Top-k Query Processing with Conditional Skips

Edward Bortnikov
Yahoo Research, Israel
bortnik@yahoo-inc.com

David Carmel
Yahoo Research, Israel
dcarmel@yahoo-inc.com

Guy Golan-Gueta*
VMWare Research, Israel
ggolangueta@vmware.com

ABSTRACT

This work improves the efficiency of dynamic pruning algorithms by introducing a new posting iterator that can skip large parts of the matching documents during top-k query processing. Namely, the conditional-skip iterator jumps to a target document while skipping all matching documents preceding the target that cannot belong to the final result list. We experiment with two implementations of the new iterator, and show that integrating it into representative dynamic pruning algorithms such as *MaxScore*, *WAND*, and *Block Max WAND* (*BMW*) reduces the document scoring overhead, and eventually the query latency.

Keywords

Top-k query Processing, Dynamic Pruning, Conditional-Skip Iterator

1. INTRODUCTION

The search process applied by most modern search engines is usually based on two phases [26]. In the first phase, a large subset of documents that "match" the query is efficiently scanned and scored by a simple and easy to compute scoring function such as tf-idf or BM25. The top-k scored documents are extracted by a *top-k query processing* algorithm [27]. Typically, the extracted list contains a few hundreds, or even thousands, of matching documents. In the second "re-ranking" phase, this extracted list is re-ranked using a complex scoring function that considers a rich set of features of the documents, the query, the context of the search process, and many other signals, in order to obtain a small ranked list of high-quality search results.

In this work we focus on improving the efficiency of the top-k query processing applied by the first phase of the search process. Handling this task in efficient manner is a must for search engines who are expected to serve their customers in a sub-second response time. Efficiency is harder to maintain as the size of the document collection becomes larger. Furthermore, efficiency is badly affected by the query length, yet current user queries, especially on the Web, become much longer due the high usage of query suggestion tools, query reformulation techniques, and the enlarging trend of supporting voice-enabled search [10].

A long line of research has focused on improving the efficiency of top-k query processing by applying *dynamic pruning* techniques [3, 17, 25, 18, 14, 2, 1] which reduce the number of evaluated documents while identifying the top scored results. *MaxScore* [25, 23], *WAND* [2], and *BMW* [8], are some representative dynamic-pruning algorithms for the top-k processing task. In this work we introduce a novel posting traversal method, called *conditional-skip iterator*, that can jump to a target document while skipping all matching documents preceding the target, conditionally that none of them can belong to the final result list. We show how the new iterator can improve dynamic pruning algorithms, and in particular the algorithms mentioned above, by decreasing the number of evaluated documents in significant way.

We demonstrate two implementations of the new iterator. The first one is based on existing posting list traversal APIs, whereas the second one is based on the *treap* data structure [21] (a combination of a tree and a heap). Our treap-based implementation is similar to the one described by Konow et al. [12]. They described an implementation of both *OR* and *AND* top-k processing algorithms based on a treap-based index. In contrast, our iterator is independent of a specific retrieval algorithm, and of a specific implementation, hence it can directly be applied by multiple search paradigms for improved dynamic pruning.

We perform an extensive study of the contribution of the new iterator, on two large document sets. Our experimental results reveal significant performance improvement in terms of reducing the number of document evaluations and eventual latency reduction for all algorithms we experimented with. We examine the iterator's impact on a large set of queries of varying length (1 to 10 terms). The experiments reveal, somewhat surprisingly, that the more sophisticated treap-based iterator outperforms only for very short queries, whereas the simpler implementation, which incurs much less overhead, is more efficient for long queries

The rest of the paper is organized as follows: In Section 2 we describe the for top-k query processing algorithms used in this work. We present them using a unified framework that allows us to fairly compare them on the same evalu-

*This Research was performed while Guy Golan-Gueta was in Yahoo.

WWW 2017 Companion, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/33041021.3054191

ation platform. Section 3 introduces the conditional-skip iterator abstraction, and shows how it can be invoked by dynamic pruning algorithms. In Section 4 we describe the two alternative implementations of the iterator. Section 5 presents the experimental results. Section 6 summarizes related work, and Section 7 concludes and suggests directions for future work.

2. BACKGROUND

2.1 Preliminaries

Top-k query processing is the task of identifying the top-k scored documents, out of a collection of documents, for a given query q and a given scoring function $S(q, d)$ [14, 27]. Documents are usually scored based on an estimation of their relevance to the query. We assume an additive scoring function,

$$S(q,d) = \sum_{t \in q \cap d} w(t) \times f(t,d), \tag{1}$$

where $w(t)$ is the weight of the query term t, representing its relative importance in the query, and $f(t, d)$ is the contribution of term t to the score of document d. Such a scoring model covers many popular scoring schemes in IR which are additive in nature, e.g., tf-idf, BM25, and many others [2].

The evaluation process is typically based on inverted index of the collection, in which each term is associated with a posting list of elements representing all documents containing the term. A posting element is usually a tuple containing the doc-id (d), the term contribution to d's score ($f(t, d)$), and other information related to the term occurrence in the document (e.g., a list of the term offsets within the document, needed for term proximity evaluation). Scoring a document d using such an index relates to identifying the posting elements of all query terms matching d and summing their score contribution.

There are two main paradigms for scanning the documents matching the query. The *term-at-a-time* (TAAT) approach [3, 27] sequentially traverses the posting lists of the query terms, in a term-by-term manner, while documents scores are sequentially accumulated. The final document score is determined only after all query terms are processed. After completing the traversal of all lists, the k documents with highest accumulated score are extracted. Alternatively, in *document-at-a-time* (DAAT) approach [25, 2, 23], we simultaneously scan all posting lists in parallel, and the score of a document is fully evaluated before moving to the next document. For DAAT based methods all posting lists must be sorted by the same key, typically by increasing doc-id.

