Top-k Publish-Subscribe for Social Annotation of News

Alexander Shraer Maxim Gurevich Marcus Fontoura Vanja Josifovski

{*shralex, gmax, marcusf, vanjaj*}*@google.com*

Google, Inc. Work done while the authors were at Yahoo! Inc.

ABSTRACT

Social content, such as Twitter updates, often have the quickest first-hand reports of news events, as well as numerous commentaries that are indicative of public view of such events. As such, social updates provide a good complement to professionally written news articles. In this paper we consider the problem of automatically annotating news stories with social updates (tweets), at a news website serving high volume of pageviews. The high rate of both the pageviews (millions to billions a day) and of the incoming tweets (more than 100 millions a day) make real-time indexing of tweets ineffective, as this requires an index that is both queried and updated extremely frequently. The rate of tweet updates makes caching techniques almost unusable since the cache would become stale very quickly.

We propose a novel architecture where each story is treated as a subscription for tweets relevant to the story's content, and new algorithms that efficiently match tweets to stories, proactively maintaining the top-k tweets for each story. Such *top-k pub-sub* consumes only a small fraction of the resource cost of alternative solutions, and can be applicable to other large scale content-based publish-subscribe problems. We demonstrate the effectiveness of our approach on real-world data: a corpus of news stories from Yahoo! News and a log of Twitter updates.

1. INTRODUCTION

Micro-blogging services as twitter.com are becoming an integral part of the news consumption experience on the web. With over 100 million users, Twitter often has the quickest first-hand reports of news events, as well as numerous commentaries that are indicative of the public view of the events. As such, micro-blogging services provide a good complement to professionally written stories published by news services. Recent events in North Africa illustrate the effectiveness of Twitter and other microblogging services in providing news coverage of events not covered by the traditional media [17].

Proceedings of the VLDB Endowment, Vol. 6, No. 6
Copyright 2013 VLDB Endowment 2150-8097/13/04... $ 10.00.

A popular emerging approach to combining traditional and social news content is annotating news stories with related micro-blogs such as Twitter updates. There are several technical difficulties in building an efficient system for such social news annotation. One of the key challenges is that tweets arrive in real time and in very high volume of more than 100 millions a day. As recency is one of the key indicators of relevance for tweets, news stories need to be annotated in real time. Second, large news websites have high number of news pageviews that need to be served with low latency (fractions of a second). In this context we consider a system that sustains hundreds of millions to billions of serving requests per day. Finally, there is a non-trivial number of unique stories that need to be annotated ranging in hundreds of thousands.

In this paper we propose a *top-k publish-subscribe* approach for efficiently annotating news stories with social content in real-time. To be able to cope with the high scale of updates (tweets) and story requests, we use news stories as *subscriptions*, and tweets as *published items* in a pub-sub system. In traditional pub-sub systems published items trigger subscriptions when they match a subscription's predicate. In a top-k pub-sub each subscription (story) scores published items (tweets), in our case based on the content overlap between the story and a tweet. A subscription is triggered by a new published item only if the item scores higher than the k-th top scored item previously published for this specific subscription.

For each story, we maintain the current result set of top-k items, reducing the story serving cost to an in-memory table lookup made to fetch this set. In the background, on an arrival of a new tweet, we identify the stories that this tweet is related to, and adjust their result sets accordingly. We show how top-k pub-sub makes news annotation *feasible* from efficiency standpoint for a range of scoring functions. In this paper we do not address the issue of ranking quality, however the presented system can accommodate most of the popular ranking functions, including cosine similarity, BM25, and language model scoring [3]. Moreover, our system can be used as a first phase of selecting annotation candidates, from which the final annotation set can be determined using other methods, which may, for example, take context and user preferences into account (e.g., using Machine Learning).

The pub-sub approach is more suitable for high volume updates and requests than the traditional "pull" approach, where tweets are indexed using real-time indexing and news pageview requests are issued as queries at serving time, for the following two reasons: first, due to the real-time indexing component, cached results would be invalidated very frequently; and second, due to the order-of-magnitude higher number of serving requests than tweet updates, the cost of query evaluation would be very high. The combination of these two issues would dramatically increase the cost of serving. Our evaluation shows that on average, only a very small fraction of tweets related to a story end up annotating the story, and that cache invalidation rate in a real-time indexing approach would be 3 to 5 orders of magnitude higher than actually required in this setting.

Our approach is applicable to other pub-sub scenarios, where the subscriptions are triggered not only based on a predicate match, but also on their relationship with previously seen items. Examples include content-based RSS feed subscriptions, systems for combining editorial and user generated content under high query volume, or updating cached results of "head" queries in a search engine. Even in cases when the stream of published items is not as high as in the case of Twitter, the pub-sub approach offers lower serving cost since the processing is done on arrival of the published items, while at query time the precomputed result is simply fetched from memory. Another advantage of this approach is that it allows the matching to be done off-line using more complex matching logic than in the standard caching approaches where the cache is filled by results produced by online evaluation. Top-k pub-sub has been considered previously in a similar context [16], for personalized filtering of event streams. The work presented in this paper allows for order of magnitude larger subscription sizes with orders of magnitude better processing times (Section 5 provides a detailed discussion of previous work).

Even in a pub-sub setting, there are still scalability issues with processing incoming tweets and updating annotation lists associated with news stories. Classical document retrieval achieves scale and low latency by using two families of top-k algorithms: document-at-a-time (DAAT) and term-at-a-time (TAAT). In this work we show how to adapt these algorithms to the pub-sub setting. We furthermore examine optimizations of top-k pub-sub algorithms achieving in some cases reduction of the processing time by up to 89%. The key insight that allows for this improvement is maintaining "threshold" scores the new tweets would need to meet in order to enter the current result sets of stories. Intuitively, if the upper bound on a tweet's score is below the threshold, the tweet will not enter the result set and thus we can skip the full computation of story-tweet score. Score computation is the key part of processing cost and thus by skipping a significant fraction of score computations we reduce the CPU usage and the processing time of incoming tweets accordingly. Efficiently maintaining these thresholds for ranges of stories allows applying DAAT and TAAT skipping optimizations, saving up to 95% of score computations, which results in significant reduction of processing latency.

In summary, our main contributions are as follows:

- We show how the top-k pub-sub paradigm can be used for annotating news stories with social updates in real time by indexing the news stories as subscriptions and processing tweets as published items. This approach removes the task of matching tweets with news articles from the critical path of serving the articles, allowing for efficient serving, and at the same time guarantees maximal freshness of the served annotations.
- We introduce novel algorithms for top-k pub-sub that allow for orders of magnitude larger subscriptions with significant reduction in query processing times compared to previous approaches.
- We adapt the prevalent top-k document retrieval algorithms to the pub-sub setting and demonstrate variations that reduce the processing time by up to 89%.
- Our experimental evaluation validates the feasibility of the approach over real-world size corpora of news and tweets.

The paper proceeds as follows. In the next section we formulate news annotation as a top-k publish-subscribe problem and overview the proposed system architecture. In Section 3 we describe our algorithms. Section 4 presents an experimental evaluation demonstrating the feasibility of the proposed approach and the benefit of the algorithmic improvements. Section 5 gives an overview of related approaches and discusses alternative solutions. We conclude in Section 6.

2. NEWS ANNOTATION AS PUB-SUB

2.1 Annotating news stories with tweets

We consider a news website serving a collection $\mathcal{S}$ of news stories. A story served at time t is annotated with the set of k most relevant social updates (tweets) received up to time t. Formally, given the set $\mathcal{U}^t$ of updates at serving time t, story s is annotated with a set of top-k updates $\mathcal{R}_s^t$ (we omit superscripts t when clear from the context) according to the following scoring function:

$$\texttt{score}(s,u,t) \triangleq \texttt{cs}(s,u)\cdot\texttt{rs}(t,t_u),$$

where $\texttt{cs}$ is a content-based score function, $\texttt{rs}$ is a recency score function, and t_u is the creation time of update u. In general, we assume $\texttt{cs}$ to be from a family of state of the art IR scoring functions such as cosine similarity or BM25, and $\texttt{rs}$ to monotonically decrease with $t - t_u$, at the same rate for all tweets. We say that tweet u is *related* to story s if $\texttt{cs}(s,u) > 0$.

