An Incremental Threshold Method for Continuous Text Search Queries

Kyriakos Mouratidis [1], HweeHwa Pang [2]

School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
[1]kyriakos@smu.edu.sg
[2]hhpang@smu.edu.sg

***Abstract*—A text filtering system monitors a stream of incoming documents, to identify those that match the interest profiles of its users. The user interests are registered at a server as continuous text search queries. The server constantly maintains for each query a ranked result list, comprising the recent documents (drawn from a sliding window) with the highest similarity to the query. Such a system underlies many text monitoring applications that need to cope with heavy document traffic, such as news and email monitoring. In this paper, we propose the first solution for processing continuous text queries efficiently. Our objective is to support a large number of user queries while sustaining high document arrival rates. Our solution indexes the streamed documents with a structure based on the principles of the inverted file, and processes document arrival and expiration events with an incremental threshold-based method. Using a stream of real documents, we experimentally verify the efficiency of our approach, which is at least an order of magnitude faster than a competitor constructed from existing techniques.**

I. Introduction

The increased use of digital information channels, such as email, electronic news feeds and automation of business reporting functions, coupled with the importance of making timely decisions, raise the need for a *continuous text search* model. In this model, new documents arrive at a monitoring server in the form of a stream. The server hosts many text search queries that are installed once and remain active until terminated by the users. Each query Q continuously retrieves, from a sliding window of the most recent documents [1], the k that are most similar to a fixed set of search terms.

For instance, a security analyst who monitors email traffic for potential terror threats would register several standing queries to identify recent emails that most closely fit certain threat profiles (e.g., emails that mention names of explosives or possible biological weapons). As another example, an investment manager who is interested in a portfolio of industries and companies would monitor newsflashes from his information provider (e.g., Reuters, Bloomberg, etc) to identify those that are relevant to his portfolio. Words related to the industries of interest can be formulated as standing text queries over the newsflashes. With the same news report stream, another user could be an entrepreneur who is tracking developments about competing products. All the above cases can be modeled as *continuous text search queries*.

We consider *count*-based and *time*-based sliding windows, whose sizes are expressed in numbers of documents or time units, respectively. An example of a query with a time-based window is "Monitor the 10 documents received in the last 15 minutes that best match the string {weapons of mass destruction}". The count-based counterpart of this example is "Continuously report the 10 documents among the 500 most recent ones that best match the string {weapons of mass destruction}". For heavy-traffic streams, the system must be able to update the ranked results of all user queries efficiently enough to keep pace with the document arrivals. That is the primary technical challenge addressed in this work.

Previous studies on document filtering (e.g., [2]) have focused on techniques for adaptively setting a similarity threshold to determine whether each document is relevant to a query. However, the problem of efficiently maintaining the list of the k most relevant documents has not been considered. Existing schemes for continuous top-k processing (e.g., [3]), on the other hand, rely on spatial index structures. In text retrieval, each term in the dictionary is considered a dimension. Since a realistic dictionary typically contains more than 100,000 terms, the dimensionality far exceeds the capabilities of any spatial index structure, thus ruling out the application of those schemes. In this paper, we present the first solution for continuous text search over high-volume document streams.

II. Problem Formulation

A text search query Q specifies a set of terms (i.e., words of interest) and a parameter k ($k \in \mathbb{N}$). Q requests for the k documents in a dataset D that are most similar to the query terms. A similarity measure that is effective in practice is the cosine similarity, which defines the score $S(d|Q)$ of a document $d \in D$ as:

$$S(d|Q) = \sum_{t \in Q} w_{Q,t} \cdot w_{d,t} \quad (1)$$

where $w_{Q,t} = \frac{f_{Q,t}}{\sqrt{\sum_{t' \in Q} f_{Q,t'}^2}}$, $w_{d,t} = \frac{f_{d,t}}{\sqrt{\sum_{t' \in T} f_{d,t'}^2}}$, T is the dictionary of all possible terms and $f_{Q,t}$ ($f_{d,t}$) is the number of times that term t appears in query Q (in document d, respectively). We focus on the cosine similarity function, but our techniques are applicable to other measures, such as the Okapi formulation.