In practice, both approaches suffer from huge amount of matching documents per query that are needed to be scored in order to identify the top-k scored ones. Dynamic pruning techniques [27] try to reduce the amount of documents to be scored. Dynamic pruning methods are classified to "safe" methods which guarantee that the top-k results are identical to the results of the corresponding non-pruned method, and are sorted by the same order. "Non-safe" methods are typically more efficient however do not guarantee identical results nor the same order. In this work we only consider safe DAAT methods.

We assume that the inverted index supports the following posting traversal methods:

- $t.doc()$: returns the doc-id of the posting element that the term's cursor currently points to.
- $t.next()$: advances the cursor to the next element in the list of term t.
- $t.skip(d)$: advances the cursor to the first element in the list with document $d' \geq d$.

In addition we assume that each query term holds a global upper bound on its potential contribution to any document in the index. Such an upper bound can be determined by (offline) traversal over the term's posting list to select the maximum entry, i.e., $t.UB = w(t) \times max_d f(t, d)$ [25, 2]. Complementary, if documents are split into n blocks, a term can maintain an array of tighter block upper bounds, $t.UB[1..n]$, where each entry in the array bounds the term's contribution to all documents in the block [8]. Term upper bounds are essential for all dynamic pruning methods discussed in this work.

2.2 Algorithms

We now describe four (safe) DAAT algorithms for top-k query processing that we analyze in this work. We assume that all postings in the index are sorted by increasing doc-id. To ease the comparison and analysis, we generalize the algorithms using a unified framework, while preserving their correctness and their main principles.

2.2.1 *Top-k* OR

The top-k *OR* algorithm traverses all matching documents to the query, i.e., all documents containing at least one query term, without any pruning. Algorithm 1 presents its pseudo code.

Algorithm 1 Top-k: *OR*

```
1: input:
2: $termsArray$ - Array of query terms
3: $k$ - Number of results to retrieve
4: Init:
5: for $t \in termsArray$ do $t.init()$
6: $heap.init(k)$
7: $\theta \leftarrow 0$
8: $Sort(termsArray)$
9: Loop:
10: while ($termsArray[0].doc() < \infty$) do
11:     $d \leftarrow termsArray[0].doc()$
12:     $i \leftarrow 1$
13:     while ($i < numTerms \cap termArray[i].doc() = d$) do
14:         $i \leftarrow i + 1$
15:     $score \leftarrow Score(d, termsArray[0..i-1]))$
16:     if ($score \geq \theta$) then $\theta \leftarrow heap.insert(d, score)$
17:     $advanceTerms(termsArray[0..i-1])$
18:     $Sort(termsArray)$
19: Output: return $heap.toSortedArray()$
20:
21: function $advanceTerms(termsArray[0..pTerm])$
22:     for ($t \in termsArray[0..pTerm]$) do
23:         if ($t.doc() \leq termsArray[pTerm].doc()$) then
24:             $t.next()$
```

The algorithm begins with initializing the query terms, including initiating the terms' posting iterators and setting the terms' global upper bounds. It then initializes a min-heap of size k that accumulates the top-k scored results, and sorts the terms in increasing order of their cursor which points to their first matching document. The main loop scans over all matching documents; for each matching document it collects all matching terms (called "pivot terms"). It then scores the document, following Equation 1, and pushes the document and its calculated score into the heap, using the $heap.insert()$ method which returns the minimum value in the heap, θ. After heap insertion all pivot terms are advanced to their next posting element, and the terms are then re-sorted based on their updated cursor. Note that the algorithm does not take any advantage of the $skip()$ operator, neither the term upper bounds and θ, the minimum value in the heap.

2.2.2 *Top-k* MaxScore

The *MaxScore* algorithm [25, 23] is a safe algorithm with respect to *OR*, i.e., it returns the same top-k results in the same order, that dynamically prunes the scanned posting lists. It splits the query terms to "required" and "non-required". The main observation is that a document containing only non-required terms cannot belong to the final result list and thus can be skipped. *MaxScore* maintains a max_score value for each term which is the sum of all term upper bounds that are equal to or smaller than the term's upper bound. Thus, the max_score value provides an upper bound on the score of a document containing this term and maybe other terms with smaller upper bound. If the max_score of a term is smaller than θ, the minimum heap value, the score of a document containing only this term, and other lower valued terms, will not be inserted into the heap. Therefore, such a term is marked as "non-required". During the main loop, the algorithm only analyzes documents that contain at least one required term. After any heap update, it checks all terms if they are still required.

2.2.3 *Top-k* WAND

WAND [2] is a safe algorithm with respect to *OR*[1]. Similarly to *MaxScore*, it also analyzes the correspondence between term upper bounds and the heap minimum value θ. At each stage the algorithm searches for the "pivot" document defined as the first one with a potential score to be included in the final result set. It first identifies $pivotTerm$, the first term in the order that the sum of upper-bounds of all preceding terms exceeds θ. If all terms behind $pivotTerm$ match its document, the pivot, it is scored and pushed into the heap. Otherwise, one term behind is selected and advanced up to the pivot. Term selection is performed by estimating the term with the farthest potential jump.

2.2.4 *Top-k* BMW

Block-Max Wand (*BMW*) [8], an extension of *WAND*, keeps an additional array of block upper-bounds for each term, each bounds the scores of all documents in a block of posting elements.

BMW identifies the pivot document exactly as *WAND* does. However, it additionally compares the sum of current block upper bounds of the matching terms with the minimum value in the heap, θ. The document will be evaluated only when the sum is larger than θ, otherwise the algorithm will search for the next pivot. Since block upper bounds are naturally tighter than the global upper bound, much more candidates will be skipped by *BMW* with respect to *WAND*.

[1]There is a very efficient unsafe version of *WAND* [2] which we do not cover in this work.

This improvement comes with the cost of an extra memory used by *BMW*. While *WAND* keeps only one upper bound per term, *BMW* maintains two additional lists per term – one for keeping the block boundaries and the other for block upper bounds.

