Continuous Top-k Monitoring on Document Streams

Leong Hou U, Junjie Zhang, Kyriakos Mouratidis, and Ye Li

Abstract—The efficient processing of document streams plays an important role in many information filtering systems. Emerging applications, such as news update filtering and social network notifications, demand presenting end-users with the most relevant content to their preferences. In this work, user preferences are indicated by a set of keywords. A central server monitors the document stream and continuously reports to each user the top-k documents that are most relevant to her keywords. Our objective is to support large numbers of users and high stream rates, while refreshing the top-k results almost instantaneously. Our solution abandons the traditional frequency-ordered indexing approach. Instead, it follows an identifier-ordering paradigm that suits better the nature of the problem. When complemented with a novel, locally adaptive technique, our method offers (i) proven optimality w.r.t. the number of considered queries per stream event, and (ii) an order of magnitude shorter response time (i.e., time to refresh the query results) than the current state-of-the-art.

Index Terms—Top-k query; Continuous query; Document stream.

1 Introduction

In the era of big data, the amount of information made available to users far exceeds their capacity to discover and understand it. For instance, a user on Twitter may receive an overwhelming volume of notifications if her message is retweeted by too many people in a short period. Moreover, the timeliness of information filtering and delivery is of great importance. For example, a user would like to receive instant updates of the hottest topics on social news and entertainment websites (e.g., on reddit.com). Thus, the efficient *filtering* and *monitoring* of rapid streams is key to many emerging applications.

We consider *continuous top-k queries on documents* (CTQDs), a topic which has received a lot of attention recently [1], [2], [3]. In this context, a central server monitors a document stream and hosts CTQDs from various users. Each CTQD specifies a set of keywords, as explicitly given by the issuing user or extracted from her online behaviour [4], [5]. The task of the server is to continuously refresh for every CTQD the top-k most relevant documents to the keywords, as new documents stream in and old ones become too stale to be of interest.

Stock news notifications are an application domain for CTQDs. The investment decisions of a stock broker are very sensitive to news about the stocks in her portfolio. To enable timely decisions, presenting her with the most relevant news as soon as they become available is key to the success of the notification system. Similar applications can be found in monitoring live Web content, such as RSS/news feeds, blog entries, posts on social media, etc. Widely available notification systems, such as Google Alerts (google.com/alerts) and Yahoo! Alerts (alerts.yahoo.com), attest to the significance of these applications. On the other hand, these systems either work in a semi-offline manner by delivering periodic updates (e.g., daily) or allow for coarse filtering only (e.g., based on general topics, rather than sets of specific keywords). Another application domain for CTQDs are microblog real-time search services [6], [7]. Currently, these services allow the user to query (in an on-demand, one-off way) for posts that match a set of keywords. CTQDs could extend the functionality of these services by offering continuous monitoring/notifications about new posts that match the keywords.

In traditional text search, there are *snapshot* (i.e., one-off) top-k queries over static document collections. The inverted file is the standard index to organize documents [8]. It comprises a list for every term in the dictionary; the list for a term holds an entry for each document that contains the term. By sorting the lists in decreasing term frequency, and with appropriate use of thresholding (e.g., [9]), a snapshot query can be answered by processing only the top parts of the relevant lists. Due to the said sorting, we refer to that paradigm as *frequency-ordering*. This common practice for snapshot queries has been followed by most approaches for *continuous* top-k search, albeit adapted to the "standing" nature of the continuous queries and the highly dynamic characteristics of the document stream, e.g., [1], [2].

In this work, we depart from frequency-ordering, and adopt a different paradigm, namely, *identifier-ordering* (ID-ordering). Past studies on snapshot top-k queries revealed that, for sparse types of data, it may be more effective to sort the lists of the inverted file by document ID [10], thus enabling "jumps" within the relevant lists, i.e., disregarding contiguous fractions of the lists. This

- *L. H. U, J. J. Zhang and Y. Li are with the Department of Computer and Information Science, University of Macau. E-mails:* ryanlhu@umac.mo, zhjjie89@gmail.com, yb47438@umac.mo
- *K. Mouratidis is with the School of Information Systems, Singapore Management University. E-mail:* kyriakos@smu.edu.sg

is an interesting fact, which however is not directly applicable to continuous top-k queries. An application of ID-ordering to document streams would incur costly index maintenance, and also it would require repetitive query reevaluation, as it entails no mechanism to reuse past query results in response to updates.

We propose an ID-ordering methodology for CTQDs. Our methodology involves three dimensions. **First**, we reverse the role of the documents and the queries. That is, we index the (relatively static) queries and probe the streaming documents against that index, in order to eliminate the need for index maintenance due to stream events. The general idea of indexing the queries instead of the data in a streaming context is commonly referred to as *query indexing*, and has been used for many types of continuous queries (e.g., [11]). **Second**, since we index user queries which, unlike the documents, typically comprise just a few terms (i.e., they are hugely sparse), we may effectively apply ID-ordering *to the query index*. The adaptation of ID-ordering to a query index, however, is far from trivial and requires a careful redesign of its inner workings, as we explain in Section 4.2. By incorporating the first two dimensions, we already have a preliminary CTQD method (albeit just a stepping stone to our complete, most comprehensive solution), termed *Reverse ID-Ordering* (RIO). RIO is already faster than existing CTQD approaches, but we do not stop there. **Third**, we complement RIO with a novel, locally adaptive technique that produces tighter processing bounds. This technique renders the overall CTQD method optimal w.r.t. the number of considered queries per stream event, i.e., we prove that it computes the score of an arriving document w.r.t. the smallest possible number of queries, for any algorithm that follows the ID-ordering paradigm and guarantees correctness. The resulting method is our most advanced technique, called *Minimal RIO* (MRIO).

Through an extensive experimental evaluation with streams of real documents, we demonstrate that MRIO outperforms the current state-of-the-art CTQD solution by an order of magnitude. Furthermore, the "internal" comparison between MRIO and RIO reveals that the vast performance improvements achieved are primarily due to the third dimension sketched above, i.e., due to our locally adaptive technique. The contributions we make in this paper are summarised as follows:

- Our advanced approach (MRIO) outperforms the current state-of-the-art by one order of magnitude.
- MRIO employs novel bounds that offer proven *optimality* w.r.t. the number of considered queries per stream event.
- MRIO is more than two times faster than RIO, demonstrating that a skillful adaptation of ID-ordering to CTQDs alone (as in RIO) is not enough to derive the improvements achieved in this work.
- We further improve the performance of MRIO by restructuring its query index (i.e., rearranging the queries inside) to better exploit locality and strengthen the pruning effectiveness of its bounds.
- Our evaluation has a broader experimental value too, because it involves (besides the state-of-the-art for CTQDs) methods for different formulations, which perform competitively, and were never put in the same testbed before.

2 Related Work

In *information filtering* the objective is to remove from an information stream those items that are of no interest to the end users. Information filtering approaches have been studied for text streams [12], however, their focus is to determine an appropriate relevance threshold, based on the user's profile and the stream's characteristics [13]. The actual filtering involves fixed thresholds (and therefore binary relevance assessments per stream item), rather than relative similarity and ranking.

Publish−subscribe is a messaging pattern where the publishers of messages categorize their messages into classes, and the subscribers receive only those messages that fall in their classes of interest [14], [15]. Unlike CTQD, there is typically a set of predefined classes (instead of terms) and there is no notion of relative ranking. [16] does consider relative similarity, however, its goal is to identify the k most relevant queries for every newly published message. [17] proposes a probabilistic algorithm that keeps a select subset of the messages in a sliding window to support approximate top-k processing. Still in the publish−subscribe setting, [18] considers the social annotation of news articles. Specifically, given a set of news stories (documents), it maintains for each of them the k most related tweets posted. Although in [18] the documents (news stories) play the role of the standing queries, it could be applied to our setting (by treating user queries as news stories), although it is not tailored to it. We include this method in our experiments, abbreviated as TPS (for *top-k publish−subscribe*).

The top-k query is relevant to our work. Given a set of options and a scoring function defined over their attributes, the goal is to report the k options with the highest scores. Top-k processing methods have been extensively studied in relational databases; [19] offers an extensive survey. Among them, the *threshold algorithm* [9] is central to our competitors. It assumes that the options are indexed by a number of lists, each of which is responsible for one option attribute, and keeps options sorted in descending order of that attribute. The main idea is to consider options from the sorted lists in a round-robin fashion and maintain an upper bound (threshold) for the score of any unseen option. The algorithm terminates when the k-th best option found so far scores no lower than the threshold.

In the context of text search engines, similarity search is typically framed as a top-k problem over a set of documents. Terms (in queries and documents) are treated as attributes, weighted based on a standard scheme (e.g., tf-idf or Okapi BM25). The score of a document for a query is defined as a function over their common

terms, such as cosine similarity. To facilitate search, the documents are indexed by an inverted file; [8] surveys different types of inverted files and query processing techniques. The inverted file includes a sorted list per term. In the *frequency-ordering* paradigm, the sorting key is term frequency (weight), whereas in *ID-ordering* it is the document ID. In the former case, processing follows similar principles to the threshold algorithm in order to consider only the top parts of the sorted lists. In the latter case, the lists are read in their entirety but jumps over ID ranges are made possible; in Section 4.1, we describe in more detail the most efficient processing approach in this paradigm [10], [20].

