Efficient Evaluation of Continuous Text Search Queries

Kyriakos Mouratidis, HweeHwa Pang

Abstract—Consider a text filtering server that monitors a stream of incoming documents for a set of users, who register their interests in the form of continuous text search queries. The task of the server is to constantly maintain 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 in main memory with a structure based on the principles of the inverted file, and processes document arrival and expiration events with an incremental threshold-based method. We distinguish between two versions of the monitoring algorithm, an eager and a lazy one, which differ in how aggressively they manage the thresholds on the inverted index. Using benchmark queries over a stream of real documents, we experimentally verify the efficiency of our methodology; both its versions are at least an order of magnitude faster than a competitor constructed from existing techniques, with lazy being the best approach overall.

1 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*. The continuous text search model involves two key notions:

- Sliding window: A common concept in the data stream literature that reflects the interest of the users in the newest available documents, the sliding window may be defined in two alternative ways. A *count*-based window contains the N most recent documents for some constant number N, whereas a *time*-based window contains only documents that arrived within the last N time units. Our rationale for adopting the sliding window model is that it fits naturally the purpose of monitoring applications. Documents reported by the text filtering server may trigger a set of predefined operations or be presented to a human user (e.g., a security analyst) for his further consideration/action. Therefore, although a document may be very relevant to a query, it is ignored (by the filtering server at least) when it falls outside the window, because it is considered as having already been brought to the attention of the end-user or the trigger mechanisms. The size of the sliding window reflects the time given to the user/trigger process to consider the document.

- Ranked query result: As the sliding window may contain numerous documents that are relevant to a query Q, the user's interest focuses on the k most relevant ones, ordered by their similarity scores. The similarity between a document d and Q is a function of their term compositions. Common similarity functions include cosine distance and the Okapi measure [2]. One could argue that the server should ignore ranking, and report all documents that score above a certain threshold (instead of the top-k ones). In instances where the search terms of a query appear in many documents (e.g., many "innocent" emails may include terrorism-related keywords/discussions after an attack takes place), the end-user or trigger mechanism that receives the results of the server will be overwhelmed by massive numbers of documents. On the other hand, on occasions when too few documents score above the threshold, the user or trigger process will be left idle, despite having the capacity to examine more documents (with lower scores). In contrast, top-k ranking regulates in a natural way the size of the output and may be configured (using parameter k) according to the capacity of the end-user or the trigger processes.

• *Kyriakos Mouratidis and HweeHwa Pang are with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902.*
E-mail: {kyriakos, hhpang}@smu.edu.sg

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 1000 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 document arrivals. That is the primary technical challenge addressed in this work.

Previous studies on document filtering (e.g., [3]) 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., [4]), 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. At its core lies a memory-based index similar to the conventional inverted file, complemented with fast update techniques and book-keeping structures. We compute the first-time result of a query with a threshold-based algorithm on the inverted lists. The thresholds derived are used for subsequent result maintenance, and specifically (i) to avoid unnecessarily processing document arrivals/expirations that cannot affect the query, and (ii) to incrementally and efficiently update the result whenever some stream event does affect the query. Depending on how promptly/aggressively the thresholds are updated when the query result changes, we distinguish between the *Eager* and *Lazy* versions of our *Incremental Threshold* algorithm (EIT and LIT, respectively). Experiments with real document streams and benchmark queries confirm the efficiency of our scheme on the whole, and show that it vastly outperforms a baseline competitor that we constructed from previous methods for related problems.

The rest of the paper is structured as follows. Section 2 provides a background on text retrieval and reviews related work. Section 3 formulates the addressed problem and describes a competitor based on existing techniques. Section 4 presents our proposed solution, including the index structures and the query processing algorithms employed. Section 5 experimentally evaluates our approach and its variants. Finally, Section 6 concludes the paper.

2 Background

2.1 Text Retrieval by Similarity

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. Text search engines typically rate the similarity of each document $d \in D$ to query Q based on these heuristics [2]: (i) terms that appear many times in a document are given more weight; (ii) documents that contain many terms are given less weight; and (iii) terms that appear in many documents are given less weight. The heuristics are encapsulated in a similarity function which uses some composition of the following statistical values:

- $f_{Q,t}$: number of times that term t appears in query Q;
- $f_{d,t}$: number of times that term t appears in document d;
- f_t: the number of documents that contain term t.

A similarity measure that is effective in practice is the cosine similarity, which defines the score of a document d with respect to a query Q, $S(d|Q)$, to be:

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

where

$$\begin{aligned} w_{Q,t} &= \frac{f_{Q,t}}{\sqrt{\sum_{t' \in Q} f_{Q,t'}^2}} \\ w_{d,t} &= \frac{f_{d,t}}{\sqrt{\sum_{t' \in T} f_{d,t'}^2}} \end{aligned}$$

and T is the dictionary of all possible terms.

Given a query, a straightforward evaluation algorithm is to compute $S(d|Q)$ for each document d in turn, and return those documents with the highest similarity scores at the end. The execution time of this algorithm is proportional to the number of documents, which is not scalable to large collections. Instead, search engines make use of an index that maps terms to the documents that contain them. The most efficient and widely used index for this purpose is the inverted file. Below we describe its most recommended variant, the *frequency-ordered inverted index* [2]. For brevity, we refer to it simply as inverted index.

The inverted index consists of two components; a *dictionary* of terms and a set of *inverted lists*. The dictionary stores for each distinct term $t \in T$ a pointer to its inverted list L_t. The inverted list of t holds an impact entry $\langle d, w_{d,t} \rangle$ for each document d that contains t, where

- d is (the identifier of) a document that contains t,
- $w_{d,t}$ is the frequency of term t in document d, as defined in Formula (1).

The entries in each list are sorted in decreasing $w_{d,t}$ order. Figure 1 shows an example of a frequency-ordered inverted index, adapted from [2].

With the inverted index, a query Q is processed as follows. The inverted lists for the terms $t \in Q$ are scanned and the partial $w_{d,t}$ scores of each encountered document d are accumulated to produce $S(d|Q)$. The k documents with the highest scores at the end are returned as the result. The frequency-ordered structure of the index allows for efficient query processing. Specifically, with the use of thresholding, only the top parts of the inverted lists need to be considered [5], [6], [7]. Such thresholding approaches are similar in principle to the general threshold algorithm described in Section 2.2.

Term t	Inverted list L_t
and $\mapsto$	$\langle 6, 0.15\rangle$
big $\mapsto$	$\langle 2, 0.14\rangle$ $\langle 3, 0.08\rangle$
house $\mapsto$	$\langle 3, 0.08\rangle$ $\langle 2, 0.07\rangle$
in $\mapsto$	$\langle 6, 0.15\rangle$ $\langle 2, 0.14\rangle$ $\langle 5, 0.12\rangle$ $\langle 1, 0.05\rangle$ $\langle 7, 0.04\rangle$ $\langle 8, 0.03\rangle$ ...
keep $\mapsto$	$\langle 5, 0.09\rangle$ $\langle 1, 0.08\rangle$ $\langle 3, 0.07\rangle$
keeps $\mapsto$	$\langle 5, 0.08\rangle$ $\langle 1, 0.05\rangle$ $\langle 6, 0.02\rangle$
night $\mapsto$	$\langle 5, 0.17\rangle$ $\langle 4, 0.12\rangle$ $\langle 1, 0.05\rangle$
tower $\mapsto$	$\langle 7, 0.10\rangle$ $\langle 1, 0.08\rangle$ $\langle 5, 0.07\rangle$ $\langle 8, 0.05\rangle$ ...
use $\mapsto$	$\langle 2, 0.14\rangle$ $\langle 4, 0.12\rangle$ $\langle 1, 0.08\rangle$ $\langle 3, 0.03\rangle$
dark $\mapsto$	$\langle 4, 0.12\rangle$ $\langle 6, 0.07\rangle$
visit $\mapsto$	$\langle 6, 0.07\rangle$
white $\mapsto$	$\langle 6, 0.08\rangle$ $\langle 2, 0.06\rangle$ $\langle 4, 0.04\rangle$ $\langle 3, 0.03\rangle$ ...

Fig. 1. Example of Inverted File

Unlike the most common processing paradigm of considering each inverted list in sequence (term-at-a-time), in *document-ordered ranking* inverted lists are processed simultaneously so that the score of a document $S(d|Q)$ is fully computed once d is encountered (in a document-at-a-time fashion) [8], [9], [10]. Consider that the inverted lists are sorted on document identifier. If impact entries are fetched in parallel from the lists involved in Q, then the partial scores of d for each $t \in Q$ are fetched concurrently from the corresponding lists, and $S(d|Q)$ is directly derived. This approach may process the entire lists, but it need not maintain partial score accumulators for each encountered document (i.e., only the list of the k highest ranking documents found so far is kept).

