Distributed Ranked Data Dissemination in Social Networks

Kaiwen Zhang[1,2], Mohammad Sadoghi[1,2,3], Vinod Muthusamy[3], Hans-Arno Jacobsen[1,2]
[1]*Middleware Systems Research Group*
[2]*University of Toronto*
[3]*IBM T.J. Watson Research Center**

***Abstract*—The amount of content served on social networks can overwhelm users, who must sift through the data for relevant information. To facilitate users, we develop and implement dissemination of ranked data in social networks. Although top-k computation can be performed centrally at the user, the size of the event stream can constitute a significant bottleneck. Our approach distributes the top-k computation on an overlay network to reduce the number of events flowing through. Experiments performed using real Twitter and Facebook datasets with 5K and 30K query subscriptions demonstrate that social workloads exhibit properties that are advantageous for our solution.**

I. Introduction

The importance of social networks is prevalent with their ever growing user base, projected to soon reach the billion mark[1]. Social networks (SN) at this scale introduce new challenges for both the underlying infrastructure and the perceived user experience.

Examples are the feeds produced and consumed by users and their friends' updates of news feeds, videos, application feeds, and location-based events, offers, and coupons.

Other sources of information are rooted in thousands of applications running on SNs which also generate content on behalf of their users (e.g., FarmVille). More recently, the linked data movement is becoming yet another source of publishing information; for instance, social tags (e.g., keyword annotations, "like" tag) on many websites is a new way of publishing data on SNs. Social tags will (soon) be enabled on billions of web pages, which are visited by millions of users daily. The generated (and delivered) amount of information is overwhelming.

In order to improve the user experience, we aim to enable users to (globally) rank their feeds and be notified only by the top-k results of interest to them. In SNs, the homepage feed contains only the most relevant stories. More importantly, many of these feeds naturally have notions of proximity and locality associated to them. For example, in location-based services, people with close proximity and similar interests (e.g., students on campus) constitute an ideal basis for joint top-k filtering. Similarly, traditional coupon distribution is also highly localized: People in the same area may receive similar offers, which constitutes a further opportunity to cluster similar interests based on location.

* The work in this paper was performed at MSRG in Toronto.

[1]As of October 2012, Facebook has over a billion active users; http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

From the infrastructure perspective, the social network must float large volumes of data between its users. From the user experience perspective, large volumes of data must be adequately (pre-)processed to be consumable by users. Thus, it is essential for the user to simplify consumption of the volumes of data, which requires the capabilities to filter and rank information and to selectively pick information. This puts the burden on service providers to offer fine-grained control over the delivered information; yet, it also brings forward new opportunities to optimize the infrastructure to reduce message traffic through the adequate placement of interest queries and information routing operations towards interested users.

The underlying infrastructure that processes these sheer volumes of data, which are prevalent in today's social network domain such as [1], [2], [3], [4], is inevitably running in a distributed computing environment. Indeed, a rising trend is to establish a decentralized architecture in social networks such as Diaspora. The pub/sub model, known for its scalability and decoupled nature, then becomes a logical candidate for the dissemination substrate of distributed social networks, and pub/sub designs for social networks have recently surfaced [11], [12]. Therefore, our goal is to provide a solution for *distributed ranked data dissemination* for pub/sub systems within the context of social networks.

Thus, in this work, we make the following contributions: (1) Present an lightweight approach to aggressively filter at the dissemination network's edges to substantially reduce the overall network traffic (Sec. II), (2) develop novel algorithms that achieve early pruning of messages that are guaranteed not to be part of a subscriber's top-k results while maintaining semantic correctness (Sec. III), (3) propose an effective buffering protocol to avoid re-sending messages that fall in the top-k windows of multiple subscribers (Sec. IV), (4) evaluate and compare the performance of our solutions vis-à-vis a baseline algorithm, offer a sensitivity analysis of the various parameters related to top-k semantics and those specific to our solution, and employ datasets from Facebook and Twitter, containing 5K and 30K subscription queries, to project the scalability of our pub/sub architecture by identifying the major properties of social network workloads that impact our solution (Sec. V).

II. Top-k semantics and correctness criterion

We develop a solution for supporting top-k data dissemination within the context of a pub/sub broker overlay network [5]. This model allows publishers to *publish* data to a *broker* which forwards the data to consumers who are *subscribed* to the data while remaining decoupled. Brokers in the pub/sub system form an overlay and collaborate to disseminate publications from publishers to the appropriate subscribers.

Subscribers define their interest by specifying a conjunction of predicates. Publications contain a set of attribute-value pairs. Those attributes are compared against the predicates of each subscription. If a publication satisfies the predicates of a given subscriber, the publication is considered *matched* and must be delivered to the subscriber.

We extend the functionalities of pub/sub systems to allow top-k filtering on a publication stream. Subscribers include a scoring function with their subscription which assigns a numerical score to an input publication. Windows of publications are derived from the stream of matching publications according to parameters provided by the subscriber. For each window, the top-k publications with the highest score are determined and delivered to the subscriber, while the rest are discarded. Windows can be either count- or time-based. A count-based window contains a fixed number of events, while a time-based window contains publications within a certain time range. The focus of this work will be on the former only.

Although the top-k stream can be computed a posteriori (at the consumers), our approach distributes the top-k computation to the upstream brokers as shown in Fig. 1(a). Top-k filtering is applied on partial streams collected upstream and disseminated towards a target subscriber. These top-k streams are smaller than the original streams; thus, reducing traffic within the system. These are then merged downstream to form the final top-k results.

With each subscription, the subscriber specifies the following top-k parameters: (1) W: the window size. W is the number of elements each window contains. (2) k: the number of highest ranking elements within a window to be delivered. (3) δ: the window shift size. δ is the number of elements (publications) to shift over for each successive window.

We suggest a correctness criterion for top-k algorithms called *stream reconstructability*: Given a set of top-k events T generated by a solution for any finite stream of events E using any window semantics WS, there exists at least one permutation (interleaving) E' of events in E possible under the specified pub/sub model where applying top-k semantics over E' using WS will produce T.

The correctness criterion states that a correct solution selects the same set of top-k events as the centralized solution, where events are all delivered to the subscriber before top-k is applied. Due to the asynchronous nature of pub/sub, several interleavings of publications exist, depending on the reliability guarantees provided by the system. Because our windows are count-based, a different permutation of publications may produce different results. For the remainder of the paper, we will focus on a model where messages in the network can be arbitrarily delayed but never lost or reordered. Per-publisher order is therefore enforced: Multiple publications from a single publisher are delivered to all interested subscribers in FIFO order. According to our correctness criterion, there exists one interleaving of publications that can be processed by the centralized solution to obtain a set of top-k results which is identical to our distributed solution. Because this correctness criterion is agnostic to the scoring function, correctness is only maintained if a distributed solution considers every window from a possible interleaving of all publications. Note that the criterion does not guarantee a specific order for the correct set of top-k events.

We show a naive distributed solution which is incorrect: Source brokers simply filter local publications and select the top publications out of each window to disseminate downstream. Non-source brokers then simply forward any received publications to the subscriber. Fig. 1(b) is a sample execution of the naive distributed algorithm for sliding ($\delta = 1$) count-based windows. Each source broker independently selects and forwards the top-k publications out of their respective windows. For instance, the top broker forwards publications $[a, d]$ for the first window and only $[b]$ for the second window (publication d is selected again but since it has already been disseminated, it is omitted). The top-k results from each source broker are delivered in a certain interleaving to the subscriber broker.