Continuous versions of the top-k query have also been studied. Top-k monitoring was originally addressed over a stream of low-dimensional records [21], [22]. The proposed methods relied on spatial indices and geometric reasoning (e.g., dual space transformations), and were thus tailored to data in just a handful of dimensions. Bound by the dimensionality curse, these approaches are not applicable to document streams, because if terms were dealt with as attributes, dimensionality would be in the order of hundreds of thousands.

Rao et al. [23] consider streams of documents, but address a special version of continuous top-k queries where the query weights are equal (equivalently, the query terms are unweighted). In this version of the problem, if the search terms in a query $\mathbf{q}$ are a superset of those in another $\mathbf{q}'$, then the score of a document w.r.t. $\mathbf{q}$ is always larger than its score w.r.t. $\mathbf{q}'$. This means that if we compute the score of a stream document $\mathbf{d}$ w.r.t. $\mathbf{q}$, and that score is already smaller than the score of the k-th document in the result of $\mathbf{q}'$, we can directly infer that $\mathbf{q}'$ is not affected by $\mathbf{d}$. The proposed solution utilizes this "coverage" relationship between queries to safely ignore some of them when a document streams in. It is inapplicable to our problem, where query weights are generally not equal. Even if an extension were possible, the chances of an ad-hoc user query being completely covered by another would be too slim.

Closest related to our work are methods for continuous top-k queries (with ad-hoc term weights) on document streams. [2] assumes the sliding window model and indexes the valid documents by a (frequency-ordered) inverted file. It uses the threshold algorithm to compute the initial top-k results, and maintains pointers in the sorted lists so as to resume processing from these positions when result refill is necessary. [1] proposes an approach that also relies on frequency-ordering and the threshold algorithm, but indexes the queries instead of the stream documents. It is shown to outperform [2] and is the current state-of-the-art. We refer to it as *reverse threshold algorithm* (RTA). The same authors extended RTA to heterogeneous scoring functions, by considering hotness in addition to similarity score [24].

Vouzoukidou et al. [3] propose a CTQD method, called SortQuer. For every term in the dictionary, they represent each query $\mathbf{q}$ that includes the term as a point in a two-dimensional space – one axis corresponds to the score of the current top result document and the other axis to the query weight for that term. When a document (which includes the term) arrives, it is mapped to a region. Only queries that fall in that region could be affected by the document. Vouzoukidou et al. [3] evaluate the $k = 1$ case; in that case, SortQuer outperforms RTA. Although SortQuer was designed with the $k = 1$ case in mind, it applies easily to $k > 1$ as well, thus we include it in our experiments, and offer a comprehensive evaluation against our methods and previous art.

Some stream processing frameworks for multidimensional objects are also related to our work. Koudas et al. [25] propose an approximate k-nearest neighbor monitoring technique for streams of low-dimensional points. However, their solution is inapplicable to CTQDs, because even if stream documents were mapped to points in term space, their dimensionality would be in the order of many thousands. Zhang et al. [26] study a shared processing framework for multiple aggregation queries on a stream. It is an interesting idea to share computations among queries. However, this work is inapplicable to CTQDs because it cannot handle weighted sum aggregates for arbitrary weights. That is, even if two CTQDs share some common terms, their respective weights for these terms are generally different.

3 PRELIMINARIES

In this section, we first define the similarity metric between queries and documents, and present the model of focusing on the fresher stream content. Next, we formalize the *continuous top-k query on documents* (CTQD).

3.1 Similarity Measure

We treat the query (i.e., the set of keywords it specifies) and the documents as vectors. Letting $\mathbb{T}$ be the dictionary of all terms, a query or a document vector includes one weight per term in the dictionary. That is, a query $\mathbf{q}$ and a document $\mathbf{d}$ are represented as:

$$\mathbf{q} = \langle w_1, w_2, ..., w_{|\mathbb{T}|} \rangle; \quad \mathbf{d} = \langle f_1, f_2, ..., f_{|\mathbb{T}|} \rangle$$

The term weights w_i and f_i can be assigned to queries and documents, respectively, using any standard weighting scheme, such as tf-idf or Okapi BM25. Without loss of generality, we assume that all query and document vectors are normalized to unit magnitude. Note that typically user queries include just a few terms, which means that the query vectors are extremely sparse (i.e., include numerous zero values). The documents, too, include just a fraction of the terms in the dictionary, thus, there are many zeros in their vectors.

As per common practice [27], the textual similarity between a query $\mathbf{q}$ and a document $\mathbf{d}$ is defined as the cosine similarity of their vectors:

$$c(q, d) = \frac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}| \ |\mathbf{d}|} = \sum_{1 \leq i \leq |\mathbb{T}|} w_i f_i \tag{1}$$

The cosine similarity measure $c(q,d)$ takes values in $[0,1]$; the higher its value, the greater the textual similarity.

3.2 Document Freshness

In stream monitoring applications, the freshness of information is essential. Hence, a focusing on the most recent stream documents is required. The two prevalent formulations to achieve this focusing are the *sliding window* and the *decay model*. A sliding window only considers as valid the documents that arrived most recently; the sliding window includes either a fixed number of documents (count-based version) or those that arrived within a fixed number of time units before current time (time-based). On the other hand, in the decay model the score of the documents drops over time by applying a decay function, so that the more recent documents are favored in query answering. Our work applies to both models, but the latter (and in particular its forward-decay version) is better suited to our targeted applications, as suggested in [28], [3]. The forward-decay model works as follows.

Consider a document **d** with arrival time τ_d and cosine similarity $c(q,d)$ to query **q**. The *score* of **d** w.r.t. **q** is defined as

$$S(q,d) = c(q,d)/e^{-\lambda\Delta\tau_d} \quad (2)$$

where λ is an application-specified decay parameter, and $\Delta\tau_d$ is the difference between the document's arrival time τ_d and a reference time τ_0 in the past (e.g., the system startup time). Unlike the vanilla decay model, this forward-decay formulation associates an invariable score to each arriving document (that is, $S(q,d)$ does not change over time) while at the same time it effectively captures the freshness requirement of streaming applications [28]. Note that in this model, arriving documents receive increasingly larger scores as time goes by[1].

3.3 Problem Definition

A stream of documents flows into a central processing server, which hosts a set of CTQDs. Each CTQD specifies a set of keywords (modeled as a query vector **q**) and a positive integer k. For the sake of notation, we denote by m the number of keywords it specifies. The result of a CTQD includes the k stream documents with the highest scores $S(q,d)$ seen so far. The task of the stream server is to update all query results as new documents arrive. Document arrivals are referred to as *stream events*. The primary performance metric in our work is the time required to refresh (update) all CTQD results in response to stream events.

Although in our default setting document arrivals are the only type of stream events, in Section 6 we consider the handling of other events types, such as query insertions, query deletions, and document expirations. The handling of document expirations enables, among others, the application of our framework to the sliding window model too.

1. If at some point the scores grow beyond available numerical precision, the reference time τ_0 is reset to the current time and the scores of documents that belong to the top-k results are recomputed accordingly.

4 ID-Ordering Techniques

4.1 Index & Query Processing for Snapshot Queries

In this section, we overview the ID-ordering paradigm for snapshot queries. The documents are indexed by an inverted file, comprising a list L_i for every term t_i in the dictionary. L_i holds an entry $\langle d_{ID}, f_i \rangle$ for every document that includes term t_i (where d_{ID} is the ID of the document, and f_i its weight for term t_i). All lists are sorted in ascending document ID. The execution strategy to process a (snapshot) query **q** on this index evaluates the documents one after another from the sorted lists, but it performs "jumps" over zones of document IDs. The most efficient processing approach is *Weighted AND* [10], [20]; in the following we refer to this approach.

The maximum f_i value in each list is pre-computed and stored with it – we denote it as μ_i. Posed a query, the algorithm executes in a number of *iterations* involving only the relevant lists. For every list L_i, a cursor c_i is used to store the ID of the next unconsidered document in the list. Assume that the query involves terms $t_1, t_2, ..., t_m$. At the beginning of an iteration, the *processing order* among the lists is decided based on their c_i, i.e., by placing first the list whose cursor points at the smallest document ID, then the list whose cursor points at the next smallest document ID, etc. Assume that the processing order is $L_1 \rightarrow L_2 \rightarrow ... \rightarrow L_m$ (equivalently, in the beginning of the iteration $c_1 \leq c_2 \leq ... \leq c_m$). The invariant of the method is that, for every $i \in [1,m)$, any list after the i-th in the processing order includes no entry for document IDs in $[c_1, c_i)$. That is, the score of every document ID in the zone $[c_1, c_{i+1})$ is upper bounded by:

$$UB(i) = \sum_{1 \leq j \leq i} w_j \mu_j \quad (3)$$

Recall that μ_j is the maximum term frequency in the j-th list.

Let S_k denote the score of the k-th best document found so far. The algorithm identifies the smallest $i \in [1,m]$ for which $UB(i) \geq S_k$ and sets the corresponding cursor c_i as the *pivot*. Effectively, all documents in the zone $[c_1, c_i)$ can be safely pruned[2]. Thus, all cursors advance ("jump") to the first ID in their list that is no smaller than c_i. Now, c_i is *evaluated* (i.e., its score is computed and the query result updated accordingly) only if $c_1 = c_2 = ... = c_i$; otherwise, we proceed to the next iteration[3]. The process terminates when all cursors reach the end of their lists.