3. THE CONDITIONAL-SKIP ITERATOR

We now introduce our new posting iterator, $t.condSkip(d, \tau)$, which skips the term's posting-list up to d, conditionally that all skipped entries have a score lower than τ. First we describe how the iterator can be used to improve DAAT algorithms, and then two implementations of it, one is based on existing traversal APIs and the second on organizing the posting list as a treap [21, 12].

The conditional-skip iterator, $t.condSkip(d, \tau)$, skips the cursor in t's posting-list beyond the target doc-id d, conditionally that all term scores of the skipped entries are smaller than τ. More precisely, it returns the first entry in the posting list with doc-id $d' \geq t.doc()$, such that either $d' \geq d$ or $w(t) \times f(t, d') \geq \tau$. It is easy to see that the previously defined posting iterators can be instantiated with the new iterator; $t.skip(d) \equiv t.condSkip(d, \infty)$ and $t.next() \equiv t.condSkip(t.doc() + 1, 0)$.

3.1 Usage

Consider a one-term query $q =< t >$. Extracting the top-k scored documents can be done by scoring any document in t's posting-list and selecting the top-k scored ones. However, using the new iterator, we can skip large parts of the list by repeatedly calling $t.condSkip(\infty, \theta)$, where θ, the minimum value in the min-heap, is dynamically updated during traversal. The conditional-skip iterator will only consider documents with a score larger than θ, which becomes larger as we proceed through the list. The larger θ is, the larger the skip.

The iterator can also be used, in a similar manner, for multi-term queries. Assume $q =< t_1, \ldots, t_n >$, where t_1 is the most backward term and the cursor of t_2, the next term in the order, is advanced farther. The current document, $t_1.doc()$, has already been handled and now we should advance t_1's cursor. Since any document in the interval $[t_1.doc() + 1, t_2.doc())$ can only match t_1, we can safely call $t_1.condSkip(t_2.doc(), \theta)$. The iterator will only stop on a candidate preceding $t_2.doc()$ with a score larger or equal θ, while skipping all entries with a lower score than θ which cannot make it into the heap.

The conditional-skip iterator can be used for modifying $advanceTerms()$, the method called by all algorithms for advancing the the pivot terms. Figure 1 demonstrates the conditional skips applied by the modified method. The pivot is the current document after being handled; $< t_1, \ldots, t_3 >$ are the pivot terms, each associated with its upper-bound, $nextDoc$ is the current document pointed by the cursor of the first term in the order that does not match the pivot (t_4), and the current threshold $\theta = 9$. For simplicity we assume that $w(t) = 1$ for all terms.

Let t_3 be the first term to advance. The sum of upper bounds of all pivot terms excluding t_3, $t_3.othersUB = 5$. We can safely advance t_3 with the call $t_3.condSkip(nextDoc, 4)$ since the score of any document d' in the interval $[pivot + 1, nextDoc)$, with $f(t_3, d') < 4$, is guaranteed to be less than $\theta = 9$ even if it matches all pivot terms. If t_3's cursor lands before $nextDoc$, this document is set to be the new $nextDoc$. Similarly, for the next term to advance, $t_2.othersUB = 3$. Hence, it can be safely advanced by calling $t_2.condSkip(nextDoc, 6)$. Since the scores of all documents preceding $nextDoc$ in t_2's list are less than 6, it lands on the current $nextDoc$. Finally, $t_1.othersUB = 0$ hence it can be advanced by calling $t_1.condSkip(nextDoc, 9)$, landing on the first document beyond $nextDoc$.

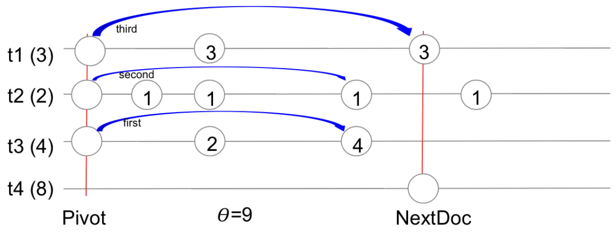


Figure 1: Advancing pivot terms using the conditional-skip iterator. The circles represent posting entries and the values withing the circles represent $f(t, d)$). A posting entry in a term's list can be conditionally skipped, if its score, together with the sum of other pivot terms' upper-bounds, is smaller than θ.

Algorithm 2 describes the code of the updated $advanceTerms()$ method. The method receives an additional input; θ, the minimum value in the heap, and the sum of upper-bounds of the pivot terms $sumUB$. At first, the function determines $nextDoc$, the limit on advancing the pivot terms. If the sum of upper-bounds is less than θ we advance all pivot terms beyond $nextDoc$. Otherwise, it repeatedly selects a (non-selected yet) pivot term t, for advancing its cursor using the conditional-skip iterator[2].

We first set $othersUB$ to be the sum of term upper-bounds excluding $t.UB$. Then t can be safely advanced with the call $t.condSkip(nextDoc, \theta - othersUB)$. After term's advancement, if t's cursor is before $nextDoc$, we update it to become the new $nextDoc$. Finally, we decrease $sumUB$ with $t.UB$ as t cannot match any document preceding $nextDoc$. The process ends after advancing all pivot terms.

For BMW, the $advanceTerms()$ method can be modified to take further advantage of the tighter block upper bounds. The modified version receives two additional parameters, the minimum block boundary of all pivot terms, $minBlockBoundary$, and the sum of their current block upper bounds, $sumBlockUB$. If $nextDoc \leq minBlockBoundary$, i.e. the interval for potential skips of the pivot terms is covered by all current blocks, we replace the global upper bounds with the block upper bounds (line 14 is replaced with $othersUB \leftarrow (sumBlockUB - t.UB[t.currentBlock])$. Otherwise, we cannot use the blocks since they do not cover the whole interval, hence, the conditional-skip iterator with the global upper bounds must be applied.