Suppose the order is $[a, d][2, 3][b][4, 5]$. Such interleaving of top-k results can be derived from a reconstructed stream of original publications as $[a, b, c, d, 1, 2, 3, 4, b, c, d, e, 2, 3, 4, 5]$. However, applying our sliding window semantics to this stream would also require results for windows such as $[b, c, d, 1]$ and $[2, 3, 4, b]$. In a naive distributed solution, such windows are not considered since each source broker considers only windows of publications originating from their own publishers. Because it is impossible to construct an interleaving of publications where there does not exist at least one sliding count-based window which includes a publication from both brokers, this solution fails the correctness criterion.

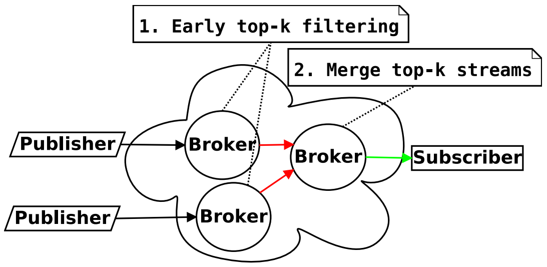
III. Distributed chunking top-k solution

Our algorithm for efficient top-k pub/sub *dissemination* consists of two key ideas: (1) Maintain correctness using *chunking* and (2) avoid sending duplicate messages using *buffering*.

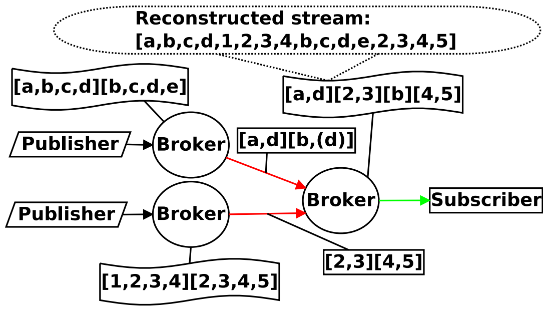
Note that efficient (centralized) top-k *computation* by a single broker is complementary to the focus of this paper and has been thoroughly studied elsewhere [6], [7]. We are concerned with the *distribution* of the top-k computation, not the actual computation itself.

Chunking: As shown in the previous section, the naive distributed solution is incorrect because it fails to consider windows which cut across publisher boundaries. In other words, when publishers are located at different brokers, the naive distributed solution will omit certain results because certain combinations of publications coming from multiple publishers are not considered.

The intuition behind our solution is to disseminate additional publications downstream such that the broker connected to the subscriber can process those missing windows.



(a) Distributed approach



(b) Naive solution example

Figure 1. Top-k matching overview

This means that every publication found in those windows must be sent downstream. Our solution is therefore hybrid in nature: Publisher edge brokers switch between full forwarding of publications necessary to maintain correctness, and dissemination of top-k results only. Additionally, the missing windows are ones which contain publications from multiple sources. We thus need a mechanism to pick an interleaving of publications which reduces the occurence of such windows.

We introduce the notion of *chunks* and *guards* to maintain correctness. Each publisher edge broker stream is divided into contiguous chunks. Every broker or subscriber can independently choose the chunk size (a parameter referred to as C).Chunks can be count- or time-based independent of the properties of the top-k window semantics.

For each chunk C, the first and last W events must be included in the chunk and serve as *guards* (left guard and right guard, respectively). For all windows of length W within the chunk, the top-k events are computed and forwarded. Publications which are not part of any top-k or guards are discarded.

The broker may forward events in a chunk as they arrive or as a whole. In either case, the downstream broker must process the events in the chunk as a whole. In particular, it must not interleave other events within those in a chunk.

Upon receipt of a chunk, an edge broker can compute the final top-k results for its subscribers through a process called *dechunking*. Publications located within a chunk (outside of the guards) are already part of processed windows upstream and are simply delivered to the subscribers. The guards, which are complete windows of publications, are processed to compute intra-chunk windows (i.e., windows which contain publications from two different chunks). Only publications which have been identified as top-k within those guards will ultimately be delivered. The example of Fig. 2 is dechunked as follows: Events from LG_1 marked as top-k events will be delivered. Then, windows w_1 to w_n will be delivered as is. Finally, top-k events will be computed and delivered for all windows with publications between RG_1 to LG_2. The process is repeated for chunk 2 and any subsequent chunks.

The correctness of our solution depends on two lemmas:

Lemma 1. *The interleaving of chunks will reconstruct a possible interleaving of events.*

Lemma 2. *All events contained in an inter-chunk window are guard events.*

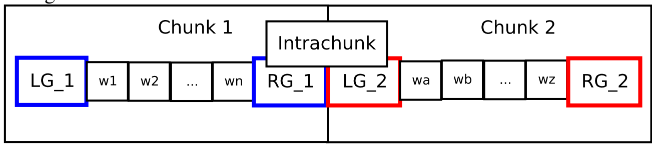


Figure 2. Chunk structure and intra-chunk overlaps

These two lemmas establish the stream reconstructability property: They demonstrate that our chunking algorithm creates valid interleavings of events, and that the guards provide enough publications to compute all the necessary windows. The proofs are included in our technical report [8]. Extensions to the chunking component, which includes support for subscription covering and adaptive rechunking (which allows intermediary brokers to recombine multiple chunks together) are also included in the report.

Buffering: Due to the heterogeneous top-k specifications of the subscriptions, it is possible for the same publication to be matched at different times for different subscribers. For instance, it is possible for a publication to be a guard event for some subscription and later be matched as the top-k of a window for another subscription.

In such cases, the same publication may be forwarded down a different path through the overlay network. However, it is possible that the paths to the different subscriptions share common brokers. Each broker is equipped with standard knowledge to understand which outgoing links a publication can *potentially* be forwarded to. We require only storage of the standard subscription set and subscription last hop. This information already exists in the pub/sub broker through the operation of pub/sub routing (e.g., [5]).

The brokers can, therefore, determine if a publication needs to be buffered to serve later requests and propagate the publication down a previously unforwarded link. Upstream brokers then simply need to signal the buffered brokers which links they now have to serve. If these signals are piggybacked on existing messages, we obtain a net reduction in traffic.

IV. Chunking and Buffering Algorithms

We now present each component in the broker processing pipeline. The chunking component has two variants: A synchronous one that queues all events in a chunk and outputs one chunk at a time and an incremental one that outputs the events with less queuing.

Matching: For each incoming event e, the matching component simply outputs a set of event-subscription pairs for each subscription s that matches e. As well, the output event is tagged as a matching event. Event tags are used in the incremental chunking algorithm.