2. Because $UB(i-1) < S_k$. See also previous discussion on Equation 3.

3. If document c_i indeed belongs to the result, it will be evaluated in a future iteration. That deferred evaluation leads to fewer unnecessary evaluations [10].

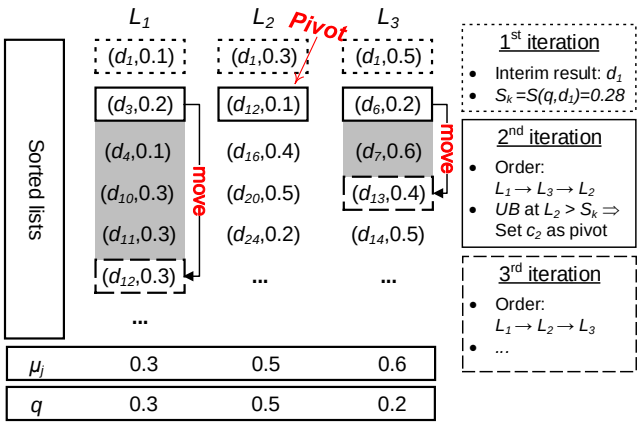


Fig. 1. Query processing in the ID-ordering paradigm

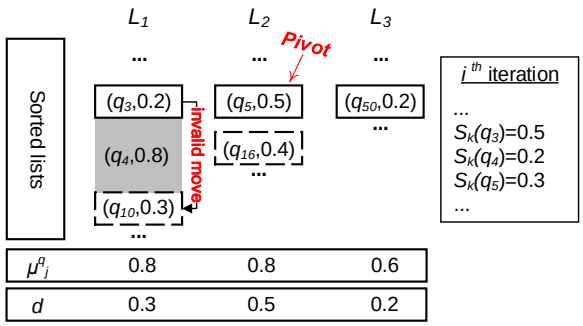


Fig. 2. Example of an invalid move

To demonstrate, we use the example in Figure 1, where $\mathbf{q} = \langle 0.3, 0.5, 0.2 \rangle$ and $k = 1$. In the first iteration, all cursors point at $\mathbf{d}_1$, which is evaluated and becomes the interim result with score $S_k = 0.28$. The cursors advance to the next position, i.e., $c_1 = 3$, $c_2 = 12$, $c_3 = 6$. When the second iteration commences, the processing order is $L_1 \rightarrow L_3 \rightarrow L_2$ because $c_1 < c_3 < c_2$. First, the upper bound at L_1 is computed as $w_1\mu_1 = 0.09$, which is smaller than S_k. Then, the upper bound at L_3 is computed as $w_1\mu_1 + w_3\mu_3 = 0.21$, which is also smaller than S_k. The upper bound at the last list (L_2), however, is $w_1\mu_1 + w_3\mu_3 + w_2\mu_2 = 0.46$, which is greater than S_k. Thus, the document pointed by the cursor of L_2 (i.e., $\mathbf{d}_{12}$) becomes the pivot. The cursors of L_1 and L_3 move to d_{12} and d_{13}, respectively, and the documents in the shaded parts of the lists are safely pruned. $c_3 \neq 12$, hence $\mathbf{d}_{12}$ is not evaluated (not in this iteration, at least). Then, a new iteration commences, and so on. When all lists are exhausted, the interim result becomes final.

4.2 Reverse ID-Ordering for CTQD processing

A straightforward approach is to index documents as per normal, and evaluate all CTQDs as explained in Section 4.1. Whenever a new document arrives, we need to (i) update the index and (ii) reevaluate each and every query. This approach is impractical because it requires excessive processing for index maintenance and, primarily, for query reevaluation from scratch. A key observation to remedy the problem is that an arriving document may affect the result of just a fraction of the queries. To effectively identify the affected queries and to avoid heavy index maintenance costs, instead of the documents, we index the queries, following the general query indexing principle.

The application of query indexing to the ID-ordering paradigm, however, is far from trivial. For snapshot queries (Section 4.1), the index holds documents that compete *among each other* in order to enter a *single result*, thus comparing against a single "result admission" score S_k. In our case, however, the index holds queries that do not compete nor otherwise affect each other, and which have their own independent S_k scores. We use a simple example to demonstrate that a direct application of ID-ordering is problematic.

Assume for simplicity that $\lambda = 0$, i.e., that there is no decay and therefore $S(q,d) = c(q,d) = \sum_{1 \leq i \leq |\mathbb{T}|} w_i f_i$ (this is an assumption we will drop later in this section). Consider Figure 2, where the sorted lists of the index hold a set of queries. Assume that a new document $\mathbf{d} = \langle 0.3, 0.5.0.2 \rangle$ streams in, and must be probed against the index to update the affected query results. Suppose the cursors of the three relevant lists point at q_3, q_5, and q_{50}, hence, the processing order in this iteration is $L_1 \rightarrow L_2 \rightarrow L_3$. Assume that the current S_k score for $\mathbf{q}_3$, $\mathbf{q}_4$, and $\mathbf{q}_5$ is 0.5, 0.2, and 0.3, respectively. A direct application of ID-ordering would identify the smallest $i \in [1,3]$ for which $UB(i) > S_k(q_i) \Leftrightarrow \sum_{1 \leq j \leq i} f_j \mu_j^q > S_k(q_i)$ (where $S_k(q_i)$ is the S_k score for $\mathbf{q}_i$, and μ_j^q is the maximum w_j value in list L_j) and set the corresponding cursor c_i as the pivot. This would set c_2 as pivot (pointing at $\mathbf{q}_5$), and move c_1 from $\mathbf{q}_3$ to $\mathbf{q}_{10}$, thus jumping over (i.e., disregarding) query $\mathbf{q}_4$. This move, however, is invalid since in reality the result of $\mathbf{q}_4$ is affected by $\mathbf{d}$, that is, $S(q_4, d) = 0.24$ is greater than the current $S_k(q_4)$ of 0.2.

The main reason that a direct application of ID-ordering fails is that every query imposes its own condition to determine whether a document arrival affects its result, i.e., it requires comparison of the list upper bounds against its own S_k value. In the *Reverse ID-ordering* (RIO) method we remedy the situation by adaptively scaling the w_i values in every query, such that they all compare the list upper bounds with the exact same S_k score. In particular, for every query, we divide its w_i values by the query's own S_k value, such that the score of its k-th result (i.e., S_k) is always 1. Accordingly, the list upper bounds $UB(i)$ are now compared to 1. That is, in each iteration RIO sets the pivot to the cursor of the first list (in the processing order) where $UB(i) > 1 \Leftrightarrow \sum_{1 \leq j \leq i} f_j \mu_j^q > 1$.

RIO proceeds like the algorithm in Section 4.1, the difference being that when the pivot in an iteration is evaluated, we need to compute the score of the arriving document $\mathbf{d}$ for the corresponding query $\mathbf{q}$ (i.e., the

query at the pivot position), and update the result of **q** if **d** scores higher than its current S_k score (i.e., higher than 1). If so, the S_k of **q** also needs to be updated and, along with it, the w_j values of **q** must be rescaled such that the new S_k is normalized to 1. The latter requires that the entries of **q** in the lists where it appears must be updated accordingly, and these updates might also affect the μ_j^q values kept for those lists[4]. We show the correctness of RIO in Lemma 1.

Lemma 1. *RIO reports the correct result for every query when a new document arrives.*

Proof: We prove the lemma by contradiction. Suppose **q** is a query affected by the new document **d**, but disregarded by RIO (i.e., "jumped over"). Without loss of generality, assume that in the iteration that jumped over **q**, the processing order of the lists was $L_1 \rightarrow L_2 \rightarrow ... \rightarrow L_m$ (where m represents the number of terms in **d**) and the pivot was c_i.

For **q** to be jumped over, its ID must be in $[c_1, c_i)$ and, thus, it cannot appear in any list past the i-th in the processing order, i.e., **d** and **q** could have in common only terms $t_1, t_2, ..., t_{i-1}$. For the pivot to be c_i, it must hold that $UB(i-1) \leq 1$ (because $UB(i)$ is the first upper bound in the processing order that exceeds 1). Thus, $\sum_{1 \leq j \leq i-1} f_j \mu_j^q \leq 1$. On the other hand, for any w_j value in **q** it holds that $w_j \leq \mu_j^q$. From the last two inequalities, we derive that $\sum_{1 \leq j \leq i-1} f_j w_j \leq 1$. The left part of the inequality is the score of **d** for query **q**. Since the score is no greater than 1, **d** does not belong to (equivalently, does not affect) the result of **q**, which contradicts the hypothesis. □

Taking decay into account: For simplicity, so far we have assumed that $\lambda = 0$. In the general case where $\lambda \geq 0$, the only necessary modification regards the upper bound computation as

$$UB(i) = \sum_{1 \leq j \leq i} f_j \mu_j^q / e^{-\lambda \Delta \tau_d} \tag{4}$$

To see why this is necessary, we refer to the proof of Lemma 1. The proof for $\lambda \geq 1$ follows the same lines, however, the derivation from $UB(i-1) \leq 1$ (in the second paragraph of the proof) now translates to $\sum_{1 \leq j \leq i-1} f_j \mu_j^q / e^{-\lambda \Delta \tau_d} \leq 1 \overset{w_j \leq \mu_j^q}{\Rightarrow} \sum_{1 \leq j \leq i-1} f_j w_j / e^{-\lambda \Delta \tau_d} \leq 1 \Rightarrow S(q,d) \leq 1$ (and therefore **d** does not affect **q**).