While the original dynamic pruning algorithms advance each of the pivot terms independently to its next entry, the modified $advanceTerms()$ method promotes the term's cursor while considering the status of all other pivot terms. This enables bigger skips, as more matching documents can

Algorithm 2 Improving $advanceTerms()$ using the conditional-skip iterator

```
1: function $advanceTerms(termsArray[0..pTerm], \theta, sumUB)$
2:     if $(pTerm + 1 < |termsArray|)$ then
3:         $nextDoc \leftarrow termsArray[pTerm + 1].doc()$
4:     else
5:         $nextDoc \leftarrow \infty$
6:     if $(sumUB < \theta)$ then
7:         for all $(t \in termsArray[0..pTerm])$ do
8:             $t.skip(nextDoc)$
9:     else
10:        $nonSelected \leftarrow \{t | t \in termsArray[0..pTerm]\}$
11:        repeat
12:            $t \leftarrow pickTerm(nonSelected)$
13:            $nonSelected \leftarrow nonSelected - \{t\}$
14:            $othersUB \leftarrow (sumUB - t.UB)$
15:            $newDoc \leftarrow t.condSkip(nextDoc, \theta - othersUB)$
16:            if $(newDoc < nextDoc)$ then
17:                $nextDoc \leftarrow newDoc$
18:            $sumUB \leftarrow othersUB$
19:        until $nonSelected = \emptyset$
```

be safely surpassed when the sum of upper bounds of potential matching terms is lower than the current threshold.

4. ITERATOR IMPLEMENTATION

In this section we describe two implementations of $t.condSkip(d, \tau)$ that we experimented with in this work.

nextCondSkip. This implementation repeatedly calls $t.next()$ until the stopping condition is met, i.e., it stops on the first document with doc-id d' which satisfies $(d' \geq d) \bigcup (w(t) \times f(t, d') \geq \tau)$.

Note that while using the conditional-skip iterator saves many redundant document evaluations, this simple implementation comes with the cost of calculating the stopping condition for many pruned documents. On the other hand, it does not require any additional data structures, hence there is no extra memory overhead.

treapCondSkip. This implementation closely follows the ideas behind *treap* [21] – a randomized binary search tree. In a treap, the data is organized as a search tree by *key*, and as a heap by *priority*. For a root u, the keys of all nodes in its left subtree are less or equal to $u.key$, the keys of all nodes in its right subtree are greater than $u.key$, and the priorities of all nodes in its whole subtrees are less or equal to $u.priority$.

Our index follows a similar data layout, resembling [12], with doc-ids serving as keys and scores serving as priorities. The posting element with maximal score is selected as a root; all elements with smaller doc-ids form the left subtree, and all elements with larger doc-ids form the right subtree, recursively. The construction strives to create a balanced partition by selecting the root as median, by doc-id, among multiple elements with the same score. In addition, each node maintains the maximal doc-id in its rooted sub-tree, together with its doc-id and score.

The iterator is initialized by pointing to the leftmost leaf node in the tree. Upon calling to $t.condSkip(d, \tau)$, as long as the stopping conditions is not met, we traverse the tree in-order by doc-id, while pruning the whole subtrees in which both the maximal score is smaller than τ, and the maximal

[2]In our implementation, the order of term selection is determined according to their idf value.

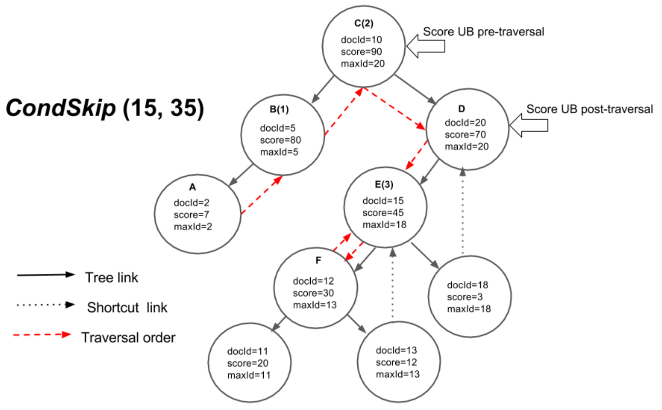


Figure 2: Treap-based index for a term's posting list. Each tree node maintains the doc-id as a key, and the score as priority. The iterator performs a $t.condSkip(d = 15, \tau = 35)$ starting from the leftmost node. It returns, in turn, the nodes B, C, and E. Note that the entire subtree rooted by F is skipped because both the maximal score of the subtree is 30 ($\leq \tau$), and the maximal doc-id is 13 (≤ 15). Pre-iteration, node C's score is the dynamic upper bound on suffix of the list; post-iteration, it is surpassed, hence node D score is the new upper bound.

doc-id is smaller than d. As an optimization, we maintain in-order shortcut links that allow hopping to the next node in constant time while backtracking uptree.

In addition to the basic iteration, the treap provides a dynamic upper bound ($t.dUB()$) on the scores of non-traversed yet nodes, to support further optimization. Formally, we define the dynamic upper bound of a term t, on the suffix of its posting list, to be $t.dUB() \overset{def}{=} w(t) \times \max_{d' \geq t.doc()} f(t, d')$. Initially, $dUB()$ points to the root node returning its score as the dynamic upper bound. Once the iteration surpasses the node that holds the upper bound, $t.dUB()$ updates the pointed node to be its right son. Figure 2 illustrates a tree traversal, including subtree pruning and dUB update.

Note that the treap nodes hold fixed-sized values (integers and references). Hence, the entire structure can be easily embedded in linear (array) storage, similarly to standard posting list implementations. It is amenable to state-of-the-art gap encoding algorithms that optimize storage space (e.g. [16]). Specifically for treaps, Konow et. al. [12] show how the treap representation can be stripped of any extra pointers, through a recursive (or alternatively, stack-based) traversal. In this work, we forgo further discussion of space compression as it is tangential to our contribution.

Iterator Performance.