2.2 Related Work

A topic that is related to our problem is information filtering. The TREC conference has an active track on this field [11]. In information filtering, the objective is to deliver newly arrived documents that are above some relevance threshold to each user query [3]. The focus is on tuning the relevance threshold to adapt to the corpus and query characteristics, e.g., [12]. Each standing query is assumed to be periodically re-evaluated from scratch. This approach is prohibitively expensive for high-volume streams, which is why we focus on incremental mechanisms to update the top-k results efficiently. Another distinction is that information filtering applies a boolean relevance judgment for each arriving document, whereas we consider the more challenging problem of monitoring the top-k relevant documents for each user query. This top-k ranking is especially important in our model where high-volume streams may produce many documents that are similar to the user query at any given time, and the user needs to only focus on the most relevant ones.

The generic top-k query is also related to our problem. Given a dataset D and a preference function f, a top-k query retrieves the k tuples in D that possess the highest scores according to f (where k is a user-specified parameter). Research in this field has focused on preprocessing to accelerate top-k evaluation [13], [14], mapping of top-k queries into traditional selections [15], [16], and computing the top-k records among the results of a join operation over multiple relations [17], [18], [19], [20].

In top-k computation over distributed data repositories, there are multiple (say m) repositories, each of which keeps a list of the data tuples that is sorted on a different attribute. The *threshold algorithm* [21] combines the partial scores from these repositories (sorted lists) and locates the top-k data tuples according to a user-specified, monotone preference function f over the m attributes. The method works as follows.

Assume that f increases monotonically on the m attributes, and the repository lists are sorted in decreasing order. The lists are probed in a round-robin fashion; the next entry is popped from their head and the score of the corresponding tuple is computed from all of its attributes. A threshold c_i is maintained for each of the m lists, which is dynamically set to the value of its next entry. The value of f over all list thresholds c_i defines a global threshold τ which constitutes an upper bound on any non-encountered tuple further down the lists. When τ drops below the score of the k-th best tuple found so far, the search terminates and returns the k best tuples as the result. A top-k processing example with (an adapted version of) this technique is given in Section 4.2. The threshold algorithm has been used in various domains with many application-specific optimizations (e.g., [22], [23], [24], [25]). We note that the above method corresponds to the *random access* flavor of the threshold algorithm. Although there exists a *no random access* variant too, we focus on the former, because it is more relevant to our document streaming context as we show next.

Going beyond *snapshot* (i.e., one-time) top-k queries which report the result over the database instance at query-time and terminate, Yi et al. [26] propose algorithms for efficient maintenance of materialized top-k views in the presence of updates. A top-k' view is maintained (instead of a top-k one), where k' changes at runtime between k and some $k_{max} > k$. Specifically, when the view R is initially computed, it is populated with the top-k_{max} tuples, where $k_{max} > k$. Subsequently, updates are dealt with as follows. If an inserted tuple has a score larger than the last one in the view, then it is included in R (evicting, if necessary, R's last entry so that the number of maintained top tuples does not exceed k_{max}).

If a deleted tuple is currently in R, then it is removed. If that causes the number of entries in R to drop below k, a new search is initiated to refill R with the top-k_{max} tuples in the database. Assuming that the attribute values are uniformly distributed in their respective domains and independent from each other, the authors present an analysis that suggests setting $k_{max} = k + \sqrt{N}$, where N is the size of the database.

In [4], Mouratidis et al. address the problem of *continuous top-k monitoring*. In their model, a set of top-k queries request continuous evaluation over a stream of multidimensional points. The points are stored in main-memory and indexed in a regular grid; i.e., the space is partitioned into cells of equal side-lengths, and each cell maintains the points that fall within its extent. The authors propose a method for the first-time top-k search (when a query is registered), which considers the minimum number of cells that may contain result records, based on a geometrically inferred visiting order among them. Subsequent top-k maintenance is restricted to point updates within only the processed cells. Two maintenance policies are described: (i) recomputing the top-k result whenever some of the current top-k points expire; (ii) partially pre-computing future results, by reducing the problem to a skyline maintenance over a subset of the data points.

Although a continuous text search query can be seen as a special case of continuous top-k monitoring, the above method is not applicable to our problem. 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 the regular grid (as spatial indexing suffers from the *dimensionality curse* [27]). This is obvious from the experiments in [4], where the processing cost increases exponentially with the dimensionality and becomes prohibitive beyond 6 dimensions.

Another piece of relevant work in the data stream literature is by Babcock and Olston [28], who introduce the concept of *distributed top-k monitoring*. Their goal is to continuously report the k objects with the largest cumulative score over a set of stream sources. In order to reduce the communication cost, they maintain arithmetic constraints at the stream sources to ensure that the most recently reported answer remains valid. When a constraint is violated, the corresponding source reports it to the central server, which updates the top-k set and assigns new constraints to the sources. Our target problem is different from [28] since we deal with *text* queries over a *single* document stream. Furthermore, while Babcock and Olston aim at minimizing the network overhead, our goal is to minimize the CPU cost at the server.

Query processing aside, research in text retrieval has considered (inverted) index maintenance in the presence of updates, assuming disk-based storage and focusing mostly on document insertions [29], [30]. Instead of directly updating the index for new documents, batch insertions are used to reduce the number of disk accesses. Specifically, in the *intermittent merging* approach [31], [32], the inverted index is broken into two parts: (i) an in-memory index for recently inserted documents, and (ii) an on-disk index, which is periodically merged with the in-memory one. Although our solution involves a dynamic index, in our setting the index and documents are stored in main memory, where the above batching/merging approach is not beneficial [2]. Instead, we follow a direct update model (see Section 4.1).

3 Preliminaries

In this section, we present the problem formulation. We also describe a preliminary solution which combines existing techniques, and serves as a baseline to evaluate our methods. Table 1 summarizes the notation used in the paper.

Symbol	Description
T	dictionary of all possible terms of interest
λ	document arrival rate
N	sliding window size
D	set of valid documents (in the current window)
Q	a continuous text query
q	number of queries
n	number of query terms
R	query result list
k	number of result documents
$S(d\|Q)$	similarity score of document d w.r.t. Q
S_k	score of the k-th document in R
τ	global/influence threshold
$\theta_{Q,t}$	local threshold of query Q for term t

TABLE 1
Notation

3.1 Problem Formulation

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. As in most stream processing systems, we store all the data in main memory in order to cope with frequent updates, 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[1], and $w_{d,t}$ is as defined in Section 2. We consider the *append-only* data stream model [1], i.e., each document arriving at the server is new (as opposed to being an update over an existing document). We assume a count-based sliding window of size N so only the N most recent documents participate in query evaluation; we call these documents *valid* and collectively denote the current window contents by D. Adaptation of our methods to time-based windows is straightforward as we will show later.

In our monitoring model (assuming a count-based window), a stream event refers to the arrival of a document and the resulting expiration of another. Each stream event is fully processed (i.e., the results of all queries are updated accordingly) at the time it occurs, before handling the next event. While the alternative of periodically processing events in batches

1. As in conventional document retrieval systems, the dictionary T of all possible terms of interest is static, i.e., no terms are deleted or inserted.

at fixed timestamps may allow for computational savings, in our work stream events are processed one by one as soon as they occur, because we target time-critical applications where query results must be up-to-date. Note that in the case of a time-based window, document arrivals and expirations are decoupled; therefore, each arrival or expiration is handled as a separate event.

Each user query Q specifies the number of desired result documents k, and a set of search terms. k is query-specific, so different queries can request for a different number of result documents. 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). Any query terms that are not in the dictionary are ignored. At any given time, the query result R for Q is an ordered list of k entries, $R = \{\langle d_1, S(d_1|Q)\rangle, \langle d_2, S(d_2|Q)\rangle, \ldots, \langle d_k, S(d_k|Q)\rangle\}$, where:

- Each document $d \in R$ is inside the current window D, and has similarity score $S(d|Q)$;
- The result entries are sorted in non-increasing score order;
- All documents $d \notin R$ that are inside D have similarity scores no larger than $S(d_k|Q)$; we denote $S(d_k|Q)$ as S_k.

In this work we adopt the cosine similarity function as defined in Formula (1), which involves only the term frequencies (TF). Other measures for cosine similarity (as well as the Okapi formulation [2]) additionally exploit the notion of inverse document frequency (IDF), by introducing a multiplier $\log \frac{N}{f_t}$ (where N is the total number of documents) to the formulae for $w_{Q,t}$ and $w_{d,t}$. In the context of streaming documents, however, it is not clear whether the IDF should be measured on the entire corpus, or be revised periodically according to the recent documents. Since the choice entails an evaluation of their impact on the quality of the query results and is beyond the scope of this paper, we exclude the IDF here. When a consensus emerges on the IDF definition, it can be incorporated easily in our solution as weights associated with the query terms.

