Fast Parallel Top-K Retrieval of Verbose Queries

Gali Sheffi
Yahoo Research, Oath
gsheffi@oath.com

Dmitry Basin
Yahoo Research, Oath
dbasin@oath.com

Edward Bortnikov
Yahoo Research, Oath
ebortnik@oath.com

David Carmel
Amazon
david.carmel@gmail.com

Idit Keidar
Technion and Yahoo Research, Oath
idish@ee.technion.ac.il

Abstract

Top-k document retrieval is emerging as a performance bottleneck in query processing as verbose queries are becoming mainstream, while state-of-the-art algorithms fail to process such long queries in real time. To date, attempts to accelerate top-k evaluation via approximate result computation and intra-query parallelism have produced limited results.

We present Sparta – a practical parallel algorithm that exploits multi-core hardware to run approximate top-k retrieval within interactive latency bounds. Its design is inspired by Fagin et al.'s seminal Threshold Algorithm for aggregation in databases. Sparta scales through lightweight coordination and context sharing among concurrent threads. The resulting algorithm is highly scalable, both in the number of query terms and in the searched index size. For example, when applied to the 50M-document public ClueWeb09B dataset, Sparta processes 12-term queries 3.6 times faster than the state-of-the-art. On a tenfold bigger index, Sparta continues to process queries at the same speed, whereas today's best-in-class algorithm is more than 60 times slower. Sparta further significantly improves throughput.

2012 ACM Subject Classification C.1.4 Parallel Architectures and D.1.3 Concurrent Programming

Keywords and phrases Parallel computing, web-search, top-k

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Internet users are changing the ways in which they interact with search engines. While early days web queries were short (2.4 terms on average [27]), modern search experiences (query suggestion, reformulation, conversational interfaces, etc.) stimulate their users to submit much longer queries. For example, more than 5% of queries submitted via voice search on mobile devices exceed 10 terms [18]. At the same time, usability studies (e.g., [6]) show that users are extremely sensitive to end-to-end delays beyond 500 ms, and any excess delay beyond 250 ms leads to material degradation in their experience. Maintaining this *service-level agreement* (*SLA*) is becoming a major challenge as more queries become longer.

Most search systems today evaluate queries via a two-stage process [33]. The first stage is *top-k retrieval*: it roughly matches the top-k documents most relevant to the query (typically, hundreds to thousands) based on some simple relevance scoring function like TF-IDF or BM25 [7]. The second re-ranks the documents via some sophisticated (e.g., machine-learned) function, to produce the final result list (typically, a few tens of documents). The first stage sifts through huge volumes of data and dominates the execution time. We therefore focus on the latency induced by this backend search tier at a given search node that serves an index shard.

Obtaining the exact top-k matches out of a large corpus is often too slow to meet stringent SLA requirements. Given that the top-k selection is only the first query processing step, perfect results are often not essential, as later query processing stages can be satisfied by approximate results [22]. We leverage this observation and focus on *approximate* (sometimes called *non-safe*) query evaluation, tuned to achieve a certain high recall (e.g., 97% or more).

State-of-the-art sequential top-k algorithms, e.g., *Block-Max WAND* (*BMW*) [14], serve traditional (short) web queries with impressive speed. However, these algorithms do not scale well when applied to verbose queries and big data collections (as pointed out in [10] and confirmed by our experiments). A promising approach to address this challenge is to parallelize query evaluation on multiprocessor hardware. The current trend in server architecture, which favors parallelism over sequential speed, makes this approach particularly attractive. For example, Rojas et al. [26] parallelize BMW – we refer to their algorithm as *pBMW* – with a considerable performance improvement. Nevertheless, our experiments show that pBMW has limited scalability, both in the number of query terms and in the index size, and cannot meet expected SLAs on long queries or large indices.

We present *Sparta* (Scalable PARallel Threshold Algorithm) – a novel concurrent algorithm that substantially improves search time for verbose queries over large search indices. Its query latencies fit well within the real-time SLA on standard server hardware.

Sparta's design is inspired by the seminal *Threshold Algorithm (TA)* by Fagin et al. [16] for retrieving the top-k objects from a database based on an aggregation of attributes that may reside in multiple nodes. Perhaps surprisingly, this algorithm was largely ignored in the web search literature. In the web setting, term posting lists represent attributes, and the ranking function is a linear aggregation of term scores. TA has two variants [16]: (1) *Random Access* (*RA*), which relies on random access to posting lists; and (2) *No Random Access* (*NRA*), which only traverses them sequentially. The former is ineffective in the context of real-time search because it requires (1) a secondary random-access index (which doubles the required index space), and (2) random I/O to the on-disk secondary index, which incurs high overhead when the index is too big to be kept entirely in RAM. We therefore base our algorithm on NRA.