Optimization (early termination): During the execution of RIO, some lists may be exhausted. If the upper bound on the remaining lists is already not greater than 1, the score of any unseen document cannot be greater than 1 too, thus RIO can safely terminate.

4. A lazy update strategy is possible in the index. The w_j values do not appear in the $UB(i)$ expression (recall that $UB(i) = \sum_{1 \leq j \leq i} f_j \mu_j^q$), thus the only information that needs to be explicitly updated to preserve RIO correctness are the μ_j^q values. If the previous w_j values of **q** were not defining any μ_j^q value, then the updated w_j of **q** need not be reflected explicitly in the index.

Algorithm 1 presents the pseudo-code for RIO. Lines 3-4 implement the early termination optimization. For easy presentation, we assume that the order decided in line 5 is $L_1 \rightarrow L_2 \rightarrow ... \rightarrow L_m$ (equivalently, that $c_1 \leq c_2 \leq ... \leq c_m$). An implementation note on line 7 is that $UB(i)$ computation can be performed incrementally (i.e., based on $UB(i-1)$). The savings from incremental upper bound computation are significant, if one considers that m can be very large.

Algorithm 1 RIO(INCOMING DOCUMENT **D**)

```
1:  set all cursors c_i to the beginning of their lists
2:  while the relevant lists are not exhausted do
3:      if sum of f_j μ_j^q for all non-exhausted lists ≤ 1 then
4:          return
5:      decide execution order of relevant lists
6:      for i ← 1 to m do
7:          if UB(i) > 1 then
8:              advance c_1, c_2, ..., c_{i-1} until their IDs ≥ c_i
9:              if c_1, c_2, ..., c_i point at same query q then
10:                 if S(q, d) > 1 then
11:                     insert d into the top-k result of q
12:                     reflect new S_k(q) to w_j values of q
13:                     advance c_1, c_2, ..., c_i to next position
14:             break (goto line 2)
```

5 MINIMAL REVERSE ID-ORDERING

In this section, we present *Minimal RIO* (MRIO), our most advanced algorithm. MRIO builds on RIO, but enhances (tightens) its bounds via a novel, locally adaptive approach. This approach renders MRIO optimal (minimal) in terms of the number of iterations required to process a document arrival. In Section 5.1, we analyze (analytically and quantitatively) RIO to gain insight into the main factors that determine its performance, and to motivate MRIO. In Section 5.2, we describe MRIO and prove its optimality. Finally, in Section 5.3, we describe an optimization regarding the structure of the inverted file that significantly improves performance.

5.1 Analysis of RIO

Let $\mathcal{I}$ be the number of iterations executed by RIO and $|L|$ be the average length of a sorted list in the query index. Table 1 presents the key cost factors in RIO, referring to the specific lines in Algorithm 1. Recall that m denotes the number of terms in the arriving document **d**. We break down the costs into two types, ρ_i and ρ_c, i.e., those dependent on the number of iterations $\mathcal{I}$ and those independent of it. The second category includes only the advancement of cursors, accounting for a total of $O(m \cdot |L|)$ since the m relevant lists are scanned in their entirety. The overall complexity of RIO is $O(\mathcal{I} \cdot m \log m + m \cdot |L|)$.

To quantify ρ_i and ρ_c, we measure them empirically in our default experiment setting (details to be given in

TABLE 1
Complexity of Key Steps in Algorithm 1

Line #	Action	Time	Cost Type
5	*sorting*	$O(\mathcal{I} \cdot m \log m)$	ρ_i
7	*UB computation*	$O(\mathcal{I} \cdot m)$	
9	*position check*	$O(\mathcal{I} \cdot m)$	
10	*score computation*	$O(\mathcal{I} \cdot m)$	
8 and 13	*moving cursors*	$O(m \cdot \lvert L \rvert)$	ρ_c

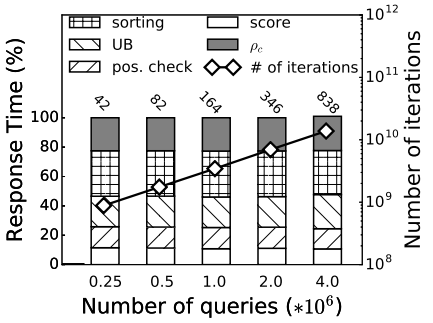


Fig. 3. Time breakdown and no. of iterations in RIO

Section 7). Specifically, in Figure 3 we vary the number of CTQDs in the system and present the breakdown of the response time in RIO into the different costs of Table 1. The left vertical axis represents percentage. The number on top of each bar is the total response time (in ms). In the same chart we also plot the number of iterations $\mathcal{I}$, illustrated by diamonds; the right vertical axis corresponds to $\mathcal{I}$ and is in logarithmic scale. We observe that ρ_i vastly dominates the overall cost[5]. We also observe that the number of iterations grows quickly. A reason is that as the number of queries increases, so does the length of the lists. Another reason is that *the more the queries the larger the μ_j^q values, i.e., the less effective the pruning by upper bounds*. This motivates our most advanced approach, MRIO, which uses (provably) the tightest possible bounds, regardless of the number of queries or any other system parameter.

5.2 Minimizing Iterations in MRIO

Having established that the response time of RIO is dominated by the ρ_i type costs, and given that all costs of that type are proportional to $\mathcal{I}$ (see Table 1), the ideal way to improve performance is to reduce the number of iterations required. To achieve that, we will need to perform as large jumps in the relevant lists as possible. In turn, what determines the length of the jumps (equivalently, the amount of pruned queries) is the tightness of the upper bounds $UB(i)$. Tightening the upper bounds is what we are set to achieve in this section.

The upper bound $UB(i)$ in RIO (Equation 4) is very loose because it is derived from the maximum w_j value in each involved list *in its entirety*, i.e., μ_j^q. The key idea in MRIO is to replace μ_j^q with the maximum w_j value among exactly those queries considered for pruning, i.e., those that will be jumped over if c_i is set as the pivot.

To formalize, we introduce the concept of the *processing zone*. Assume that the processing order is $L_1 \rightarrow L_2 \rightarrow ... \rightarrow L_m$. Recall from the discussion on Equation 3 for standard ID-ordering that the i-th upper bound regards IDs in $[c_1, c_{i+1})$ (except for $i = m$ where the ID range is $[c_1, c_m]$) – it is exactly these IDs that are candidates for pruning when the i-th upper bound is considered. We call that ID range the i-th processing zone. We also introduce function $\psi_j(i)$ which returns the maximum w_j value in list L_j for query IDs in the i-th processing zone:

$$\psi_j(i) = \begin{cases} \text{max. } w_j \text{ for query IDs in } [c_1, c_{i+1}) & \text{if } i \in [1, m) \\ \text{max. } w_j \text{ for query IDs in } [c_1, c_m] & \text{if } i = m \end{cases} \quad (5)$$

The upper bounds $UB^*(i)$ in MRIO are defined as:

$$UB^*(i) = \sum_{1 \leq j \leq i} f_j \psi_j(i) / e^{-\lambda \Delta \tau_d} \quad (6)$$

Essentially, Equation 6 is derived from Equation 4 by replacing the μ_j^q values with their local counterparts $\psi_j(\cdot)$ for the respective processing zone, i.e., with the *exact* maximum w_j values for *those and only those* queries that will be pruned if $UB^*(i) \leq 1$. The resulting pruning is therefore safe, and at the same time the upper bounds $UB^*(i)$ are the tightest possible. We refer to $UB^*(i)$ as the *local* upper bound for list L_i. The pseudo-code for MRIO is the same as Algorithm 1 by replacing the condition in line 7 with $UB^*(i) > 1$.

Lemma 2. *MRIO invokes the minimum possible number of iterations subject to the ID-ordering execution paradigm.*

Proof: The invariant of the ID-ordering paradigm [10] (described above Equation 3) ensures that the iterations partition the sequence of all query IDs into disjoint ranges; the range for an iteration extends from the smallest cursor in the beginning of the iteration to the pivot it eventually sets. The paradigm itself imposes two limitations: (i) the pivot must be selected among the original cursor positions in the beginning of the iteration, and (ii) only the pivot can be evaluated, i.e., have its score accumulated, within the iteration[6]. To demonstrate minimality, it suffices to show that in MRIO these ranges are the longest possible to uphold correctness and abide by the said limitations, equivalently, that each iteration sets the pivot to the largest possible among the initial cursor positions. Because $\psi_j(i)$ is the exact maximum w_j value in the i-th processing zone, and since we cannot accumulate the score of any non-pivot query across sorted lists,

5. For completeness, we mention that sorting is the main bottleneck of the total response time. This is because a document typically contains several hundreds or thousands of terms.

6. This is the quintessence of ID-ordering. If the limitation is dropped, every query between c_1 and the currently examined c_i could be evaluated, thus leading to greater jumps. That, however, defies the purpose of ID-ordering which is to avoid as many evaluations as possible.