Content-based score. In this work we consider two popular IR relevance functions: cosine similarity and BM25. We adopt a variant of cosine similarity similar to the one used in the open-source Lucene[1] search engine:

$$\texttt{cs}(s,u) \;=\; \sum_i u_i \cdot idf^2(i)\cdot\sqrt{\frac{s_i}{|s|}},$$

where s_i (resp. u_i) is the frequency of term i in the content of s (resp. u), $|s|$ is the length of s, and $idf(i) = 1+\log(\frac{|\mathcal{S}|}{1+|\{s\in\mathcal{S}|s_i>0\}|})$ is the inverse document frequency of i in $\mathcal{S}$. With slight abuse of notation we refer to the score contribution of an individual term u_i by $\texttt{cs}(s,u_i)$, e.g., in the above function $\texttt{cs}(s,u_i) = u_i\cdot idf^2(i)\cdot\sqrt{\frac{s_i}{|s|}}$.

The BM25 score function is defined as follows:

$$\texttt{cs}(s,u) \;=\; \sum_i u_i\cdot idf(i)\cdot\frac{s_i\cdot(k_1+1)}{s_i+k_1\cdot(1-b+b\cdot\frac{|s|}{\text{avg}_{s\in\mathcal{S}}|s|})},$$

where k_1 and b are parameters of the function (typical values are $k_1 = 2, b = 0.75$).

[1]lucene.apache.org

While these are simplistic scoring functions, they are based on query-document overlap and can be implemented as dot products similarly to other popular scoring functions, and incurring a similar runtime cost. Designing a high-quality scoring function for matching tweets to stories is beyond the scope of this paper. We note that these scoring functions can be used in first phase retrieval, producing a large set of annotation candidates, after which a second phase may employ an arbitrary scoring function (based, for example, on Machine Learning) to produce the final ordering of results and determine the annotations to be displayed.

Recency score. Social updates like tweets are often tied (explicitly or implicitly) to some specific event, and their relevance to current events declines as time passes. In our experiments we thus discount scores of older tweets by a factor of 2 every time-interval τ (a parameter), i.e., use exponential decay recency score:

$$\mathtt{rs}(t_u, t) = 2^{\frac{t_u - t}{\tau}}.$$

Although we consider the above exponential decay function, our algorithms can support other monotonically decreasing functions. Note, however, that the efficient score computation method described in Section 2.4 may not be applicable to some functions.

2.2 A top-k pub-sub approach

We focus on high-volume websites serving millions to billions daily pageviews. The typical arrival rate of tweets is 100 millions a day, while new stories are added at the rate of thousands to tens of thousands a day. We focus on annotating the "head" stories that get the majority of pageviews; these are typically new stories describing recent events, published during the current day or the few preceding days. It is these stories whose annotation has to be updated frequently as new related tweets arrive.

Pageviews are by far the most frequent events in the system. We are thus looking for a scalable solution that would do as little work as possible on each pageview. It therefore makes sense to maintain the current, up-to-date, annotations (sets of tweets) for each story. Let $\mathcal{R}_s$ be the set of up-to-date top-k tweets for a story $s \in \mathcal{S}$ (i.e., at time t the top-k tweets from $\mathcal{U}^t$). For each arriving tweet we identify the stories it should annotate, and add the tweet to these stories' result sets. On pageviews, the precomputed annotations $\mathcal{R}_s$ are fetched with no additional processing overhead.

The architecture we propose is described in Figure 1. The Story Index is the main component we develop in this paper. It indexes stories in $\mathcal{S}$, is "queried by" the incoming tweets, and updates the current top-k tweets $\mathcal{R}_s$ for each story. We describe the Story Index in detail in the following sections. A complementary Tweet Index can be maintained and used to initialize annotations of new stories that are being added to the system. Such initialization (and hence the Tweet index) is optional and one may prefer to consider only newly arriving tweets for annotations. We note, however, that initializing the list of annotations sets a "higher bar" for newly incoming tweets, which is beneficial to the user as well as allows for faster queries of $\mathcal{S}$ (optimizations presented later in this section allow skipping the evaluation of an incoming tweet against stories for which the tweet cannot improve the annotation set).

We note that our approach implements a content-based publish-subscribe system where stories are standing subscriptions on tweets. It is referred to as *push* pub-sub system, where updates are proactively computed and pushed to the subscribers (precomputed annotations in our case). For a more detailed discussion of previously proposed content-based pub-sub systems, see Section 5.2.

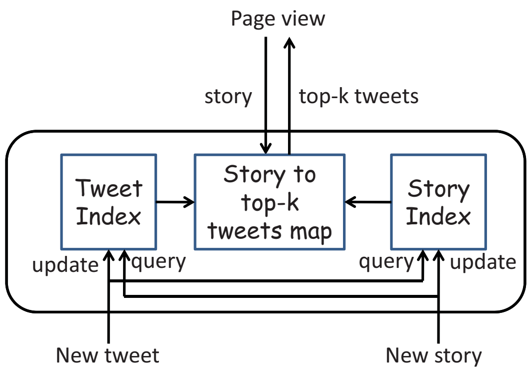


Figure 1: A pub-sub based solution.

This design has three main scenarios: (1) every new tweet is used as a query for the Story Index and, for every story s, if it is part of the top-k results for s, we add it to $\mathcal{R}_s$. We also add the new tweet to the Tweet Index; (2) for every new story we query the Tweet Index and retrieve the top-k tweets, which are used to initialize $\mathcal{R}_s$. We also add the new story to the Story Index; (3) for every page view we simply fetch the top-k set of tweets $\mathcal{R}_s$.

The major advantages of this solution are the following: (1) the Story Index is queried frequently, but it is updated infrequently; (2) for the Tweet Index, the opposite happens - it is updated frequently but queried only for new stories which are orders of magnitude less frequent than the number of tweet updates; (3) the page views, which are the most frequent event, are served very efficiently since we only need to return the precomputed set of tweets $\mathcal{R}_s$.

2.3 The Story Index

The main idea behind the Story Index is to index stories *instead* of tweets, and to run tweets as queries on that index. Inverted indices is one of the most popular data structures for information retrieval. The content of the documents (stories in our case) is indexed in an inverted index structure, which is a collection of *posting lists* $L_1, L_2, \ldots, L_m$, typically corresponding to terms (or, more generally, features) in the story corpus. A list L_i contains postings of the form $\langle s, \mathtt{ps}(s,i)\rangle$ for each story that contains term i, where s is a story identifier and $\mathtt{ps}(s,i) \triangleq \frac{\mathtt{cs}(s,u_i)}{u_i}$ is the *partial score* — the score contribution of term i to the full score $\mathtt{cs}(s,\cdot)$. For example, for cosine similarity, $\mathtt{ps}(s,i) = idf^2(i) \cdot \sqrt{\frac{s_i}{|s|}}$. The factor u_i multiplies the partial score at the evaluation time giving $\mathtt{cs}(s,u_i)$. Postings in each list are sorted by ascending story identifier. Given a query (in our case a social update) u, a scoring function $\mathtt{cs}$, and k, a typical IR retrieval algorithm, shown in Algorithm 1, traverses the inverted index of the corpus $\mathcal{S}$ and returns the top-k stories for u, that is, the stories in $\mathcal{S}$ with the highest value of $\mathtt{cs}(s,u)$.

Note that the above described semantics is different from what we want to achieve. We do not want to find the top-k stories for a given tweet, but rather *all* stories for which the

Algorithm 1 Generic top-k retrieval algorithm

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: Number of results $k$
4: Output: $R$ – min-heap of size $k$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$
6: $R \leftarrow \emptyset$
7: for every story $s \in \bigcup L_i$ do
8:     Attempt inserting $(s, \texttt{cs}(s,u))$ into $R$
9: return $R$
```

tweet is among *their* top-k tweets. This difference precludes using off-the-shelf retrieval algorithms.

Algorithm 2 shows the top-k pub-sub semantics. Given a tweet u and the current top-k sets for all stories $\mathcal{R}_{s_1}, \ldots, \mathcal{R}_{s_n}$, the tweet u must be inserted into all sets for which u ranks among the top-k matching tweets. (Here we ignore the recency score $\texttt{rs}$ and handle it in Section 2.4).

Algorithm 2 Generic pub-sub based algorithm

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$ – min-heaps of size $k$ for all
   stories in $\mathcal{S}$
4: Output: Updated min-heaps $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$
6: for every story $s \in \bigcup L_i$ do
7:     Attempt inserting $(u, \texttt{cs}(s,u))$ into $\mathcal{R}_s$
8: return $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
```

2.4 The recency function

Until now we focused on content-based scoring only and ignored the recency score. Recall that our recency score function $\texttt{rs}(t_u, t) = 2^{\frac{t_u - t}{\tau}}$, decays exponentially with the time gap between the creation time of tweet t_u and the pageview time t. It is easy to see that this function satisfies the following invariant:

OBSERVATION 2.1. *As t grows, the relative ranking between the scores of past tweets does not change.*

The above invariant means that we do not need to recompute scores and rerank tweets in $\mathcal{R}_s$ between updates caused by new tweets.