The efficiency of the two implementations depends on the traversal scenario. *nextCondSkip* is faster when the skips are small because it only performs a few consecutive calls to $t.next()$, and therefore achieves a better memory access locality. *treapCondSkip*, on the contrary, is more efficient when the skips are big, and hence better covered by binary search. Our experiments (Section 5) demonstrate that the latter scenario is prevalent only for very short queries (1-2 terms), hence (somewhat surprisingly) the simpler implementation is best in most cases.

5. EXPERIMENTS

This section presents experimental results that compare the performance of the top-K retrieval algorithms (Section 2) with their versions enhanced with the conditional-skip iterator (Section 3).

5.1 Setup

We experimented with the dynamic pruning algorithms over two datasets: a dump of the the English Wikipedia collection from 12/01/2015 which contains about 4.3M documents (*https://dumps.wikimedia.org/enwiki/*), and the ClueWeb09 dataset, Category B (*http://lemurproject.org/clueweb09*) which contains about 50M English Web documents. We constructed the search indices using the Lucene open-source search engine, version 4.10.4 (*https://lucene.apache.org/*). The data was tokenized using the Lucene's English tokenizer, including lower-casing, stop-word removal (using Lucene's default stop-list) and Porter stemming.

In order to simulate in-memory search, the common practice typically applied by modern search engines, we uploaded all query term posting lists into memory before query execution. The posting-lists were stored in in-memory arrays, sorted by increasing doc-id. For supporting *treapCondSkip* we added a treap-based index on top of the posting array (a more sophisticated implementation [12] could avoid this overhead). The term global upper bounds were calculated during upload, while dynamic upper bounds were dynamically updated by the treap API during iteration. For *BMW*, we experimented with a few block sizes and fixed the size to 1024 posting entries for all terms based on its good performance on both datasets.

For document evaluation we applied a simple tf-idf scoring function with document length normalization. Equation 1 was implemented by setting $f(t, d)$ to be the frequency of term t in document d, multiplied by $1/\sqrt{|d|}$, where $|d|$ is the number of tokens in d, and $w(t)$ is the term's idf.

Our query test-set contains 1000 queries randomly sampled from a large query-log of a commercial Web search engine. The set was constructed by sampling 100 queries for each query length $1 \leq l \leq 10$. The query length was determined after tokenization, using the same tokenizer used for indexing. We filtered out all queries with terms that do not appear in the data in order to avoid misspelled or malformed terms to be included. Hence, for each query length l, our test-set contains exactly 100 queries, each includes l terms, where all terms appear in at least one of the search indices[3].

We note that the distribution of the query length in our test-set is very different from the typical length distribution of queries usually served by search engines. The average query length in our test-set is 5.5 while on the Web, for example, the average query length is 3.08 [24]. Furthermore, a typical query length distribution is long-tailed while our distribution is uniform. As we will show in the following, the query length is one of the major factors affecting the search performance. Moreover, the longer the query, the more challenging the pruning of the search process. Hence, our query set provides an extremely challenging test-bed for

[3]This query set has been intentionally selected to provide longer queries than those in existing publicly available query sets. It will be made available to the public.

		Wikipedia		ClueWeb	
Query	Mode	#DocEvals	Latency	#DocEvals	Latency
1-term	*OR*	187,437	26.56	1,600,569	229.4
	OR-next	3,667.5	2.15	6,237.4	2.84
	OR-treap	3,667.5	2.01	6,237.4	2.1
2-terms	*OR*	241,910.3	35.17	5,641,462.7	742.22
	OR-next	34,680.4	6.78	760,975.3	85.68
	OR-treap	34,680.4	6.05	760,975.3	56.75
3-terms	*OR*	492.150.4	72.6	10,332,329.8	992.56
	OR-next	193,510.2	39.6	3,050,727.5	463.9
	OR-treap	193,510.2	41.6	3,050,727.5	494.2

Table 1: Average performance (in doc-evaluations and query latency, measured in ms) for one-, two-, and three-terms *OR* queries, in original mode and with the two different implementations of *condSkip*, over the two datasets and with $K = 1000$.

the search algorithms and the dynamic pruning methods we experimented with.

We ran each query in the test-set using *OR*, *MaxScore*, *WAND*, and *BMW*, in their original mode as described in Section 2, and with the new *advanceTerms*() method which exploits the conditional-skip iterator (Algorithm 2) in the two implementation modes. Each query was processed by all algorithms with three different heap sizes (10,100,1000). Performance was evaluated by two measures:

- **#DocEvals** – the average number of doc-evaluation calls per query. This is our major measurement as the main goal of all dynamic pruning algorithms is to cut the number of evaluated documents while extracting top scored results. This measure has the advantage of being independent of both the software and the hardware environment, as well as the specific implementation.

- **Query latency** – the average execution time of the search process per query measured in milliseconds.This metric is the one directly experienced by the end user. It strongly correlates with #DocEvals as the computation overhead is dominated by document evaluation time. The more expensive the document evaluation function, the higher the impact of dynamic pruning on the latency.

Finally, statistical significance tests were performed using a two-tailed paired t-test with $p \leq 0.05$.

5.2 Conditional Skips

Short Queries

At first we report the results for one-term queries separately, as all algorithms perform exactly the same for such queries. The first rows of Table 1 show the average results of the *OR* search algorithm for 100 one-term queries over the two datasets, with $K = 1000$, in original mode (first row) and with the two different implementations of the conditional-skip iterator.

As can be seen, the new iterator cuts the number of document evaluations in about 98%. The full scan evaluates all documents in the term's posting-list (187.4K and 1.6M evaluations on average for Wikipedia and ClueWeb, respectively) while the new iterator evaluates much less (3.6K and 6.2K respectively), in order to extract the top-1000 scored documents. This huge cutoff is reflected by statistically significant cut of the run-time latency per query.