Synchronous chunking: The synchronous chunking component, used by a publisher edge broker, outputs a chunk for a set of incoming events (Alg. 1). A chunk c has two parts: $c.sub$ refers to the subscription associated with the chunk and $c.evts$ is the sequence of events in the chunk. This component internally maintains a data structure $s.evts$ to store a sequence of events associated with subscription s. The $getFirstWindow(evts)$ and $getLastWindow(evts)$ functions return the set of events in the first and last windows in the sequence of events. $getAllWindows(evts)$ returns the set of windows, each containing W events, in the sequence of events. These functions encapsulate the window semantics. Note that the algorithm computes top-k of windows within a chunk, even those with events already included in a guard. If a publication is part of a top-k it will be tagged as top-k to ensure it will be delivered to the subscriber.

Algorithm 1: Synchronous chunking

```
1  on receive event e matching subscription s do
2      $s.evts$.add($e$) ;
3  on finish chunk for subscription s do
4      $leftguard \leftarrow \{e|s.evts.seqOf(e) \leq W\}$ ;
5      $rightguard \leftarrow \{e|s.evts.seqOf(e) > |s.evts| - W\}$ ;
6      $topk \leftarrow \emptyset$ ;
7      $\mathbb{W} \leftarrow$ getAllWindows($s.evts$) ;
8      foreach window w in $\mathbb{W}$ do
9          $topk \leftarrow topk \cup$ getTopK($w, k$) ;
10         //getTopK tags events as top-k
11     $c.sub \leftarrow s$ ;
12     $c.evts \leftarrow leftguard \cup topk \cup rightguard$ ;
13     sort($c.evts$) ;
14     forward($c$, $s.lasthop$) ;
```

Incremental chunking: The incremental chunking component performs chunking but does not wait until the chunk is complete before forwarding events (Alg. 2). The component indicates which chunk c each event belongs to, and also tags outgoing events so that downstream brokers can distinguish between the guards and top-k events. As well, it sends messages to indicate when each chunk begins and ends.

The algorithm forwards the first W events as the left guard, and when the chunk ends, the last W events are forwarded as the right guard. As each event arrives, it computes a new window (according to the window semantics) and forwards the top k events that have not already been forwarded.

Internally, the component stores in the $s.evts$ data structure the most recent two *disjoint* windows of incoming events that match subscription s. We need two windows to ensure there is one full window of events in addition to one window of right guard events in case the chunk ends soon. As well, $s.sentevts$ stores the events the components has already forwarded. $s.chunkid$ is the unique ID of the current chunk for subscription s.

Algorithm 2: Incremental chunking

```
1  on receive event e matching subscription s do
2      $s.evts$.add($e$) ;
3      if $|s.sentevts| = 0$ then
           // Start a new chunk.
4          $s.chunkid \leftarrow$ generateUniqueId() ;
5          send($\langle startchunk, s.chunkid, s\rangle, s.lasthop$) ;
6      if $|s.sentevts| < W$ then
           // e is a left guard.
7          $s.sentevts$.add($e$) ;
8          send($\langle e^{guard}, s.chunkid\rangle, s.lasthop$) ;
9      else if getNumDisjointWindows(s.evts) $< 2$ then
           // Wait for more events.
10     else
           // Forward top-k events.
11         $w \leftarrow$ getFirstWindow($s.evts$) ;
12         $topk \leftarrow$ getTopK($w, k$) ;
13         foreach event $e' : e' \in topk \wedge e' \notin s.sentevts$ do
14             $s.sentevts$.add($e'$) ;
15             send($\langle e'^{topk}, s.chunkid\rangle, s.lasthop$) ;
           // Expire events.
16         $w' \leftarrow$ getLastWindow($s.evts$) ;
17         foreach event $e' : e' \in w \wedge e' \notin w'$ do
18             $s.evts$.remove($e'$) ;
19             $s.sentevts$.remove($e'$) ;
20 on finish chunk for subscription s do
21     $w \leftarrow$ getLastWindow($s.evts$) ;
22     foreach event $e : e \in w$ do
23         send($\langle e^{guard}, s.chunkid\rangle, s.lasthop$) ;
24     send($\langle endchunk, s.chunkid\rangle, s.lasthop$) ;
25     $s.chunks \leftarrow s.chunks \setminus \mathbb{C}$ ;
26     $s.evts \leftarrow \emptyset$ ;
27     $s.sentevts \leftarrow \emptyset$ ;
```

Dechunking: This component, used by a broker for subscribers directly connected to it, computes the final top-k stream by concatenating chunks, computing and forwarding top-k results over guards and forwarding pre-computed top-k results.

Deduplication: The deduplication and rehydration components operate as a pair to avoid sending duplicate messages between brokers. The deduplication component operates at the tail end of an upstream broker's pipeline and essentially "compresses" the outgoing chunked event stream. The rehydration component works at the head of a downstream broker's pipeline and reconstructs the original chunks. To clarify, rehydration restores deduplicated chunks and may be performed at any broker while dechunking refers to the reconstruction of the final top-k results at the subscriber edge.

The deduplication component maintains a history of the events forwarded to each neighbour and never forwards duplicate messages over the same overlay link (Alg. 3). To ensure the downstream broker can reconstruct the chunk stream, the deduplication component forwards the IDs of the guard events; this is relatively lightweight information but must be done for each chunk. The top-k events that fall between the guards, however, can be inferred by the rehydration component at the downstream broker. The details of this algorithm are described below in the discussion about the rehydration component.

Internally, the deduplication component maintains information about the incoming chunk stream for each subscription s. In particular, $s.chunkid$ is the current chunk ID, $s.guardids$ are the event IDs of the guard events, and $s.numTopK$ is the number of top-k events in the chunk.

Algorithm 3: Deduplication

```
on receive $\langle startchunk, c, s\rangle$ do
    $s.chunkStartTime \leftarrow NOW$ ;
    $s.chunkid \leftarrow c.id$ ;
    $s.guardids \leftarrow \emptyset$ ;
    $s.numTopK \leftarrow 0$ ;
on receive event $e^{guard}$ for chunk for subscription $s$ do
    sendOnce($e^{match}, s.lasthop$) ;
    $s.guardids$.add($e.id$) ;
on receive event $e^{topk}$ for chunk for subscription $s$ do
    sendOnce($e^{match}, s.lasthop$) ;
    if $s.numTopK = 0$ then
        // Left guard is done.
        send($\langle Lguard, s.guardids, s.chunkid, s\rangle, s.lasthop$) ;
        $s.guardids \leftarrow \emptyset$ ;
    $s.numTopK \leftarrow s.numTopK + 1$ ;
on call sendOnce($e, nexthop$) do
    send($\langle Rguard, s.guardids, s.numTopK, s.chunkid, s\rangle$,
    $s.lasthop$) ;
    if $e \notin nexthop.evts$ then
        send($e^{match}, nexthop$) ;
    // Expire old events.
    $oldestChunkStartTime \leftarrow \min_s(s.chunkStartTime)$ ;
    foreach neighbour $n$ do
        $n.evts$.removeOlderThan($oldestChunkStartTime$) ;
```