However, it might seem that whenever we attempt to insert a new tweet into $\mathcal{R}_s$, we have to recompute scores of tweets that are already in $\mathcal{R}_s$ in order to be able to compare these scores to the score of the new tweet. Fortunately, this recomputation can also be avoided by writing the recency score as

$$\texttt{rs}(t_u, t) = \frac{2^{t_u/\tau}}{2^{t/\tau}},$$

and noting that the denominator $2^{t/\tau}$ depends only on the current time t, and at any given time is equal for all tweets and all stories. Thus, and since we do not use absolute score values beyond relative ranking of tweets, we can replace $2^{t/\tau}$ with constant 1, giving the following recency function:

$$\texttt{rs}(t_u) = 2^{t_u/\tau}.$$

The above function depends only on the creation time of the tweet and thus does not have to be recomputed later when we attempt to insert new tweets.[2]

To detach accounting for the recency score from the retrieval algorithm, when a new tweet arrives we compute its $\texttt{rs}(t_u)$ and use it as a multiplier of term weights in the tweet's query vector u, i.e., we use $2^{t_u/\tau} \cdot u$ to query the inverted index. Clearly, when computing the tweet's content-based score $\texttt{cs}$ with such a query vector, we get the desired final score:

$$\begin{aligned} \texttt{cs}(s, 2^{t_u/\tau} \cdot u) &= \sum_i 2^{t_u/\tau} \cdot \texttt{cs}(s, u_i) = \\ 2^{t_u/\tau} \cdot \texttt{cs}(s, u) &= \texttt{score}(s, u, t). \end{aligned}$$

3. RETRIEVAL ALGORITHMS FOR TOP-K PUB-SUB

In this section we show an adaptation of several popular top-k retrieval strategies to the pub-sub setting, and then evaluate their performance empirically in Section 4. Although top-k retrieval algorithms were evaluated extensively, [8, 23, 18, 7, 22] to name a few, the different setting we consider necessitates a separate evaluation.

We first describe an implementation of the pub-sub retrieval algorithm (Algorithm 2) using the term-at-a-time strategy (TAAT).

3.1 TAAT for pub-sub

In term-at-a-time algorithms, posting lists corresponding to query terms are processed sequentially, while accumulating the partial scores of all documents encountered in the lists. After traversing all the lists, the accumulated scores are equal to the full query-document scores ($\texttt{cs}(s,u)$); documents that did not appear in any of the posting lists have zero score.

A top-k retrieval algorithm then picks the k documents with highest accumulated scores and returns them as query result. In our setting, where query is a tweet and documents are stories, the new tweet u may end up being added to $\mathcal{R}_s$ of any story s for which $\texttt{score}(s,u,t) > 0$. Thus, instead of picking the top-k stories with highest scores, we attempt to add u into $\mathcal{R}_s$ of all stories having positive accumulated score, as shown in Algorithm 3, where μ_s denotes the minimal score of a tweet in $\mathcal{R}_s$ (recall that u_i denotes the term weight of term i in tweet u).

3.2 TAAT for pub-sub with skipping

An optimization often implemented in retrieval algorithms is skipping some postings or the entire posting lists when the scores computed so far indicate that no documents in the skipped postings can make it into the result set. One such optimization was proposed by Buckley and Lewit [8]. Let $\texttt{ms}(L_i) = \max_s \texttt{ps}(s,i)$ be the maximal partial score in list L_i. The algorithm of Buckley&Lewit sorts posting lists in the descending order of their maximal score, and processes them sequentially until either exhausting all lists or satisfying an early-termination condition, in which case the

[2]Since the scores would grow exponentially as new tweets arrive, scores may grow beyond available numerical precision, in which case a pass over all tweets in all $\mathcal{R}_s$ is required, subtracting a constant from all values of t_u and recomputing the scores.

Algorithm 3 TAAT for pub-sub

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$ – min-heaps of size $k$ for all stories in $\mathcal{S}$
4: Output: Updated min-heaps $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$, in the descending order of their maximal score
6: $A[s] \leftarrow 0$ for all $s$ – Accumulators vector
7: for $i \in [1, 2, \ldots, |u|]$ do
8:     for $\langle s, \mathtt{ps}(s, i) \rangle \in L_i$ do
9:         $A[s] \leftarrow A[s] + u_i \cdot \mathtt{ps}(s, i)$
10: for every $s$ such that $A[s] > 0$ do
11:     $\mu_s \leftarrow$ min. score of a tweet in $\mathcal{R}_s$ if $|\mathcal{R}_s| = k$, 0 otherwise
12:     if $\mu_s < A[s]$ then
13:         if $|\mathcal{R}_s| = k$ then
14:             Remove the least scored tweet from $\mathcal{R}_s$
15:         Add $(u, A[s])$ to $\mathcal{R}_s$
16: return $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
```

remaining lists are skipped and the current top-k results are returned. The early-termination condition ensures that no documents other than the current top-k can make it into the true top-k results of the query. This condition is satisfied when the k-th highest accumulated score is greater than the upper bound on the scores of other documents that are currently not among the top-k ones, calculated as the (k+1)-th highest accumulated score plus the sum of maximal scores of the remaining lists. More formally, let the next list to be evaluated be L_i, and denote by A_k the k-th highest accumulated score. Then, lists $L_i, L_{i+1}, \ldots, L_{|u|}$ can be safely skipped if

$$A_k \; > \; A_{k+1} + \sum_{j \geq i} u_j \cdot \mathtt{ms}(L_j).$$

In our setting, since we are not interested in top-k stories but in top-k tweets for each story, we cannot use the above condition and develop a different condition suitable to our problem. In order to skip list L_i, we have to make sure that tweet u will not make it into $\mathcal{R}_s$ of any story s in L_i. In other words, the upper bound on the score of u has to be below μ_s for every $s \in L_i$:

$$A_1 + \sum_{j \geq i} u_j \cdot \mathtt{ms}(L_j) \;\; \leq \;\; \min_{s \in L_i} \mu_s. \qquad (1)$$

When this condition does not hold, we process L_i as shown in Algorithm 3, lines 8-9. When it holds, we can skip list L_i and proceed to list L_{i+1}, check the condition again and so on. Note that such skipping makes some accumulated scores inaccurate (lower than they should be). Observe however, that these are scores of exactly the stories in L_i that we skipped because tweet u would not make it into their $\mathcal{R}_s$ sets even with the full score. Thus, making the accumulated score of these stories lower does not change the outcome of the algorithm.

3.2.1 Efficient fine-grained skipping

Although Condition 1 allows us to skip the whole list L_i, it is less likely to hold for longer lists, while skipping such lists is what could make the bigger difference for the evaluation time. Even a single story with $\mu_s = 0$ at the middle of a list would prevent skipping that list. We thus resort to a more fine-grained skipping strategy: we skip a segment of a list *until* the first story that violates Condition 1, i.e., first s in L_i for which $A_1 + \sum_{j \geq i} u_j \cdot \mathtt{ms}(L_j) > \mu_s$. We then process that story by updating its score in the accumulators (line 9 in Algorithm 3), and then again look for the *next* story in the list that violates the condition. We thus need a primitive $\mathtt{next}(L_i, pos, UB)$ that given a list L_i, a starting position *pos* in that list, and the value of $UB = A_1 + \sum_{j \geq i} u_j \cdot \mathtt{ms}(L_j)$, returns the next story s in L_i such that $A_1 + \sum_{j \geq i} u_j \cdot \mathtt{ms}(L_j) > \mu_s$.

Note that $\mathtt{next}(L_i, pos, UB)$ has to be more efficient than just traversing the stories in L_i and comparing their μ_s to UB, as this would take the same number of steps as the original algorithm would perform traversing L_i. We thus use a tree-based data structure for each list L_i that supports two operations: $\mathtt{next}(pos, UB)$ corresponding to the next primitive defined above, and $\mathtt{update}(s, \mu_s)$ that updates the data structure when μ_s of a story s in L_i changes. Specifically, for every posting list L_i we build a balanced binary tree $\mathcal{I}_i$ where leafs represent the postings $s_1, s_2, \ldots, s_{|L_i|}$ in L_i and store their corresponding μ_s values. Each internal node n in $\mathcal{I}_i$ stores $n.\mu_s$, the minimum μ value of its sub-tree. The subtree rooted at n contains postings with indices in the range *n.range_start* to *n.range_end*, and we say that n is *responsible* for these indices. Figure 2 shows a possible tree $\mathcal{I}_i$ for L_i with five postings.

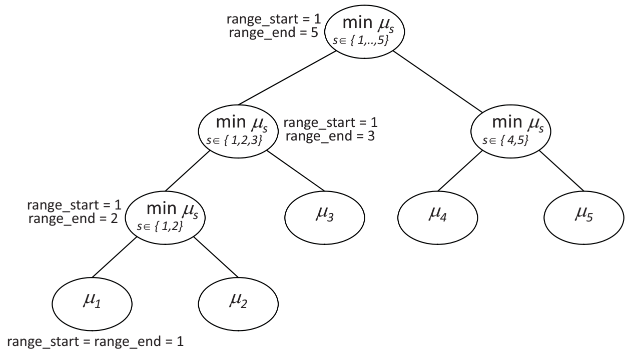

range_start = 1
range_end = 5
min μ_s $s \in \{1,..,5\}$
min μ_s $s \in \{1,2,3\}$
range_start = 1
range_end = 3
min μ_s $s \in \{4,5\}$
range_start = 1
range_end = 2
min μ_s $s \in \{1,2\}$
μ_3
μ_4
μ_5
μ_1
μ_2
range_start = range_end = 1

Figure 2: Example of a tree $\mathcal{I}_i$ representing a list L_i with 5 postings.

Algorithm 4 presents the pseudo-code for $\mathtt{next}(pos, UB)$ on a tree $\mathcal{I}_i$. It uses a recursive subroutine `findMaxInterval`, which gets a *node* as a parameter (and *pos* and *UB* as implicit parameters) and returns *endIndex* — the maximal index of a story s in L_i which appears at least in position *pos* in L_i and for which $\mu_s \geq UB$ (this is the last story we can safely skip). If *node.μ* $> UB$ (line 9), all stories in the sub-tree rooted at *node* can be skipped. Otherwise, we check whether *pos* is smaller than the last index for which *node*'s left child is responsible (line 12). If so, we proceed by finding the maximal index in the left subtree that can be skipped, by invoking `findMaxInterval` recursively with *node*'s left child as the parameter. If the maximal index to be skipped is not the last in *node*'s left subtree (line 14) we surely cannot skip any postings in the right subtree. If all postings in the left subtree can be skipped, or in case *pos* is bigger than all indices in *node*'s left subtree, the last posting to be skipped may be in *node*'s right subtree. We

Algorithm 4 Pseudo-code for operation `next` of tree $\mathcal{I}_i$