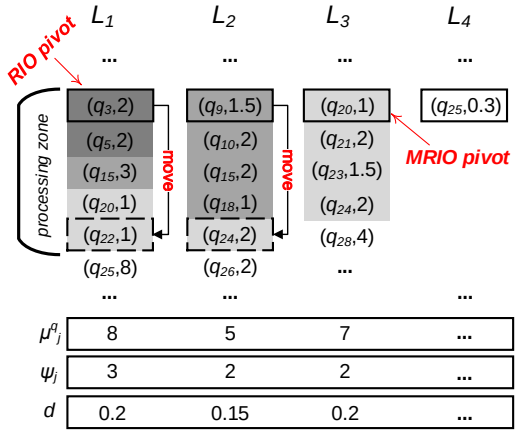


Fig. 4. Example of MRIO

$UB^*(i)$ is already the tightest upper bound estimation possible that guarantees correctness of the jump. Due to the monotonicity of $UB^*(i)$ to i, MRIO always chooses the largest permissible pivot position. □

Note that MRIO is minimal w.r.t. the structure of the inverted index, meaning that the algorithm has no control over the query index, and therefore a better structured index could lead to fewer iterations. Minimality w.r.t. a given index is common among optimality proofs for index-based algorithms. For instance, the I/O optimality of the well-known *best-first* nearest neighbor algorithm holds for a fixed R-tree index [29].

In Figure 4, we present an example of MRIO processing and demonstrate its difference from RIO, assuming for simplicity that $\lambda = 0$. The μ_j^q values and the vector of the arriving document $\mathbf{d}$ are shown at the bottom of the figure. In the beginning of the current iteration, the cursors in the four relevant lists are 3, 9, 20, and 25, thus the processing order is $L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow L_4$. In RIO, the first cursor $c_1 = 3$ would be set as pivot because $UB(1) = f_1\mu_1^q = 1.6 > 1$. In MRIO, however, the 1-st processing zone is $[c_1, c_2) = [3, 9)$, covering the darker-gray entries in L_1, thus, $\psi_1(1) = 2$ (the maximum w_1 among these entries) and $UB^*(1) = f_1\psi_1(1) = 0.4 < 1$. Hence, MRIO continues the current iteration and moves to L_2. The processing zone becomes $[c_1, c_3) = [3, 20)$, thus, $\psi_1(2) = 3$ and $\psi_2(2) = 2$, taking into account additionally the intermediate-gray entries. $UB^*(2) = f_1\psi_1(2) + f_2\psi_2(2) = 0.9 < 1$, hence, MRIO proceeds to L_3. The 3-rd processing zone is $[c_1, c_4) = [3, 25)$ which updates the $\psi_j(\cdot)$ values to account also for the lighter-gray entries. Although this does not increase $\psi_j(\cdot)$ for the first and second list, it sets $\psi_3(3) = 2$. Therefore, $UB^*(3) = f_1\psi_1(3) + f_2\psi_2(3) + f_3\psi_3(3) = 1.3 > 1$. The iteration terminates here by setting $c_3 = 20$ as the pivot. This is a significant improvement (i.e., a greater jump) compared to RIO whose pivot in the same example is $c_1 = 3$. Larger jumps translate to fewer iterations and, in turn, to shorter response time.

Implementation of $\psi_j(\cdot)$: We consider three alternative implementations for function $\psi_j(\cdot)$.

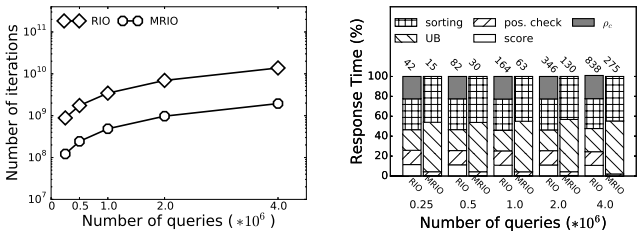
I) Sequential scan: Given a processing zone, we sequentially scan L_j from the position of c_j to the last query ID that falls in the zone, using a temporary cursor (we advance the actual c_j only after the pivot is set). Note that within the same iteration, when moving from the i-th processing zone to the next, $\psi_j(i+1)$ can be computed incrementally from $\psi_j(i)$.

II) Segment tree: We index each list L_j by a balanced, binary segment tree [30] on query ID, where every non-leaf node keeps the maximum w_j value in the queries under its subtree. Given a processing zone, the maximum weight $\psi_j(i)$ can be determined in logarithmic time by traversing the tree. When the S_k score of a query is updated, its new w_j values are reflected in the respective sorted lists, each taking logarithmic time.

III) Square root block decomposition: We partition each list L_j into $\sqrt{|L_j|}$ blocks of $\sqrt{|L_j|}$ contiguous query IDs (where $|L_j|$ is the length of the list). We then compute and store the maximum w_j value for each block. Finding the maximum weight $\psi_j(i)$ in a processing zone takes $O(\sqrt{|L_j|})$ time – that is the cost to scan the contents of the first and last block that overlap the zone, and to scan the precomputed maxima for the intermediate blocks. Update of a w_j value requires $O(\sqrt{|L_j|})$ time to recompute the corresponding block's maximum.

The segment tree and the block decomposition options offer better asymptotic performance and enable an accurate analysis of the response time, i.e., by plugging a $O(log|L|)$ or $O(\sqrt{|L|})$ cost per $UB^*(i)$ computation in the analysis of Table 1. However, in our experiments (Section 7) the sequential scan option performs better overall and is used as the default choice.

The complexity of MRIO is dominated by the upper bound computation. On the positive side, MRIO incurs no cost for moving cursors since they are already advanced to their next positions within the upper bound computation process. To offer a preliminary assessment of the effectiveness of MRIO, in Figure 5 we compare it with RIO, in the same setting as Figure 3. MRIO reduces the number of iterations by more than 7 times, which leads to 2 to 3 times shorter response time than RIO.



(a) Number of iterations (b) Response time

Fig. 5. Preliminary comparison between MRIO and RIO

5.3 Inverted File Optimization

As we mentioned in the context of Lemma 2, and similarly to traditional text retrieval systems [31], [32],

the structure of the inverted file affects the performance of MRIO. Specifically, it would be meaningful to assign surrogate query IDs such that queries with similar w_j values receive similar IDs, and are thus placed nearby in the sorted lists. This would lead to tighter local upper bounds and, hence, to longer jumps and fewer iterations.

ID rearrangement in inverted files has received considerable attention in the past decade. Silvestri [33] suggests to assign document IDs based on the alphabetical ordering of their URLs. This simple heuristic works because documents with similar URLs tend to hold related content. Blanco and Barreiro [34] model the ID rearrangement problem as a Traveling Salesman Problem (TSP). Term similarity is used to model the distance from a document to another. A TSP algorithm is used to find the shortest route that passes through all of them. Documents receive IDs according to their order of appearance in that route. The effect of ID rearrangement in the ID-ordering paradigm was evaluated in [35].

Unlike past work, in MRIO the inverted file holds queries instead of documents, which entails important differences. Typically, the number of terms in a query is much smaller than in a document; while a document includes hundreds or thousands of terms, user queries include just 2.35 on the average [36]. More importantly, prior studies [37], [38] observed that query term frequencies follow a long-tailed distribution, i.e., some representative terms appear in the majority of the queries.

Based on this observation, as a *first step* in our ID assignment approach, we extract representative topics from the query set and group each query under the topic it has the highest similarity to. Queries in a group will receive contiguous IDs. Probabilistic topic models are standard practice in text retrieval for topic extraction [39], [40], [41] but take considerable time to run. Each extracted topic comprises a set of terms, based on the term appearance probabilities. Inspired by this, we employ a fast yet effective approach, where the query terms with the highest appearance frequencies are each used as a topic. Subsequently, we assign each query to the topic it has the highest (cosine) similarity to.

To order queries even more cohesively, in a *second step*, we locally apply the TSP technique to arrange the IDs *within* each group. We start the TSP route from the query that is most similar to the group's topic, and move to its nearest query in the group. Iteratively, we move to the latter's own (unvisited) nearest neighbor, and so on. For efficiency, each nearest neighbor search only examines ϵ adjacent queries in the group, i.e., the ϵ most similar candidates based on their similarity to the group topic. We empirically observed that a small ϵ (e.g., 5) strikes a good preprocessing time-response time tradeoff.

Our ID assignment strategy reduces the response time of MRIO by 30%. Note that the ID rearrangement optimization does not apply to RIO, because its global μ_j^q maxima are irrelevant to the ordering within the lists.

6 Alternative Stream Models

In our default setting, we assume the decay model, where document freshness is incorporated into the scoring function, and document arrivals are the only type of stream events. A common alternative is the sliding window model, described in Section 3.2. Be it a count- or time-based window, we now have two types of stream events; newly arriving documents are considered as *insertions*, while documents that fall out of the window (i.e., expire) as *deletions*. Insertions are treated in the same way as document arrivals in previous sections, but deletions require special handling.

Together with each valid document $\mathbf{d}$, we keep the IDs of the queries that include it in their result. If $\mathbf{d}$ is deleted, the said queries need to be reevaluated and their results refilled. Reevaluation requires a linear scan of all valid documents[7]. To reduce the frequency of expensive reevaluations, we adopt the method of [42]. Whenever a query is (re)evaluated, we compute its top-k' result for some $k' > k$. Insertions are dealt with as per normal, using however the score of the k'-th result document instead of S_k. When a result document is deleted, we remove it, but we only resort to reevaluation when the size of the updated result drops below k. [42] provides an analysis for deciding the value of k'; in practice, we found that setting k' to $2 \cdot k$ delivers good performance with small space overhead.