3.2 Naïve Solution

The most straightforward approach to evaluate the continuous queries defined above is to scan the entire window contents D after every update or in fixed time intervals, compute all the document scores, and report the top-k documents. This method incurs high processing costs due to the need for frequent recomputations from scratch. An intuitive way to improve performance is incremental evaluation, where the current query result is derived by only computing updates over the previous one.

A simple incremental technique is the following. D is stored in a first-in-first-out list, so that updates due to window sliding can be performed efficiently. When a query is evaluated for the first time, its result is computed from scratch. Assume that a document d_{ins} arrives subsequently, forcing an existing one d_{del} to expire. First, we process d_{ins}; if $S(d_{ins}|Q) \geq S_k$, we insert d_{ins} into R. Next, we deal with the expiring d_{del}; 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 (in case of a tie at the last position, we evict the oldest document). Otherwise, if R contains fewer than k documents, we recompute R by scanning through D.

The above technique is incremental, but still performs many recomputations from scratch. To reduce their frequency (and, thus, boost performance), we combine it with the result maintenance technique of [26] described in Section 2.2. We refer to the combined method as *Naïve* algorithm. When a query Q is first submitted to the server, its top-k_{max} documents are computed and placed into R; k_{max} is set according to the analysis in [26] (i.e., $k_{max} = k + \sqrt{N}$). Following that, R is maintained by the procedure in Algorithm 1. Note that in line 4 the result R of a query contains k' documents, where $k \leq k' \leq k_{max}$. The pseudo-code assumes a count-based sliding window. If a time-based one is used, then arrivals and expirations do not occur concurrently. In that case, an arrival (expiration) is handled by the same algorithm, except for lines 1, 8-11 (lines 2, 4-7, 12, 13, respectively) which are omitted.

Algorithm 1 *Naïve* Maintenance Algorithm

```
   algorithm Naïve Maintenance(Arriving $d_{ins}$, Expiring $d_{del}$)
1: Delete document $d_{del}$ from $D$ (the system document list)
2: Insert document $d_{ins}$ into $D$ (the system document list)
3: for all queries $Q$ in the system do
4:     Let $S_{k'}$ be the smallest score in $R$
5:     Compute $S(d_{ins}|Q)$
6:     if $S(d_{ins}|Q) \geq S_{k'}$ then
7:         Add $d_{ins}$ to $R$
8:     if $d_{del} \in R$ then
9:         Remove $d_{del}$ from $R$
10:    if $R$ contains fewer than $k$ documents then
11:        Scan $D$ and place in $R$ the top-$k_{max}$ documents w.r.t. $Q$
12:    if $R$ contains more than $k_{max}$ documents then
13:        Keep in $R$ only the top-$k_{max}$ among them
```

The *Naïve* approach is inefficient in handling updates. For *each* arriving document d_{ins}, it needs to compute $S(d_{ins}|Q)$ for *every* query Q in the system (line 5). Also, for each expiring document d_{del}, *Naïve* needs to look into the result R of every query to determine whether $d_{del} \in R$ (line 8). An additional problem of the algorithm is in its top-k computation (for first-time query evaluation and line 11 in maintenance procedure), where scanning through the entire D and computing the score of each valid document is time-consuming. Due to these inefficiencies, *Naïve* is not expected to be able to cope with high document arrival rates and a large number of user queries.

4 Incremental Threshold Algorithm

In this section, we introduce our *Incremental Threshold* algorithm. The quintessence of the algorithm is to employ threshold-based techniques to derive the initial result for a query, then continue to update the thresholds to reflect document arrivals and expirations. The thresholds are used to incrementally maintain the query result, and also to avoid processing documents that have too low a similarity score to affect the result. We first describe the supporting data structures and indexes in Section 4.1, then elaborate on the top-k computation module for first-time query evaluation in

Section 4.2. Following that, Sections 4.3 and 4.4 present the *Eager* and *Lazy* result maintenance strategies, respectively. Finally, Section 4.5 discusses the adaptation of our techniques to query-specific sliding windows.

4.1 Data Structures

Figure 2 depicts the data structures in our system. The valid documents D are stored in a single list, shown at the bottom of the figure. Each element of the list holds the stream information of a document (identifier, text content, composition list, arrival time). D contains the most recent documents for both count-based and time-based windows. Since documents expire in a first-in-first-out manner, D is maintained efficiently by inserting arriving documents at the end of the list and deleting expiring ones from its head.

On top of the list of valid documents we build an inverted index. The structure at the top of the figure is the dictionary of search terms. It is an array that contains an entry for each term $t \in T$. The dictionary entry for t stores a pointer to the corresponding inverted list L_t. L_t holds an impact entry for each document d that contains t, together with a pointer to d's full information in the document list. When a document d arrives, an impact entry $\langle d, w_{d,t} \rangle$ (derived from d's composition list) is inserted into the inverted list of each term t that appears in d. Likewise, the impact entries of an expiring document are removed from the respective inverted lists. To keep the inverted lists sorted on $w_{d,t}$ while supporting fast (logarithmic) insertions and deletions, we implement them as red-black trees [33].

For each inverted list L_t, we maintain a book-keeping structure termed *threshold tree*. It contains an entry $\langle \theta_{Q_i,t}, Q_i \rangle$ for each query Q_i that includes t. Its use and that of the $\theta_{Q_i,t}$ values, called *local thresholds*, will be explained shortly. At this point, however, we mention that the threshold tree should facilitate the efficient retrieval of all query identifiers for which $\theta_{Q_i,t}$ is less than a given number w. To achieve this, we implement the threshold tree as a red-black tree on the $\theta_{Q_i,t}$ values.

Finally, we keep a query table (omitted from the figure) which maintains the query information. Specifically, for each Q_i, the system stores the query string, the current result R, and the *influence threshold* τ (discussed later). Actually, list R in our method stores the top-k result plus some extra documents that are necessary for fast result maintenance; two separate lists could be used instead, but we choose to have a single R list for ease of presentation and implementation.

4.2 Initial Top-k Search

When a query is first submitted to the system, its top-k result is computed using the initial search module. The process is an adaptation of the threshold algorithm described in Section 2.2. Here, the inverted lists L_t of the query terms play the role of the sorted attribute lists. Unlike the original threshold algorithm, however, 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, inspired by [34], we probe the list L_t with the highest $c_t = w_{Q,t} \cdot w_{d_{nxt},t}$ value, where d_{nxt} is the next document in L_t. The global threshold τ, a notion used identically to the original algorithm, is the sum of c_t values for all the terms in Q. We exemplify this procedure in Figure 3, using the documents and index in Figure 2.

Consider query Q_1 with search string {white white tower} and $k = 2$. Let term t_{20} be "white" and t_{11} be "tower". According to Formula (1), $w_{Q_1,t_{11}} = 1/\sqrt{5}$ and $w_{Q_1,t_{20}} = 2/\sqrt{5}$; for ease of presentation, we eliminate the denominator $\sqrt{5}$ from both weights in our example. First, the server identifies the inverted lists L_{11} and L_{20} (using the dictionary hash-table), and computes the values $c_{11} = w_{Q_1,t_{11}} \cdot w_{d_7,t_{11}} = 1 \cdot 0.10$ and $c_{20} = w_{Q_1,t_{20}} \cdot w_{d_6,t_{20}} = 2 \cdot 0.08$. In Iteration 1, since c_{20} is larger[2], the first entry of L_{20} is popped; the similarity score of the corresponding document, d_6, is computed by accessing its composition list in D (assume that $S(d_6|Q_1) = 0.19$), and inserted into the tentative result list R. c_{20} is then updated to 0.12, reflecting the next entry in L_{20} (i.e., $\langle 0.06, d_2 \rangle$). Accordingly, the global threshold is set to $\tau = c_{11} + c_{20} = 0.22$.

Iteration 2 pops the entry of d_2 from L_{20} (because again $c_{20} > c_{11}$), computes its score, and inserts it into R. Iteration 3 proceeds similarly, probing however L_{11}. There are two interesting points in this iteration. First, R contains 3 entries. Since k is only 2, one could argue that the last entry in R cannot be part of the result and, thus, it can be evicted. However, our algorithm does not discard any encountered documents from R, but exploit them for query result maintenance in subsequent re-evaluations (as explained in the next section). Second, the entry $\langle d_6, 0.19 \rangle$ in R is *verified*. Since the score of d_6 exceeds the global threshold τ, there cannot be any other document with a higher score. For this reason, we show this entry in bold. Continuing the example, τ drops sufficiently low to verify the second entry in R in Iteration 4. At this point, the search procedure halts and returns $\{\langle d_6, 0.19 \rangle, \langle d_2, 0.17 \rangle\}$ as the top-2 result. Note that the "extra" entry $\langle d_7, 0.15 \rangle$ need not be verified and cannot be returned as a third top result, although it remains in R.