Transforming NRA into an efficient concurrent algorithm is challenging because coordination around shared state can become a major bottleneck unless carefully designed. On the one hand, sharing state among threads is essential in order to benefit from TA's early stopping feature. Indeed, a shared-nothing (partitioned) parallelization of NRA performs two times worse than a single-threaded implementation. On the other hand, a naïve attempt to parallelize NRA using shared memory also results in worse performance than the sequential algorithm. Sparta judiciously shares information among threads while significantly reducing the synchronization overhead and memory overlap among them, leading to major performance gains.

Our results show that Sparta scales well with both query length and corpus size. E.g., on the 50M-document TREC ClueWeb09B dataset [12], Sparta can serve verbose 12-term queries within less than 200 ms, whereas a parallel execution of RA (denoted, pRA) processes them in 400 ms, and pBMW takes more than 600 ms. On a 500M-document index, its latency is virtually unchanged, whereas pRA requires 2 seconds and pBMW's latency surges to almost 10 seconds. The latter occurs because unlike Sparta and pRA, pBMW's processing time grows linearly with the index size.

Sparta's throughput on a production-grade query mix (with the query length distribution measured in [18]) is twice that of pBMW on the small corpus, and 25x faster on the large one.

Summing up, we present a practical scalable parallel algorithm that significantly improves the state-of-the-art latency of long query processing on large search indices. The remainder of this paper is organized as follows. Section 2 defines the top-k retrieval problem and associated metrics. Section 3 gives background on existing algorithms. Section 4 presents Sparta, and Section 5 evaluates it. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Top-k Retrieval

We focus on the fundamental primitive of top-k retrieval, commonly used by search engines for initial selection of documents over which more refined search is performed [33]. Given a query q, the primitive retrieves the top k scored documents in the corpus according to a scoring function $score(D, q)$. More specifically, a *query* is given as a list of terms (to search for). Given an m-term query $q = t_1, \dots t_m$ and a document D, the score of D for query q is $score(D, q) \triangleq \sum_{i=1}^{m} ts(D, t_i)$ where $ts(D, t_i)$ is the term score of term t_i to document D.

An *exact* top-k retrieval primitive returns a list of the k documents with the highest scores for the query. An *approximate* solution returns a list of k results that approximates the top k. Let L be the exact list of top-k documents sorted in decreasing order of document scores, and let A be the list returned by an approximate solution. The quality (or accuracy) of A is measured by its *recall*, which is the fraction of the actual top-k included in A.

3 Background

We provide background on state-of-the-art top-k algorithms and Fagin et al.'s TA [16].

3.1 State-of-the-art Top-k Algorithms

Search algorithms use a preprocessed inverted index of the corpus. The index is organized according to terms. For each term, the index holds a *posting list* listing all documents associated with that term. Top-k retrieval algorithms typically traverse posting lists sequentially; multiple lists may be scanned simultaneously. For big datasets, only a small portion of the index can reside in RAM at any given time. However, the I/O overhead is low because contiguous chunks of the lists are infrequently fetched from disk into memory.

In order to avoid scoring a huge number of documents per query, state-of-the-art algorithms reduce the number of evaluated documents while identifying the top scored results. *MaxScore* [28, 32], *WAND* [11], and *Block-Max WAND (BMW)* [14] are popular examples of such algorithms, widely used in production systems. They simultaneously scan all relevant posting lists in order of document id, evaluating the full score of each document before moving to the next one. They track the top-k documents among those scored so far (typically, in a heap). A variable Θ – called the *threshold* – holds the score of the k-th (lowest-ranked) document in the heap; any document whose score is below this threshold is not a candidate for the final top-k list. As long as the heap contains fewer than k documents, Θ remains zero.

3.2 The Threshold Algorithm

TA [16] was originally presented in a database setting, where the partial scores of an item (term scores in our context) reside at different nodes. We cast it here in the IR setting, where partial scores are obtained from posting lists rather than nodes, and query evaluation occurs on a single machine, possibly using multiple threads accessing shared memory. Although Fagin et al. have mentioned the applicability of TA to top-k retrieval over inverted files [15], it has been largely overlooked by the IR community.