Algorithm 4: Rehydration

```
on receive event $\langle e^{match}\rangle$ do
    foreach subscription $s$ that matches $e^{match}$ do
        $s.evts$.add($e$) ;
on receive $\langle Lguard, \{eId\}, c, s\rangle$ do
    send($\langle startchunk, c, s\rangle$, $s.lasthop$) ;
    foreach $id \in \{eId\}$ do
        $e \leftarrow s.evts$.get($id$) ;
        send($\langle e^{guard}, c\rangle$, $s.lasthop$) ;
        $s.evts$.removeOlderThan($e$) ;
on receive $\langle Rguard, \{eId\}, n, c\rangle$ do
    $guardevts \leftarrow \emptyset$ ;
    foreach $id \in \{eId\}$ do
        $guardevts \leftarrow guardevts \cup s.evts$.get($id$) ;
    $potentialtopkevts \leftarrow s.evts$.getOlderThan($guardevts$) ;
    $topkevts \leftarrow potentialtopkevts$.getTopRanked($n$) ;
    // Compute and send top-k events.
    ;
    foreach $e \in topkevts$ do
        send($\langle e^{topk}, c\rangle$, $s.lasthop$) ;
    foreach $e \in guardevts$ do
        send($\langle e^{guard}, c\rangle$, $s.lasthop$) ;        // Send guard
        events.
        ;
        $s.evts$.removeOlderThan($e$) ;
```

Rehydration: The rehydration component reconstructs the chunk streams (Alg. 4). For each incoming event, it first finds the matching subscriptions and records the event in a buffer associated with each subscription. This buffer is the sequence of *potential* events within each subscription's chunk. The guards within each chunk are explicitly specified by the upstream broker's deduplication component. To determine the top-k events within each chunk, it waits for the count n of top-k events within the chunk (sent within the right guard message $Rguard$ from the deduplication phase), and selects the n highest ranked events in the buffer within the guards.

V. Evaluation

We experimentally evaluate our various count-based algorithm implementations. Our experiments are divided into two parts. The first part employs a synthetic workload and contains a performance analysis of the following algorithms: (1) Baseline where all computations are performed at the subscriber's edge broker (EDGE), (2) incremental chunking (CHUNK), (3) incremental chunking with the buffering deduplication/rehydration option (B-CHUNK), with a sensitivity analysis of the last solution. The second part evaluates the B-CHUNK algorithm within the context of the online social network use case and is performed using real datasets.

A. Performance and sensitivity analysis

Setup: The algorithms are implemented in Java for the PADRES pub/sub prototype[2]. Experiments are conducted on the SciNet testbed in the General Purpose Cluster (GPC) using 24 machines[3].

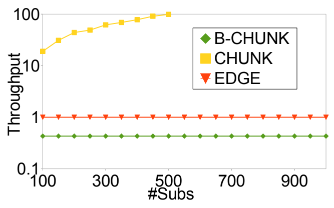
The workload used for the baseline comparison and the sensitivity analysis is synthetic: Publishers send data every 4 seconds. To maximize the top-k overhead at that scale (worst case scenario), the subscription predicates are identical and match every publication sent. However, the scoring function associated with each subscription can differ, as described below. The workload for the online social networks use case is described separately in Sec. V-D.

The overlay topology consists of several core brokers connected in a chain. In addition, these core brokers are each connected to 5 additional edge brokers. Publishers are located on edge brokers called source brokers, while subscribers are uniformly distributed on the remaining brokers. This setup models a network of data centers connected through gateways and focuses on measuring the impact of our algorithms on delivery paths with multiple broker hops.

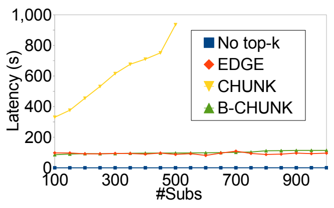
A score is assigned to each publication for each subscriber, generated using a certain distribution function. The top-k consists of the k publications with the highest scores within the current window. Since we are focused on the dissemination aspect, we minimize the overhead of the scoring function. Thus, the overhead of the scoring function is not a bottleneck in our experiments, but could be a factor if a real function was used. We also employ a deterministic scoring function (DET) which assigns strictly decreasing scores over time. This deterministic scoring function is used when we want different subscribers to assign the same score to the same publications.

Parameters: The various window related parameters, such as W, k and δ vary during the experiments. In particular, we are interested in a sliding window ($\delta = 1$) and a tumbling window ($\delta = W$). We also vary the number of subscribers, publishers, and the top-k distribution. In a uniform distribution, each publication is equally likely to be

[2]http://padres.msrg.toronto.edu/
[3]http://www.scinet.utoronto.ca/



(a) Throughput (source broker)



(b) End-to-end latency

Figure 3. Performance comparison

part of the top-k of subscription, whereas in a Zipfian distribution, the top-k of each subscription are more overlapping. Finally, we also vary the publication delay. In a *mixed* setup, publications are injected as subscribers join the system. In a *2-phase* setup (2P), subscriptions are first submitted before publications start to flow.

Metrics: *Outgoing traffic* - The main advantage of early top-k matching is to allow unselected publications to be discarded early, which reduces the amount of traffic in the system. We are therefore interested in the traffic reduction of our distributed solutions.

End-to-end latency - Although the matching overhead associated to top-k is negligible in our evaluation, publications can be deferred until they are selected as part of the top-k. This delay is dependent upon the window parameters used. We measure the end-to-end latency at the subscribers which are the furthest away from the publishers. In our topology, publishers are 6 broker hops away from the measured subscribers.

B. Performance comparison

We measure the performance of the various solutions presented over an increasing number of subscribers. Traffic throughput have been measured at different points in the network, namely at the source (publisher) broker, core brokers and edge (subscriber) brokers. We employ a single publisher and each subscriber is interested in the top-k of every publications published using a deterministic scoring function. We use $k = 1$, $W = \delta = 20$, and $C = 100$ (chunk size). These top-k parameters are applied to every subscription.

Fig. 3(a) shows the relative throughput at the publisher broker of each solution, normalized relative to the EDGE solution. This solution always forwards every publication received since no top-k processing occurs at the source. The chunking solution without buffering performs top-k filtering at the source; however, the top-k publications are sent separately for each subscription. Thus, a subscription could be sent multiple times over the same link. This duplication means the traffic increases linearly to the number of subscribers. In our experiments, the throughput of CHUNK starts at an order of magnitude higher in traffic to that of EDGE. In fact, the throughput is saturated at around 500 subscribers. With buffering enabled, the chunking solution (B-CHUNK) scales much better than the centralized solution. B-CHUNK shows a constant 57% traffic reduction with varying subscriber loads. In our setup, a large portion of the remaining traffic is used to forward guard publications.

At the subscriber brokers, the throughput results are straightforward: Each solution reduces the traffic by 95%.

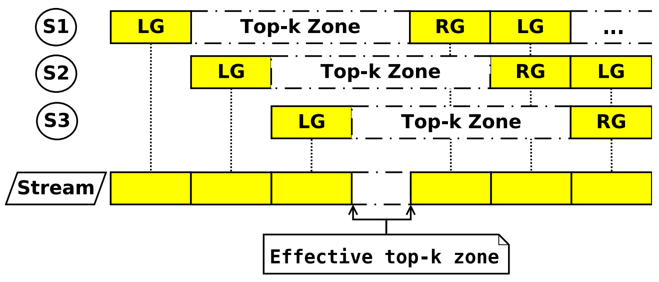


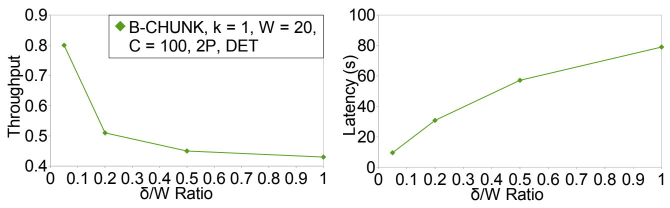
Figure 4. Chunk guards offset

Since the selectivity of the top-k filter is 1 out of 20, only 5% of the publications are ultimately sent to the clients. No further processing is required by the clients themselves: The processed stream sent by the edge broker to a client corresponds exactly to the final result. Therefore, all three solutions are able to produce that stream at the edge broker.