The described handling of document insertions and deletions is not bound to the sliding window model. For example, the same techniques can be used for streams of arbitrary insertions and deletions, or streams where each document is associated with an expiration time.

In most monitoring systems, the query set is fixed. Nevertheless, our framework extends easily to scenarios where new queries may be registered and old ones terminated. When a new query is registered, we assign it the largest query ID so that all its entries in the inverted file are placed at the end of the respective lists. Its result is computed using the evaluation approach described above. In RIO, if a w_j value in the new query is higher than the respective μ_j^q, the latter is updated.

When a query is terminated, we simply remove it from the inverted file. In RIO, the μ_j^q values of the affected lists do not need to be eagerly updated, since the current upper bounds still hold, albeit becoming progressively looser. A lazy update strategy is the most practical option. For example, we could be recomputing the μ_j^q values after a certain number of query insertions/deletions. In MRIO, there is no need for μ_j^q update, because it computes the exact $\psi_j(\cdot)$ values at run time. To enjoy the benefits described in Section 5.3, after a number of query registrations/terminations, the query IDs may need to be rearranged.

7. An alternative is to keep valid documents indexed by an inverted file, but this entails large index maintenance costs, which generally outweigh the gains in query reevaluation.

TABLE 2
Real Document and Query Sets

Name	Type	Cardinality	Avg. length
Wiki	Documents	7,012,610	190.7
20news	Documents	38,825	54.1
WSJ	Documents	172,961	104.7
AOL	Queries	21,011,340	1.5
Hypertable	Queries	485,066	1.8

TABLE 3
Experiment Parameters

Parameter	Values
No. of queries (x10^6)	0.25, 0.5, **1.0**, 2.0, 4.0
Query length m	3, **5**, 7, 11, 15, 19
Result size k	1, 5, **10**, 15, 20
Decay parameter λ	0, 1, **2**, 4, 8, 16
Document stream	**Wiki**, 20news, WSJ
Synthetic queries	Uniform, **Connected**, Clustered, Random
Real queries	**AOL**, Hypertable
Streaming model	**Decay**, Sliding window
No. of groups	1, 10, **20**, 30, 40

7 EXPERIMENTS

We evaluate RIO and MRIO against RTA [1], SortQuer [3], and TPS [18]. RTA is widely considered the state-of-the-art due to its performance and generality for CTQDs, however, we also compare with SortQuer because for $k = 1$ it was shown to outperform RTA [3]. TPS was proposed for a different setting/problem, but it is included in our evaluation because it is easily adaptable to CTQDs. The internal parameters of competitors are set to the values recommended in the respective papers.

By default, we consider processing in the decay model (see Section 3.2), but we also present experiments in the sliding window model. Performance is evaluated in terms of response time, i.e., the average time required to refresh all query results upon a stream event [8].

7.1 Experiment Setup

Document stream: We use three real document sets to simulate the document stream. The default document set is *Wiki*. It contains all available pages of Wikipedia from 2001 to 2006. The other document sets are *20news* and *WSJ*. The former (*20news*) contains $38,825$ posts from 20 different newsgroups and the latter (*WSJ*) contains $172,961$ articles published in Wall Street Journal from 1986 to 1992. Following standard practice in text retrieval systems, we remove all high frequency terms (i.e., stop words) from the documents. We simulate the document stream by randomly selecting $20,000$ documents from the corresponding document set. The first 20% of the documents are used for system warm-up and the rest to evaluate performance. To disassociate the decay function in Equation 2 from the definition of time units, we express $\Delta\tau_d$ in terms of the number of documents since the beginning of the stream.

Query workload: We experiment with synthetic and real query workloads. The synthetic ones allow more flexibility in varying/testing the query characteristics, thus a more comprehensive evaluation. The real ones verify practicality using actual user queries.

We use 4 types of synthetic queries. The first is *Random*, where the query terms are drawn randomly and independently from the dictionary of the document stream. To generate more realistic workloads, in the other 3 synthetic types we model the fact that terms in a query are generally not independent. In particular, we create a co-occurrence graph, where each node corresponds to a term, and every edge is weighted by the frequency in which its end-nodes (terms) appear in the same document, as computed over the entire document stream. We generate a query $\mathbf{q}$ by choosing its first term randomly, with a probability that follows the appearance frequency of the terms in the document stream. The rest of the terms in $\mathbf{q}$ are drawn among the neighbors of the first term in the co-occurrence graph. Each of the neighbors has a probability to be selected that is dependent on its co-occurrence frequency (weight) w with the first term. Specifically, the probability is proportional to value w^α, where α is a parameter that controls the coherence of the queries. In *Uniform* query workloads, we set $\alpha = 0$. In *Connected* we set $\alpha = 1$, and in *Clustered* we set $\alpha = 2$. The default synthetic queries are *Connected*. Another remark regards the length m of generated queries, which is one of the investigated parameters. For a given m, not all queries have the exact same length, but their individual lengths follow a Gaussian distribution with mean m and standard deviation $\sigma = 1$.

The real query sets are AOL and Hypertable. The first contains 21M web queries collected from 650K users over 3 months. The second includes 485K queries from 100K users, available at hypertable.com. Table 2 summarises the characteristics of the real document and query sets.

Experiment parameters: All methods were implemented in Java 1.7 and run on a machine with Intel 3.4GHz i7-2600 CPU, operating Ubuntu 14. Investigated parameters and their tested ranges are listed in Table 3. Unless otherwise specified, in each experiment we vary one parameter and set the remaining ones to

8. A side aspect of performance is space requirements, where our methods incur minimal overhead; that is, $O(Q \cdot k)$ to keep the results and S_k of all queries, and $O(Q \cdot m)$ to keep their ID and w_j values in the inverted file, where Q is the number of queries and m is the query length. Even in our largest scale experiment (with 4 million queries), space consumption is below 615MBytes.

TABLE 4
Response Time for the Default Setting (ms)

Stream	Queries	MRIO	RIO	SortQuer	RTA	TPS
Wiki	Unif.	34.4	94.8	234.2	272.3	243.7
Wiki	Conn.	62.7	163.8	241.5	653	427.8
Wiki	Clus.	89.3	213.8	209.4	1866	552
Wiki	Rand.	11.2	20.7	63.8	183.2	44.8
20News	Conn.	5.8	6.8	32.7	58.5	18.8
WSJ	Conn.	21.6	48.2	116.5	173	136.8

their defaults (shown in bold). The last row in the table corresponds to the number of groups produced by topic-based clustering for ID rearrangement. The default number of groups we use is 20. As explained in Section 5.3, this optimization is only applicable to MRIO.

An important note is that we test a wide range of query lengths because we want to capture both settings where users manually specify their query terms (in which case the query length is short [36]) but also applications where the terms of interest are extracted from the users' recorded behaviour (where queries are longer [4], [5]).

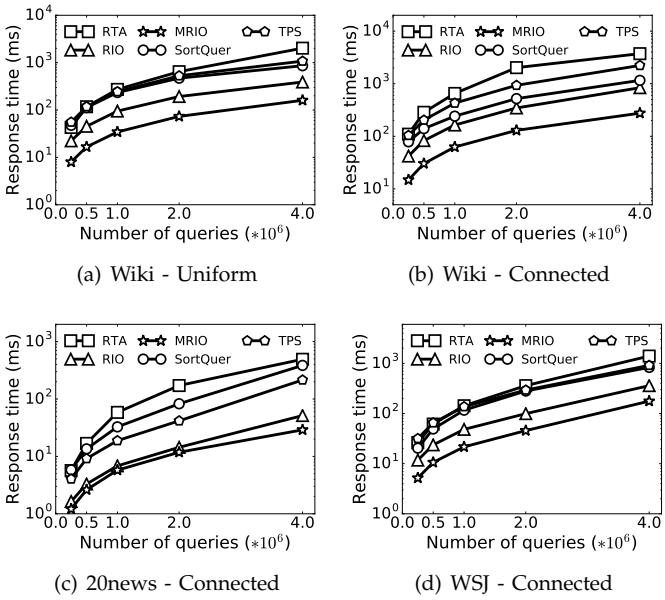
7.2 Experiment Results

Performance for the default setting: In Table 4, we show the response time of all competing methods for the default setting. In the first 4 rows we use the default document stream (*Wiki*) with each of the synthetic query workloads. In the last 2 rows, we use the non-default document streams with the default query workload (*Connected*). Performance is generally better for *Uniform* and *Random* queries, because they tend to include rarer terms than *Connected* and *Clustered*, thus, the stream events are less likely to affect them. Performance for the non-default document sets (*20news*, *WSJ*) is better than *Wiki*, because their documents contain fewer terms (see average length in Table 2).

MRIO is always the most efficient method, achieving in most cases 2 to 3 times shorter response time than the runner-up, RIO. The other three competitors are lagging further behind. A surprising fact is that TPS and SortQuer perform comparably or better than RTA. Note that TPS was never before evaluated for CTQDs nor compared with RTA (or any other CTQD method). On the other hand, SortQuer was previously evaluated only for $k = 1$ and only against RTA, as explained in Section 2. There is no clear winner in the comparison between TPS and SortQuer but, as we will see shortly, SortQuer suffers for larger k and/or query length (where it becomes the slowest among all competitors).