TA uses term posting lists sorted in decreasing order of term score. To evaluate a query $q = t_1, \dots t_m$, it dynamically traverses the m corresponding posting lists of the query terms in an interleaved manner. An *upper bound* vector, $UB[m]$, bounds the term scores of documents that were not yet visited in each term's posting list. Figure 1 shows an example posting list traversal and the corresponding values in UB. Here, $m = 3$ and the scores of the last traversed items in each list are UB[1] = 38, UB[2] = 32, and UB[3] = 41.

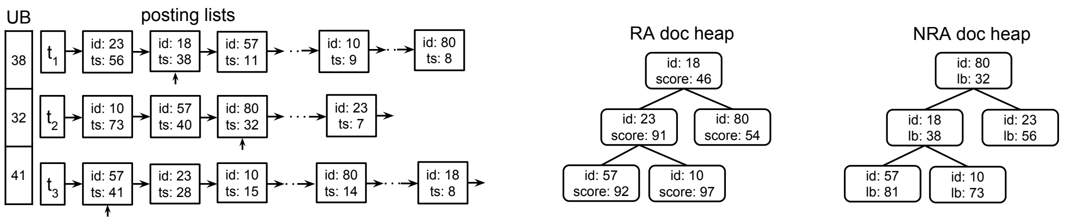


Figure 1 Traversal example in RA and NRA variants of the Threshold Algorithm (TA). Posting lists are sorted by decreasing term score. Vertical arrows depict iterator positions. UB holds upper bounds on the terms' contributions to scores of untraversed documents. The RA document heap is ordered by full document score, and the NRA heap by partial score (lower bound).

TA also maintains a threshold Θ, i.e., the score of the k-th document in the heap. It stops when no candidate can exceed the threshold score. Fagin et al. present two flavors of the algorithm, which we now describe.

3.2.1 Random Access (RA)

The RA variant assumes that given a document id, we can use random access in order to obtain all its term scores in order to compute its total score. It thus computes the full score for every document it encounters. If the score is higher than the threshold, it is inserted into the heap. Then, Θ and the corresponding term's UB are updated. The algorithm stops when the following *upper bound stopping condition* holds:

$$\mathit{UBStop} \triangleq \sum_{i=1}^{m} UB[i] \leq \Theta. \tag{1}$$

At this point, no non-traversed document can achieve a high enough score to be included in the heap. RA's output is the set of documents in the heap, ordered by their scores.

RA is an exact algorithm as it returns the top-k results. In our evaluation, we implement an approximate variant of RA by stopping whenever the heap does not change for some parameter Δ ms. Note that the Threshold Algorithm is amenable to such early stopping because it traverses posting lists in order of score. High scoring documents are likely to become evident early, and finding new high scoring documents becomes less and less likely as the algorithm progresses. This is in contrast with algorithms like BMW (and consequently, pBMW), which traverse posting lists in order of document id, and hence finding high scoring documents remains equally likely throughout their execution.

Unfortunately, random access is costly, in particular for large data sets that do not reside fully in RAM. Whereas a sequential posting list traversal requires infrequent I/O (at the end of each data block) and exhibits cache-friendly locality of access, each random document access entails an I/O request and a cache miss. RA has an additional drawback in the IR setting, as it needs to maintain a secondary index by document id in addition to the posting lists sorted by term score, which doubles its footprint.

3.2.2 No Random Access (NRA)

The alternative NRA method refrains from computing the full score for each traversed document, and instead maintains lower and upper bounds for candidate documents based on *partially* computed scores. For a document D and a term t_i, we define the upper bound $UB(D, t_i)$ to be the term score $ts(D, t_i)$ if it has already been encountered, and otherwise $UB[i]$, which provides an upper bound on t_i's term score. We similarly define its lower bound $LB(D, t_i)$ to be the term score if it is known, and zero otherwise. We then aggregate these scores to compute the document's upper and lower bounds:

$$UB(D) \triangleq \sum_{i=1}^{m} UB(D, t_i) \; ; \; LB(D) \triangleq \sum_{i=1}^{m} LB(D, t_i).$$

E.g., in Figure 1, $UB(D_{57}) = 38 + 40 + 41 = 119$ and $LB(D_{57}) = 40 + 41 = 81$, whereas its actual score is $11 + 40 + 41 = 92$.

NRA maintains the top-k heap according to the document *lower bounds*, and Θ holds the smallest value among them. Its output is the set of documents in the heap, sorted by LB.

NRA's safe variant stops when (1) the *UBStop* stopping condition of RA holds, and (2) all the visited documents that are not in the heap have *upper bounds* lower than or equal to Θ. These two conditions are complementary: (1) ensures that no non-traversed documents are among the final top-k, whereas (2) ensures the same for traversed documents that are not among the current top-k. While NRA does return the exact top-k results, unlike RA, it does not necessarily preserve the order among them, since some returned documents may be partially scored. As in RA, the approximate variant stops after the heap has not changed for Δ ms.