```
1: Input: $pos \in [1, |L_i|]$
2: Input: $UB$
3: Output: $\texttt{next}(L_i, pos, UB)$
4: $endIndex \leftarrow \texttt{findMaxInterval}(\mathcal{I}_i.root)$
5: if $(endIndex = |L_i|)$ return $\infty$  //skip remainder of $L_i$
6: if $(endIndex = \perp)$ return $pos$ //no skipping is possible
7: return $endIndex + 1$

8: procedure findMaxInterval($node$)
9:    if $(node.\mu > UB)$ return $node.range\_end$
10:   if (isLeaf($node$)) return $\perp$
11:   $p \leftarrow \perp$
12:   if $(pos \leq node.left.range\_end)$ then
13:      $p \leftarrow \texttt{findMaxInterval}(node.left)$
14:      if $(p < node.left.range\_end)$ return $p$
15:   $q \leftarrow \texttt{findMaxInterval}(node.right)$
16:   if $(q \neq \perp)$ return $q$
17:   return $p$
```

therefore proceed by invoking `findMaxInterval` with node's right child as the parameter.

In case no skipping is possible, the top-level call to `findMax-Interval` returns $\perp$, and `next` in turn returns *pos*. If `findMax-Interval` returns the last index in L_i, `next` returns ∞, indicating that we can skip over all postings in L_i. Otherwise, any position *endIndex* returned by `findMaxInterval` is the last position that can be skipped, and thus `next` returns $endIndex + 1$.

Although `findMaxInterval` may proceed by recursively traversing both the left and the right child of *node* (in lines 13 and 15, respectively), observe that the right sub-tree is traversed only in two cases: 1) if the left sub-tree is not traversed, i.e., the condition in line 12 evaluates to false, or 2) if the left child is examined but the condition in line 9 evaluates to true, indicating that the whole left sub-tree can be safely skipped. In both cases, the traversal may examine the left child of a node, but may not go any deeper into the left sub-tree. Thus, it is easy to see that $\texttt{next}(pos, UB)$ takes $O(\log |L_i|)$ steps. $\texttt{update}(s,\ \mu_s)$ is performed by finding the leaf corresponding to s and updating the μ values stored at each node in the path from this leaf to the root of $\mathcal{I}_i$.

In order to minimize memory footprint, we use a standard technique to embed such a tree into an array of size $2|L_i|$. We optimize further by making each leaf in $\mathcal{I}_i$ responsible for a range of l consecutive postings in L_i (instead of a single posting) and use the lowest μ_s of a story in this range as the value stored in the leaf. While this modification slightly reduces the number of postings the algorithm skips, it reduces the memory footprint of trees by a factor of l and the lookup complexity by $O(\log l)$, thus being overall beneficial. We did not investigate methods for choosing the optimum value of l but found experimentally that setting l between 32 and 1024 (depending on the index size) results in an acceptable memory-performance tradeoff. Algorithm 5 maintains a set $\mathcal{I}$ of such trees, consults it to allow skipping over intervals of posting list as described above, and updates the affected trees once μ_s for some s changes. Note that when such change occurs, we must update all trees which contain s (Algorithm 6 lines 9 and 10). Enumerating these trees is equivalent to maintaining a forward index whose size is of the same order as the size of the inverted index of $\mathcal{S}$.

Algorithm 5 Skipping TAAT for pub-sub

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$ – min-heaps of size $k$ for all stories in $\mathcal{S}$
4: Output: Updated min-heaps $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$, in the descending order of their maximal score
6: Let $\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{|u|}$ be the trees for the posting lists
7: $A[s] \leftarrow 0$ for all $s$ – Accumulators vector
8: for $i \in [1, 2, \ldots, |u|]$ do
9:    $UB \leftarrow A_1 + \sum_{j \geq i} u_j \cdot \texttt{ms}(L_j)$
10:   $pos \leftarrow \mathcal{I}_i.\texttt{next}(1, UB)$
11:   while $pos \leq |L_i|$ do
12:      $\langle s, \texttt{ps}(s, i) \rangle \leftarrow$ posting at position $pos$ in $L_i$
13:      $A[s] \leftarrow A[s] + u_i \cdot \texttt{ps}(s, i)$
14:      $pos \leftarrow \mathcal{I}_i.\texttt{next}(pos, UB)$
15: for every $s$ such that $A[s] > 0$ do
16:    processScoredResult($s, u, A[s], \mathcal{R}_s, \mathcal{I}$)
17: return $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
```

Algorithm 6 A procedure that attempts to insert a tweet u into $\mathcal{R}_s$ and updates trees

```
1: Procedure processScoredResult($s, u, score, \mathcal{R}_s, \mathcal{I}$)
2: $\mu_s \leftarrow$ min. score of a tweet in $\mathcal{R}_s$ if $|\mathcal{R}_s| = k$, 0 otherwise
3: if $\mu_s < score$ then
4:    if $|\mathcal{R}_s| = k$ then
5:       Remove the least scored tweet from $\mathcal{R}_s$
6:    Add $(u, score)$ to $\mathcal{R}_s$
7:    $\mu'_s \leftarrow$ min. score of a tweet in $\mathcal{R}_s$ if $|\mathcal{R}_s| = k$, 0 otherwise
8:    if $\mu'_s \neq \mu_s$ then
9:       for $j \in$ terms of $s$ do
10:         $\mathcal{I}_j.\texttt{update}(s,\ \mu'_s)$
```

To increase skipping we use an optimization of ordering story ids in the ascending order of their μ_s. This reduces the chance of encountering a "stray" story with low μ_s in a range of stories with high μ_s in a posting list, thus allowing longer skips. Such a (re)ordering can be performed periodically, as μ_s of stories change. We do not further explore this optimization and in our evaluation we ordered stories only once at the beginning of the evaluation.