Upon termination, we set the *influence threshold* of Q_1 in the query table, and insert its *local thresholds* into the corresponding threshold trees. The influence threshold is the last value of τ (i.e., 0.16). There is one local threshold $\theta_{Q_1,t}$ for each $t \in Q_1$, which is set to the $w_{d_{nxt},t}$ value of the corresponding L_t. The threshold $\theta_{Q_1,t}$ is inserted into the threshold tree of L_t in the form of an entry $\langle \theta_{Q_1,t}, Q_1 \rangle$. In our example, $\theta_{Q_1,t_{11}} = 0.08$ and $\theta_{Q_1,t_{20}} = 0.04$ are inserted in the threshold trees of L_{11} and L_{20}, respectively. The influence and local thresholds are necessary for subsequent result maintenance as discussed next.

We stress here that R contains both verified and unverified documents. The former category includes documents that are guaranteed to have the top scores in D, while there is no such guarantee for the latter. The distinction between the two subsets is implicit – *those and only those R documents that have a score higher than or equal to the influence threshold τ are verified*. This property is retained in subsequent result

2. In Figure 3, the larger value between c_{11} and c_{20} in each iteration is typeset in bold.

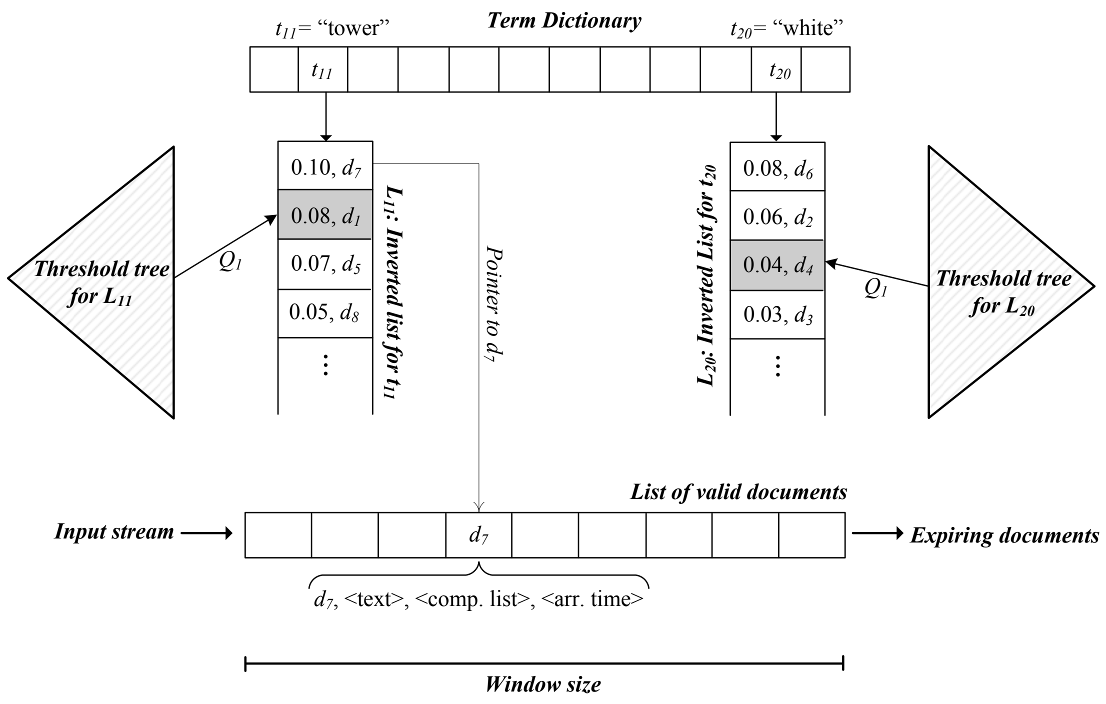


Fig. 2. Data structures

Iteration	c_{11}/c_{20}	τ	Pop Entry	$\mathcal{R}$
1	0.10/**0.16**	0.26	$\langle 0.08, d_6\rangle$ from L_{20} (with $S(d_6\|Q_1) = 0.19$)	$\langle d_6, 0.19\rangle$
2	0.10/**0.12**	0.22	$\langle 0.06, d_2\rangle$ from L_{20} (with $S(d_2\|Q_1) = 0.17$)	$\langle d_6, 0.19\rangle$, $\langle d_2, 0.17\rangle$
3	**0.10**/0.08	0.18	$\langle 0.10, d_7\rangle$ from L_{11} (with $S(d_7\|Q_1) = 0.15$)	$\langle d_6,$**0.19**$\rangle$, $\langle d_2, 0.17\rangle$, $\langle d_7, 0.15\rangle$
4	0.08/0.08	0.16	-	$\langle d_6,$**0.19**$\rangle$, $\langle d_2,$**0.17**$\rangle$, $\langle d_7, 0.15\rangle$ $\leftarrow$ *End*

Fig. 3. Initial top-2 ($k = 2$) computation for query {white white tower}

maintenance steps.

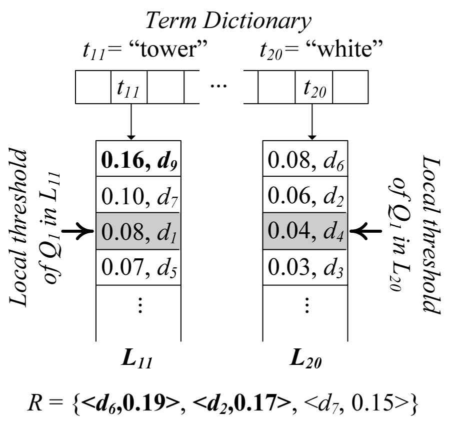
4.3 *Eager* Result Maintenance

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:

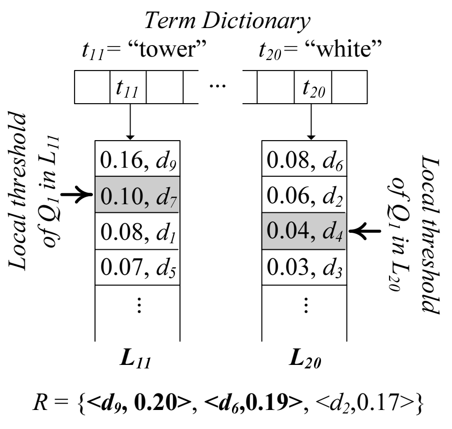
- Most of the document arrivals/expirations do not affect the query result. A query Q monitors only a very small number of search terms (say, up to a dozen from a dictionary with 100,000 terms). Consequently, most documents (either arriving or expiring) share no common terms with Q, and hence have a 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 spent 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 to only 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. The 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



(a) d_9 arrives



(b) After rolling up local thresholds

Fig. 4. Arrival handling in EIT

$(S(d|Q_i) > S_k)$, 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 4.2. 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* 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 c_t value. d is processed only once for each Q_i even if d ranks higher than several of Q_i's local thresholds, to avoid redundant computations.

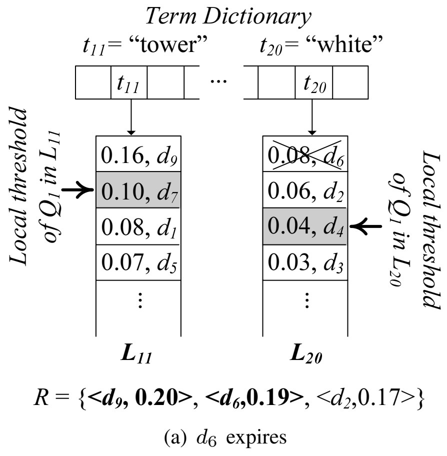
To illustrate, consider again the example in Figure 2, and assume that document d_9 arrives at the server. Its impact entry is highlighted in bold in the updated L_{11} in Figure 4(a). The new document is inserted above $\theta_{Q_1,t_{11}}$, the local threshold of Q_1 in L_{11}, therefore it is placed 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, yielding $c_{11} = 0.10$ and $c_{20} = 0.12$. Since $c_{11} < c_{20}$, we consider L_{11} for roll-up. Lifting its local threshold to d_7 would result in $\tau = 0.10+0.08 = 0.18$, which is still below S_k and is thus feasible; we update $\theta_{Q_1,t_{11}} = 0.10$ inside the threshold tree of L_{11}, set the influence threshold of Q_1 to 0.18, and delete d_7 from R because it is now below all the local thresholds of Q_1. Figure 4(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. Altering the example, if the score of d_9 was below S_k (although its impact entry is above $\theta_{Q_1,t_{11}}$ in L_{11}), we would still include it in R as an unverified entry.

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 $S(d|Q_i) \geq S_k$ (i.e., d is inside the current top-k result), and we delete d from R; we also need to "refill" the result since R now contains fewer than k verified documents.