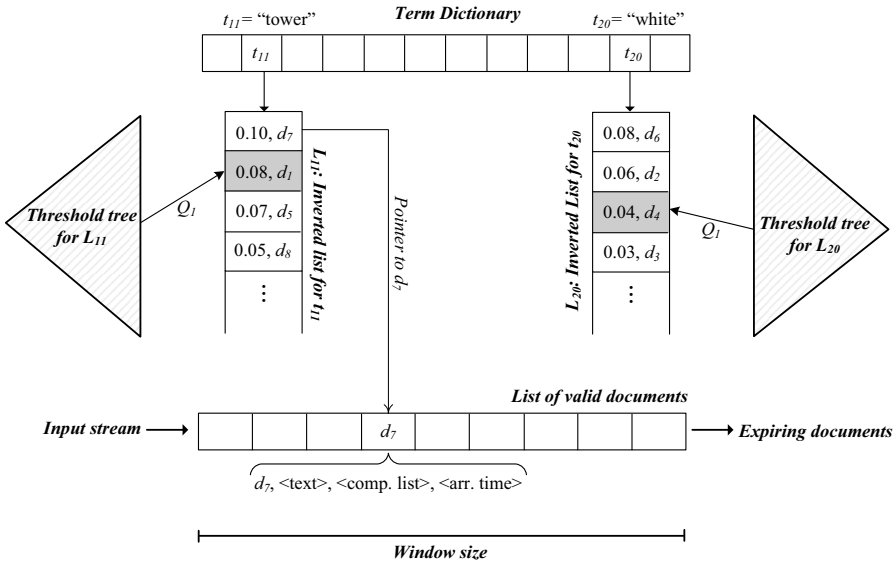


Fig. 1. Data structures

In our model, a stream of documents flows into a central server. The users register text queries at the server, which is then responsible for continuously monitoring/reporting their results. In order to cope with frequent updates, we store all the data in main memory, and design our methods with the primary goal of minimizing the CPU cost.

Each element of the input stream comprises a text document d, a unique document identifier (for simplicity, we also denote the identifier as d), the document arrival time, and a *composition list*. The composition list contains one $\langle t, w_{d,t}\rangle$ pair for each term $t \in T$ in the document. We assume a count-based sliding window [1] of length N; i.e., only the N most recent documents participate in query evaluation. We call these N documents *valid* and collectively denote them as set D. Our method can be easily adapted to time-based windows.

Each user query Q specifies the number of desired result documents k, and a set of search terms. The query string is translated to $Q = \{\langle t_1, w_{Q,t_1}\rangle, \langle t_2, w_{Q,t_2}\rangle, \ldots, \langle t_n, w_{Q,t_n}\rangle\}$, where the $w_{Q,t}$ frequencies are defined in Formula (1). At any given time, the result R of Q contains the k valid documents with the highest scores. We denote as S_k the smallest score across all documents in R.

A simple monitoring technique (termed *Naïve*) is the following. When a query Q is evaluated for the first time, R is computed by scanning all valid documents and computing their scores. In a subsequent timestamp, assume that a document d_{ins} arrives, forcing an existing one d_{del} to expire. If $S(d_{ins}|Q) > S_k$, we insert d_{ins} into R. If $d_{del} \in R$, we delete it from R. After these updates, if R contains more than k documents, we evict the last one. If R contains fewer than k documents, we recompute R by scanning through D.

Update handling in *Naïve* is very inefficient. For *each* arriving document d_{ins}, *Naïve* needs to compute $S(d_{ins}|Q)$ for *every* user query Q. Also, for each expiring document d_{del}, it needs to check whether $d_{del} \in R$ for every query. Furthermore, top-k recomputation (by scanning D) is expensive.

III. Incremental Threshold Algorithm

In this section, we introduce our *Incremental Threshold Algorithm* (ITA). ITA employs a threshold-based technique for computing the initial result for each query. The derived thresholds are subsequently updated to reflect document arrivals and expirations, so that the query results can be incrementally maintained, without processing documents that cannot affect the query.