3.3 DAAT for pub-sub

Document-at-a-time (DAAT) is an alternative strategy where the current top-k documents are maintained as min-heap, and each document encountered in one of the lists is fully scored and considered for insertion to the current top-k. Algorithm 7 traverses the posting lists in parallel, while each list maintains a "current" position. We denote the current position in list L by $L.curPosition$, the current story by $L.cur$, and the partial score of the current story by $L.curPs$. The current story with the lowest id is picked, scored and the lists where it was the current story are advanced to the next posting. The advantage compared to TAAT is that there is no need to maintain a potentially large set of partially accumulated scores.

Algorithm 7 DAAT for pub-sub

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$ – min-heaps of size $k$ for all stories in $\mathcal{S}$
4: Output: Updated min-heaps $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$
6: for $i \in [1, 2, \ldots, |u|]$ do
7:     Reset the current position in $L_i$ to the first posting
8: while not all lists exhausted do
9:     $s \leftarrow \min_{1 \le i \le |u|} L_i.cur$
10:    $score \leftarrow 0$
11:    for $i \in [1, 2, \ldots, |u|]$ do
12:        if $L_i.cur = s$ then
13:            $score \leftarrow score + u_i \cdot L_i.curPs$
14:            Advance by 1 the current position in $L_i$
15:    $\mu_s \leftarrow$ min. score of a tweet in $\mathcal{R}_s$ if $|\mathcal{R}_s| = k$, 0 otherwise
16:    if $\mu_s < score$ then
17:        if $|\mathcal{R}_s| = k$ then
18:            Remove the least scored tweet from $\mathcal{R}_s$
19:        Add $(u, score)$ to $\mathcal{R}_s$
20: return $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
```

3.4 DAAT for pub-sub with skipping

Similarly to TAAT algorithms, it is possible to skip postings in DAAT as well. One of the popular algorithms is WAND [7]. In each iteration it orders posting lists in the ascending order of the current document id and looks for the *pivot* list – the first list L_i such that the sum of the maximal scores in lists $L_1, \ldots, L_{i-1}$ is below the lowest score θ in the current top-k:

$$\sum_{j<i} u_j \cdot \mathtt{ms}(L_j) \ \le\ \theta.$$

Then, if the current document in the pivot list – the *pivot document* – equals to the current document in list L_1, the pivot document is scored and considered for insertion into the current top-k. Otherwise, the current positions in lists $L_1, \ldots, L_{i-1}$ are *skipped* to a document id greater than or equal to the pivot document. This skipping is possible since by the ordering of the lists, and by definition of the pivot list, the maximal score of the documents with ids lower than that of the pivot document is below θ.

Similarly to our adaptation of Buckley&Lewit's algorithm for the pub-sub setting (Section 3.2), we modify WAND's skipping condition and skip only stories s in list L_i for which:

$$\sum_{j\le i} u_j \cdot \mathtt{ms}(L_j) \ \le\ \mu_s. \tag{2}$$

In Algorithm 8 we again make use of the tree-based technique described in Section 3.2.1 to efficiently find for every list L_i the first story from the current position in L_i onward that violates Condition 2. From the set of these stories we choose the *pivot story* to be the minimal according to story id. The list containing the pivot story is said to be the *pivot list.* Then, as in the regular WAND, the pivot story is either scored and the processed tweet u is considered for insertion to $\mathcal{R}_s$, or the lists are skipped to a story greater than or equal to the pivot story. Algorithm 6 is used to update $\mathcal{R}_s$ and the affected trees.

Algorithm 8 Skipping DAAT for pub-sub