While #DocEval is indifferent to the specific implementation of $condSkip$, $treapCondSkip$ outperforms $nextCondSkip$, in terms of run-time latency for one and two terms queries. Interestingly, for longer queries, $nextCondSkip$ outperforms the treap-based implementation, as shown in Table 1 for three terms *OR*. This can be explained by the fact that large skips are better handled by the treap-based implementation while $nextCondSkip$ is superior for small skips. The average skip size in the ClueWeb index is 36.65, 12.42, 1.92, 1.60, and 1.42, for 1-to-5-term queries, respectively. We can see that the average skip size drops significantly with the query length. The relatively large skips for one and two term queries account for the better performance of the treap based implementation while the smaller skips for longer queries are better handled by $nextCondSkip$.

Document Evaluations

Figure 3 presents the #DocEvals of all algorithms on ClueWeb across different query lengths, in their original mode and with the $nextCondSkip$ implementation of conditional-skip[4].

The first clear observation is that for all algorithms, as could be expected, the number of evaluated docs monotonically increases with query length. Second, *BMW* outperforms all other algorithms significantly (please note that the *BMW* performance is two orders of magnitude better than the other algorithms hence it is represented on a different scale). We observe some difference in performance between the other algorithms; *WAND* outperforms *MaxScore* while both outperform *OR*. For larger heap sizes, the difference between these two algorithms drops, and in fact almost disappears for $K = 1000$. Additionally, the gap between *BMW* to the other algorithms is decreased.

Third, and most important, for all algorithms we observe a significant improvement in performance when the new iterator is used. For each query length, the new iterator brings statistically significant gain as validated by paired two-tailed t-test ($p < 0.05$). The second row in Figure 3 presents the cutoff percent of #DocEvals across the different query lengths and for the different heap sizes. In general, the cutoff percent decays with query length. However, even under extreme cases of 10 terms queries and a large heap size, the new iterator cuts the number of evaluated docs by more than 20% for *WAND*, and by more than 10% for *MaxScore* and *OR*. The cutoff percent for *BMW*, the most efficient algorithm, is reduced to 8%.

Query Latency

As mentioned above, for multi-term queries, $nextCondSkip$, outperforms $treapCondSkip$, in terms of query evaluation time, since the average skip size significantly decreases with the query length. As most of the queries in our test-set are long, we compare the latency induced by all algorithms using the $nextCondSkip$ implementation.

Figure 4 depicts the average query run-time latency of all algorithms on Wikipedia and on ClueWeb, with and without $nextCondSkip$, across different query lengths and for a large heap size ($K = 1000$). Obviously, the latency monotonically increases with query length. The difference in performance between *OR*, MaxScore, and *WAND* diminishes

[4]The results obtained for the Wikipedia dataset are omitted since they reveal exactly the same pattern.

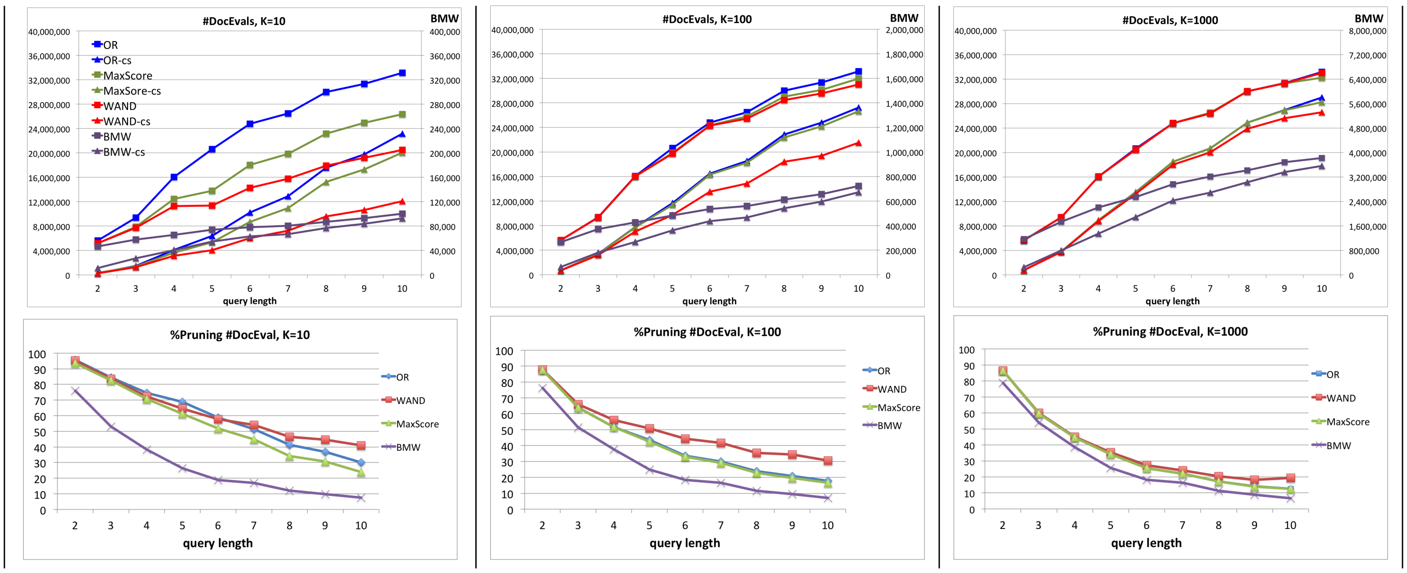


Figure 3: Top row presents the average #DocEvals per query on ClueWeb, across different query lengths and different heap sizes, performed by the algorithms in their original mode (rectangle marks) and when the conditional-skip iterator is used (triangle marks). Due to different scales in performance, *BMW* results are presented by the right vertical axis. Bottom row presents the cutoff percent of #DocEvals when the conditional-skip iterator is used.

for this heap size; *BMW* outperforms the other algorithms. Applying the conditional skip iterator reduces the latency of all algorithms, while the amount of cutoff is decreased with the query length. These results reflect the same dependency pattern of #DocEval on the query length (Figure 3).

