_chad	null	1.3
chad_alice	t0="Alice is awake", t1="Alice is hungry", ...	3.3

Figure 3: Table ConsumerPivoted

ensure low latency, although data is also persisted to disk to survive failures. Other database systems besides PNUTS could be used to implement our techniques if they similarly provided range query capabilities.

4.2 Schema and View Definitions

In this section, we describe the data tables in our implementation in more detail. Note that for brevity we omit columns from tables that are not vital to understanding the design, such as metadata, timestamps, etc.

The `ConnectionNetwork` table stores the connection graph between consumers and producers. An example is shown in Figure 2. The key of the table is the composite (producer, priority, consumer). The priority is calculated from the consumer's query rate and the producer's event rate, according to Lemmas 1 and 2. We sort the table by producer, and then priority, to simplify the decision making about whether to push an event. When a new event arrives, we do a range query over the records prefixed with the event's producer, starting at priority R, where R is the optimal threshold derived in Section 3 (e.g. H/L_j). This allows us to quickly retrieve only the producer/consumer pairs with a priority greater than R, which are the consumers to push to.

Events are stored in the `ProducerPivoted` table; an example is shown in Figure 2. The key of this table is (producer,timestamp) so that events are clustered by producer, and ordered by time within a producer (reflecting the per-producer event streams). This ordering facilitates efficient retrieval of the latest events from a given producer using a range scan with a specified limit on the number of events to retrieve (since we only show the most recent N events.)

The `ConsumerPivoted` table stores materialized feed records for each consumer. An example is shown in Figure 3. The key of this table is (consumer,producer), so that there is a separate record for each producer that a consumer follows. If the consumer decides to follow a new producer, a new `ConsumerPivoted` record is inserted, initially with no events (as with `bob_chad` in the figure.) When the view maintainer is notified of a new event, the maintainer performs a range scan on the `ConnectionNetwork` table to find the list of consumers to push the event to, and updates the appropriate `ConsumerPivoted` record for each consumer. The result is that some feed records have events materialized, while others have `null`. The feed record also includes the priority for the (consumer,producer) pair; this is the same priority stored in the `ConnectionNetwork` table. If a feed record has no events (e.g., null), we need only pull from `ProducerPivoted` if the priority is below the threshold H/L_j. A null record with priority above the threshold indicates that events will be pushed, but the producer has not yet generated any.

Note that potentially multiple events are stored in the "Events" column of the `ConsumerPivoted` table. PNUTS makes this process easier by supporting a complex column type. Also, the same event may be stored for multiple consumers. This is the result of producer fanout; multiple consumers may be interested in events from the same producer. If we fully materialized `ConsumerPivoted` (that is, pushed every event) the size of `ConsumerPivoted` would be significantly larger than `ProducerPivoted`. However, because we selectively materialize producer/consumer pairs, storage requirements are less, and `ConsumerPivoted` may be larger or smaller than `ProducerPivoted`.

4.3 Adapting Priority

As producer event rates and consumer query rates change, we must adapt the priorities for producer/consumer pairs. We measure and maintain event rates for producers and consumers. When the rates change, we re-compute priorities for producer/consumer pairs and then update `ConnectionNetwork` and `ConsumerPivoted` records. We maintain approximate statistics on rates, updating them periodically to reduce write load. To reduce the number of writes to update priorities we only perform the update if it would change the producer/consumer pair from push to pull (or vice versa). We include an experiment on adapting priorities in Section 5.5.

5. EVALUATION

We now examine our experimental results. Our experiments used the implementation described in Section 4 to handle the event load obtained from a real follows application (Twitter). We examine the impact on system load and user feed latency as we vary several different characteristics of the workload.