Compared to EDGE, core brokers will receive more inbound traffic for CHUNK, since the source broker generates more traffic. Each publication received is then forwarded down a single link to the corresponding subscriber. Thus, the outgoing traffic is exactly equal to the inbound traffic. Since core brokers have 6 outgoing edges, each publication is forwarded to only 16.67% of the links. However, the duplication of publications at the source broker causes core brokers to forward a publication down a specific link multiple times to satisfy different subscribers. Therefore, CHUNK generates more traffic at the core brokers than EDGE, even if individual inbound publications are forwarded down only a single link. For B-CHUNK, every publication received is forwarded down every link, since every subscriber uses the same scoring function. The reduction in traffic at core brokers comes only from the fact that there is less inbound traffic from the source broker.

We compare the end-to-end latency between the various solutions (see Fig. 3(b)). Because top-k is a stream processing operation, publications can sit in a queue waiting for the window to fill up. In a count-based solution, the delay experienced due to queuing can vary widely between individual publications. In our experiments, we control the variation by using a tumbling window with deterministic scoring. A single publication is sent for every 20 publications received. The sent publication is the one which entered the window first, since publications have monotonously decreasing scores.

For CHUNK, the latency is dependent on the number of publications and performs much worse than other solutions. This is because that traffic is duplicated per each subscriber. For B-CHUNK and EDGE, the latency is dominated by the queuing time (as seen by the difference when no top-k is used). The queuing time is mostly dependent upon three factors: The scoring function used, the window size, and the publication rate. All three of these factors affect both solutions equally, hence, there is no advantage in choosing one solution over another with respect to latency. Thus, the overall latency is comparable between B-CHUNK and EDGE. The difference is that B-CHUNK queues publications mostly at the publisher brokers, while EDGE queues at the subscribers' edges.



(a) Throughput (Source broker) (b) Latency

Figure 5. δ/W ratio analysis

As explained before, the scoring function used in the experiment incurs minimal overhead. Therefore, it does not factor in our latency measurement. However, the matching overhead is distributed differently depending on the solution used. In B-CHUNK, the source brokers must process a single publisher stream for multiple subscribers, while in EDGE, the opposite is true. Thus, a workload which contains more publishers than subscribers will distribute the matching overhead more evenly using B-CHUNK.

Summary: The CHUNK solution is simply not scalable due to the publication duplication issue and always performs worse than the EDGE solution in both throughput and latency even for small number of subscribers. B-CHUNK demonstrates lower traffic than EDGE due to the source filtering of top-k publications. Latency for both solutions is comparable since it is dominated by the queuing time, which is the same.

C. Sensitivity analysis

The parameters used in Sec. V-B can be considered optimal in terms of traffic reduction for the B-CHUNK algorithm. We now evaluate the impact of each parameter and justify its effect within the context of our solution.

Shift: The shift parameter has a significant impact on the throughput at the core brokers. With a tumbling window, the throughput flows at a constant rate to the subscriber: For every W publications, k publications will be sent to the subscriber. For a sliding window, the rate can vary depending on the actual score of the publications. In the best case, a publication can dominate up to $W - 1$ publications. For instance, for $k = 1$, a publication p which has a higher score than the next $W - 1$ publications following it will only trigger one message (used to forward p). It is only when that publication falls off the window (at the W^{th} publication following p) that a new top-k publication will be selected. In the worst case, a publication can be sent out for every incoming publication if the top-k is always at the head of the window. At every shift, one of the current top-k publications fall off the window and is replaced by another which must be forwarded. In this case, once the window is initially filled, maximal throughput is achieved: Every incoming publication generates an outgoing publication.

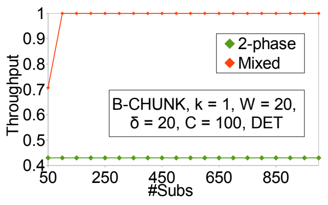
We evaluate the impact of the shift in the worst case scenario, where the scoring function used generates progressively decreasing scores. Fig. 5(a) shows the impact of the shift parameter on the throughput. With a window size of 20, the first data point corresponds to a sliding window shift of 1 (ratio of 0.05). Except for the first window of each chunk, the throughput is maximized and there is no traffic reduction in those subsequent windows. At the other extreme, the tumbling window (ratio of 1.0) provides the best traffic reduction at 43% of the baseline traffic at the source broker.

The throughput is inversely proportional to latency (see Fig. 5(b)). A smaller shift will create a more frequent publication rate, which minimizes the queuing time. A longer shift empties the window more quickly, which delays the next top-k computation until the buffer is filled. We observe that this particular tradeoff is not specific to our solution, but rather comes from the window semantics themselves. A workload with decreasing publication scores is sensitive to the throughput issue in sliding windows, but will result in shorter publication delays.

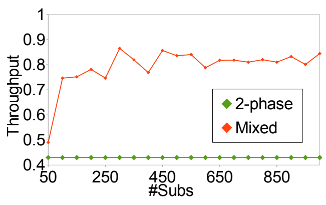
Publication delay: We evaluate B-CHUNK with varying publication delay. Publication delay refers to the time elapsed before publications are released in the system. In a *mixed* setup, publishers start their publications while subscribers join the system. This means that earlier subscribers will receive more publications than those who join later. In a *2-phase* setup, we first disseminate all the subscriptions before sending publications. Subscribers thus receive the same stream of publications when using the 2-phase setup in conjunction with the deterministic scoring function.

Publication delay has a major impact on the throughput of our B-CHUNK solution. This is due to the fact that publication forwarding is a binary decision: A broker **must** forward a publication down a certain link if at least one subscriber requires this publication at that path. For the source brokers, this means a publication can be dropped only if **all subscribers** do not require that publication. This occurs when the publication is not selected as part of any top-k window of all subscribers **and** the publication is not part of any guards of any chunks. In a mixed setup, the subscriptions will start their first chunk at different times, which correspond to the first publication each subscription matches. This means that chunks belonging to different subscriptions can offset one another such that guards for one subscription will overlap into the top-k zone of another subscription. When this happens, publications in that overlap must be forwarded by the source broker to satisfy the guard requirements, even if they are not part of any top-k window computed by other subscriptions. This is illustrated in Fig. 4. The source broker can only drop publications which are located only in the *effective top-k zone*, where no subscription is in a guard. In the worst case, a publication stream can be completely covered by guard zones, in which case every publication must be forwarded regardless of the actual top-k computations being performed. In the best case, guard zones are completely disjoint with each others' top-k zones, which maximizes the effectiveness of the top-k computations.