5.3 Dynamic Upper Bounds

The treap-based index, in addition to supporting the conditional-skip iterator, dynamically updates the upper bound over the tail of the term's posting list, from its current cursor until the end of the list (See Section 4). These dynamic upper bounds (dUB) contribute to all pruning algorithms by improving the conditional-skip iterator, as they are tighter than the global upper bounds over the whole posting lists.

In order to examine the contribution of dUB to the search algorithms, independently of the *condSkip* iterator, we updated the original mode of *WAND* and *BMW*, the two algorithms that can potentially benefit from dynamic upper bounds, to use dUB rather than global upper bounds. Figure 5 shows the effect of using dynamic upper bounds rather than global upper bounds by the two algorithms, evaluated over ClueWeb with heap size $K = 1000$.

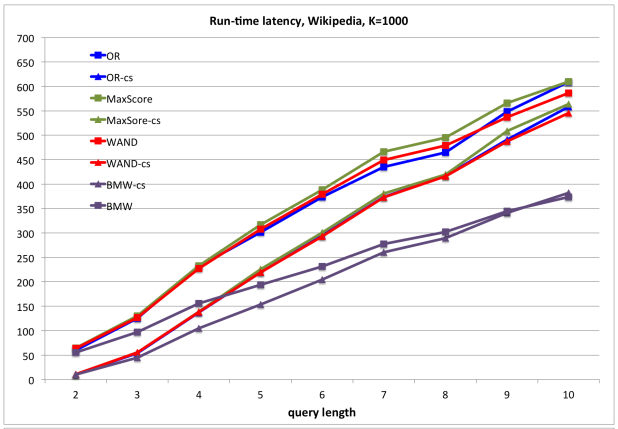
The results reveal the benefit of *WAND* from using dynamic upper bounds rather than global upper bounds, especially for long queries, however, with significant inferior performance with respect to the full application of the conditional-skip iterator. *BMW*, on the other hand, does not benefit at all from dUB while used by its original mode, probably due to the tighter block upper bounds it already uses. Therefore, the whole benefit in performance of *BMW* inferred directly from conditional skips.

6. RELATED WORK

Dynamic pruning methods for top-k query processing have been studied for decades [3, 18, 14, 25, 2, 1, 27]. Many pruning strategies have been developed for TAAT strategies (e.g., [3, 18, 14, 1]). These methods are typically unsafe as not all documents are fully evaluated due to early termination. A comprehensive overview on retrieval techniques in general, and dynamic pruning for TAAT strategies in particular, can be found in [27].

One of the advantages of DAAT strategies is that pruning can be done in a safe mode, by fully evaluating the candidate documents while skipping documents that cannot be part of the final result list [25, 2, 23, 8, 4, 20]. Turtle and Flood [25] introduced the *MaxScore* algorithm, which has a TAAT version, and a DAAT version. Strohman et al. [23] extended *MaxScore* to pre-compute "topdoc" list for each term, ordered by the term frequency. The algorithm starts with a union of the topdoc lists of the query terms to determine an initial candidates set to be scored. Our iterator can be efficiently applied for the task of identifying topdoc lists (See Table 1). Broder et al. [2] described *WAND* which can be applied in safe and in unsafe mode. These algorithms have been extended in several directions. Macdonald et al. [13] learned a better and tighter approximation of term upper bounds, which can improve the dynamic pruning algorithms. Fontoura et al. [9] suggested an efficient in-memory version for *WAND*. Shan et al [22] show that the performance of *WAND* and *MaxScore* is degraded when a document static score is added to the ranking function, and proposes how to efficiently handle such a case.

The *Block-Max WAND* (*BMW*) algorithm has been firstly described in [8], and independently by Chakrabarti et al. [4].. Dimopoulos et al. [7] experimented with *WAND* and *MaxScore*, with and without Block-Max Indexes, and suggested a new recursive query processing algorithm that uses a hierarchical partitioning of the posting lists into blocks. Petri et al. [19] analyzed *WAND* and *BMW* in the context of Language Model (LM) scoring function and showed that when the distribution of term scores is non-skewed, as in the LM



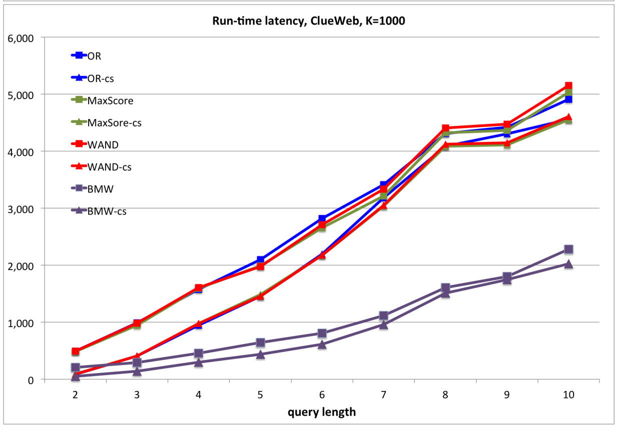


Figure 4: Average query run-time latency of all algorithms (in ms) over (top) Wikipedia, and (bottom) ClueWeb, across query length, with and without the $nextCondSkip()$ iterator.

