Efficient Term Cloud Generation for Streaming Web Content

Odysseas Papapetrou, George Papadakis,
Ekaterini Ioannou, and Dimitrios Skoutas

L3S Research Center, Hannover, Germany
{papapetrou,papadakis,ioannou,skoutas}@L3S.de

Abstract. Large amounts of information are posted daily on the Web, such as articles published online by traditional news agencies or blog posts referring to and commenting on various events. Although the users sometimes rely on a small set of trusted sources from which to get their information, they often also want to get a wider overview and glimpse of what is being reported and discussed in the news and the blogosphere. In this paper, we present an approach for supporting this discovery and exploration process by exploiting term clouds. In particular, we provide an efficient method for dynamically computing the most frequently appearing terms in the posts of monitored online sources, for time intervals specified at query time, without the need to archive the actual published content. An experimental evaluation on a large-scale real-world set of blogs demonstrates the accuracy and the efficiency of the proposed method in terms of computational time and memory requirements.

1 Introduction

The popularity of online news sources has experienced a rapid increase recently, as more and more people use them every day as a complement to or replacement of traditional news media. Therefore, all major news agencies make nowadays their content available on the Web. In addition, there exist several services, such as Google News or Yahoo! News, that aggregate news from various providers. At the same time, more and more people maintain Web logs (blogs) in a regular basis as a means to express their thoughts, present their ideas and opinions, and share their knowledge with other people around the globe. Studies about the evolution of the Blogosphere [3, 19] report around 175,000 new blogs and 1.6 million new blog posts every day. Micro-blogging has also emerged as a special form of such social communication, where users post frequent and brief text messages, with Twitter constituting the most popular example. This results in an extremely large and valuable source of information that can be mined to detect themes and topics of interest, extract related discussions and opinions, and identify trends. Perhaps the most interesting aspect that distinguishes these sources from other information available on the Web is the temporal dimension. Several recent research activities have focused on analyzing and mining news

and blogs for detecting events or stories (e.g., [2, 1, 15, 19]), identifying trends (e.g., [5, 11]), and extracting opinions and sentiments (e.g., [8, 18, 22]).

In the Web, users very often need to explore large collections of documents or pages, without having a-priori knowledge about the schema and the content of the contained information. Various techniques, such as clustering, faceted browsing, or summarization, are used in such scenarios to help the user more effectively navigate through the volume of the available information, to obtain an overview, and to drill down to the items of interest. One simple visualization technique that has become very popular, especially in Web 2.0 applications, is presenting to the user a visual overview of the underlying information in the form of a *tag cloud*. This is typically a set comprising the most popular (i.e., frequent) tags contained in or associated to the underlying data collection, where most frequent tags are visualized in a more prominent way (i.e., using larger font size). In many applications, the cloud is constructed from the tags assigned to the data explicitly by users. For example, Flickr provides tag clouds of the most popular tags entered by the users in the last 24 hours or over the last week[1]. In other cases, where no such tags or keywords are available, the cloud can be constructed from terms (or other semantically richer items, such as entities) automatically extracted from the text, as for example in Blogscope [2]. The advantage of tag clouds lies mainly in the fact that they are very simple, intuitive, and visually appealing to the user, while serving two purposes; first, they provide an easy and quick way to the user to get hints about what is contained in the underlying data, e.g., what is being reported and discussed in the news. Second, they allow the user to navigate and explore the available information, by selecting an interesting term in the cloud and viewing the corresponding documents.

In this work, our goal is to exploit these benefits of tag clouds to facilitate users in exploring and obtaining an overview of Web content made available in a streaming fashion, such as news-related information provided by online media and the blogosphere. Once the candidate terms are available, either explicitly provided or automatically extracted, the construction of the cloud comprises just a single task: computing the frequencies of these terms, and selecting the most frequent ones to visualize in the cloud. This computation is straightforward and can be done on the fly in most cases, when dealing with a relatively small set of documents. However, this is not scalable to very large collections of items, as in our scenario, which should scale, for example, to millions of news and blog posts, and over potentially very long time periods. Moreover, we do not want to restrict the interests of the users to pre-defined time periods, e.g., for the current day, as typically happens in existing applications. Instead, the users should be able to request the generation of a term cloud for arbitrarily small or large, as well as for arbitrarily more or less recent, time intervals, such as for the last hour(s), day(s) or week(s), for the last year, for the summer of 2008, etc.

To deal with these challenges, we cast the above problem as one of finding frequent items in incoming streams of data. Specifically, we assume a stream consisting of the terms extracted from the newly published posts that are made

[1] http://www.flickr.com/photos/tags/

available to the system. Our goal is to maintain space-efficient data structures that allow the computation of the top-k frequent terms in the stream, for arbitrary time intervals and as accurately as possible. This formulation allows us to exploit the advantages of efficient algorithms that have been proposed for mining frequent items in streaming data. In particular, our main contributions are as follows.