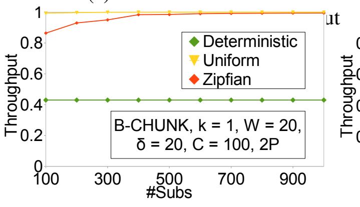
In our mixed setup, subscriptions are submitted every second (publication rate is once per 4 seconds). At the source broker, the throughput is already degraded after 50 subscribers and is saturated at 100 subscribers and more (see Fig. 6(a)). At the core brokers (Fig. 6(b)), there is stable throughput reduction of 20% at the core brokers. Since core



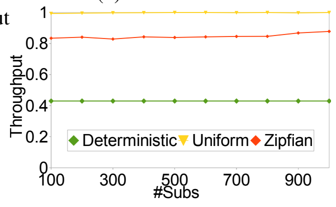
(a) Source broker



(b) Core brokers



(a) Source broker



(b) Core brokers

Figure 7. Throughput vs. scoring function

brokers have multiple outgoing edges, it is less likely for subscriptions down a certain link to offset each other. In our solution, there are 50 subscribers per outgoing edge, which is not enough to saturate the throughput at the core brokers, even in a mixed setup.

Nevertheless, our solution remains sensitive to offsets in subscription chunks due to the presence of guard zones which can nullify top-k filtering. It is therefore particularly useful to employ rechunking techniques to synchronize chunks such that their guards overlap each other, maximizing the effective top-k zones.

Scoring function: We evaluate the impact of the scoring function used. In our random selection tests, subscribers select different top-k streams according to the score they individually assign to each publication. Again due to the binary nature of publication matching, the throughput of our solution is sensitive to the scoring function used. Scoring functions that produce disjoint sets of top-k publications result in a higher throughput since publications matching at least one subscriber must be forwarded. In constrast, similar scoring functions can be leveraged to reduce traffic by forwarding publications which will match multiple top-k windows.

We simulate subscriptions with similar interests using a Zipfian scoring function. A random global score is assigned to each publication and each subscription derives its personal score for the subscription by modifying the global score with a random value chosen from a Zipfian distribution. Thus, subscriptions will have similar scores for each publication. In contrast, the uniformly random scoring function assigns an arbitrary value for each publication per subscription.

Fig. 7(a) shows the relative throughput of the various scoring functions at the source broker. The deterministic scoring function is the most efficient since all the subscribers' top-k publications are the same. Zipfian scoring provides some clustering of top-k publications, which provides a 17% traffic reduction (with respect to EDGE) at 100 subscriptions. However, the throughput saturates beyond 400 subscribers. The uniform scoring function does not provide any benefit at the source broker, even with a small number of subscriptions.

At the core brokers, we observe that the throughput stays constant with respect to the number of subscribers in all cases (see Fig. 7(b)). This is due to the fact that each outgoing link for a core broker has at most 50 subscribers. Therefore, a core broker can safely eschew forwarding of a publication down a specific link if none of those 50 subscribers require it. This is advantageous for the Zipfian scoring function, which provides a sizable reduction in traffic when the downstream link has a small number of subscribers (e.g., 50).

In general, subscriptions with similar scoring functions should be clustered together. While the source brokers will not benefit from such clustering, core brokers will as they skip forwarding down unmatched links.

Number of publishers: We increase the number of publishers and test various deployment strategies. In a *cluster* strategy, we increase the number of publishers at a single source broker, whereas in an *uniform* strategy, the publishers are uniformly distributed around the topology. We keep the rate of publications constant (15 pubs/min), regardless of the number of publishers.

We first observe that the various algorithms are not sensitive to the number of publishers with regards to throughput. Using the same setup as the performance evaluation, but varying the number of publishers, the same pattern is followed: B-CHUNK uses 43% of the original traffic at the source broker, while CHUNK performs increasingly worse as the number of subscribers increase. This pattern holds true no matter how the publishers are distributed.

This result is justified by the distributed and decoupled nature of our B-CHUNK algorithm. When multiple publishers are connected to the same source broker, all publishers' event streams are treated together as a single merged event stream for that source broker. Because the scoring function is deterministic and is based on the order of publications, the top-k computation produces the same results. When multiple publishers are connected to different brokers, each broker locally performs top-k computations independently. Thus, they will behave and perform identically.

For end-to-end delay, we find that the placement strategy used, rather than the actual number of publishers, is a major factor. Fig. 8 shows that in a clustered strategy, the latency remains constant over an increasing number of publishers. This is due to the decoupled property mentionned previously: The output events from the different publishers are merged into a single input stream for the source broker, which is no different from a single publisher publishing at their aggregated rate. When publishers are located on different source brokers (uniform strategy), the latency grows exponentially in the number of publishers. Because the publication rate is kept constant, a greater number of publishers also imply that each publisher will publish at a reduced rate. Since each publisher is publishing at a different broker, each window fills up at a slower rate, thus increasing the queuing time for publications. Furthermore, the edge broker at a subscriber's end will process one chunk at a time, originating from a single source broker. The edge broker will therefore select one chunk to process and buffer the other chunks. Since each source broker is filling its chunk concurrently, as soon as the selected chunk is completed, the edge broker

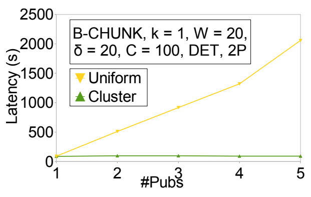


Figure 8. End-to-end latency vs. publisher placement

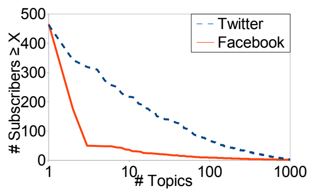


Figure 9. Social popularity tail distribution

will quickly process the completely buffered chunks before selecting a new chunk among the batch of newly formed chunks. This means the queuing time of publications in buffered chunks also includes the time required to finish the currently selected chunk. This phenomenon motivates the use of publisher clustering techniques.

Summary: The shift parameter, the publication delay and the scoring function are three major performance factors for B-CHUNK. Selecting a small shift value can lead to high throughput in certain workloads. However, one must be aware of the throughput-latency tradeoff surrounding the shift parameter. Offset chunk guards can dominate top-k filtering and saturate the throughput. Without chunk synchronization, a mixed workload will not scale as the number of subscriptions increases and the likelihood of guards offset to occur. Synchronizing chunks is necessary in order to correct the offsets. The scoring function can vary the amount of overlap between the top-k sets of the subscriptions. As the number of subscribers increase, the publications to be forwarded is likely to increase. Eventually, every publication must be forwarded by the publishers. However, clustering subscriptions with similar scoring functions in the same parts of the overlay will yield benefits for core brokers with multiple links.

With regards to latency, The chunking solutions are sensitive to the number of source brokers and not the number of publishers. As the number of source brokers increases, the number of concurrent chunks also increases, with subscribers only able to process one chunk at a time. This provides an incentive to cluster publishers to reduce the number of source brokers, as well as reducing the chunk size to allow the subscribers to switch between chunks faster.

D. *Online social networks use case*

We evaluated our solution in the context of online social networks. We employ workloads extracted from public datasets from Facebook [9] and Twitter [10].