Figure 1 depicts the data structures in our system. The valid documents D are stored in a first-in-first-out list. Arriving documents are inserted at the end of the list and expiring ones are deleted from its head. On top of the document list we build an inverted index [4]. The structure at the top of the figure is the dictionary of search terms. The dictionary entry for t points to the corresponding inverted list L_t. L_t holds an impact entry $\langle d, w_{d,t}\rangle$ for each document d that contains t, together with a pointer to d's full information in the document list. The entries in L_t are sorted in decreasing $w_{d,t}$ order. When a document d arrives (expires), an impact entry $\langle d, w_{d,t}\rangle$ is inserted into (deleted from) the inverted list of each term t in d.

For each inverted list L_t, we maintain a book-keeping structure termed *threshold tree*. It contains an entry $\langle \theta_{Q,t}, Q\rangle$ for each query Q that includes t. The $\theta_{Q,t}$ values are called *local thresholds*. The threshold tree is used to efficiently retrieve all the queries whose local thresholds are below a given parameter.

A. Initial Top-k Search

When a query Q is first submitted to the system, its top-k result is computed using an adaptation of the threshold algorithm [5]. The inverted lists of the query terms are iteratively probed from their first entry downwards. Unlike

the original threshold algorithm, we do not probe the lists in a round-robin fashion. Since the similarity function associates different weights $w_{Q,t}$ with the query terms, we favor those lists with higher such weights. Specifically, we probe the L_t with the highest $w_{Q,t} \cdot c_t$ value, where c_t is the frequency $w_{d,t}$ of the next entry in L_t.

When the next document in a list is encountered, its score is computed and it is inserted into R. Threshold $\tau = \sum_{t \in Q} w_{Q,t} \cdot c_t$ is maintained throughout the process. Once τ drops below the score of an encountered document d, we say that d is *verified*. The search terminates when k documents are verified; these documents form the query result. We keep in R all encountered documents, verified or not. The ranking of unverified documents cannot be guaranteed but they are necessary for subsequent result maintenance. Upon termination, we record τ as the *influence threshold* of Q. Also, we set local thresholds $\theta_{Q,t}$ to the the latest c_t values for each term, and insert them into the corresponding threshold tree.

B. Result Maintenance

After the initial result computation, a straightforward monitoring approach is to recompute from scratch the top-k result of each Q (using the above procedure) after every update. This, however, leads to unnecessary costs because:

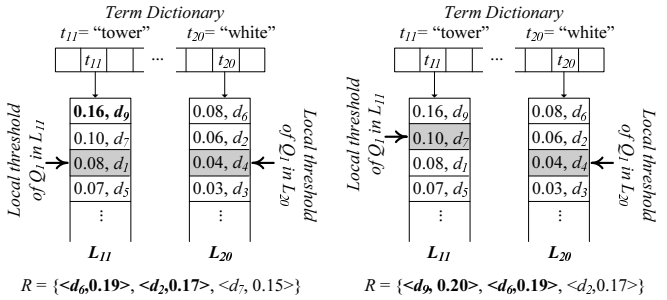
- Most of the document arrivals/expirations do not affect the query result. A query Q monitors only a very small number of search terms (say, up to a dozen from a dictionary with 100,000 terms). Consequently, most documents (either arriving or expiring) share no common terms with Q, and hence have a zero similarity score. Even when a document d contains some common terms t, the document's $w_{d,t}$ frequencies may be too small to affect the current top-k result.
- Re-scanning the inverted lists from scratch, especially those corresponding to popular terms that appear in many documents, is an expensive procedure. Having incurred the cost to compute the initial top-k result, we should reuse the work done to accelerate future re-evaluations.

Motivated by the above considerations, we propose an incremental maintenance strategy that restricts processing only to those updates that may affect the current top-k result. We base our method on the fact that *in order for an arriving/expiring document d to alter the top-k result, its score must be at least S_k, the score of the k-th document in R*. Revisiting the initial top-k search module, we observe that instead of S_k per se, we could use the influence threshold τ and its break-down into the local thresholds. Specifically, *an arriving/expiring document d may affect the result of Q if and only if it is inserted ahead of Q's local threshold in at least one of the inverted lists L_t where $t \in Q$*. Otherwise, d cannot cause any change to R and can be ignored safely. This observation allows us to reduce the maintenance cost for Q, by restricting processing to only a small region of the term frequency space. In the rest of this section we detail the processing of document arrivals and expirations separately.