4 Sparta

Sparta is a parallel algorithm that adapts NRA to shared-memory multiprocessor hardware platforms. Like our implementation of RA and NRA described above, it can be configured to provide approximate results by stopping after the heap does not change for some Δ time.

Section 4.1 describes the algorithm's data structures. Section 4.2 explains how we divide the work involved in query processing among threads. Section 4.3 describes synchronization around the shared data structures. Finally, Section 4.4 discusses the properties of our algorithm.

4.1 Sparta Data Structures

Table 1 defines the data structures used by Sparta and Figure 2 illustrates them. As in NRA, the algorithm maintains the current top-k results in a heap, *docHeap*, and its lowest value in Θ. It keeps m pointers to the next elements to traverse in all posting lists (not listed in Table 1) and an array UB of upper bounds on non-traversed term scores. UB is used for computing the documents' upper bounds as well as for checking NRA's stopping conditions.

The hash map *docMap* maps document ids encountered thus far to document *DocType* objects. A *DocType* holds a vector of term scores observed thus far for this document as well as a lower bound on the document score computed as their sum. NRA's first stopping condition is checked by the macro *UBStop*, and the second stopping condition is checked as follows:

$$\forall D \in \textit{docMap} \setminus \textit{docHeap} : UB(D) \leq \Theta. \tag{2}$$

name	value	name	value
DocType	$\langle$ int id, int score[m], int LB $\rangle$	*docMap*	init empty
docHeap	init empty	*heapUpdTime*	init now
Θ	init 0	*done*	init false
$UB[m]$	init ∞	*termMap*[m]	init pointer to *docMap*
UBStop	$\sum_{i=1}^{m} UB[i] \leq \Theta$	*tmpDocMap*	init empty
$UB(D)$	$\sum_{i=1}^{m} \big(D.score[i] > 0 \;?\; D.score[i] : UB[i]\big)$		

Table 1 Sparta's data structures and initial values.

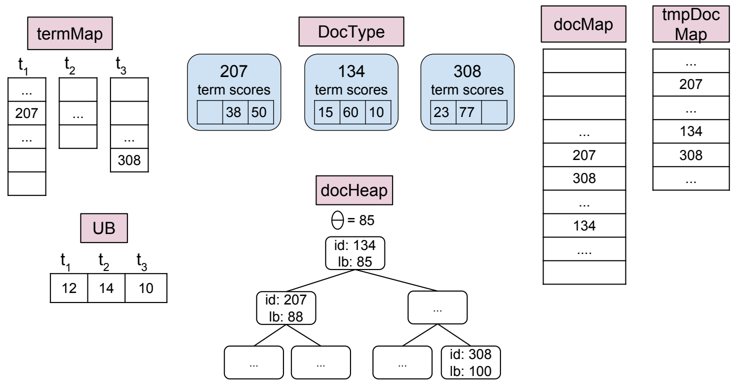


Figure 2 Sparta data structures. The *docMap* keeps track of partially scored documents; *tmpDocMap* and *termMap* are local partial copies of *docMap*.

The stopping conditions are evaluated by a *cleaner* task, which also checks the heap's latest update time *heapUpdTime* and sets the *done* flag once the algorithm can stop.

In addition, to reduce the synchronization overhead and improve cache locality, Sparta uses two local data structures that hold partial copies of the global *docMap*, namely the *termMap* array with a (local) hash map per term, and the *tmpDocMap* used by a dedicated maintenance thread. The role of these will become evident when we discuss synchronization and locality below.

4.2 Splitting the Work

A naïve attempt to parallelize NRA would be to share the data structures among all threads. (Note that sharing state is essential because the partial document scores, and consequently the lower bounds, are affected by multiple threads that generate term scores). This approach leads to high contention, primarily around *docMap*.

To reduce contention, we consider the point in time when the first stopping condition (Equation 1) holds. From this time on, no new document's score can surpass the lower bound of any document in the shared *docHeap*. Therefore, adding new documents to *docMap* is no longer helpful (a similar observation was made in [24]). On the other hand, it is possible to shrink *docMap* by removing documents whose upper bounds are smaller than Θ. This is cardinal for the concurrent implementation, which can stop sharing *docMap* among the threads once it is sufficiently small, thereby eliminating the synchronization overhead.