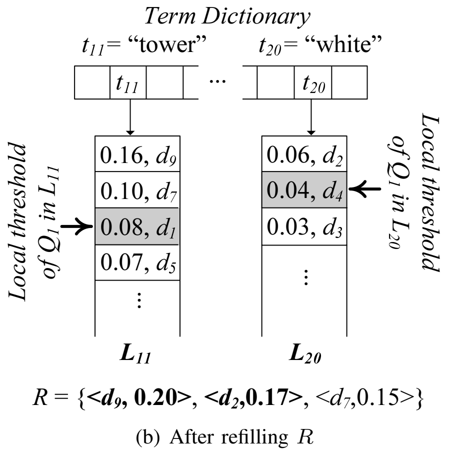
To perform this "refill", we do not re-run from scratch the top-k search of Section 4.2. Instead, we resume the search from where it had stopped previously, using the current list R (that contains both verified and unverified documents) and looking inside the involved inverted lists L_t from their local 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 for which 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 (i.e., keep them inside R).

Continuing the example in Figure 4(b), suppose that d_6 expires as shown in Figure 5(a). While deleting its impact entry from L_{20}, we detect that it is above the local threshold of Q_1. We then look into R, determine that it is part of the current top-2 result, and delete it from R. To refill the result, we resume the top-2 search from the local thresholds downwards. The current c_t values are $c_{11} = 0.10$ and $c_{20} = 0.08$. Since $c_{11} > c_{20}$, we retrieve d_7 from L_{11} and insert it into R with a score of 0.15. The search terminates here with the top-2 result $\{\langle d_9, 0.20\rangle, \langle d_2, 0.17\rangle\}$ because the updated threshold $\tau = w_{d_1,t_{11}} + 2 \cdot w_{d_4,t_{20}} = 0.16$ is lower than the new $S_k = 0.17$. The final system state and R contents are shown in Figure 5(b).

Algorithm 2 summarizes the above top-k maintenance procedure. Combined with the initial top-k search module, we term the resulting method *Eager Incremental Threshold* algorithm (EIT), in accordance with its aggressive adjustment of the local thresholds to their tightest position, i.e., to their



(a) d_6 expires



(b) After refilling R

Fig. 5. Expiration handling in EIT

highest possible value. Note that in case of a time-based window where arrivals and expirations do not necessarily happen concurrently, the former are handled by lines 1-13 and the latter by lines 14-24.

Algorithm 2 EIT Maintenance Algorithm

```
    algorithm EIT Maintenance(Arriving $d_{ins}$, Expiring $d_{del}$)
 1: Insert document $d_{ins}$ into $D$ (the system document list)
 2: for all terms $t$ in the composition list of $d_{ins}$ do
 3:     Insert the impact entry of $d_{ins}$ into $L_t$
 4:     Probe the threshold tree of $L_t$
 5:     for all queries $Q$ where $w_{d_{ins},t} \geq \theta_{Q,t}$ do
 6:         if $Q$ has not been considered for $d_{ins}$ in another $L_t$ then
 7:             Compute $S(d_{ins}|Q)$
 8:             Insert $d_{ins}$ into $R$
 9:             if $S(d_{ins}|Q) \geq$ old $S_k$ then
10:                 Update $S_k$ (since $d_{ins}$ enters the top-$k$ result)
11:                 Keep rolling-up local thresholds while $\tau \leq S_k$
12:                 Set new $\tau$ as influence threshold for $Q$
13:                 Update local thresholds of $Q$ (in threshold trees)
14: Delete document $d_{del}$ from $D$ (the system document list)
15: for all terms $t$ in the composition list of $d_{del}$ do
16:     Delete the impact entry of $d_{del}$ from $L_t$
17:     Probe the threshold tree of $L_t$
18:     for all queries $Q$ where $w_{d_{del},t} \geq \theta_{Q,t}$ do
19:         if $Q$ has not been considered for $d_{del}$ in another $L_t$ then
20:             Delete $d_{del}$ from $R$
21:             if $S(d_{del}|Q) \geq$ old $S_k$ then
22:                 Resume top-$k$ search from local thresholds
23:                 Set new $\tau$ as influence threshold for $Q$
24:                 Update local thresholds of $Q$ (in threshold trees)
```

4.4 *Lazy* Result Maintenance

In this section, we present an alternative top-k maintenance strategy, which in conjunction with the initial search module, forms our *Lazy Incremental Threshold* algorithm (LIT). The maintenance technique of LIT is based on the observation that tight (i.e., as high as possible) local and influence thresholds do not necessarily guarantee the best performance.

Consider the arrival and expiration examples in Figures 4 and 5 again. The arrival of d_9 rolls up the local threshold in L_{11} and removes d_7 from R (spending some cost on these updates), only for the next stream event (expiration of d_6) to push the threshold down and rediscover/reinsert document d_7 into R (spending some additional cost on resuming the top-k search and readjusting the threshold). We could avoid the wasted effort by allowing some leeway in the thresholds, i.e., we do not roll-up $\theta_{Q_1,L_{11}}$ but keep d_7 in R after the arrival of d_9. On the other hand, overly loose thresholds allow many document arrivals and expirations to trigger update handling unnecessarily (because they satisfy lines 5 and 18 in Algorithm 2 for many queries). The main issue that we address in this section is how loose should the thresholds be and when to tighten them. Our method is adaptive to the stream trends and, although it involves an analysis, it dynamically makes decisions based on easy-to-maintain actual measurements that are taken on the fly.

Before presenting our analysis and decision making strategy, we first explain how LIT maintenance differs from EIT. The method essentially follows Algorithm 2, but we impose a *Roll-Up Condition* (RUC) on threshold roll-up after a document's addition to the top-k result, i.e., we insert the test "if RUC = *TRUE*" before lines 11-13. Intuitively, the threshold leeway can be seen as a dynamic staircase function of time that grows until RUC is satisfied, at which point the function reverts to 0 and the thresholds are set to their tightest values. The leeway is then allowed to increase again until RUC resets it to 0, and so on.

Figure 6 visualizes the term frequency space and the thresholds of EIT and LIT at an arbitrary system snapshot, assuming the two-term query Q_1 in our running example. Let τ_E and τ_L be the influence thresholds of EIT and LIT. By design $\tau_E \geq \tau_L$ at all times, which means that R for LIT is a superset of that for EIT. Let R_E be the R list of EIT, and R_L be that of LIT.

The hollow and solid marks on each axis correspond to the local thresholds of EIT and LIT, respectively. Line l_E corresponds to EIT and is defined (by τ_E and the similarity function) as $\tau_E = w_{d,t_{11}} \cdot 1/\sqrt{5} + w_{d,t_{20}} \cdot 2/\sqrt{5}$. Above l_E, i.e., in the dark gray triangle, lie all the verified documents in R_E for EIT (i.e., *exactly* $k = 2$ *documents*). The two light gray trapezoids contain all the remaining (i.e., unverified) documents in R_E. Line l_L is similarly defined by τ_L and

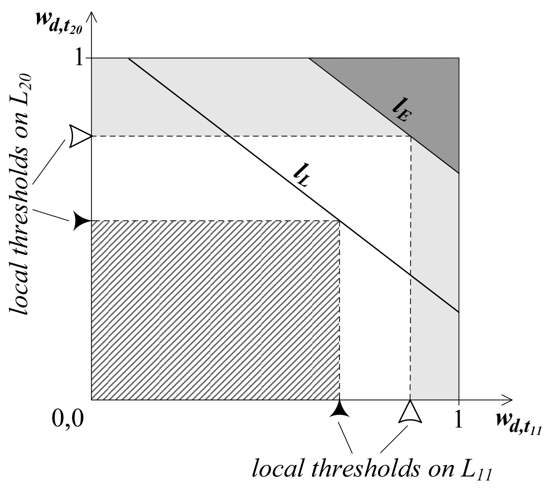


Fig. 6. LIT versus EIT monitoring in the frequency space

corresponds to LIT. LIT maintains in R_L all valid documents that lie outside the striped rectangle, i.e., R_L corresponds to the area above l_L and the trapezoids defined by its local thresholds. Documents that lie above l_L are verified (there are *more than or equal to* $k = 2$ *such documents*).

Returning to the description of LIT, assume that RUC for some query Q is checked at some point. This check involves the estimates listed in Table 2. $P_k(EX)$ is the probability that a document expiration in D leads to a deletion from the top-k result; such deletions correspond to the dark gray triangle in Figure 6. $P_L(AR)$ and $P_L(EX)$ are the probabilities that a stream event leads to an insertion into/deletion from R_L. The above three probabilities are estimated on the fly, by counting the number of intervening stream events between the two most recent ones of the corresponding type; e.g., if between the last two insertions into R_L there were 4 other stream events, then $P_L(AR) = \frac{1}{4+1} = 20\%$.