Consider the arrival of a document d. We first scan its composition list and insert impact entries into the corresponding inverted lists. For each of these lists L_t we perform the following steps. We probe its threshold tree to identify all those queries Q_i where $\theta_{Q_i,t} \leq w_{d,t}$; these queries are potentially affected by d. For each of them, we compute $S(d|Q_i)$. If $S(d|Q_i) \leq S_k$, the top-k result of Q_i is not altered, but we insert d into R; this is in the same spirit as the inclusion of extra (unverified) documents into R in the initial top-k search, and its purpose will be discussed shortly. Otherwise ($S(d|Q_i) > S_k$), we insert d into the top-k result and "roll-up" accordingly the local thresholds of Q_i in all involved inverted lists, reversing the steps of the threshold algorithm in Section III-A. Note that now the c_t values are defined by the *preceding* entry in L_t, and we roll-up the list with the *smallest* $w_{Q,t} \cdot c_t$ value each time. The roll-up process stops at the last iteration where τ is still smaller than or equal to the new S_k. The rationale behind the roll-up is that, since S_k has increased, we should "shrink" the monitored region of the term frequency space in order to reduce the number of future updates that need to be handled. This is also the reason for choosing to roll-up the list with the smallest $w_{Q,t} \cdot c_t$ value. d is processed only once for each Q_i even if d ranks higher than several of Q's local thresholds, to avoid redundant computations.

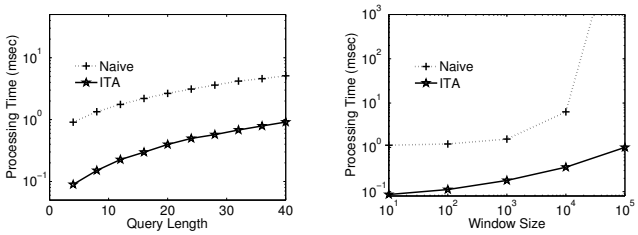
Consider the example in Figure 1. Let Q_1 be a query with search string {white white tower} and $k = 2$. Assume that its initial result is $\{\langle d_6, 0.19\rangle, \langle d_2, 0.17\rangle\}$ and that the local thresholds for t_{11} ("tower") and t_{20} ("white") are set at the entries shown shaded in their inverted lists. Suppose now that document d_9 arrives at the server as illustrated in Figure 2(a); its impact entry is highlighted in bold in the updated L_{11} . The new document is inserted above $\theta_{Q_1,t_{11}}$, the local threshold of Q_1 in L_{11}, therefore it is placed into R and its score $S(d_9|Q_1) = 0.20$ is computed. Since $S(d_9|Q_1)$ is larger than the current $S_k = 0.17$, it enters the top-2 result which now becomes $\{\langle d_9, 0.20\rangle, \langle d_6, 0.19\rangle\}$ with $S_k = 0.19$. Next, we need to roll-up the lists. The c_t values are defined by d_7 and d_2, respectively. Assuming that $w_{Q_1,t_{11}} \cdot c_{11} < w_{Q_1,t_{20}} \cdot c_{20}$, we consider L_{11} for roll-up. Assume that lifting its local threshold to d_7 would lead to $\tau = 0.18$. Since this τ is below S_k, the roll-up is feasible, $\theta_{Q_1,t_{11}}$ is set to $w_{d_7,t_{11}}$ (i.e., 0.10), and d_7 is deleted from R because it is now below all the local thresholds of Q_1. Figure 2(b) shows the updated system state. No further roll-up is possible at this time, otherwise τ would exceed S_k and leave us without enough verified documents.

Consider now the expiration of a document d. To remove it from the system, we delete its impact entry from the L_t of every term t in d. For each of these lists L_t, we probe its threshold tree to locate all the potentially affected queries Q_i, i.e., queries where $\theta_{Q_i,t} \leq w_{d,t}$. For each such query Q_i, we know that d is inside R, be it verified or not. Also, we know its score $S(d|Q_i)$; it is stored in R, so we do not need to calculate it anew. If $S(d|Q_i) < S_k$ (i.e., d is not among the top-k documents), we simply remove it from R. Otherwise, d is in the current top-k set; we delete d from R and "refill" the result (since R now contains fewer than k verified documents).