```
1: Input: Index of $\mathcal{S}$
2: Input: Query $u$
3: Input: $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$ – min-heaps of size $k$ for all stories in $\mathcal{S}$
4: Output: Updated min-heaps $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
5: Let $L_1, L_2, \ldots L_{|u|}$ be the posting lists of terms in $u$
6: Let $\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{|u|}$ be the trees for the posting lists
7: for $i \in [1, 2, \ldots, |u|]$ do
8:     Reset the current position in $L_i$ to the first posting
9: while true do
10:    Sort posting lists in the ascending order of their current story ids
11:    $p \leftarrow \perp$ – index of the pivot list
12:    $UB \leftarrow 0$
13:    $s \leftarrow L_{|u|}.cur$
14:    for $i \in [1, 2, \ldots, |u|]$ do
15:        if $L_i.cur \ge s$ then
16:            break
17:        $UB \leftarrow UB + u_i \cdot \mathtt{ms}(L_i)$
18:        $pos \leftarrow \mathcal{I}_i.\mathtt{next}(L_i.curPosition, UB)$
19:        if $pos \le |L_i|$ then
20:            $s' \leftarrow$ story at position $pos$ in $L_i$
21:            if $s' < s$ then
22:                $p \leftarrow i$
23:                $s \leftarrow s'$
24:    if $p = \perp$ then
25:        break
26:    if $L_0.cur \ne L_p.cur$ then
27:        for $i \in [1, 2, \ldots, p-1]$ do
28:            Skip the current position in $L_i$ to a story $\ge s$
29:    else
30:        $score \leftarrow 0$
31:        $i \leftarrow 0$
32:        while $L_i.cur = L_p.cur$ do
33:            $score \leftarrow score + u_i \cdot L_i.curPs$
34:            Advance by 1 the current position in $L_i$
35:            $i \leftarrow i + 1$
36:        processScoredResult$(s, u, score, \mathcal{R}_s, \mathcal{I})$
37: return $\mathcal{R}_{s_1}, \mathcal{R}_{s_2}, \ldots, \mathcal{R}_{s_n}$
```

4. EXPERIMENTAL RESULTS

This section describes the evaluation of our algorithms. We used an 8-core Linux machine equipped with Intel Xeon 1.86GHz processors and 16GB of memory. We report in-memory performance of a single-threaded code after loading all the data (including indices) into the main memory.

4.1 Test collections

We used a corpus of 100K news stories in English, randomly selected from the set of stories available during a single day on Yahoo! News. We extracted the main textual content (the body) of each story, as well as keywords from its title and abstract (16 terms on average). The body of a story contained 310 terms on average from which we retained 190 after filtering out 800 common stopwords. We thus experimented with two Story indices – (1) Keywords, indexing only the title and the abstract of each story, and (2) FullText, indexing the main body of each story. The total number of unique terms in Keywords and FullText is 83K and 305K respectively. These indices reflect different tradeoffs between the precision and the recall of relevant tweets. The FULLTEXT index maximizes recall while KEYWORDS improves precision.

We obtained a log of more than 100M tweets posted during the day the stories in our corpus were displayed. From these, 35M were retained, filtering out (mostly non-English) tweets containing non-ASCII characters. This number translates to about 24K incoming tweets per minute. To approximate the behavior of a real system we used the first 90% of tweets (ordered by creation time) to initialize the annotation sets ($\mathcal{R}_s$) of the stories, and performed our measurements on a random sample from the last 10%.

We experimented with both cosine similarity (denoted CS) and BM25 content-based score functions (their definitions appear in Section 2.1).

4.2 News annotation statistics

In this section we analyze some basic statistics of the news annotation problem. We first examine the average rate of incoming tweets that are related to a story s, i.e., such that $\texttt{cs}(s, u) > 0$. Such tweets can potentially be used for annotating s. Note that the rate is the same for both CS and BM25, since we're simply looking for any textual overlap.

Index	Tweets per minute
KEYWORDS	3.06
FULLTEXT	37.92

Table 1: Rate of tweets related to an average story.

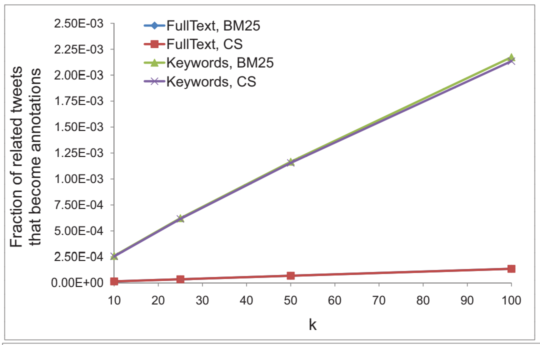
The table above shows that out of 24K tweets that arrive each minute, 3 are related to an average story in KEYWORDS, while as many as 38 are related to an average story in FULLTEXT. This would be the average cache invalidation rate *per story* have we decided to cache story annotations and refresh them using real-time tweet indexing. In a corpus of 100K stories we consider, this would translate to as many as 300 thousands and 3.8 million invalidation events per minute for KEYWORDS and FULLTEXT indices respectively.

We next evaluate the fraction of related tweets that actually get to annotate s, i.e., inserted into the set of annotations $\mathcal{R}_s$. Clearly, this fraction depends on the size k of $\mathcal{R}_s$ as well as on τ, the decay parameter of the recency score. Figure 3 shows that the chances of an incoming related tweet to get into $\mathcal{R}_s$ increase linearly with k: as k grows, it is easier for the new tweet to score higher than the k-th best tweet for the story. Similarly, as τ grows, the scores of older tweets in $\mathcal{R}_s$ decay slower, and it is more difficult for a new tweet to get added to $\mathcal{R}_s$ replacing an older tweet.

Figure 3 shows that while the rate of related tweets is high, the actual set of annotations of a story is updated 3 to 5 orders of magnitude slower. We conclude that to minimize processing cost it is not enough to find the set of stories *related* to an incoming tweet, but it is also crucial to efficiently identify the subset of these related stories whose annotation sets the tweet will eventually be added to.

4.3 Pub-sub algorithms for news annotation

This section evaluates our new algorithms and compares their effectiveness at processing incoming tweets. We first consider the two basic algorithms: TAAT (Algorithm 3) and DAAT (Algorithm 7).



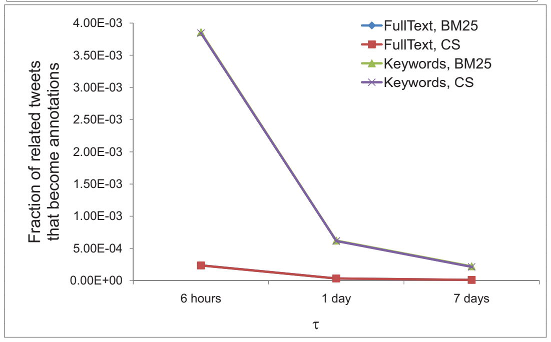


Figure 3: Fraction of related tweets that are inserted into $\mathcal{R}_s$ of an average story as a function of k (for $\tau = 1$ day) and as a function of τ (for $k = 25$).

Figure 4 shows the average processing time of an incoming tweet, for the KEYWORDS index. As the figure demonstrates, our experiments showed that the performance and the effect of different algorithm parameters is quite similar regardless of whether CS or BM25 is used as the scoring function. Consequently, in the following experiments we show measurements with BM25 and point out the differences between CS and BM25 only when they are noteworthy.

As Figure 4 shows, the higher k is, the more likely a new tweet is to score higher than the worst tweet in an annotation set of a story, and, consequently, our algorithms have to update the annotation sets of a larger number of stories. However, as the non-skipping TAAT and DAAT process posting lists corresponding to tweet terms in whole, the dependence is not strong, e.g., for k increasing from 10 to 100, the processing latency increases by less than a factor of 2. The dependence on τ is even lower.

4.4 The effect of skipping

We next focus on TAAT+SKIPPING (Algorithm 5) and DAAT+SKIPPING (Algorithm 8). We analyze the relative fraction of postings that are skipped using our tree-based technique, and its effect on tweet processing latency.

Figure 5 demonstrates the fraction of skipped postings as a function of k and τ. DAAT+SKIPPING skips up to 95% of the postings, whereas TAAT+SKIPPING skips up to 85%. Increasing k and decreasing τ directly reduces μ_s of stories, making incoming tweets more likely to enter $\mathcal{R}_s$, and consequently reducing the opportunities to skip postings. For any given k and τ, the fraction of the postings skipped by TAAT+SKIPPING and DAAT+SKIPPING is significantly higher for the FULLTEXT index than for KEYWORDS. This indicates that our skipping techniques scale well with the

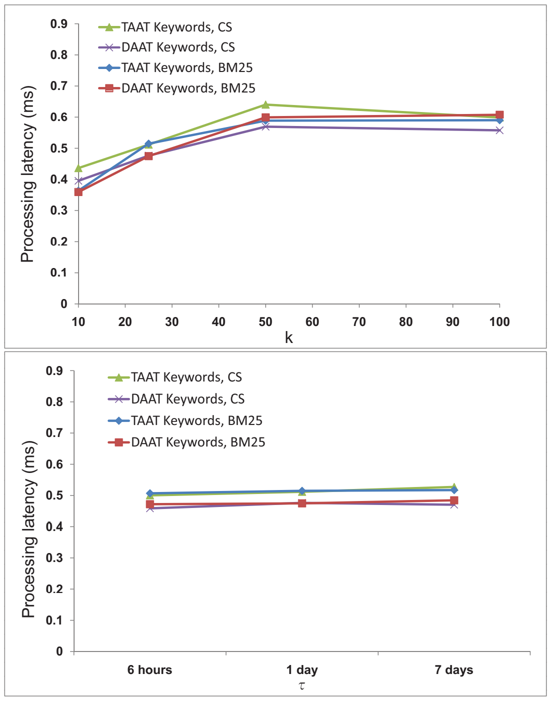


Figure 4: Average tweet processing latency with Keywords index (top: $\tau = 1$ day, bottom: $k = 25$).

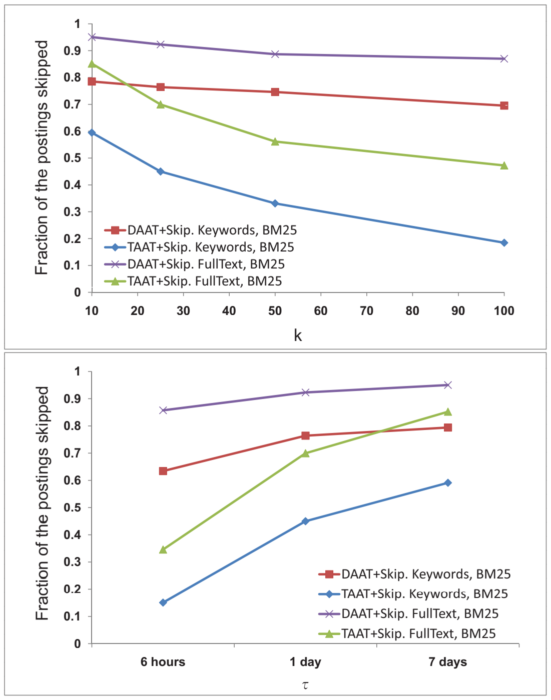


Figure 5: Fraction of postings skipped (top: $\tau = 1$ day, bottom: $k = 25$).

index size – the number of postings our algorithms examine depends sub-linearly on the size of the index.