Number of queries: Figure 6 demonstrates the effect of the number of queries, which varies from 250K to 4M, using four different document-query combinations. The response time of all methods increases, since the number of queries that are affected by a document arrival grows proportionally to their total number. MRIO is the fastest method in all settings. Its running time is up to 8, 10, and 25 times shorter than TPS, SortQuer, and RTA, respectively. The internal comparison between MRIO and RIO shows improvements of up to 4 times. The reason is that the pruning efficiency of MRIO is unaffected by the number of queries, i.e., its local bounds retain their absolute tightness guarantee, whereas (as discussed at the end of Section 5.1) the bounds of RIO become increasingly looser.

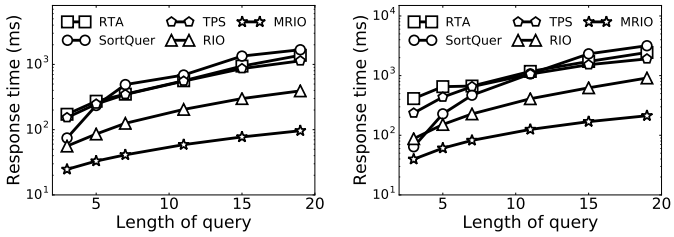
Query length: In Figure 7, we measure the effect of the query length m. For brevity, we plot results for two



(a) Wiki - Uniform (b) Wiki - Connected

(c) 20news - Connected (d) WSJ - Connected

Fig. 6. Effect of number of queries

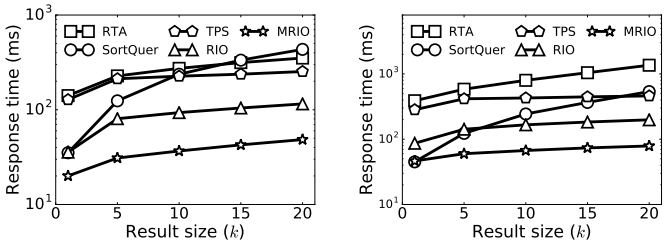


(a) Wiki - Uniform (b) Wiki - Connected

Fig. 7. Effect of query length

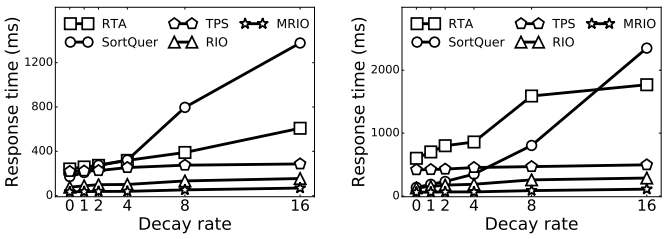
representative document-query combinations, i.e., '*Wiki - Uniform*' and '*Wiki - Connected*'. The response time of all methods increases with m. Their relative performance is similar to that in Figure 6, with the exception of SortQuer. The performance of SortQuer worsens severely with query length, because it needs to search in a two-dimensional space for every query term.

Parameter k: In Figure 8, we examine the effect of k. As k becomes larger, the likelihood that a document arrival affects a query result increases. Therefore, the response time of all methods increases with k. Our techniques,



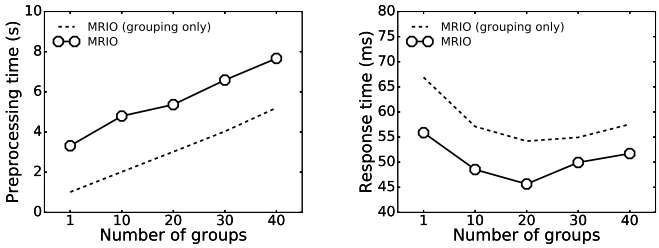
(a) Wiki - Uniform (b) Wiki - Connected

Fig. 8. Effect of k



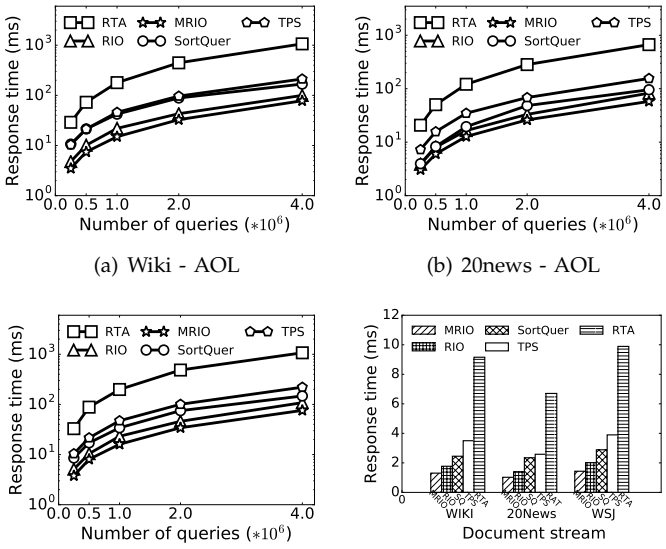
(a) Wiki - Uniform (b) Wiki - Connected

Fig. 9. Effect of decay parameter λ



(a) Preprocessing time (b) Response time

Fig. 11. Tests on ID rearrangement (Wiki - Connected)



(a) Wiki - AOL (b) 20news - AOL

(c) WSJ - AOL (d) Hypertable queries

Fig. 10. Experiments on real query sets

MRIO and RIO, are significantly less sensitive to k than RTA. This result attests to the stronger pruning ability of ID-ordering than frequency-ordering for the CTQD problem. Another important observation is that SortQuer suffers with k. The reason is that in its two-dimensional mappings, one axis corresponds to the score of the k-th result document. A larger k implies a smaller score (coordinate) for the mapped queries, which in turn affects negatively the effectiveness of spatial pruning in the two-dimensional spaces.

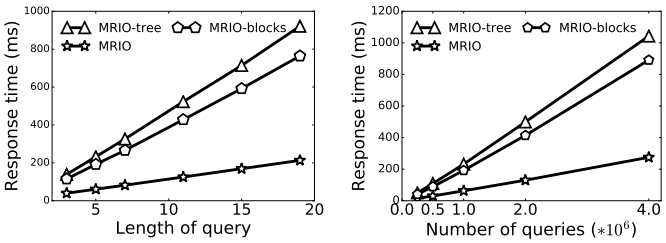
Decay parameter λ: Figure 9 investigates the effect of the decay parameter λ. Referring to Equation 2, a larger λ value implies that documents "age" faster, i.e., that newly arriving documents are favored even more aggressively compared to the old ones. In effect, this leads to more frequent result updates and, therefore, to longer response time for all methods. Our techniques, and especially MRIO, not only outperform competitors, but they are also very robust to λ. Interestingly, SortQuer scales poorly with λ. That is because the higher frequency of result updates leads to large maintenance cost for its numerous two-dimensional indices.

Real query workloads: For generality, in Figure 10 we present results for real queries. Sub-figures (a), (b), and (c) consider AOL queries over our different document streams, for varying number of queries (randomly drawn from AOL). Sub-figure (d) considers Hypertable queries for all document streams in the default setting. Our methods, and especially MRIO, retain their superiority over competitors, however, the improvement is not as pronounced as in previous experiments. The reason is that AOL and Hypertable queries include 1.5 and 1.8 terms on the average. This means that a large fraction of the queries have a single term. The ideal (and trivial) way to answer a single-term query is via a frequency-ordered inverted file, by simply reporting the first k documents in the list for that term. I.e., ID-ordering is not ideal for single-term queries, however, the processing time for such queries is anyway very short.

Inverted file optimization: In Section 5.3, we described an ID rearrangement optimization that improves the structure of the inverted file and enhances the performance of MRIO. The process involves two steps, grouping and TSP assignment within each group. In Figure 11, label 'MRIO' corresponds to the default, full rearrangement. The second method represented applies only grouping, but not the TSP step. Figure 11(a) presents the preprocessing time required for ID rearrangement for different numbers of groups, while Figure 11(b) shows the achieved response time for actual stream processing.

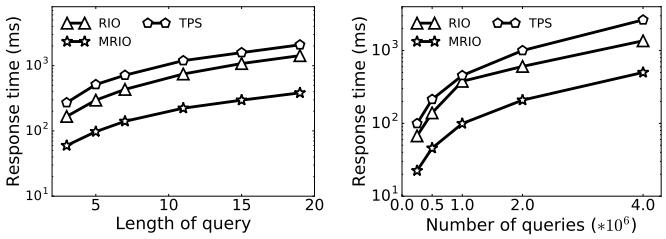
More groups always imply greater preprocessing cost, because the assignment to groups (in the first step) needs to consider more alternatives, thus performing more similarity computations. The running time, however, initially decreases and then increases. The reason is that too few groups mean that each group is less cohesive. On the other hand, too many groups may force physical clusters of queries to be split, placed into different groups, receive too distant IDs, and thus be treated as more different than they actually are.

Effect of $\psi_j(\cdot)$ implementation: In Section 5.2, we described three alternative implementations for $\psi_j(\cdot)$ computation, i.e., alternative techniques to derive the local maximum weight in a given processing zone. Figure 12 compares them for different query lengths and for different numbers of queries. The default implementation, MRIO (using sequential scan), clearly outperforms MRIO-tree (segment tree) and MRIO-block (block decomposition). The main reasons are its simplicity and



(a) Wiki - Connected (b) Wiki - Connected

Fig. 12. Effect of $\psi_j(\cdot)$ implementation



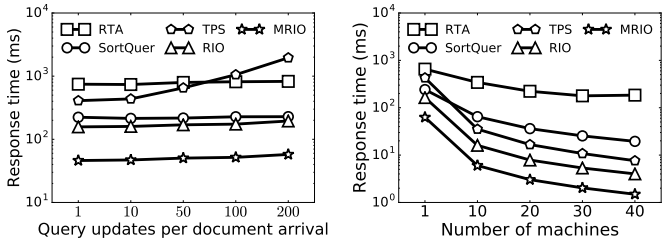
(a) Wiki - Connected (b) Wiki - Connected

Fig. 13. Performance in sliding window model

the maintenance overhead required in the other two methods when S_k values are updated.