Our load metric was **aggregate CPU load** across all servers, as measured by the Unix `sar` command. In our system (as in many real-life deployments of follows applications) we provision sufficient main memory so that our data and view tables fit in memory; this decision reduces latency and improves throughput. Data is also written to disk for persistence, but reads are performed from the memory copy. As a result, CPU is the main bottleneck, and hence the best metric for system load. System load directly correlates with the amount of hardware that must be dedicated to the system, or equivalently, determines the amount of throughput sustainable by a given hardware deployment. Network bandwidth may also be a bottleneck in some deployments. In our experiments we measured network utilization and found the same trends as the aggregate CPU results reported here.

We also measure latency as **response time**, observed from the client, for retrieving a consumer's feed. Users are latency sensitive, and may stop using the website if their feed pages take too long to load.

In summary, our results show:

- The hybrid push/pull mechanism most effectively minimizes system load, even as the aggregate rate of events varies relative to the aggregate feed retrieval rate.
- We can directly trade some extra cost for reduced latency just by adjusting the push threshold.
- The density of the connection graph has a direct impact on the cost in the system; more fanout means more event processing work.

Number of producers	67,921
Number of consumers	200,000

	Average	*Zipf parameter*
Consumers per producer	15.0	0.39
Producers per consumer	5.1	0.62
Per-producer rate	1 event/hour	0.57
Per-consumer rate	5.8 queries/hour	0.62

Table 2: Workload parameters for baseline scenario.

- The system effectively absorbs flash events (a sudden increase in event rate from some producers) by responding to the new query:event ratio for affected consumers.

5.1 Experimental data and setup

We constructed our data set using traces from real follows applications. First, we constructed the *Connection-Network* by downloading the "following" lists for 200,000 Twitter users, using the Twitter API [1]. This resulted in a connection network with 67,921 producers and 1,020,458 producer/consumer pairs. The graph was highly skewed: both the number of consumers per producer (i.e. "fan-out") and the number of producers per consumer (i.e. "fan-in") followed a Zipfian distribution. Second, for producer events we downloaded the "tweets" (status updates) from those producers via the Twitter API. We found that the distribution of event rates also followed a Zipfian distribution. Third, we constructed a sequence of feed retrieval queries by consumers. Unfortunately, the Twitter API does not provide information about how often users retrieve their feeds. Therefore, as a substitute we obtained a trace of pageviews from Yahoo!'s Social Updates platform. Again, we found the distribution of pageview frequencies to be Zipfian. We assigned a query rate to our 200,000 consumers to follow this Zipfian distribution. Our baseline workload parameters are in Table 2. Our prototype implements per-producer coherency, to ensure diversity of results for consumers.

In our experiments, we measured the effect of varying some of these parameters. The experiments reported here used this real data set, although we ran other experiments with a synthetic data set (with a connection graph that was uniform instead of Zipfian) and observed similar trends. In one experiment reported here (Section 5.4) we report results using the synthetic data, which allowed us to more easily vary the producer fanout.

We ran our experiments on server-class machines (8 core 2.5 GHz CPU, 8 GB of RAM, 6 disk RAID-10 array and gigabit ethernet). In particular, we used six PNUTS storage servers, one view maintainer server, and one query processor server. In addition, PNUTS utilizes query routers (two servers in our experiments) and a replication system (two servers in our experiments.) Although we did not explicitly examine replication policies in this work, replication is necessary to provide fault tolerance (in particular, recovery from data loss after a storage server failure.)

5.2 System cost

We start by examining which policies perform best as we vary the relative query and event rates. We used the distribution of event rates from our real dataset and varied the average query rate, retaining the Zipfian skew observed in the query trace.

First, we tuned the hybrid push/pull policy by measuring

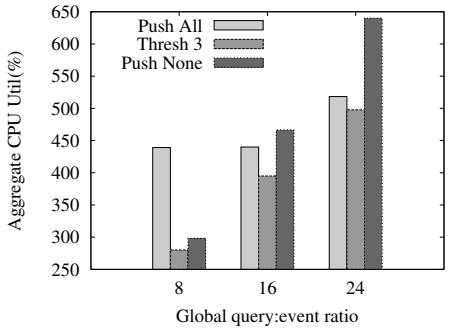

Figure 4: System Costs with Increasing query rates

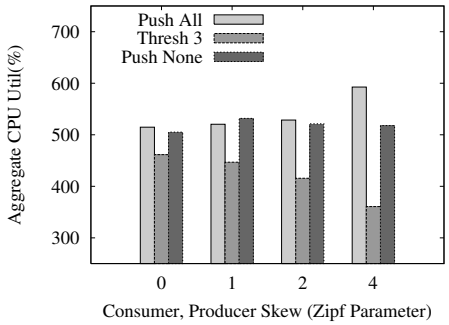

Figure 5: Relative effect of producer and consumer skew on Push None, Push All, and optimal.