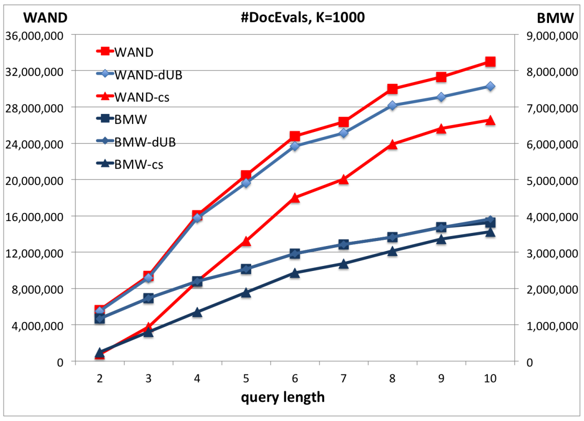


Figure 5: #DocEvals of $WAND$ and BMW on ClueWeb, across different query lengths, in three different modes: 1) with global upper bounds; 2) with dUB; 3) with $condSkip$.

case, then the efficiency of $WAND$ is degraded with respect to BMW. Rossi et al. [20] and recently Daud et al. [6] took advantage of a two-tiered index, in which the first tier is a small index, containing only higher impact entries of each inverted list. The small index is used to pre-process the query before accessing the full index, resulting in considerable speeding up the whole process.

The line of works that are mostly related to ours is of using effective data structures for supporting larger skips in the posting list. Culpepper et al. [5] and Konow et al. [11] used a *Wavelet tree* data structure [15] to represent a posting-list using the Dual-Sorted inverted list. Postings are sorted by decreasing term weights, while the wavelet tree can simulate ordering by increasing doc-ids. In recent work, Konow et al [12] introduced a new implementation for posting lists that is based on the *treap* data structure [21]. They described how a treap-based index can be encoded and how it can be used for implementing OR and AND top-k algorithms. Our work follows the usage of a treap-based index for implementing the conditional-skip iterator. We enhanced the treap tree by keeping the maximum doc-id in the sub-tree root-node, and by adding shortcut links from a node to its successor in the doc-id order, for further improvement during posting traversal. In addition, we showed how the treap tree can provide a dynamic upper bound on the suffix of the posting list that can effectively replaces the global term upper bound. In contrast to [12], our conditional-skip iterator is independent of specific retrieval algorithms, hence it can directly be applied by multiple search paradigms for dynamic pruning. Furthermore, our implementation of the baseline OR with the skip iterator, can be seen as a generalized way to realize the treap-based OR described in [12].

7. SUMMARY

In this work we introduced a new posting iterator, $condSkip(d, \tau)$, that can skip large parts of the term posting lists during top-k query processing. We demonstrated how the iterator can be applied by DAAT-based algorithms to significantly cut the number of evaluated documents. While the original dynamic pruning algorithms advance each of the matching terms independently, the new iterator enables the modified algorithms to advance the term's cursor while considering the status of all other matching terms. This consideration enables the detection of many documents that can be safely skipped when the sum of upper bounds of all potential matching terms is lower than the current heap threshold. We described two implementations of the new iterator, one based on existing APIs and the second on a treap-based organization of the posting lists. While the treap-based iterator is superior for short queries, the first one is favored for long queries where skips are expected to be very short.

Using two datasets and a large sample of queries with varying length, we demonstrated the contribution of the new iterator to some representative DAAT-based algorithms, OR, $MaxScore$, $WAND$, and BMW. Our experimental results demonstrate that the amount of contribution of the new iterator to dynamic pruning depends heavily on the query length. We showed a huge cutoff in the number of evaluated documents for short queries; the amount of cutoff is decreased for longer ones. Nevertheless, even for 10-term queries, and for a large heap size, we could still observe significant improvement in performance of all algorithms.

The new posting iterator opens many questions that were not covered in this work. One of the open issues when increasing K, the number of documents to extract, is how to initially set a decent cutoff threshold θ for heap insertion. Another open issue is the behavior of the new iterator with more complicated scoring functions, e.g. when the term scores are non-skewed [19]. We would also like to explore whether the new iterator can be used in unsafe processing mode, and by TAAT-based algorithms. We leave all these interesting and open questions for future work.

8. REFERENCES

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of SIGIR*, pages 372–379. ACM, 2006.

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

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

[4] K. Chakrabarti, S. Chaudhuri, and V. Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proceedings of ICDE*, pages 709–720, 2011.

[5] J. S. Culpepper, M. Petri, and F. Scholer. Efficient in-memory top-k document retrieval. In *Proceedings of SIGIR*, pages 225–234. ACM, 2012.

[6] C. M. Daoud, E. S. de Moura, A. Carvalho, A. S. da Silva, D. Fernandes, and C. Rossi. Fast top-k preserving query processing using two-tier indexes. *Information Processing & Management*, 2016.

[7] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of WSDM*, pages 113–122. ACM, 2013.

[8] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of SIGIR*, pages 993–1002. ACM, 2011.

[9] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[10] I. Guy. Searching by talking: Analysis of voice queries on mobile web search. In *Proceedings of SIGIR*, pages 35–44. ACM, 2016.

[11] R. Konow and G. Navarro. Dual-sorted inverted lists in practice. In *Proceedings of SPIRE*, pages 295–306. Springer-Verlag, 2012.

[12] R. Konow, G. Navarro, C. L. Clarke, and A. López-Ortíz. Faster and smaller inverted indices with treaps. In *Proceedings of SIGIR*, pages 193–202. ACM, 2013.

[13] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.*, 29(4):17:1–17:28, Dec. 2011.

[14] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, Oct. 1996.

[15] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[16] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proceedings of SIGIR*, pages 273–282. ACM, 2014.

[17] M. Persin. Document filtering for fast ranking. In *Proceedings SIGIR*, pages 339–348. ACM, 1994.

[18] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[19] M. Petri, J. S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *Proceedings of ADCS*, pages 58–65. ACM, 2013.

[20] C. Rossi, E. S. de Moura, A. L. Carvalho, and A. S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of SIGIR*, pages 183–192. ACM, 2013.

[21] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

[22] D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *Proceedings of WSDM*, pages 423–432. ACM, 2012.

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

[24] J. Teevan, D. Ramage, and M. R. Morris. #twittersearch: A comparison of microblog search and web search. In *Proceedings of WSDM*, pages 35–44. ACM, 2011.

[25] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, Nov. 1995.

[26] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of SIGIR*, pages 105–114. ACM, 2011.

[27] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.