- We exploit term clouds to facilitate the navigation and exploration of streaming Web content, such as incoming news-related posts from online media and the blogosphere.
- We present an efficient method for generating term clouds dynamically for arbitrary time periods, based on techniques for mining frequent items in data streams.
- We validate experimentally the efficiency and the effectiveness of our method, using a large-scale real-world collection of blog posts.

The remainder of the paper is organized as follows. The next section discusses related work. Section 3 presents our approach in detail. In Section 4, we present our experimental evaluation. Section 5 concludes the paper.

2 Related Work

As discussed above, our purpose is to help users to get an overview and to explore and navigate large streams of news-related information, by identifying and visualizing in the form of a cloud the most popular terms appearing in the posts. Such clouds have become a very popular visualization method, especially in Web 2.0 applications, for presenting frequent tags or keywords for exploration and navigation purposes. Blogscope [2], for instance, presents in its front page a cloud of keywords extracted from the posts of the current day. Similarly, Grapevine [1], which builds on Blogscope, displays also a cloud of the main entities related to a selected topic or story, using the font size to denote the popularity of the entity, and the font color to denote its relatedness to the topic or story. In a different domain, PubCloud [14] is an application that allows querying the PubMed database of biomedical literature, and employs term clouds as a visualization technique to summarize search results. In particular, PubCloud extracts the words from the abstracts retrieved for the given query, performs stopword filtering and stemming, and visualizes the resulting terms as a cloud. It uses different font sizes to denote term frequency and different font colors to denote the recency of the corresponding publications in which the term appears. Through a user study, the authors show that this summarization via the term cloud is more helpful in providing descriptive information about the query results compared to the standard ranked list view. TopicRank [4] generates a term cloud from documents returned as a result to a query, where the positioning of the terms in the cloud represents their semantic closeness. Page History Explorer [9] is a system that uses term clouds to visually explore and summarize

the contents of a Web page over time. Data clouds are proposed in [13] for combining the advantages of keyword search over structured data with those of tag clouds for summarization and navigation. Complex objects are retrieved from a database as responses to a keyword query; terms found in these objects are ranked and visualized as a cloud. Other tools also exist, such as ManyEyes[2] or Wordle[3], for generating word clouds from user provided text.

In all these applications, the term cloud is generated from a corpus that has a relatively limited size, e.g., blog posts of a particular day or story, query results, or a document given by the user. Hence, efficiently computing the most frequent terms for visualization in these scenarios does not arise as a challenging problem per se, and, naturally, it does not constitute the focus of these works. Instead, in our work, frequent terms need to be computed on demand from very long streams of text (e.g., news articles and blog posts spanning several months), and therefore efficiency of the computation becomes a crucial issue.

To meet these requirements, we formulate the problem as finding frequent items in streaming data. Several space-efficient algorithms have been proposed that, given a stream of items, identify the items whose frequency exceeds a given threshold or estimate the frequency of items on demand. A survey and comparison of such algorithms can be found in [6, 16]. In our case, we are interested in finding the top-k frequent items. Hence, these algorithms are not directly applicable, since there is no given frequency threshold nor is it efficient to maintain a set of *all* possible items and then calculate their frequencies on demand to select the top-k ones. The problem of finding the top-k frequent items in a stream is addressed in [21]. The authors propose two algorithms, one based on the Chernoff bound, and one that builds on the Lossy Counting algorithm from [17]. However, these solutions consider either the entire data stream or a sliding window that captures only the most recent items. Instead, we want to allow the users to construct the term clouds on arbitrary time intervals of the stream. The TiTiCount+ algorithm has been proposed in [20] for finding frequent items in ad hoc windows. This algorithm, though, finds the items with frequency above a given threshold and not the top-k frequent items. Thus, the method we present in this paper builds on the ideas and advantages of the solutions presented in [20, 21] in order to meet the requirements of our case.

3 Term Clouds for Streaming Text

In this section, we present our approach for constructing term clouds from streaming text. A term cloud is composed of the top-k frequent terms in the data collection, which in our case is a substream of text covering a specified time interval, defined by the query. First, we describe the main components of the system and we formally define the problem. Then, in Section 3.2, we describe the data structures used to support the efficient computation of the term cloud.

[2] http://services.alphaworks.ibm.com
[3] http://www.wordle.net

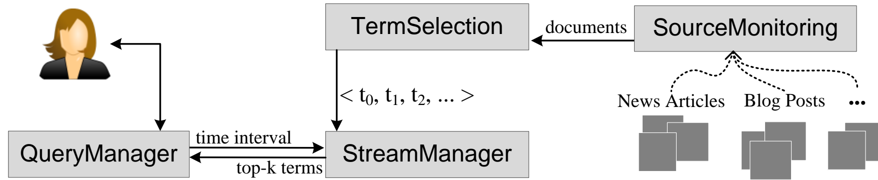

Fig. 1. System overview.

Using these data structures, we describe in Section 3.3 how the term cloud for a desired time interval is generated dynamically upon request.