Value C_u is the cost of an insertion/deletion in R_L and may be analytically estimated; e.g., with a red-black tree implementation, C_u is logarithmic to the size of R_L. C_r is the cost of the last top-k refill (line 22 in the LIT-adapted Algorithm 2) and is estimated by the recorded numbers of score computations and insertions/deletions in R_L during the last resumption of the top-k search. The cost of a single score computation may be analytically estimated in a way similar to the cost of an insertion/deletion in R_L.

$P_E(AR)$, $P_E(EX)$, and C'_u are the EIT counterparts of $P_L(AR)$, $P_L(EX)$, and C_u, i.e., they correspond to insertions/deletions into R_E. Note that LIT does not explicitly maintain R_E. Thus, estimating $P_E(AR)$, $P_E(EX)$, C'_u requires (i) recording the local threshold values when the k-th result document was verified and (ii) checking when arrivals/expirations fall above at least one of these thresholds.

To decide RUC, we estimate the cost to process the next stream event in case we do roll-up (C_{TRUE}) and that in case we do not (C_{FALSE}). We consider these costs representative of the update handling costs in the near future. If we roll-up the thresholds of Q now, the cost at the subsequent event C_{TRUE} is identical to EIT:

Symbol	Description
$P_k(EX)$	prob. that an event is an expiration in top-k result
$P_L(AR)$	prob. that an event is an arrival into R_L
$P_L(EX)$	prob. that an event is an expiration in R_L
C_u	insertion/deletion cost in R_L
C_r	last top-k refill cost
$P_E(AR)$	prob. that an event is an arrival into R_E
$P_E(EX)$	prob. that an event is an expiration in R_E
C'_u	insertion/deletion cost in R_E

TABLE 2
Maintained estimates in LIT

$$C_{TRUE} = (P_E(AR) + P_E(EX)) \cdot C'_u + P_k(EX) \cdot C_r \quad (2)$$

The factor $(P_E(AR) + P_E(EX)) \cdot C'_u$ is the cost spent for a document insertion or deletion in R_E. The second factor $P_k(EX) \cdot C_r$ is the cost when an expiration occurs inside the dark gray triangle, i.e., in the top-k result. The probability of this case is $P_k(EX)$ and its cost is C_r because the expiration triggers a result refill (since the dark gray triangle previously contained exactly k documents).

C_{FALSE}, on the other hand, is the anticipated cost for the next stream event if we choose not to roll-up the thresholds now. The insertion/deletion costs in R_L are captured as above, the difference being that probabilities $P_E(AR)$ and $P_E(EX)$ are replaced by $P_L(AR)$ and $P_L(EX)$, and cost C'_u by C_u. Assuming the general case where $\tau_E > \tau_L$, the space above l_L contains more than k documents, so even if a verified entry is deleted, we are still left with at least k others in R_L and a top-k refill is not needed. C_{FALSE} is given by:

$$C_{FALSE} = (P_L(AR) + P_L(EX)) \cdot C_u \quad (3)$$

Whenever RUC is checked (before line 11 in the LIT-adapted Algorithm 2), we compare C_{TRUE} with C_{FALSE} to determine which course of action has a lower anticipated cost. As we demonstrate empirically in Section 5, this lazy mechanism pays off, enabling LIT to significantly outperform EIT (while both of our methods are much faster than *Naïve*).

4.5 Query-Specific Sliding Windows

So far we have assumed a system-wide sliding window, within which all documents are valid for every query. In certain applications, the users may have different recency timespans of interest and require query-specific window sizes (that are upper-bounded by the system-wide window size). The adaptation of our methods is easy, with the changes concerning primarily the deletion handling procedure. Now, each query maintains its own end-of-validity pointer at some fixed offset inside the system's document list, past which documents are treated as expired. We maintain the document identifiers included in R in a hash-table. Whenever a document d expires for a query Q, we probe its hash-table to determine quickly whether the expiration affects its result R. This check replaces the condition in line 18 of Algorithm 2. Another modification affecting the initial search and the top-k refill procedure is that all impact entries additionally include the document's arrival time, so that those entries belonging to documents that are outside the window of a query can be ignored during its top-k search/refill.

5 Experiments

5.1 Experiment Set-Up

Document stream: To form the document stream in our experiments we use the WSJ corpus, which comprises 172,961 articles published in the Wall Street Journal from December 1986 to March 1992. After removing stopwords (common words like "the" and "a" that are not useful for differentiating between documents) and words that appear in only one document, we are left with 181,978 terms for the dictionary. The removal of these words is a standard step in document indexing [35], and not specific to our methods. The WSJ documents are streamed into the monitoring system, following a Poisson process with a mean arrival rate of λ.

Query workload: Our default workload comprises queries with terms selected randomly from the dictionary. The query parameters that we examine are the number of concurrent queries q, the number of search terms n, the number of result documents k, and the sliding window size N. We use a count-based window; the results for a time-based one are similar.

Our second workload comprises the TREC-2 and TREC-3 ad-hoc queries (topics 101 to 200) [36], which contain between 2 and 20 terms each. We use the TREC queries in our last experiment only (Table 4), because they are too few (only 100) and do not allow us control over n.

Parameter setting: The algorithms evaluated are *Naïve*, EIT, and LIT. In each experiment we vary a parameter while setting the remaining ones to their default values. The defaults are: number of queries $q = 1000$, number of query terms $n = 10$, number of results $k = 10$, sliding window size $N = 1000$, arrival rate $\lambda = 200$ documents/second. Table 3 presents the examined ranges for each parameter. All experiments are performed on an Intel Xeon 3GHz CPU with 1GB RAM, operating on Redhat Linux.

Parameter	Value Range	Default
Number of queries q	32 - 100,000	1,000
Number of query terms n	4 - 40	10
Number of result documents k	10 - 200	10
Sliding window size N	10 - 100,000	1,000
Arrival rate λ (documents/sec)	100 - 6,000	200

TABLE 3
Parameters and their examined values

5.2 Experiment Results

In our first experiment we vary the number of queries q from 32 to 100,000. Figure 7(a) shows the average turnaround 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. Figures 7(b) and 7(c) drill down to the time taken to process the arrivals, and the time spent to process the expirations. Figure 7(d) shows the number of queries updated per document arrival/expiration; *Naïve* is omitted from this chart because it updates all the queries for every stream event. Figure 7(e) plots the average CPU utilization.

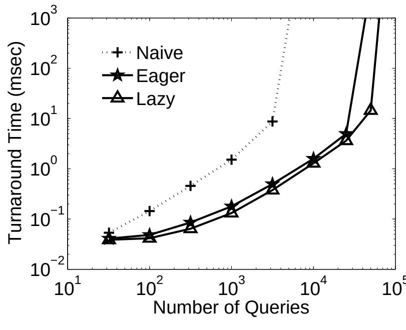
The costs of all methods increase linearly with the number of queries. *Naïve* is the slowest method, being more than an order of magnitude costlier than EIT/LIT for $q \geq 1000$. Its inefficiency, obvious in both arrival and expiration handling (Figures 7(b) and 7(c)), pushes the CPU utilization above 70% at just 4,800 queries. For $q > 10000$, the system becomes unstable and thus the last measurements for *Naïve* are missing from the figures[3]. LIT is the most efficient method, with an average 36% turnaround time improvement over EIT. Interestingly, LIT is slower than EIT in arrival handling, and updates more queries (because of its looser thresholds), but this is offset by the dominating expiration handling cost and the more frequent top-k refills of EIT. Looking at the results from a different angle, *Naïve* can process only 3,170 queries before the turnaround time exceeds one second, whereas EIT and LIT can accommodate more than 13,500 and 82,000 queries, respectively, while still delivering sub-second turnaround times.

Next, we investigate the effect of the query length n. In Figure 8 we vary the number of search terms n from 4 to 40, while setting the remaining parameters to their default values. In this and in the following figures, we omit the charts for arrival and expiration handling times as Figures 7(b) and 7(c) are representative of these costs.

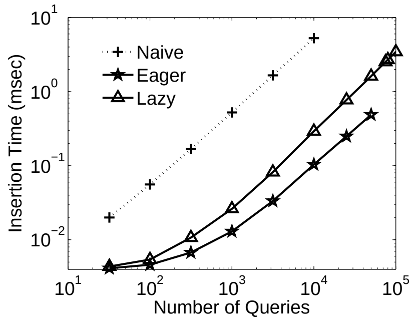
With more search terms (i.e., larger n), an arriving/expiring document has a higher chance of sharing common terms with the queries. This leads to an increase in the number of queries that need updating (see Figure 8(b)), and thus to a longer turnaround time. Overall, Figure 8(a) shows that EIT is about 10 times faster than *Naïve* for queries comprising 4 search terms, and 6 times faster for 40-term queries. The corresponding speed-up of LIT over *Naïve* is 15 and 9 times. In terms of CPU utilization (plotted in Figure 8(c)), *Naïve* does not reach the system's limit here, despite incurring significantly more computations than our algorithms.