Sliding window model: We present indicative results for the sliding window model. We use the *Wiki* document stream and *Connected* queries. The size of the sliding window is set to $4,000$ documents. We plot results for our methods, and for TPS, as the best performing competitor. In Figure 13(a), we use 1 million queries and vary the query length m from 3 to 19 – all other parameters are set to their defaults. In Figure 13(b), we use query length $m = 5$ and vary the number of queries from 250K to 4M. The reported measurements correspond to the average time required to update all query results for a pair of document arrival-document expiration stream events. Both our methods outperform TPS by a wide margin. Specifically, MRIO is 4.4 to 5.4 times faster than TPS, while RIO is 1.2 to 1.93 times faster than TPS. Performance in Figure 13 is representative for time-based windows too, since in both types of sliding windows the documents expire in first-in-first-out order.

Query updates: In Section 6, we elaborated on query updates. In Figure 14(a), we show the response time when together with each document arrival, we have a number (from 1 to 200) of query updates. Each update corresponds to a query deletion and a query insertion. The response times of RTA, SortQuer, RIO and MRIO are not very sensitive to the number of updates, since their query indices are easy to maintain. TPS is the only exception due to its more complicated query index.



(a) Effect of query updates (b) Effect of parallelization

Fig. 14. Additional experiments (Wiki - Connected)

Horizontal parallelization: All methods can benefit from horizontal parallelization, i.e., by partitioning the queries, assigning each partition to a different machine, and executing the respective CTQD method on every machine. Figure 14(b) shows the response time for different numbers of available machines.

8 Conclusion

In this paper, we propose a scalable framework for the processing of continuous top-k queries on document streams (CTQDs). A CTQD continuously reports the k most relevant documents to a set of keywords. CTQDs find application in many emerging applications, such as email and news filtering. Our preliminary approach, RIO, adapts the ID-ordering paradigm to the CTQD setting. An analysis on RIO reveals that the key factor that determines its performance is the number of iterations it executes. This motivates our advanced approach, MRIO, which not only reduces the number of iterations, but is proven to minimize it. We achieve this by introducing novel, locally adaptive bounds. Extensive experiments with streams of real documents demonstrate that MRIO is an order of magnitude faster than the previous state-of-the-art. A promising direction for future work is to extend our methodology to approximate top-k queries.

Acknowledgement: This work was supported by grant NSFC 61502548 from NSF of China and grant MYRG2014-00106-FST and MYRG2016-00182-FST from UMAC RC.

References

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

[2] K. Mouratidis and H. Pang, "Efficient evaluation of continuous text search queries," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 10, pp. 1469–1482, 2011.

[3] N. Vouzoukidou, B. Amann, and V. Christophides, "Processing continuous text queries featuring non-homogeneous scoring functions." in *CIKM*, 2012, pp. 1065–1074.

[4] A. Hoppe, "Automatic ontology-based user profile learning from heterogeneous web resources in a big data context." *PVLDB*, pp. 1428–1433, 2013.

[5] A. Lacerda and N. Ziviani, "Building user profiles to improve user experience in recommender systems," in *WSDM*, 2013, pp. 759–764.

[6] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. J. Lin, "Earlybird: Real-time search at twitter," in *ICDE*, 2012, pp. 1360–1369.

[7] L. Wu, W. Lin, X. Xiao, and Y. Xu, "LSII: an indexing structure for exact real-time search on microblogs," in *ICDE*, 2013, pp. 482–493.

[8] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, 2006.

[9] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.

[10] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien, "Efficient query evaluation using a two-level retrieval process." in *CIKM*, 2003, pp. 426–434.

[11] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch, "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," *IEEE Trans. Computers*, vol. 51, no. 10, pp. 1124–1140, 2002.
[12] S. E. Robertson and D. A. Hull, "The TREC-9 Filtering Track Final Report," in *Text REtrieval Conference*, 2000, pp. 25–40.
[13] Y. Zhang and J. Callan, "Maximum Likelihood Estimation for Filtering Thresholds," in *SIGIR*, 2001, pp. 294–302.
[14] F. Fabret, H. Jacobsen, F. Llirbat, J. L. M. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe," in *SIGMOD Conference*, 2001, pp. 115–126.
[15] W. Rao, L. Chen, A. W.-C. Fu, H. Chen, and F. Zou, "On efficient content matching in distributed pub/sub systems." in *INFOCOM*, 2009, pp. 756–764.
[16] M. Sadoghi and H.-A. Jacobsen, "Relevance matters: Capitalizing on less (top-k matching in publish/subscribe)." in *ICDE*, 2012, pp. 786–797.
[17] K. Pripuzic, I. P. Zarko, and K. Aberer, "Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w." in *DEBS*, 2008, pp. 127–138.
[18] A. Shraer, M. Gurevich, M. Fontoura, and V. Josifovski, "Top-k publish-subscribe for social annotation of news," *PVLDB*, vol. 6, no. 6, pp. 385–396, 2013.
[19] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-*k* query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.
[20] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes." *PVLDB*, pp. 1213–1224, 2011.
[21] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows." in *SIGMOD Conference*, 2006, pp. 635–646.
[22] A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-k queries." in *SIGMOD Conference*, 2012, pp. 397–408.
[23] W. Rao, L. Chen, S. Chen, and S. Tarkoma, "Evaluating continuous top-k queries over document streams." *World Wide Web*, pp. 59–83, 2014.
[24] P. Haghani, S. Michel, and K. Aberer, "Efficient monitoring of personalized hot news over web 2.0 streams," *Computer Science - R&D*, vol. 27, no. 1, pp. 81–92, 2012.
[25] N. Koudas, B. C. Ooi, K. Tan, and R. Zhang, "Approximate NN queries on streams with guaranteed error/performance bounds," in *VLDB*, 2004, pp. 804–815.
[26] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou, "Streaming multiple aggregations using phantoms," *VLDB J.*, vol. 19, no. 4, pp. 557–583, 2010.
[27] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.
[28] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems." in *ICDE*, 2009, pp. 138–149.
[29] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, 1999.
[30] J. L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Trans. Computers*, vol. 29, no. 7, pp. 571–577, 1980.
[31] S. Ding, J. Attenberg, and T. Suel, "Scalable techniques for document identifier assignment ininverted indexes," in *WWW*, 2010, pp. 311–320.
[32] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering." in *WWW*, 2009, pp. 401–410.
[33] F. Silvestri, "Sorting out the document identifier assignment problem," in *ECIR*, 2007, pp. 101–112.
[34] R. Blanco and A. Barreiro, "Tsp and cluster-based solutions to the reassignment of document identifiers." *Inf. Retr.*, pp. 499–517, 2006.
[35] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes." in *SIGIR*, 2011, pp. 993–1002.
[36] C. Silverstein, M. R. Henzinger, H. Marais, and M. Moricz, "Analysis of a very large web search engine query log," *SIGIR Forum*, vol. 33, no. 1, pp. 6–12, 1999.
[37] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui, "Heavy-tailed distributions and multi-keyword queries," in *SIGIR*, 2007, pp. 663–670.
[38] M. Whittle, B. Eaglestone, N. Ford, V. J. Gillet, and A. D. Madden, "Query transformations and their role in web searching by the general public." *Inf. Res.*, 2006.
[39] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation." *Journal of Machine Learning Research*, pp. 993–1022, 2003.
[40] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences*, vol. 101, no. Suppl. 1, pp. 5228–5235, 2004.
[41] M. Steyvers and T. Griffiths, "Probabilistic topic models," *Handbook of latent semantic analysis*, vol. 427, no. 7, pp. 424–440, 2007.
[42] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient maintenance of materialized top-k views," in *ICDE*, 2003, pp. 189–200.

Leong Hou U completed his B.Sc. in Computer Science and Information Engineering at Taiwan Chi Nan University, his M.Sc. in E-commerce at University of Macau, and his Ph.D. in Computer Science at University of Hong Kong. He is now an Associate Professor at University of Macau. His research interests include spatial and spatio-temporal databases, advanced query processing, crowdsourced query processing, information retrieval, data mining and optimization problems.

Junjie Zhang competed his M.Sc. in Computer Science at University of Macau. He is now a project manager at Waqu Inc. in China and his projects focus on video recommendations in social networks.

Kyriakos Mouratidis completed his B.Sc. at Aristotle University of Thessaloniki (AUTH), and his Ph.D. at Hong Kong University of Science and Technology (HKUST), both in Computer Science. He is currently an Associate Professor of Information Systems at Singapore Management University (SMU). His research focuses on continuous query processing, preference queries, spatial and road network databases, and spatial optimization problems.

Ye Li is a Ph.D. student at the Department of Computer and Information Science, University of Macau, under the supervision of Prof. Leong Hou U. His current research focuses on complex spatial query processing.