3.1 System Overview

The overall system architecture is depicted in Figure 1. The *SourceMonitoring* component monitors online sources for new posts (e.g., online news agencies and blogs) and updates the system when new information is published. These incoming posts are provided as input to the *TermSelection* component, which is responsible for extracting a set of terms from each post. These are the candidate terms from which the term clouds are generated. These terms consist the input to the *StreamManager* component, which maintains a set of data structures for efficiently generating term clouds. Finally, the *QueryManager* component is responsible for receiving and evaluating user queries for term clouds, using the data structures provided by the *StreamManager*.

Each component poses a set of challenges. For example, regarding the *Source-Monitoring* component, the main issues that arise are how to select or dynamically discover sources, and how to monitor and extract the useful text from sources that do not provide feeds. A main challenge for the *TermSelection* component is how to tokenize the incoming text and to select potentially useful and meaningful terms. Solutions for this may vary from simply filtering out some terms using a stopword list to applying more sophisticated NLP techniques for extracting keywords and identifying named entities. In this paper, we focus on the *StreamManager* and *QueryManager* components, which we describe in detail in the following sections. This modular architecture for the processing pipeline allows us to easily plug in and try different solutions for the other components.

3.2 Data Structures

We now describe the data structures maintained by *StreamManager* for supporting the efficient computation of term clouds.

As described above, the *TermSelection* component extracts a set of candidate terms from each newly published post that is detected by the *SourceMonitoring* component, and provides the resulting terms as an input to the *StreamManager* component. Hence, the input of *StreamManager* is a stream of terms $\mathcal{T} = < t_0,\ t_1,\ t_2, \ldots >$. For each term t_i, we also maintain a pointer to the document from which it was extracted. As will be described later, only a subset of the

incoming terms are maintained by the system. Accordingly, only the pointers to the documents containing these terms are finally stored. Maintaining these pointers is required to enable the navigation functionality in the term cloud. Also note that the actual contents of the posts are not stored in main memory; they are fetched (from the Web or from a local cache in secondary storage) only if the user requests them by navigating in the term cloud. The order in which the terms are appended to the stream is according to the publishing time of the corresponding posts. This implies that the *SourceMonitoring* component fetches the posts in batches, and sorts them by their publishing time, before pushing them further down the processing pipeline. Posts with the same publishing time are ordered arbitrarily, while terms extracted from the same post are ordered according to their appearance in the post. Hence, the first (last) term in the stream is the first (last) term of the least (most) recent post that has been processed by the system.

Efficiently computing the top-k frequent terms in the stream involves two main issues: (a) a method to estimate the frequency of a given term, and (b) a method to distinguish which terms have high probability to be in the top-k list, so that only these terms are maintained by the system to reduce memory requirements. In the following, we describe how these issues are addressed.

Instead of storing the whole stream, we maintain a compact summary of the observed terms and their frequencies (i.e., number of occurrences) using an *hCount structure* [10], as briefly described in the following. hCount is composed of a set of counters, which allow us to estimate the frequency of a term with high accuracy. Specifically, this structure is a 2-dimensional array of counters comprising h rows and m columns. Each row is associated with a hashing function, which maps a given term to an integer in the range $[0, m-1]$. When the next term t in the stream arrives, its hashing values for all the h hashing functions are computed. Then, for each row i, the counter of the column $h_i(t)$ is incremented by 1. The frequency of a term t can then be estimated as follows: the term is hashed as before, to identify the corresponding counters, and the minimum value of these counters is used as an estimation of the term's frequency. The benefit is that this operation now requires only $h \times m$ counters instead of $|\mathcal{T}_d|$ which would be required for storing the exact frequencies of all the distinct terms $\mathcal{T}_d$. hCount also provides probabilistic guarantees for the frequency estimation error of a term t in a given stream $\mathcal{T}$. According to [10], if $\frac{e}{\epsilon} \times \ln\left(-\frac{|\mathcal{T}_d|}{\ln \rho}\right)$ counters are stored, the frequency estimation error for each term is not more than $\epsilon \times |\mathcal{T}|$ with probability ρ. In our experiments in Section 4, a sufficiently accurate estimation of the term frequencies was achieved using only a very small number of counters, i.e., $(h \times m) << |\mathcal{T}_d|$, and thereby with negligible memory requirements.