In Figure 9, we vary N from 10 to 100,000 documents to study the effect of the sliding window size. A larger window holds more valid documents in the system. For *Naïve* which scans the entire D, this imposes a higher cost whenever the top-k_{max} result needs to be recomputed. For EIT and LIT, the inverted lists grow longer, leading to higher index update cost and slower arrival/expiration handling. Interestingly, the average number of queries that EIT updates upon each document arrival/expiration declines steadily in Figure 9(b). The reason is that a larger sliding window leaves high-scoring documents in the query results for longer periods, so the k-

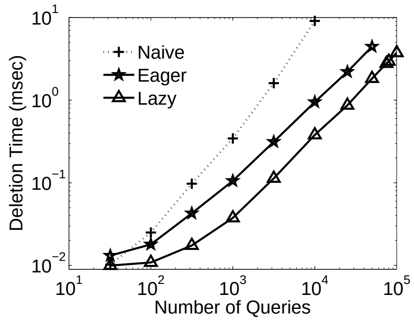
3. Note that for all three methods there is a q value (or scale in general) above which the CPU utilization becomes too high, leading to significant event queuing. In this situation, our model's requirement for up-to-date results is not feasible [37]. To deal with such a case, a practical implementation would deviate from our formal problem definition, and switch to periodic re-evaluation of all queries from scratch until the system load drops to manageable levels.



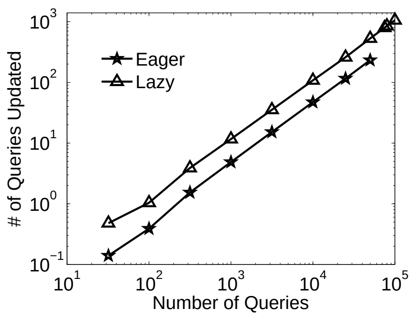
(a) Turnaround time



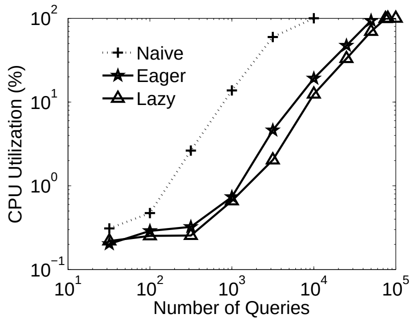
(b) Arrival-handling time



(c) Expiration-handling time

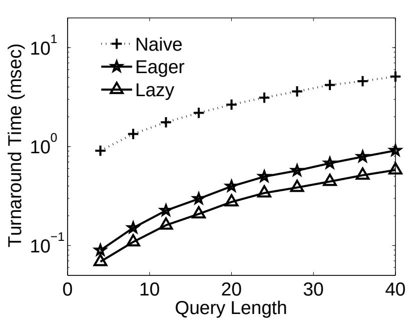


(d) Number of updated queries

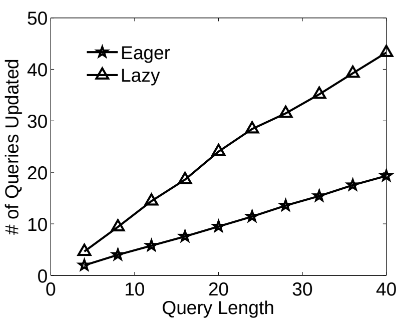


(e) CPU utilization

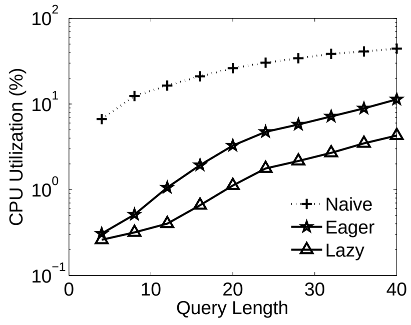
Fig. 7. Sensitivity to number of queries q



(a) Turnaround time



(b) Number of updated queries



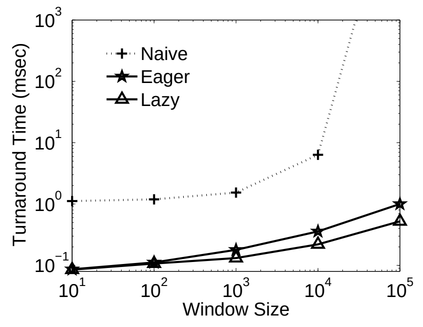
(c) CPU utilization

Fig. 8. Sensitivity to query length n

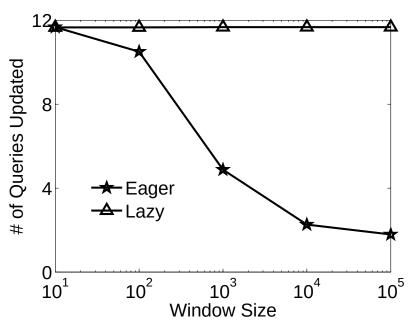
th best score S_k is higher on the average. This raises the query thresholds, with the consequence that new documents (expiring documents) are less likely to be inserted (deleted) ahead of the local thresholds in the inverted lists. This however is not the case for LIT, where the looser thresholds prevent it from benefiting from the higher S_k values. Overall, EIT is 13 times faster than *Naïve* for a window size of 10, and 18 times faster when the sliding window comprises 10,000 documents. The corresponding speed-up of LIT over *Naïve* is 14 and 28 times. Note that the CPU utilization of *Naïve* again approaches 100% for $N > 10000$ in Figure 9(c).

Next, in Figure 10 we observe the performance of the methods for various arrival rates λ. Starting from $\lambda = 100$, we increase it until we hit the limit of the methods, i.e., until the CPU saturates. Clearly, as the arrival rate increases, so does the processing cost of all the methods, since more stream events need to be handled and more changes occur in the query results. The relative performance of the methods remains the same though. *Naïve*'s turnaround time exceeds one second at just 850 documents/second, and it fails to scale beyond $\lambda = 1570$. In contrast, EIT and LIT manage to process up to 4,200 and 5,600 documents/second with sub-second turnaround times. For EIT and LIT, the CPU saturates at $\lambda = 4560$ and $\lambda = 5750$, respectively.

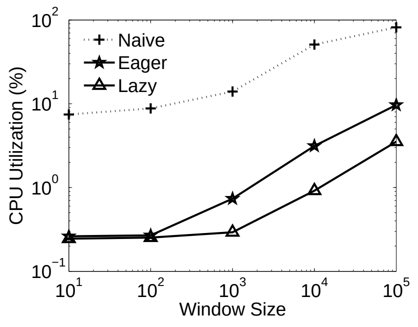
Figure 11 investigates the effect of the number of result documents k. While setting the remaining parameters to their defaults, we vary k between 10 and 200. The turnaround time of *Naïve* increases only slightly with k. In contrast, EIT initially worsens as k increases; S_k drops and so do the influence/local thresholds, leading to the processing of more updates as shown in Figure 11(b). Interestingly, beyond $k = 80$ documents, EIT's turnaround time starts to drop. On closer



(a) Turnaround time

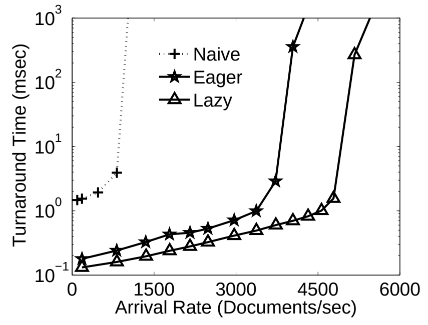


(b) Number of updated queries

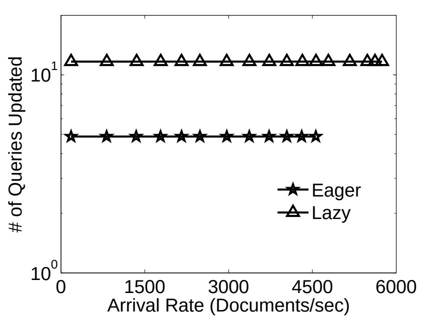


(c) CPU utilization

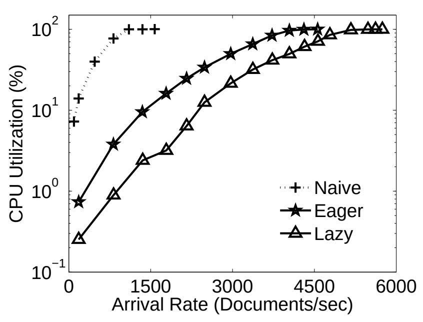
Fig. 9. Sensitivity to sliding window size N