Subscriptions: The datasets contain the social relationships between the different users in the network. In Facebook, the *friendship* relation is modeled as a pair of subscriptions which subscribes one user to another and vice-versa. In Twitter, the *follow* relation is modeled as a subscription between a *follower* to a *followee*. All these subscriptions are topic-based: Subscribers are interested in the totality of publications posted by a user. From the original datasets, we extracted a sample set contained subscriptions for 1000 users, using the technique employed in [11]. The samples retain the original properties of the original datasets, specifically with regards to the relative popularity of the topics. Fig. 9 shows that the popularity of both datasets exhibit a long tail, with the Twitter tail being broader than Facebook. The Facebook and Twitter samples contain 5K and 30K subscriptions, respectively. For each user, we combine all subscriptions together to form one subscription spanning the various topics this user is interested in. This allows our pub/sub system to perform top-k over the entire stream of matching publications for that given user, rather than compute the top-k publications over each topic separately. The intended purpose of our solution is to provide subscribers with a top-k selection of the most relevant publications among all the topics this user is subscribed to. Thus, our sample actually contains only 1K subscriptions, one for each user.

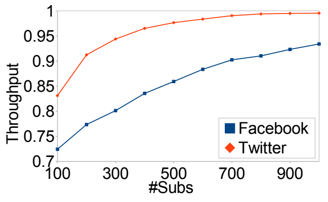
Publications: Publications are generated synthetically at a fixed rate of one per second. The publications represent *tweets* (Twitter) and *status updates* (Facebook) posted by users ie., topics, which must be disseminated to all subscribers of the user eg., followers and friends. The publications are assigned a topic selected separately for each publication. According to [13], there is a correlation between user popularity and activity. We take this into account by skewing topic selection towards popular users: The probability of a topic being selected for a publication is linearly dependent on the topic popularity.

Scoring function: Publications are scored according to the popularity of its source (namely, its topic). A publication coming from a popular topic will gather a higher score than an unpopular topic and thus be more likely to be part of the top-k scores for a window. We argue that since our sample subscriptions come from a cluster of linked users, this popularity-based scoring function is akin to collaborative filtering, which is employed in social networks.

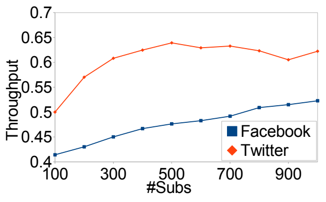
Topology: We use a topology similar to the previous experiments, with edge brokers connected to a chain of core brokers. Each edge broker is connected to a single publisher. We argue that in a real social network cluster, state updates are handled by an external service which is logically represented as a single source of data for our pub/sub system. Each edge broker is connected to a single subscriber, who is in charge of submitting subscriptions for several users. The subscriptions are allocated randomly to each subscriber. Each pub/sub subscriber logically represents a collection of users that reside on the same machine. Furthermore, even though the subscriptions are assigned randomly, the subscriptions in the sample data are closely related and follow the aforementionned long tail distribution: Subscriptions residing on the same physical machine have similar interests, which adheres to the clustering techniques prescribed by [15]. Clustering the interest of multiple users to a single subscriber is an effective technique for scaling up a pub/sub-based social network dissemination middleware to the required specifications.

Top-k parameters: We use $k = 2$, $W = 20$, $\delta = 20$ (tumbling window), and $C = 100$ (chunk size), with a 2-phase setup. The algorithm employed is B-CHUNK (chunking with deduplication/rehydration).

Performance evaluation: Fig. 10(a) shows the normalized throughput of the B-CHUNK solution over the centralized baseline for varying number of subscriptions at the



(a) Source broker



(b) Core brokers

Figure 10. Throughput vs. number of subscriptions

publisher edge broker. For Facebook, the reduction gain starts at 27.6% for 100 subscribers and deteriorates to 6.7% for 1000 subscribers, while in Twitter the reduction starts at 16.9% and drops to 0.5%. Two main properties of the workload justify these results.

First, the subscriptions have varying degrees of overlap in the publication space. This means that the windows of different subscriptions progress at different rates since they are not filled by every published event (as in our sensitivity analysis). This causes offsets in the chunks and reduces the size of the effective top-k zone, as described earlier. Due to this phenomenon, the results are worse than during the performance evaluation using synthetic data. Synchronizing guards is therefore a concern as it has practical benefits in real workloads.

Second, the skew towards popular subscriptions is beneficial to our solution. The publication workload contains a higher frequency of popular publications, which are matching a large number of subscriptions. Therefore, it is likely that for any given window of any subscriber, it contains some publications of popular topics. Due to our scoring functions, these publications are then able to "overshadow" any lower ranking publications and be selected as part of top-k. Thus, any subscription that overlap in those popular topics will likely obtain the same top-k results. This allows the publisher broker to safely filter out a large volume of publications very quickly and only keep the higher ranking ones which appear in the top-k list of a large number of subscriptions. It also follows that having a long tail, as exhibited by social network workloads, is a benefit as it allows the solution to "cut" the tail at the publisher broker and keep only a smaller core of publications. This is demonstrated in the results, where the Facebook experiments perform better than Twitter due to their narrower tails. Due to this positive effect, our results are better than the measurements obtained during the publication delay analysis, even though we are still under the chunk offset issue.

The positive gain of the popularity skew is more apparent in the core brokers performance (see Fig. 10(b)). Because the core brokers have multiple outgoing hops, the chunk offset occurs only between subscription from the same hop. The problem is therefore lessened, and we see the traffic reduction jumps 48% and 38% at 1000 subscribers for Facebok and Twitter. This means our popularity-based social network experiment outperforms the Zipfian scoring function used in the scoring sensitivity analysis and is closer to the uniform scoring performance.

Scalability tests: We also evaluate the impact of the number of brokers on traffic reduction. We extend the topology to 96 and 960 brokers while maintaining the degree of each node constant. For instance, a 960 brokers topology consist of 160 core brokers, each connected to 5 edge brokers. For the publisher edge broker, we find no impact for the number of brokers in the system. The traffic reduction is found to be the same: 0.5% for Twitter and 6.7% for Facebook at 1000 subscribers. This is because the algorithm is lightweight: The publication edge broker computes the top-k of each subscriber locally and forwards the resulting publications downstream, irrespective of the rest of the topology.

For the core brokers, we find the size of the topology to have no impact on the performance as well, other than the fact that the same number of subscriptions is now spread out on a larger number of edge brokers. Thus, the first core broker directly attached to the publisher edge broker is in charge of disseminating for all 1000 subscriptions, and will therefore exhibit traffic reduction in the range of 48% and 38% for Facebook and Twitter, as per Fig. 10(b). Core brokers towards the end of the chain have a smaller number of subscriptions to disseminate to, and reduce traffic at a higher rate.

Summary: The popularity-based scoring function and publication workload, coupled with the heavy tail distribution of subscriptions in social networks, creates large overlaps in the top-k results of the subscribers which increases the traffic reduction capabilities of our solution, especially apparent at the core brokers. On the other hand, the uneven overlaps between the subscriptions create offsets in the chunks which minimize the effective top-k zone, which indicates the solution can benefit from chunk synchronization optimizations.