The problem that arises next is how to determine for which terms to probe the *hCount* structure for their frequencies, in order to get the top-k frequent terms. A straightforward solution, albeit prohibitively expensive, is to maintain a list of all the distinct terms, and look them up in the *hCount* structure to get their estimated frequencies. To significantly reduce the memory requirements and execution time, we need to prune this list, maintaining instead only a subset

of terms, such that all the top-k frequent terms are contained in this subset with high probability. To determine this subset, we employ a method based on Chernoff bounds, as proposed in [21]. The method considers the terms in the stream in batches of fixed length l. For each batch B, the exact frequencies of all the contained terms are calculated, and the frequency of the k-th most frequent term in the batch, denoted by $freq_k(B)$, is found. The *support* of this term in terms of B is computed as $s_k(B) = freq_k(B)/l$. Then, using Chernoff bounds, we filter out all terms in the batch that do not belong in the top-k terms of the stream with probability higher than a predefined value δ. In particular, only terms that have observed support:

$$s(B) \geq s_k(B) - 2\sqrt{\frac{2 \times s_k(B) \times \ln(2/\delta)}{l}} \quad (1)$$

need to be maintained in a term pool $\mathcal{P}$, with $\mathcal{P} \subseteq \mathcal{T}_d$, and it is sufficient to use only these terms for probing the hCount structure. The final size of $\mathcal{P}$ is typically substantially smaller than the total number of distinct terms $|\mathcal{T}_d|$ observed from the beginning of the stream. In our experiments presented in Section 4, we were able to obtain highly accurate results by maintaining no more than $2 \times k$ terms in $\mathcal{P}$ (which was at least 3 orders of magnitude less than the total number of distinct terms occurring in the stream). At query time, to extract the top-k terms from $\mathcal{T}_d$, the system estimates the frequency of each term $t \in \mathcal{T}_d$ from the hCount structure, it sorts all the terms on their estimated frequency, and it returns the top-k ones to the user.

The method described so far allows us to efficiently compute the top-k frequent terms for the *whole* stream. However, different users may be interested in different time periods. Therefore, we want to generate term clouds for arbitrary time intervals specified by the query. A query is a tuple $Q =< i,\ j >$, defining a substream comprising all the terms between the positions i and j ($i < j$).

To allow users to view term clouds for arbitrary time intervals, we divide the stream into time windows, as proposed in the TiTiCount+ algorithm [20]. Each window covers a different time interval, i.e., a different substream, as illustrated in Figure 2. Specifically, the first window, i.e. w_0, has size b, which means that it contains summary information for the b most recently received terms, where b is a predefined system parameter. Each subsequent window w_i, $i > 0$, has size $2^{i-1}b$. This particular organization of the windows is motivated by the assumption that users will more often be interested in more recent time periods; hence, the more recent parts of the stream are covered by windows of higher resolution (i.e., contain less terms) than the least recent ones. In the presence of these windows, the method described above is adapted by maintaining a separate hCount structure and a term pool of potentially frequent terms for each window, instead of the whole stream. The same number of counters and pool sizes are used for all the windows; hence, all windows have the same memory requirements. Consequently, the maintained summary in windows that cover less recent parts of the stream, and therefore have larger size, has typically lower accuracy.

Initially, the hCount structures in all the windows are empty. Each new incoming term is received by the first window. When a batch of b terms has been

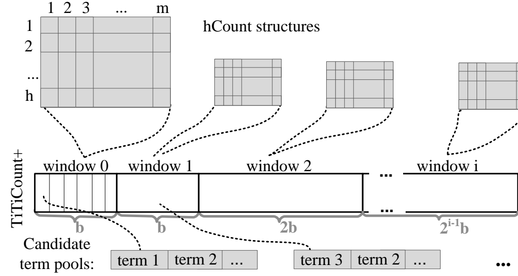

Fig. 2. An illustration of the StreamManager.

received, this window becomes full. Then, the contents of this window are shifted to the second one, so that new terms can be received. When the second window reaches its maximum size, its contents are shifted to the third window, and this continues recursively. If the window that receives the new content already contains some information, i.e., the counters in its hCount structure are not 0, the old contents need to be merged with the new one. Merging two subsequent windows involves: (a) merging the two hCount structures, as proposed in [20], and, (b) computing the new set of candidate top-k terms for the result of the merge, which is to replace the contents of the least recent of the two windows. Merging of the hCount structures is straight-forward; each counter of the resulting merged window is set to the sum of the corresponding hCount counters from the initial windows. Computing the new set of candidate top-k terms proceeds as follows. Let $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ denote, respectively, the corresponding pools of candidate terms of the original windows w_i and w_{i+1}. The new candidate terms $\mathcal{P}'_{i+1}$ are initially set to $\mathcal{P}_i \cup \mathcal{P}_{i+1}$. Then, the size of $\mathcal{P}'_{i+1}$ is further reduced by re-applying the Chernoff bound to the merged pool, with probability again δ, but now using the new window size as the length of the batch, i.e., $2^i \times b$. As before, the estimation for the Chernoff bound requires computing the support of the k most frequent term in the pool P'_{i+1}. For this, the hCount structure of the new window is used to estimate the frequencies of all terms in $\mathcal{P}_i \cup \mathcal{P}_{i+1}$ with high accuracy. Since the tightness of the Chernoff bound increases with the length of the batch (see Equation 1), the final number of terms in $\mathcal{P}'_{i+1}$ is typically substantially smaller than the number of terms in $\mathcal{P}_i \cup \mathcal{P}_{i+1}$.