the push threshold (that is, threshold ratio ϕ_{c_i}/ϕ_{p_j} from Lemma 1) that minimized system load. Our results (not shown) indicate that across a variety of query:event ratios, a push threshold of 3 minimizes system load.

Next we compared different policies. Figure 4 shows the system cost as the query:event ratio changes. The figure shows three strategies: Push All, Push None (i.e. pull all), and our hybrid scheme with push threshold 3. As the figure shows, neither Push All or Push None works well in all scenarios, confirming our hypothesis. For a scenario with a high event rate, Push All results in 47 percent higher system load than Push None. With a relatively high query rate, Push None results in 23 percent more system load than Push All. In contrast, the hybrid scheme always performs better than either strategy. Hybrid does mostly push when query rate is high, but still does some pull; similarly hybrid does mostly pull when event rate is high but still does some push. In the median scenario, where neither push nor pull is clearly better, hybrid does best by doing push for some producer/-consumer pairs and pull for others. Note that as we increase the query:event ratio, total system load for all policies increases. This is because we vary the ratio by increasing the query rate while using the same event rate, resulting in more total load on the system (regardless of the chosen policy).

5.2.1 Impact of skew

We also examined the impact of increased skew on system load. We ran an experiment with a query:event ratio of 16:1 (the median scenario from Figure 4) and varied the Zipfian parameters for per-producer rate and per-consumer rate (default settings in Table 2). Figure 5 shows the result-

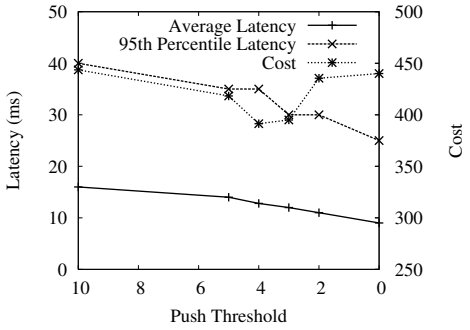

Figure 6: Push threshold vs. query latency (left vertical axis) and system cost (right vertical axis).

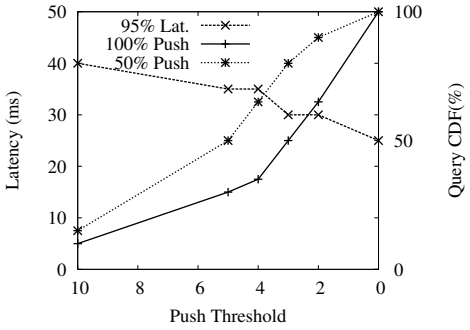

Figure 7: Push threshold vs. percentage of queries 100% pushed.

ing system load. As the figure shows, the hybrid policy is best in each case. Moreover, the improvement of the hybrid policy over Push All or Push None increases as the amount of skew increases from 14 percent at a moderate amount of skew to 30 percent when there is higher skew. When there is a great deal of skew, there are both consumers that log in very frequently (favoring push) and producers that produce events at a very high rate (favoring pull). Treating these extreme producers and consumers separately, as hybrid does, is essential to minimizing cost.

5.3 Latency

Next, we examined the latency experienced by the consumer under the hybrid technique. Figure 6 plots threshold vs. average and 95th percentile latency, again with a 16:1 query:event ratio. The 95th percentile latency is often more important to web applications than the average latency, as it reflects the latency observed by the vast majority of users. As expected, as we decrease threshold and push for more producer/consumer pairs, latency decreases.