The scale of the topology has no direct impact on the performance of the solution. The only benefit gained by increasing the number of brokers is to distribute the subscription load onto a larger number of subscriptions, which is the main factor for our solution. We envision our pub/sub model to scale adequately through partitioning of the social networks according to interests.

VI. Related Work

Generally speaking, the problems related to retrieving the most relevant answers have been studied in different contexts including database (distributed) top-k querying ([16], [17], [18], [19], [20]) and publish/subscribe (pub/sub) matching techniques [21], [22], [23], [24], [25], [26], [27].

The most widely adopted database top-k processing model [16], [17] differs with our proposed top-k model in an important respect: Our top-k model solves the reverse problem. In the database context, top-k querying means finding the most relevant tuples (events) for a given query (subscription). But in our pub/sub abstraction, matching means finding the relevant subscriptions (queries) for a given event (tuple). Furthermore, the database literature is best suited for lower dimensional data [17], while in pub/sub context, a dimensionality in the order of hundreds is commonplace [21], [22], [23], [24], [25], [26], [27], [7], which is orders of magnitude larger than capabilities of existing database techniques.

Broadly speaking, two classes of matching algorithms have been proposed for pub/sub: Counting-based [21], [23],

[25] and tree-based [22], [24], [27], [7] approaches. A fresh look at enhancing pub/sub matching algorithms is to leverage top-k processing techniques to improve matching performance. An early top-k model is presented in [28]; however, this model leverages a fixed and predetermined scoring function, i.e., the score for each expression is computed independent of the incoming event. In addition, this approach is an extension of R-Tree, the interval tree, or the segment tree structure; as a result, it is ideal for data with few dimensions [28]. In contrast, a scalable top-k model that supports up to thousands of dimensions while incorporating a generic scoring function, i.e., takes the event into consideration, is introduced in [25], which relies on a static and single-layered pruning structure [25]. To alleviate these challenges, a new dynamic and multi-layered pruning top-k structure is developed in [7]. However, our proposed top-k model attempts to solve a different problem, namely, distributed top-k processing; therefore, our model can leverage any of the existing top-k work as building blocks (e.g. [28], [25], [7]).

Another important aspect of pub/sub top-k matching is to explore and identify a plausible top-k semantics. Unlike in the database context, formalizing top-k semantics in pub/sub is more involved and not limited to a single interpretation [29], [6], [30]. The most widely used pub/sub top-k semantics is defined with respect to subscribers, i.e., consume-centric semantics, in which the subscription language is extended (with a scoring function) in order to rank each incoming publication (over a time- or count-based sliding window); thus, delivering only the top-k matched publications to each subscriber [29], [6], [30].

Alternatively, the top-k semantics can be defined with respect to a publisher, i.e., produce-centric semantics, which extends the publication language for ranking subscribers and delivering publications only to the top-k matched subscribers [28], [25]. Produce-centric semantics is suitable for targeted advertisement (e.g., targeting a specific demographic group) and diversified advertisement (e.g., reaching out to most eligible members within each demographic group). Finally, hybrid semantics can be foreseen such that both subscribers and publishers have control on how data is received and disseminated, respectively. In our current work, we focus primarily on the well-adopted consume-centric top-k semantics, although some of our techniques could be leveraged for both semantics.

VII. Conclusions

We have developed a distributed ranked data dissemination algorithm. The solution performs aggressive top-k filtering closer to the sources within an overlay network. The resulting top-k streams are propagated and recombined by downstream nodes. Early filtering discards low ranking data, thus saving traffic over a centralized solution at the end-user. The solution uses a chunking algorithm which switches between full forwarding of a sequence of events and selective forwarding of top-k events. This chunking algorithm ensures correctness by providing enough data to reconstruct a possible interleaving of the original event stream.

We have implemented our solution within the context of pub/sub overlay networks. Brokers closer to publishers are responsible for the top-k computation while brokers closer to subscribers recombine the chunks. Our evaluation shows that we obtain significant throughput reduction in the system compared to the centralized solution, while maintaining comparable end-to-end latency. A sensitivity analysis reveals the importance of rechunking and clustering to maximize the performance of our solution. We also show how the social network use case exhibits favorable properties for our solution in terms of scalability and efficiency.

References

[1] D. Eyers, et al., "Living in the present: on-the-fly information processing in scalable web architectures," in *CloudCP'12*.

[2] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDM Workshops'10*.

[3] M. Fan and R. Chen, "Real-time analytics streaming system at Zynga," *XLDB'12*.

[4] "Tumblr architecture - 15 billion page views a month and harder to scale than Twitter," *Big Scalability Blog'12*.

[5] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski, "The PADRES distributed publish/subscribe system." in *ICFI'05*.

[6] M. Drosou, E. Pitoura, and K. Stefanidis, "Preferential publish/subscribe," in *PersDB'08*.

[7] M. Sadoghi and H.-A. Jacobsen, "Relevance matters: Capitalizing on less (top-k matching in publish/subscribe)," in *ICDE'12*.

[8] K. Zhang, et al., "Distributed ranked data dissemination in social networks," University of Toronto, TR'12. http://msrg.org/papers/topktr12

[9] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *EuroSys'09*.

[10] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW'10*.

[11] V. Setty, et al., "Poldercast: Fast, robust, and scalable architecture for p2p pub/sub," in *Middleware'12*.

[12] B. Wong and S. Guha, "Quasar: A probabilistic publish-subscribe system for social networks," in *IPTPS'08*.

[13] B. A. Huberman, D. M. Romero, and F. Wu, "Social networks that matter: Twitter under the microscope," *First Monday'09*.

[14] A. Marian, N. Bruno, and L. Gravano, "Evaluating top-k queries over web-accessible databases," *ACM TODS'04*.

[15] J. M. Pujol, et al., "The little engine(s) that could: Scaling online social networks," in *SIGCOMM'10*.

[16] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS'01*.

[17] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM CSUR'08*.

[18] B. Babcock and C. Olston, "Distributed top-k monitoring," in *SIGMOD'03*.

[19] P. Cao and Z. Wang, "Efficient top-k query calculation in distributed networks," in *PODC'04*.

[20] S. Michel, P. Triantafillou, and G. Weikum, "Klee: a framework for distributed top-k query algorithms," in *VLDB'05*.

[21] T. Yan and H. Garcia-molina, "Index structures for selective dissemination of information under the Boolean model," *ACM TODS'94*.

[22] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *PODC'99*.

[23] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for fast pub/sub systems," *SIGMOD'01*.

[24] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," in *ICSE'01*.

[25] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina, "Indexing Boolean expressions," in *VLDB'09*.

[26] M. Fontoura, et al., "Efficiently evaluating complex Boolean expressions," in *SIGMOD'10*.

[27] M. Sadoghi et al., "BE-Tree: An index structure to efficiently match Boolean expressions over high-dimensional discrete space," in *SIGMOD'11*.

[28] A. Machanavajjhala, et al., "Scalable ranked publish/subscribe," *VLDB'08*.

[29] K. Pripužić, I. P. Žarko, and K. Aberer, "Top-k/w publish/subscribe: Finding k most relevant publications in sliding time window w," in *DEBS'08*.

[30] M. Drosou, K. Stefanidis, and E. Pitoura, "Preference-aware publish/subscribe delivery with diversity," in *DEBS'09*.