(a) Turnaround time

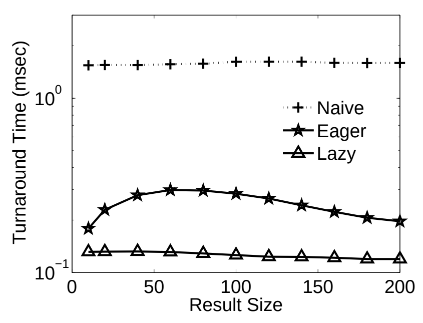


(b) Number of updated queries

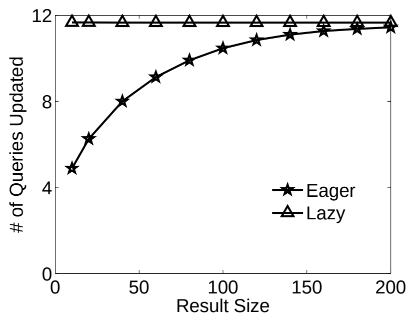


(c) CPU utilization

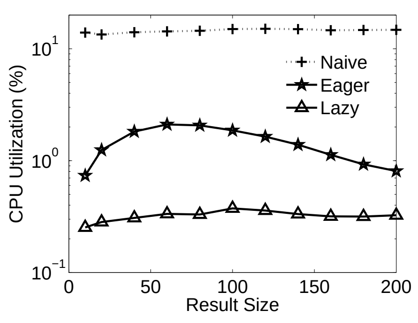
Fig. 10. Sensitivity to document arrival rate λ



(a) Turnaround time



(b) Number of updated queries



(c) CPU utilization

Fig. 11. Sensitivity to result size k

inspection, we find that a larger result size k increases the probability that there are fewer than k matching documents in the sliding window, i.e., there are not enough valid documents with non-zero similarity score. When an expiring document is removed from a query result that already contains fewer than k entries, refilling the top-k result is unnecessary (and thus avoided[4]) because D by definition does not contain any other document with non-zero score. This fact significantly lowers the expiration handling time, which is a major cost factor. LIT exhibits a similar trend to EIT, but the fluctuations of its turnaround time are much smaller. The reason is that LIT performs much fewer top-k refill operations (whose cost is most affected by k) than EIT. Overall, EIT is between 5 to 8 times faster than *Naïve*, whereas LIT achieves a 14 times speed-up over *Naïve* across all result sizes.

Finally, Table 4 presents results for the TREC queries (with $q = 100$). We use the default $k = 10$ and $N = 1000$, but set the arrival rate to $\lambda = 5000$ documents/second. The reason for choosing a higher arrival rate than the default ($\lambda = 200$) in Table 3 is that for $\lambda = 200$ the running times were too short and indistinguishable. Similar to previous experiments, our methods outperform *Naïve*, but now the performance gap

4. This is an optimization implemented in both EIT and LIT. Even if we had not implemented the optimization and instead resumed the top-k search, it would terminate directly anyway, since the local thresholds would already be at the end of the inverted lists involved.

is even wider; EIT is 30 times faster than *Naïve*, and LIT 45 times. The reason is that in the TREC workload there is some correlation between the query terms that benefits the threshold-based top-k search (compared to the synthetic queries in which term distributions are independent). The results in Table 4 testify to the practicality of our methodology and its general superiority over *Naïve*, since the TREC workload is representative of natural language queries.

Performance Measure	***Naïve***	**Eager**	**Lazy**
Turnaround time (msec)	31.02	1.01	0.68
Arrival handling time (msec)	0.057	0.015	0.036
Expiration handling time (msec)	0.089	0.108	0.048
Number of updated queries	100	7.58	17.92
CPU utilization	93.04%	72.58%	46.88%

TABLE 4
Results for TREC queries

6 Conclusion

In this paper, we study the processing of continuous text queries over document streams. These queries define a set of search terms, and request continual monitoring of a ranked list of recent documents that are most similar to those terms. 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 important problem.

We propose two incremental threshold techniques, targeted at reducing the processing cost for updating the query results. The first (EIT) eagerly adjusts the monitoring scope according to the current query result. The second (LIT) allows a dynamically adjustable leeway in the monitoring scope to facilitate faster processing of document expirations. Extensive experiments demonstrate that both approaches outperform significantly a competitor constructed from previous techniques. Between our algorithms, LIT is the method of choice in all examined settings.

Currently, our study focuses on plain text documents. A challenging direction for future work is to extend our methodology to documents tagged with metadata and documents with a hyperlink structure, as well as to specialized scoring mechanisms that may apply in these settings [38], [39].

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

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

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

[5] M. Persin, J. Zobel, and R. Sacks-Davis, "Filtered document retrieval with frequency-sorted indexes," *J. Am. Soc. Inf. Sci.*, vol. 47, no. 10, pp. 749–764, 1996.

[6] V. N. Anh, O. de Kretser, and A. Moffat, "Vector-space ranking with effective early termination," in *SIGIR*, 2001, pp. 35–42.

[7] V. N. Anh and A. Moffat, "Impact transformation: effective and efficient web retrieval," in *SIGIR*, 2002, pp. 3–10.

[8] H. R. Turtle and J. Flood, "Query evaluation: Strategies and optimizations," *Inf. Process. Manage.*, vol. 31, no. 6, pp. 831–850, 1995.

[9] M. Kaszkiel, J. Zobel, and R. Sacks-Davis, "Efficient passage ranking for document databases," *ACM Trans. Inf. Syst.*, vol. 17, no. 4, pp. 406–439, 1999.

[10] T. Strohman, H. Turtle, and W. B. Croft, "Optimization strategies for complex queries," in *SIGIR*, 2005, pp. 219–225.

[11] S. E. Robertson and D. A. Hull, "The TREC-9 Filtering Track Final Report," in *Text REtrieval Conference*, 2000, pp. 25–40.

[12] Y. Zhang and J. Callan, "YFilter at TREC9," in *Text REtrieval Conference*, 2000, pp. 135–140.

[13] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith, "The Onion technique: Indexing for linear optimization queries." in *SIGMOD Conference*, 2000, pp. 391–402.

[14] V. Hristidis and Y. Papakonstantinou, "Algorithms and applications for answering ranked queries using ranked views." *VLDB Journal*, vol. 13, no. 1, pp. 49–70, 2004.

[15] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation." *ACM Transactions on Database Systems*, vol. 27, no. 2, pp. 153–187, 2002.

[16] C.-M. Chen and Y. Ling, "A sampling-based estimator for top-k query." in *ICDE*, 2002, pp. 617–627.

[17] D. Donjerkovic and R. Ramakrishnan, "Probabilistic optimization of top N queries." in *VLDB*, 1999, pp. 411–422.

[18] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Joining ranked inputs in practice." in *VLDB*, 2002, pp. 950–961.

[19] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid, "Rank-aware query optimization." in *SIGMOD Conference*, 2004, pp. 203–214.

[20] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava, "Ranked join indices." in *ICDE*, 2003, pp. 277–288.

[21] 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.

[22] S. Chaudhuri, L. Gravano, and A. Marian, "Optimizing top-k selection queries over multimedia repositories." *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 8, pp. 992–1009, 2004.

[23] M. Theobald, G. Weikum, and R. Schenkel, "Top-k query evaluation with probabilistic guarantees." in *VLDB*, 2004, pp. 648–659.

[24] A. Marian, N. Bruno, and L. Gravano, "Evaluating top-k queries over web-accessible databases." *ACM Transactions on Database Systems*, vol. 29, no. 2, pp. 319–362, 2004.

[25] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries." in *SIGMOD Conference*, 2002, pp. 346–357.

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

[27] F. Korn, B.-U. Pagel, and C. Faloutsos, "On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 96–111, January 2001.

[28] B. Babcock and C. Olston, "Distributed top-k monitoring," in *SIGMOD Conference*, 2003, pp. 28–39.

[29] C. L. Clarke, G. V. Cormack, and F. J. Burkowski, "Fast inverted indexes with on-line update," *Tech. rep. CS-94-40, Department of Computer Science, University of Waterloo, Canada*, 1994.

[30] N. Lester, J. Zobel, and H. Williams, "Efficient online index maintenance for contiguous inverted lists," *Inf. Process. Manage.*, vol. 42, no. 4, pp. 916–933, 2006.

[31] N. Lester, A. Moffat, and J. Zobel, "Fast on-line index construction by geometric partitioning," in *CIKM*, 2005, pp. 776–783.

[32] S. Büttcher and C. L. A. Clarke, "Indexing time vs. query time: trade-offs in dynamic information retrieval systems," in *CIKM*, 2005, pp. 317–318.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[34] M. Persin, "Efficient implementation of text retrieval techniques," *Tech. rep. (thesis), Royal Melbourne Institute of Technology, Australia*, 1996.

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

[36] TREC, "Text REtrieval Conference," http://trec.nist.gov/.

[37] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," in *VLDB*, 1988, pp. 1–12.

[38] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998.

[39] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999.