Of course, decreasing the push threshold (in our system, below 3) explicitly adds cost to the system, as we may be pushing events to consumers, even if those consumers do not log in frequently enough to "deserve" the push according to the decision criteria of Lemma 1. Figure 6 demonstrates this effect, showing the impact on system cost as we decrease the threshold. Initially cost decreases, as the threshold decreases and approaches the optimal point (from a cost perspective.) As the threshold decreases further beyond the optimal, cost begins to rise again as we push more events than are necessary to minimize cost. This illustrates how we can trade some system cost to achieve the latency SLA we desire. It is up to the application designer to choose the appropriate tradeoff between latency and cost.

To help understand these results, Figure 7 shows CDFs of the number of users with 50 percent of the events in their feeds pushed, and with 100 percent of the events in their feeds pushed. As expected, a lower push threshold results in more users with events pushed, until eventually all events are pushed for all consumers, and latency (superimposed curve) reaches its minimum point.

5.3.1 Latency/cost tradeoff strategy

We next investigate our question from Section 3.3.2 on the best way to trade increased cost for reduced latency. We can shift individual producer/consumer pairs to push

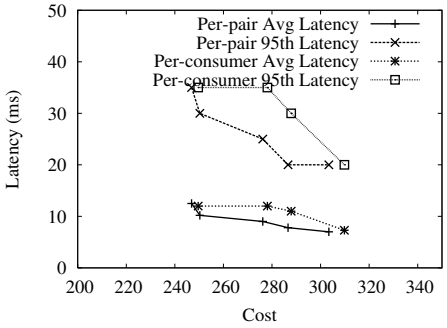

Figure 8: Cost vs. Latency, Per-pair and Per-consumer cases.

("per-pair"), or else shift whole consumers to push ("per-consumer"). In this experiment we use a query-to-event ratio of 8 since this is a scenario where pull would normally be favored for system cost, but we might want to push more events to reduce latency.

Figure 8 shows that latency is lower for the per-pair strategy, for both the average and 95th percentile latency. The per-pair strategy has more flexibility to materialize the producer/consumer pairs that provide latency benefit at minimum cost, while the need to materialize all or nothing for the per-consumer strategy handicaps its decision making. The result is that the per-pair strategy is better able to reduce latency without too much cost.

5.4 Fan-out

We next measure the impact of event fan-out on system cost. This experiment uses a synthetic *ConnectionNetwork* and a fixed 4:1 query:event ratio. We fixed the number of producers and consumers, but varied the number of consumers per producer, and so the result of increasing fan-out is to also increase fan-in; that is, the average number of producers per consumer.

Figure 9 shows the system cost as we increase fan-out, while using the hybrid policy. When fan-out increases, the system must spend more effort to fan those events out to more consumers, even though the event rate from producers is constant. This extra cost is incurred whether the event is pushed to a consumer or pulled. In fact, the decisions made for individual producer/consumer pairs does not change; there are just more producer/consumer pairs to make a push/pull decision for as we add edges to the connection network, and overall higher system cost.

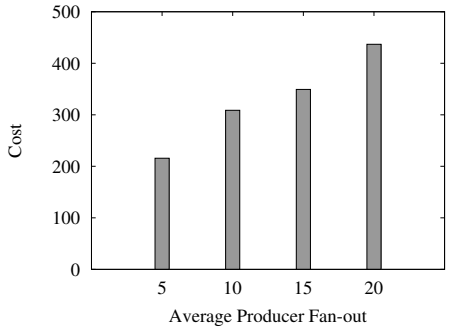

Figure 9: Impact of changing fan-out.

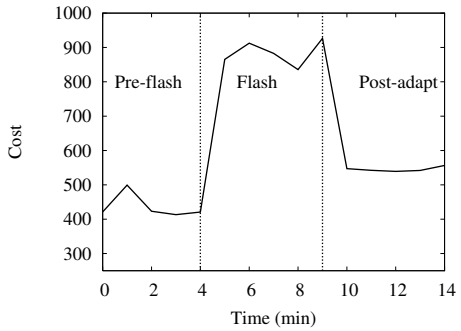

Figure 10: Impact of flash events on cost.