(a) d_9 arrives (b) After rolling up local thresholds

Fig. 2. Arrival handling in ITA



(a) Sensitivity to query length (b) Sensitivity to window size

Fig. 3. Processing time versus query length and window size

To perform this "refill", we do not invoke the top-k search of Section III-A from scratch. Instead, we resume the search from where it stopped previously, using the current list R (that contains both verified and unverified documents) and looking inside the involved inverted lists L_t from their local thresholds downwards. This downward search follows the same process as the initial top-k computation. The incremental refill is possible only because we keep and maintain upon updates all the unverified documents inside R. By definition, to be able to only examine the lists downwards, we should have maintained in R all valid documents d where at least one $w_{d,t}$ frequency is higher than the corresponding local threshold $\theta_{Q_i,t}$, and treat them in the same way as the initial search treats any encountered document prior to termination.

IV. Experiments

In Figure 3 we empirically compare ITA against *Naïve*. We enhance *Naïve* with the technique of [6], which retrieves the top-k_{max} documents (for an analytically derived k_{max} that is larger than k) whenever the result is computed from scratch, in order to reduce the frequency of subsequent recomputations. We form a document stream using the WSJ corpus, which comprises 172,961 articles published in the Wall Street Journal from December 1986 to March 1992. After standard stopword removal [7], the dictionary contains 181,978 terms. The WSJ documents are streamed into the monitoring system, following a Poisson process with a mean arrival rate of 200 documents/second. We generate 1,000 queries with $k = 10$ and terms selected randomly from the dictionary. We use a count-based window; the results for a time-based one are similar.

First, we investigate the effect of the number of search terms n on the performance of ITA and *Naïve*. In Figure 3(a), we set the window size to 1,000 documents, and vary n from 4 to 40. We measure the average processing time, i.e., the elapsed time between the arrival of a new document (which additionally causes the expiration of an existing one) and the point where all the query results are updated accordingly. The measurements are in milliseconds and are plotted in logarithmic scale. With more search terms (i.e., larger n), an arriving/expiring document has a higher chance of sharing common terms with the queries. This leads to an increase in the number of queries that need updating, and thus to a longer processing time. ITA is about 10 times faster than *Naïve* for queries comprising 4 search terms, and 6 times faster for 40-term queries.

In Figure 3(b), we study the effect of the sliding window size N. We set the query length to 10 terms, and vary N from 10 to 100,000 documents. A larger window holds more valid documents in the system. For *Naïve* this imposes a higher cost whenever the result needs to be recomputed, because it scans the entire D. For ITA, the inverted lists grow longer, leading to higher index update cost and slower arrival/expiration handling. ITA is 13 times faster than *Naïve* for a window size of 10, and 18 times faster when the sliding window comprises 10,000 documents. Note that the last measurement for *Naïve* is missing, because for $N > 10000$ the CPU utilization approaches 100% and the system becomes unstable. The above experiments, as well as others omitted due to lack of space, verify the general superiority of ITA over *Naïve*.

V. Conclusion

In this paper, we study the processing of continuous text queries over high-volume document streams. The problem arises in a variety of text monitoring applications, e.g., email and news tracking. To the best of our knowledge, this is the first attempt to address this problem. Our solution, termed ITA, follows the incremental evaluation paradigm and employs threshold-based techniques to reduce the processing time at the stream processing server. ITA is empirically shown to significantly outperform a solution compiled from existing methods.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *PODS*, 2002, pp. 1–16.

[2] Y. Zhang and J. Callan, "Maximum Likelihood Estimation for Filtering Thresholds," in *SIGIR*, 2001, pp. 294–302.

[3] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD Conference*, 2006, pp. 635–646.

[4] J. Zobel and A. Moffat, "Inverted Files for Text Search Engine," *ACM Computing Surveys*, vol. 38, no. 2, July 2006.

[5] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Journal of Computer and Systems Sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[6] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient Maintenance of Materialized Top-k Views," in *ICDE*, 2003, pp. 189–200.

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