3.3 Query Execution

The *QueryManager* component is responsible for receiving the user query and evaluating it using the data structures described above. Mainly, this involves the following operations: (a) identifying the windows corresponding to the time interval specified in the query; (b) collecting the candidate top-k terms from each window; and (c) estimating their frequencies using the corresponding hCount structures, and identifying the top-k terms.

The first operation is rather straightforward. Given a query $Q =< i,\ j >$, the windows that are involved in the query execution are those covering a time

interval that overlaps with the interval requested in the query. Once these relevant windows W_Q have been identified, the *QueryManager* constructs the set of the candidate top-k terms for Q, denoted by $\mathcal{P}_Q$. This set is the union of all the candidate top-k terms in the relevant windows, i.e., $\mathcal{P}_Q = \bigcup_{w_i \in W_Q} \mathcal{P}_i$. For each of the terms in $\mathcal{P}_Q$, the term frequency is estimated using the respective hCount structures. The terms are sorted descending on their estimated frequencies, and the top-k terms are returned to the user.

Note that, the starting and ending position specified in the query will typically not coincide with window bounds. Hence, some windows will be involved only partially in query execution. For example, consider a window of size ℓ that has an overlap of size ℓ' with the query. Assume also a term t in the pool of this window, with estimated frequency f_t provided by the hCount structure of this window. Then, this estimation is adjusted as $f'_t = (\ell'/\ell) \times f_t$. Even though this relies on a uniform distribution of the occurrences of the term in the duration of this window, our results show that the introduced error from non-uniform term distributions, i.e., bursty terms, is usually small.

4 Experimental Evaluation

4.1 Experimental Setup

To evaluate our approach, we have implemented a prototype as described in Section 3. Recall that in this paper we focus on the StreamManager and QueryManager components (see Figure 1); hence, for the TermSelection component we have only implemented standard stemming and stopword filtering, and we have omitted the SourceMonitoring component, using instead the publicly available ICWSM 2009 dataset[4], which is a crawled collection of blogs. In particular, this is a large, real-world dataset comprising 44 million blog posts from the period between August 1st and October 1st, 2008. The posts are ordered by posting time, enabling us to simulate them as a stream. They are also separated by language. In our experiments, we have used only the English posts, which yields a number of 18.5 million posts. After the stemming and stopword filtering, the total number of distinct terms in the stream was 5 million, while the total number of terms was 1.68 billion. The statistics of this data collection used in our experiments are summarized in Table 1.

In the conducted experiments, we measure the performance of our system with respect to memory usage and for different query types. The examined parameters and their value ranges are summarized in Table 2. In our implementation, we set the size of the first window to be $1,000,000$ terms, which corresponds roughly to half an hour. Moreover, we set the error parameter for the Chernoff-based filtering to $\epsilon = 0.0001$, which resulted to maintaining approximately $2 \times k$ terms per window. The results of our evaluation are presented below.

[4] http://www.icwsm.org/2009/data/

Days	47
Blog posts	18,520,668
Terms	1,680,248,084
Distinct terms	5,028,825
Average blog posts per day	394,056
Average terms per post	90.72

Table 1. Statistics for the ICWSM 2009 Blog dataset (English posts)

Parameter	Values
Counters per window and total memory required	10,000 (0.45Mb), 25,000 (1.14Mb), **40,000 (1.83Mb)**, 70,000 (3.20Mb), 100,000 (4.57Mb), 130,000 (5.95Mb)
Top-k terms to retrieve	25, **50**, 75, 100, 125
Query length	$[1, \; 1.68 \times 10^9]$, $\mathbf{10^7}$

Table 2. Parameters for the experiment. The default values are emphasized.

4.2 Accuracy versus memory

First, we investigate how the amount of memory that is available to the system affects the quality of the results. Recall from Section 3 that the amount of memory used is related to: (a) the number of counters used by the hCount structure in each window (i.e., the accuracy when estimating the frequency of a term), and (b) the number of candidate terms maintained in each window (i.e., the probability to falsely prune a very frequent term). To measure the quality of our results, we compare the list $\mathcal{L}$ of top-k frequent terms computed by our method to the exact list $\mathcal{L}_0$ of top-k terms computed by the brute-force method, i.e., by counting the exact occurrences of all the terms in the query interval. In particular, we evaluate accuracy using two standard measures, recall[5] and Spearman's footrule distance [12], which are briefly explained in the following.

The recall measure expresses the percentage of the actual top-k frequent terms that have been correctly retrieved by our system, i.e., the overlap between the two lists $\mathcal{L}$ and $\mathcal{L}_0$:

$$recall(\mathcal{L}, \mathcal{L}_0) = \frac{|\mathcal{L} \cap \mathcal{L}_0|}{k} \tag{2}$$

However, simply identifying the actual top-k frequent terms is not sufficient. Since the contents of a cloud are typically visualized with different emphasis based on their frequency, identifying also the correct ordering of the terms is important. Hence, the ordering of the terms in $\mathcal{L}$ should be as close as possible to that in $\mathcal{L}_0$. To measure this, we use Spearman's footrule distance [12], a popular distance measure for comparing rankings. In particular, we use the extended version proposed by Fagin et al. [7], henceforth referred to as *Spearman's distance* (F^*), which also handles the case where the overlap of the two compared rankings is not complete. This measure is computed as follows:

$$F^*(\mathcal{L}, \mathcal{L}_0) = \sum_{i \in \mathcal{D}} |pos(i, \mathcal{L}) - pos(i, \mathcal{L}_0)| / maxF^* \tag{3}$$

[5] Since we are retrieving a fixed number of terms k, recall and precision have always the same value.