We next show that skipping directly improves processing latency. Figure 6 shows average processing latency as a function of k for $\tau = 1$ day with the Keywords index. We see that, compared to its non-skipping counterpart, DAAT+skipping reduces processing time by 62% to 69% (for $k = 100$ and 10 respectively), while TAAT+skipping reduces processing time by 8% to 30%, compared to TAAT. Figure 7 presents similar measurements with FullText and shows that DAAT is slightly faster than TAAT, while their skipping variants now save 73% to 89% (for DAAT) and 43% to 77% (for TAAT). We see that both skipping algorithms process arriving tweets significantly faster than their non-skipping variants. The graphs additionally show that DAAT+skipping significantly outperforms TAAT + skipping for high values of k.

While the effect of τ on latency of TAAT and DAAT is negligible (see Figure 4), Figure 8 shows a much more significant effect with their skipping counterparts. Intuitively, a low τ (high decay rate) reduces scores of previously seen tweets, which in turn reduces the number of stories that can be skipped when processing a new tweet. Here too, DAAT+skipping outperforms TAAT+skipping. Figure 9 shows that DAAT+skipping with BM25 performs slightly better than with CS (we observed similar results for TAAT+skipping). It also shows the weak dependence of DAAT and DAAT+skipping on the annotation size k, suggesting that both algorithms are scalable with k.

5. RELATED APPROACHES

DAAT and TAAT algorithms for IR have been thoroughly studied and compared in the past [8, 23, 18, 7, 22, 9]. In [9], the authors of the Juru search system performed experiments comparing DAAT and TAAT algorithms for the large TREC GOV2 document collection. They found that DAAT was clearly superior for short queries, showing over 55% performance improvement. Unlike our work, which focuses on memory-resident indices, this work used the disk-based Juru index. To the best of our knowledge our work is the first to consider DAAT/TAAT algorithms in the context of content-based pub-sub. Our results indicate that similarly to their IR counterparts, DAAT algorithms for pub-sub significantly outperform TAAT.

5.1 Real-time tweet indexing

The problem we consider in this paper can be viewed as a typical IR problem, where given a pageview request of story s at time t we have to retrieve the top-k updates from a corpus U^t according to a score function `score`. A possible solution would then be maintaining a real-time incremental index of tweets, and for each pageview of a story s querying the index with an appropriate query q_s built from the content of the story. While a real-time indexing solution would work well for settings with low-to-medium traffic volume, it would be inefficient for high-volume sites, due to high overhead of querying the index of tweets for each pageview.

A partial remedy would be to cache query results for each story. Caching documents for popular web queries is widely employed in practice. Blanco et al. [5] propose a scheme to invalidate cached results when new documents arrive. Invalidation causes re-execution of the query when it is invoked by a user. In order to reduce the amount of invalidations, a *synopsis* of newly arriving documents is generated, which is a compact representation of a document's score attributes, albeit to unknown queries. The synopsis is used

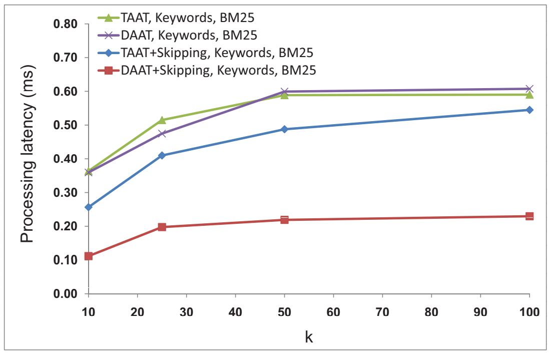


Figure 6: Average tweet processing latency (Keywords index, $\tau = 1$ day).

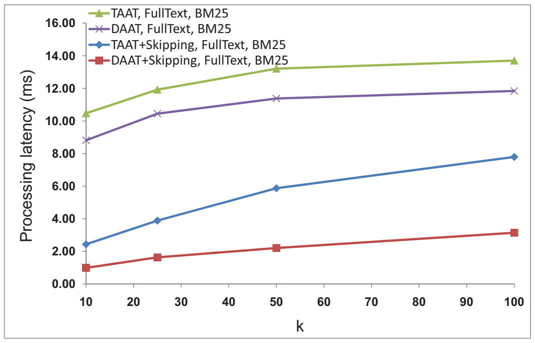


Figure 7: Average tweet processing latency (FullText index, τ=1 day).

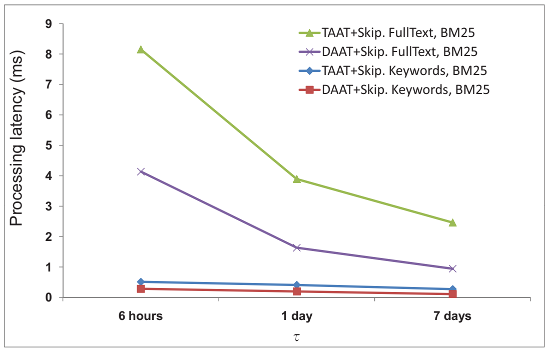


Figure 8: Average tweet processing latency ($k = 25$).

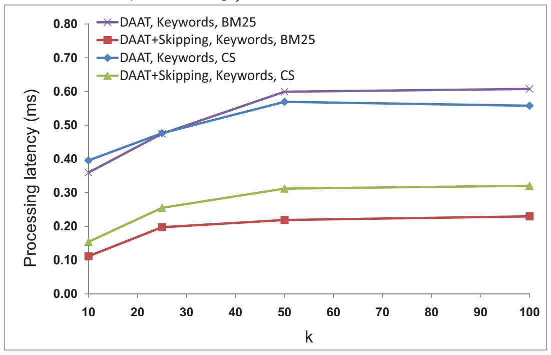


Figure 9: Average tweet processing latency (Keywords index, $\tau = 1$ day).

to identify cached results that might change, and are thus invalidated. This mechanism creates both false positives (query results are unnecessarily invalidated) and false negatives (stale query results are presented to users) and the authors study the tradeoffs between them. In our setting, only a very small fraction of incoming tweets that are related to a story eventually end up annotating it. Our approach is different in that it proactively maintains result sets of stories, instead of reactively caching and invalidating them.

Serving costs comparison to the pub-sub architecture. We continue by further comparing our approach with a caching scheme, along the lines proposed by Blanco et al. [5], which caches the top-k updates that annotate each story, and invalidates the cached list on arrival of a tweet that could potentially annotate the story. We consider the cost of annotating the 1K most-popular stories shown on a news website in a single day, where each story is represented using its main body of text. For simplicity, we assume that pageviews, as well as related incoming tweets (i.e., tweets with positive textual overlap), are distributed uniformly across the 1K popular stories.

We define the cost of each approach to be the number of queries submitted to the underlying inverted index per minute, multiplied by the cost of each query. The cost of a query depends, among other, on the index size and on the specific implementation. The Tweet index is orders of magnitude larger than the Story index: 35M new tweets (after filtering) are added to the Tweet index every day (see Section 4). The Story index, on the other hand, contains a fixed number of stories, the most-popular 1K stories in our case. For the comparison, we use a simplified cost model: we conservatively assume that although the Story index is at least 35K times smaller than the Tweet index (if we consider just tweets from a single day), a query to the Story index is merely 10 times cheaper than a query to the Tweet index[3]

In our approach each incoming tweet triggers a query to the story index, and therefore the number of queries to the Story index is simply the number of incoming tweets, i.e., 24K per minute (see Section 4.2). With the caching approach, the number of queries to the Tweet index depends on the cache miss rate. Observe that the tweet index is queried on the first pageview event of story s which follows an invalidation event of the cached list of annotations for s. From here, it is easy to see that the query rate is roughly $\min(\text{pageview rate}, \text{invalidation rate})$.

Substituting the rate of incoming related tweets from Table 1, we get almost 38K expected invalidations per minute for the cached annotations of our 1K popular stories. Figure 10 shows the invalidation and Tweet index query rates as a function of the overall pageview rate for the 1K popular stories. Observe that as long as the pagevew rate is less than the invalidation rate, caching doesn't really help since cached results are invalidated before they can be used for another pageview, in other words, every pageview results in a query. Caching starts to be beneficial as pageview rate approaches and passes the invalidation rate.

[3]We note that in our experiments, a difference of only 3.7 times in the size of FullText compared to Keywords resulted in an order of magnitude higher query latency (see, e.g., Figure 8).

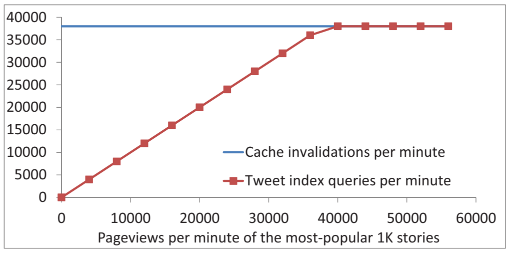


Figure 10: The effect of pageview and invalidation rates.

Figure 11 compares the overall cost of the caching approach and our solution. Here the cost is a product of the query rate to the Tweet and Story index respectively, and the cost factor of 1 for the Story index and 10 for the Tweet index. Expectedly, the cost of our solution does not depend on the pageview rate but only on the incoming tweet rate (24K per minute). The cost of caching is lower for low pageview rates, and higher for rates above $2,400$ (for all the 1K popular stories combined). Recall that major news websites receive orders of 100-s of millions of pageviews daily, hence 100-s of thousands pageviews per minute.

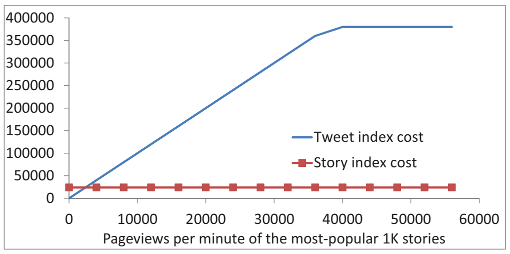


Figure 11: Comparison of query costs.

A cost analysis like the one shown in Figure 11 can guide the choice between our approach and caching, or, in a hybrid approach, can help select the number of most-popular stories whose annotations are maintained using pub-sub, while the annotations of tail stories are maintained using caching. Finally, we note an additional important factor that one must take into account: the cost of querying the Tweet index using the caching approach is incurred "online", during a pageview, whereas querying the Story index occurs when a new tweet arrives and has no effect on pageview latency.

5.2 Content based publish-subscribe