5.5 Flash events

In follows applications, it is possible to have "flash events," when there is a sudden, sharp increase in events on a particular topic. For example, there was a huge upsurge in events on Twitter during Barack Obama's inauguration in January 2009 [26]. We ran an experiment to examine the impact of such flash events. In this experiment, after running the system, we suddenly changed the event rates of 0.1% of the infrequent producers to start producing events at a significantly higher rate. We also increased the average fanout of those producers from 15 to 100, to reflect a large increase in consumers interested in the flash event. As Figure 10 shows, the experiment followed three phases: 1) A baseline before the flash events, 2) when the flash events start happening, 3) after the system has adjusted to the new query:event ratio for consumers following the flash producers. During phase 2, we have a number of producer/consumer pairs assigned to push (because they have low event rates normally) that now result in a lot of cost because of the higher pushed event rate. The system adapts by shifting all producer/consumer pairs for these producers to pull. This lowers cost nearly back down to the pre-flash level.

6. RELATED WORK

Materialized views have been extensively used to optimize query processing. Much of this work, surveyed in [15], focuses on complex query workloads, and on choosing appropriate views. For example, view selection [13, 4] chooses the best views to materialize for complex data warehousing loads. Our problem is more specialized: we consider an unbounded number of consumer feeds and an unbounded number of producer streams. Each view we consider for partial materialization is a most-recent window query that joins the `ConnectionNetwork` table (more precisely, a selection of rows from it corresponding to the given consumer) with (all connected) event streams. Thus, in contrast to existing work, it is simpler in that we only consider one kind of query, but more complex in that we simultaneously optimize a large collection of window queries over a large collection of input streams, exploiting observed query and update patterns.

Partial indexing and materialization: Our collection of partially materialized feeds is similar to partial indexes [27, 24, 25], which index only a subset of the base table to reduce maintenance cost and index size. The choice of what to index depends on the base data; for example, to avoid indexing tuples with values that are never queried. In our approach, we may materialize a base tuple for one consumer and not another, and do so based on event and query rates, not on the value distribution of the base data.

Luo [16] examines partially materialized views, materializing only frequently accessed results so that they can be returned quickly while the rest of the query is executing. In our model, this might be similar to caching events from high fan-out producers. We have shown that it is important to take the producer and consumer rates into account; even high fanout producers may not be materialized for all consumers. Top-k views are also partial materializations. Yi et al [29] examine how to "refill" the view when one of the top-k values falls out due to base data update or deletion. Although our consumer views are like top-k views, we do not need to compute an aggregate to refill the view, as in [29]; we only need to take the next time ordered value.

Other partial view types include partial aggregates to support quick, imprecise query results [14], and views in probabilistic databases, which inherently represent partial information about the actual world [22]. Our queries are non-aggregate and exact, so these techniques are less applicable.

View and index maintenance: Several web-scale database systems have begun to incorporate view maintenance mechanisms [6, 3] or scalable query result caches [11]. These scalable view and cache mechanisms could be used as the substrate for storing our data tables and views.

Several techniques for efficiently maintaining views have been developed, including techniques for determining which base table updates require a view update [7], bringing a stale view up to date when it is queried [30], maintaining auxiliary information to aid in maintaining the view [17], minimizing communication cost necessary to maintain the views [32, 2], and applying view updates as quickly as possible to minimize view unavailability [23]. Much of this work is focused on the general view maintenance problem, where queries and view definitions can be quite complex, and often where transactional consistency of queries using views must be maintained. The main problem in our setting is determining how much of the view to materialize, not how to properly maintain a fully materialized view.

Work on how to construct [20] and maintain [12] indexes in an online manner has focused on maintaining the transactional consistency of queries that use these indexes. Our focus is instead on minimizing system load, and on the high-fanout/high-skew problem.