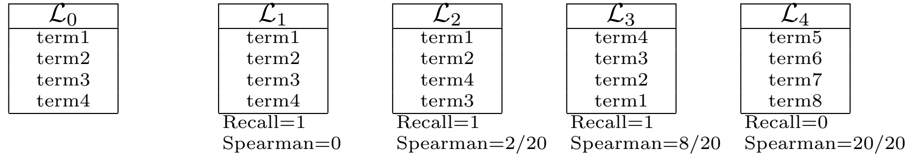

Fig. 3. Comparing various rankings, $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, $\mathcal{L}_4$, to a correct ranking $\mathcal{L}_0$.

Query	Start time	End time	Length	Involved windows
Q1	1	10000000	10000000	11
Q2	1600000001	1610000001	10000000	6, 7
Q3	1672000004	1680248081	8248078	0 - 4

Table 3. Sample queries (the most recent window is 0 and the oldest is 11).

where $\mathcal{D}$ denotes the set of terms in $\mathcal{L} \cup \mathcal{L}_0$, and function $pos(i, \mathcal{L})$ gives the position of i in the list $\mathcal{L}$ if $i \in \mathcal{L}$, or $(|\mathcal{L}| + 1)$ otherwise. $maxF^*$ denotes the maximum possible value that F^* can take, which equals to $(k + 1) \times k$. Some examples are illustrated in Figure 3.

Figure 4 displays the Spearman's distance and the recall for the three sample queries depicted in Table 3, with respect to the system's memory. We see that both quality measures increase by increasing the available system memory. This happens because with more memory available, TiTiCount+ is automatically configured with more counters per window, thereby substantially reducing the probability of collisions in the hCount structure and increasing the accuracy. The memory increase is beneficial especially for the queries that involve many windows (e.g., Q3). A near-maximum accuracy of the system is achieved by using only 3Mb memory, even for the most difficult queries that involve small parts of very old windows (e.g., Q1). Increasing the available memory beyond 3 Mb still contributes to the accuracy, though at a lower rate.

4.3 Accuracy versus query characteristics

We now examine how the accuracy of the system varies with different types of queries, and in particular with respect to (a) the query starting point, (b) the query length, and (c) the number k of most frequent terms to be retrieved. Throughout this section, we present the experimental results for three different memory configurations.

For the first set of experiments, we fix the query length to 10 million terms, and we vary the query starting point. Figure 5 presents the values for Spearman's distance and recall for the experiments with $k = 50$. As expected, the most recent queries are more accurate, since they are evaluated against windows of higher resolution. Nevertheless, even the queries that involve the least recent windows are still accurate, giving a minimum recall of 0.83 and a maximum Spearman's distance of 0.15. As shown by the previous experiments, we can further increase the accuracy for all queries by increasing the available system memory.

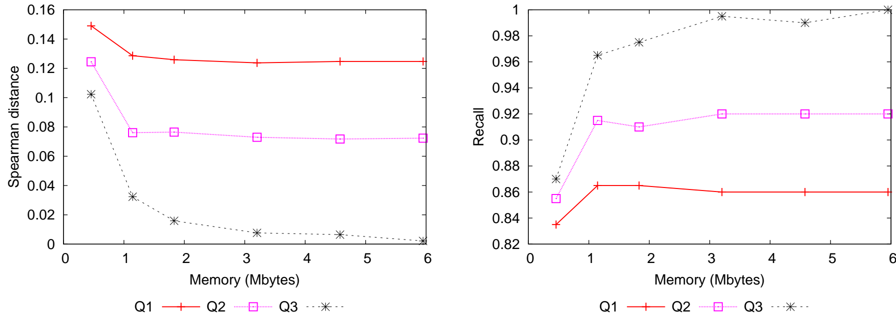

Fig. 4. Varying the allowed system memory: (a) Spearman's distance (b) Recall.