In most previous algorithms and systems for content based publish-subscribe (e.g., [1, 4, 10, 11, 12, 21]), published items trigger subscriptions when they match a subscription's predicate. In contrast, top-k pub-sub scores published items for each subscription, in our case based on the content overlap between a story and a tweet, and triggers a subscription only if it ranks among the top-k published items.

The notion of top-k publish-subscribe (more specifically, top-k/w publish-subscribe) was introduced in [20]. In top-k/w publish-subscribe a subscriber receives only those publications that are among the k most relevant ones published in a time window w, where k and w are constants defined per each subscription [20, 15]. In this model, the relevance of an event remains constant during the time window and once its lifetime exceeds w the event simply expires (i.e., the relevance becomes zero). The place of the expired object is then populated with the most relevant unexpired object by a re-evaluation algorithm. The sliding-window model was previously extensively studied in the context of continuous top-k queries in data streams (see [2] for a survey). Solutions in this model face the challenge of identifying and keeping track of all objects that may become sufficiently relevant at some point in the future due to expiration of older object (see, e.g., [6]), or alternatively use constrained memory and provide probabilistic guarantees [20, 15]. A recent work [16] proposed a solution for the top-k/w publish-subscribe problem based on the Threshold Algorithm (TA) [13], which is similar in spirit to our solution, but relies on keeping posting lists sorted according to the current minimum score in the top-k sets of subscriptions, which changes frequently.

We propose a different model, which is more natural for tweets (and perhaps for other published content) and does not require re-evaluation, where the published events (e.g., tweets) do not have a fixed expiration time. Instead, time is a part of the relevance score, which gradually decays with time. The decay rate is the same for all published events (objects) for a given subscription, and therefore older events retire from the top-k only when new events that score higher arrive and take their place. This makes re-evaluation unnecessary, and does not require storing previously seen events unless they are currently in the top-k. This, together with DAAT and TAAT algorithms that do not require re-sorting posting lists and thus are more efficient in our setting, makes our solution more than an order of magnitude faster in similar setting on a comparable hardware and somewhat *larger* dataset than [16]. Specifically, the algorithms in [16] were evaluated on shorter subscriptions of 3 to 5 random terms selected from the relatively small set of 657 unique terms, whereas the average subscription in our smallest index KEYWORDS had 16 terms from a set of 83,109 unique terms.

Query indexing is a popular approach, especially in the context of continuous queries over event streams, where it is simply impossible to store all events. Previous works, however, focused on low-dimensional data. Earlier works employed indexing techniques that performed well with up to 10 dimensions but performed worse than a linear scan of the queries for higher number of dimensions [24], and later works, such as VA-files [24] (used e.g., in [20, 6]) were shown to perform well with up to 50 dimensions. It was also shown that latency of matching events to subscriptions in the top-k/w model increases linearly with the number of dimensions [20]. Finally, the number of supported subscriptions was mostly relatively low (up to a few thousands in [20, 6]). Our work considers highly-dimensional data, and we evaluate our approach on news articles and tweets in English (therefore the number of dimensions is in hundreds of thousands), with 100K subscriptions (news articles).

A different notion of ranked content-based pub-sub systems was introduced in [19]. This system produces a ranked list of subscriptions for a given item, whereas we produce a ranked set of published items for each subscription. In [19] subscriptions were defined using a more general query language, allowing specifying ranges over numeric attributes. To support such complex queries, custom index and algorithms were designed, unlike our approach of adapting standard inverted indexing and top-k retrieval algorithms.

An interesting related problem is providing search results retroactively for standing user interests or queries, e.g., as in Google Alerts [14]. Yang et al.[25] identify such queries automatically from user search-logs. Such systems typically periodically re-execute standing queries with the search engine to find new relevant results [25]. Although the problem we consider in this paper is substantially different, we believe that our approach can provide an alternative that does not require re-execution: each standing query can be indexed similarly to the way we index news articles, and new documents can be matched to standing queries similarly to the matching of relevant tweets to stories in our system. Then, if the new document scores among the top-k documents maintained for this query, the user issuing the query can be notified. Comparing these approaches in practice is an interesting direction for future work.

6. CONCLUSION

In this paper we dealt with the problem of real-time annotation of online news stories with tweets and introduced a solution using the top-k pub-sub paradigm. Annotations are related to stories by building an index of the news stories as subscriptions and evaluating incoming tweets as published content. This approach is more efficient for high-volume websites than the classical solution based on real-time incremental indexing of tweets: we match tweets with news articles when new tweets arrive, and not during the serving of pageviews. Our solution proactively maintains annotation sets of stories under high volume of Twitter updates, allowing for efficient serving of pageviews while guaranteeing maximal freshness of annotations. We presented variations of four prevalent top-k document retrieval algorithms adapted to the publish-subscribe setting and shown how this adaptation leads to very significant reduction in processing time. Evaluation on a real-world corpus of news stories and on a log of tweets validated the effectiveness of our approach.

7. ACKNOWLEDGEMENTS

We thank Edward Bortnikov, Ronny Lempel and Benjamin Reed for helpful discussions and valuable comments.

8. REFERENCES

[1] 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*, pages 53–61, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[4] G. Banavar, T. Chanra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, pages 262–272, 1999.

[5] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *WWW*, pages 82–89, 2010.

[6] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007.

[7] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[8] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.

[9] D. Carmel and E. Amitay. Juru at 2006: Taat versus daat in the terabyte track. In *TREC*, 2006.

[10] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.

[11] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *TODS*, 28:467–516, 2003.

[12] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, pages 115–126, 2001.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[14] Google Alerts. `http://alerts.google.com/`.

[15] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[16] P. Haghani, S. Michel, and K. Aberer. The gist of everything new: personalized top-k processing over web 2.0 streams. In *CIKM*, pages 489–498, 2010.

[17] B. Keane. Twitter v the msm: covering gaddafi's war against reality. http://www.crikey.com.au/2011/03/21/twitter-v-the-msm-covering-gaddafis-war-against-reality/, 2011.

[18] P. Lacour, C. Macdonald, and I. Ounis. Efficiency comparison of document matching techniques. In *Efficiency Issues in Information Retrieval Workshop; European Conference for Information Retrieval*, 2008.

[19] A. Machanavajjhala, E. Vee, M. N. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *PVLDB*, 1(1):451–462, 2008.

[20] K. Pripuzic, I. P. Zarko, and K. Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS*, pages 127–138, 2008.

[21] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *SOSP*, pages 160–173, 2001.

[22] T. Strohman, H. R. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, pages 219–225, 2005.

[23] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[24] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[25] B. Yang and G. Jeh. Retroactive answering of search queries. In *WWW*, pages 457–466, 2006.