Temporal and stream views: Maintaining temporal views requires special handling. Yang and Widom [28] examine what auxiliary information must be stored in a data warehouse to make temporal views self-maintainable. This work focuses on "push" only to avoid querying the base data.

Our system is similar in some respects to a stream system, in that the join between producer event streams and `ConnectionNetwork` in our system is similar to a continuous window join. Babu, Munagala and Widom [5] examine which join subresults to materialize in such a setting, focusing on complex, multi-way joins. For our restricted problem (a large collection of joins of two relations), the problem is to choose which joins to materialize based on per-join relative update and consumption intervals. Other stream work has focused on how to filter pushed data updates at the source to reduce the cost of updating a downstream continuous query [21], or to pull data updates from sources when needed [18]. This work focuses primarily on aggregation queries over numeric values, and trades precision for performance. The tradeoff in our scenario is latency versus system cost, and of course, there is no aggregation in our setting.

Pub/sub: Publish/subscribe systems provide a mechanism for fanning out events from producers to consumers [31, 8], using the push strategy. It may be possible to apply our hybrid techniques to pub/sub systems.

7. CONCLUSIONS

Many popular websites are implementing a "follows" feature, where their users can follow events generated by other users, news sources or other websites. Building follows applications is inherently hard because of the high fanout of events from producers to consumers, which multiplicatively increases the load on the database system, and the high skew in event rates. We have abstracted many instances of the follows problem into a general formulation and, under assumptions about system resources that reflect common practice, shown that the best policy is to decide whether to push or pull events on a per producer/consumer basis. This technique minimizes system cost both for workloads with a high query rate and those with a high event rate. It also exposes a knob, the push threshold, that we can tune to reduce latency in return for higher system cost. These observations are validated experimentally using traces and data distributions from real social websites. Overall, our techniques provide the foundation for a general follows platform that an application developer can use to easily build a scalable, low-latency follows application.

8. REFERENCES

[1] Twitter API. http://apiwiki.twitter.com/.

[2] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.

[3] P. Agrawal et al. Asynchronous view maintenance for VLSD databases. In *SIGMOD*, 2009.

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

[5] S. Babu, K. Munagala, and J. Widom. Adaptive caching for continuous queries. In *ICDE*, 2005.

[6] M. Cafarella et al. Data management projects at Google. *SIGMOD Record*, 34–38(1), March 2008.

[7] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[8] B. Chandramouli and J. Yang. End-to-end support for joins in large-scale publish/subscribe systems. In *VLDB*, 2008.

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

[10] I. Eure. Looking to the future with Cassandra. http://blog.digg.com/?p=966.

[11] C. Garrod et al. Scalable query result caching for web applications. In *VLDB*, 2008.

[12] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, March 2006.

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

[14] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

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

[16] G. Luo. Partial Materialized Views. In *ICDE*, 2007.

[17] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. A comparison of three methods for join view maintenance in parallel RDBMS. In *ICDE*, 2003.

[18] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *TODS*, 30(1):122–173, March 2005.

[19] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience, 1990.

[20] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *SIGMOD*, 1992.

[21] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[22] C. Re and D. Suciu. Materialized views in probabilistic databases: for information exchange and query optimization. In *VLDB*, 2007.

[23] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, 2000.

[24] P. Seshadri and A. Swami. Generalized partial indexes. In *ICDE*, 1995.

[25] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[26] E. Weaver. Improving running components at Twitter. http://blog.evanweaver.com/articles/2009/03/13/-qcon-presentation/.

[27] S. Wu, J. Li, B. Ooi, and K.-L. Tan. Just-in-time query retrieval over partially indexed data on structured P2P overlays. In *SIGMOD*, 2008.

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

[29] K. Yi et al. Efficient maintenance of materialized top-k views. In *ICDE*, 2003.

[30] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, 2007.

[31] Y. Zhou, A. Salehi, and K. Aberer. Scalable delivery of stream query results. In *VLDB*, 2009.

[32] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.