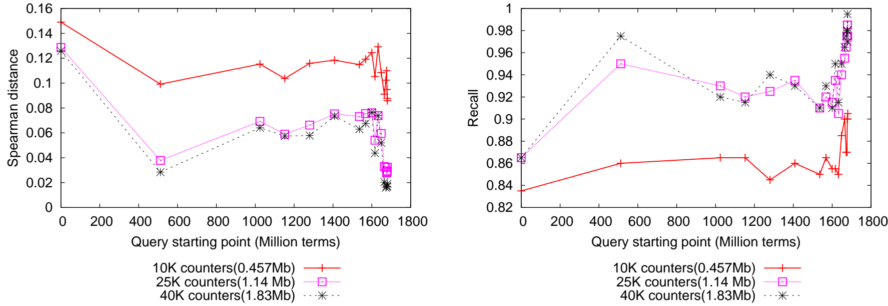

Fig. 5. Varying the query starting point: (a) Spearman's distance (b) Recall.

We also notice that both quality measures have some small peaks, e.g., at the queries starting at ca. 500 million terms. These peaks are due to the existence or absence of bursty terms in the particular query range: As explained in Section 3, when a query does not fully overlap with a window but involves only a part of it, then the computation of the term frequencies for the overlapping part are computed by multiplying the frequency given by hCount with the ratio of the overlap with the window length. This assumes a uniform distribution of the occurrences of the term in the duration of this window; therefore, if bursty terms are included in this overlap period, then a small error is introduced. In future work, we plan to deal with this issue by identifying these bursty terms and handling them differently, thereby providing even more accurate results.

We also conducted experiments varying the length of the query. For this, we fix the ending point of the query to the most recent term read from the stream, and we increase the query length with exponential steps. Figure 6 presents the two quality measures related to these experiments, for three example memory configurations. We see that the query length has a small effect on the system's accuracy. In particular, the accuracy is higher for the queries that involve the most recent windows, since these have higher resolution. Nevertheless, also the queries involving older windows are still answered with high accuracy.

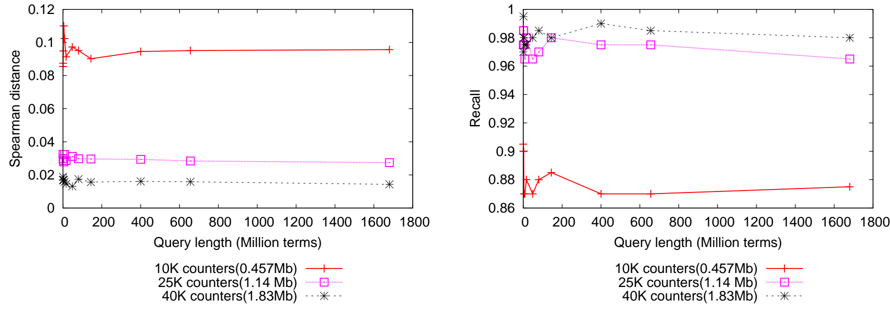

Fig. 6. Varying the query length: (a) Spearman's distance (b) Recall.

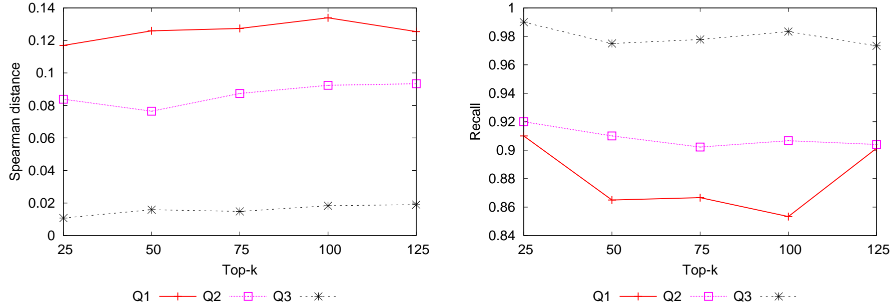

Fig. 7. Varying the desired number of top-k terms: (a) Spearman's distance (b) Recall.

Finally, we performed experiments varying the desired number of k most frequent terms to retrieve, keeping the rest of the system parameters fixed. Figure 7 presents the results for the three queries described in Table 3, for different values of k. These results were obtained using 1.83 Mb memory (40000 counters per window). We see that the value of k has no noticeable effect on the system's accuracy. This is expected, because as explained in Section 3, the system adapts to the value of top-k.

4.4 Execution time versus query characteristics

We also investigated how query execution time changes with system memory and with query characteristics. Figure 8(a) plots the execution time per query while varying the query length, for three example memory configurations. The results are averaged over 100 query executions, on a single 2.7 GHz processor. We see that query execution time scales linearly with query length. This is expected, since the query length determines the number of the involved windows, thereby also the number of number of hCount probes that need to be executed. Nevertheless, even for the query involving the whole stream range, the required execution time is negligible, below 200 msec. The execution time is also slightly

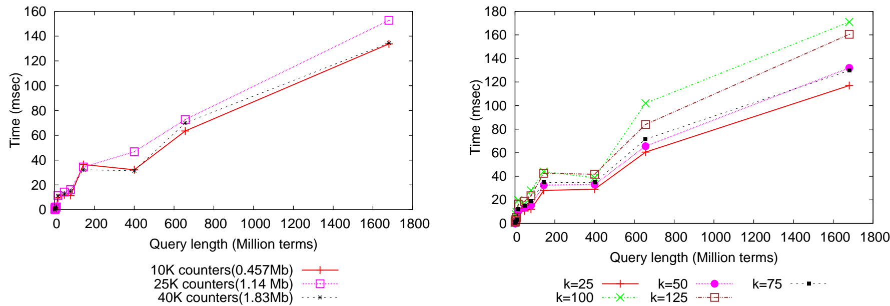

Fig. 8. Execution time: (a) varying the query length and the setup configuration, and (b) varying the query length and the number of top-k terms.

affected by the system configuration, i.e., the number of counters, but this effect is also negligible.

Figure 8(b) plots the query execution time in correlation to query length, for five values of k. The execution time scales sublinearly to the value of k. This happens because with a higher k, the number of candidate terms that need to be probed in the windows increases slightly, requiring more time. Nevertheless, all queries are executed at less than 200 msec, even for $k = 125$.

5 Conclusions

We have presented a system for generating term clouds from streaming text. The proposed method can be used to answer queries involving arbitrary time intervals, and it is very efficient with respect both to execution time and memory requirements. Our approach works by summarizing the term frequencies in compact in-memory structures, which are used to efficiently detect and maintain the top-k frequent terms with high accuracy. We have also conducted a large-scale experimental evaluation of the approach, using a real-world dataset of blog posts. The experimental results show that the proposed system offers high accuracy for identifying and correctly ordering the top-k terms in arbitrary time intervals, measured both with recall and Spearman's distance. Owing to the small memory footprint and the small query execution time, the system can be used for summarizing huge streams, and for efficiently answering top-k queries.

Our current and future work focuses mainly on two directions. First, we are working on increasing the resolution of the windows with respect to terms with uneven distribution, i.e., bursty terms. As shown in our experiments, when such terms occur in the query range, they introduce a small error in the accuracy of the results. This error can be avoided by identifying and handling such uneven term distributions differently. Furthermore, we will also focus on the other two components, the *TermSelection* and the *SourceMonitoring* component, to address the additional challenges mentioned in Section 3.1.

Acknowledgments

This work was partially supported by the FP7 EU Projects Glocal (contract no. 248984) and Sync3 (contract no. 231854).

References

1. A. Angel, N. Koudas, N. Sarkas, and D. Srivastava. What's on the grapevine? In *SIGMOD*, pages 1047–1050, 2009.
2. N. Bansal and N. Koudas. Blogscope: spatio-temporal analysis of the blogosphere. In *WWW*, pages 1269–1270, 2007.
3. N. Bansal and N. Koudas. Searching the blogosphere. In *WebDB*, 2007.
4. I. Berlocher, K.-I. Lee, and K. Kim. TopicRank: bringing insight to users. In *SIGIR*, pages 703–704, 2008.
5. Y. Chi, B. L. Tseng, and J. Tatemura. Eigen-trend: trend analysis in the blogosphere based on singular value decompositions. In *CIKM*, pages 68–77, 2006.
6. G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *PVLDB*, pages 1530–1541, 2008.
7. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, pages 28–36, 2003.
8. B. He, C. Macdonald, J. He, and I. Ounis. An effective statistical approach to blog post opinion retrieval. In *CIKM*, pages 1063–1072, 2008.
9. A. Jatowt, Y. Kawai, and K. Tanaka. Visualizing historical content of web pages. In *WWW*, pages 1221–1222, 2008.
10. C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *CIKM*, pages 287–294, 2003.
11. A. Juffinger and E. Lex. Crosslanguage blog mining and trend visualisation. In *WWW*, pages 1149–1150, 2009.
12. M. Kendall and J. D. Gibbons. *Rank Correlation Methods*. Edward Arnold, London, 1990.
13. G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. In *EDBT*, pages 391–402, 2009.
14. B. Y.-L. Kuo, T. Hentrich, B. M. Good, and M. D. Wilkinson. Tag clouds for summarizing web search results. In *WWW*, pages 1203–1204, 2007.
15. J. Leskovec, L. Backstrom, and J. M. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD*, pages 497–506, 2009.
16. N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.*, 68(4):415–430, 2009.
17. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.
18. P. Melville, W. Gryc, and R. D. Lawrence. Sentiment analysis of blogs by combining lexical knowledge with text classification. In *KDD*, pages 1275–1284, 2009.
19. M. Platakis, D. Kotsakos, and D. Gunopulos. Searching for events in the blogosphere. In *WWW*, pages 1225–1226, 2009.
20. F. I. Tantono, N. Manerikar, and T. Palpanas. Efficiently discovering recent frequent items in data streams. In *SSDBM*, pages 222–239, 2008.
21. R. C.-W. Wong and A. W.-C. Fu. Mining top-k frequent itemsets from data streams. *Data Mining and Knowledge Discovery*, 13:193–217.
22. W. Zhang, C. T. Yu, and W. Meng. Opinion retrieval from blogs. In *CIKM*, pages 831–840, 2007.