Sparta's pseudocode appears in Algorithm 1. It exploits up to m *worker* threads per query, but can run with fewer threads if less are available. We divide posting list traversals to segments of size *segSize*, and use a job queue to allocate posting list segments to threads (line 2). The

Algorithm 1 Sparta algorithm.

```
1:  for $i = 1$ to $m$ do            ▷ processing $m$-term query
2:      add PROCESSTERM($i$) to job queue
3:  spawn up to $m$ threads to run jobs from queue
4:  wait until *UBStop*  ▷ all candidates are in *docMap*
5:  add CLEANER() to job queue
6:  wait until *done*
7:  return *docHeap*

8:  procedure PROCESSTERM($i$)
9:      if *UBStop* $\wedge$ $|docMap| < \Phi$ then
                                   ▷ *docMap* shrinking and small
10:         if $termMap[i] = docMap$ then
11:             INITMAP($i$)
12:     for $j = 1$ to *segSize* do
13:         if *done* then return
14:         $\langle id$, score$\rangle \leftarrow$ next entry in $i$th posting list
15:         $D \leftarrow termMap[i](id)$
16:         if $D = \perp$ then              ▷ document missing
17:             if $\neg$*UBStop* then        ▷ hash incomplete
18:                 create new document object $D$
19:                 add $D$ to $termMap[i](id)$
20:             else continue
21:         $D.score[i] \leftarrow$ score    ▷ update term score
22:         if $\Sigma_{j=1}^{m} D.score[j] > \Theta$ then
23:             UPDATE_HEAP($D$)
24:     $UB[i] \leftarrow$ score  ▷ update term's upper bound
25:     add PROCESSTERM($i$) to job queue

26: procedure INITMAP($i$)
                                   ▷ create local copy of *docMap*
27:     *termMap*[i] $\leftarrow$ new hash map
28:     for all $D \in docMap$ s.t. $D.score[i] = 0$ do
29:         add $D$ to *termMap*[i]

30: procedure UPDATE_HEAP($D$)
31:     lock *docHeap*
32:     if $D \notin docHeap$ then
33:         insert $D$ to *docHeap*
34:         for all $d \in docHeap$ do
35:             $d$.LB $\leftarrow \Sigma_{j=1}^{m} d.score[j]$
36:             move $d$ to correct place in heap
37:         if $|docHeap| > k$ then
38:             remove lowest scored doc
39:         if $|docHeap| = k$ then
                                   ▷ set $\Theta$ to $k^{th}$ lowest score
40:             $\Theta \leftarrow$ lowest score in *docHeap*
41:         *HeapUpdateTime* $\leftarrow$ current time
42:     unlock *docHeap*

43: procedure CLEANER
44:     if $|docMap| > \Phi$ then          ▷ shrink *docMap*
45:                       ▷ keep only docs that are still relevant
46:         *tmpDocMap* $\leftarrow$ new hash map
47:         for every doc $D \in docMap$ do
48:             if $UB(D) > \Theta \vee D \in docHeap$ then
49:                 add $D$ to *tmpDocMap*
50:         replace *docMap* by *tmpDocMap*

                           ▷ check algorithm's stopping conditions
51:     $Stop2 \leftarrow \big(|docMap| = |docHeap|\big)$
52:     if $Stop2 \vee HeapUpdateTime + \Delta <$ now then
                                   ▷ in exact version $\Delta = \infty$
53:         *done* $\leftarrow$ true
54:     else add CLEANER() to job queue
```

PROCESSTERM(i) function processes the next segment of term t_i. A thread that finishes its assigned segment inserts into the queue a new task for scanning the next segment in the same term's posting list (line 25). Thus, we progress on all posting lists at the same rate modulo the segment size. In case m threads are available, a large segment size can be used.

In addition to posting list-traversing tasks, the CLEANER function (lines 43– 54) performs maintenance on the *docMap*. This task is invoked once Equation 1 holds and so *docMap* no longer grows. The cleaner serves two purposes. First, as its name suggests, it removes entries that ceased to be top-k candidates from *docMap*. Since Sparta is memory-intensive, a smaller *docMap* allows it to run much faster; (a similar observation, in a sequential setting, was made in [17]). Second, it detects the stopping conditions (line 51). In the approximate version, it checks whether the heap has not changed for Δ time (the exact version is obtained by setting $\Delta = \infty$). It also checks the condition of Equation 2: once *docMap* is the same size as *docHeap* we know that the two are identical because *docMap* always includes all *docHeap* entries. At this point, *docHeap* holds the top-k scored results, and stopping is safe. Once the algorithm stops, the main thread returns the heap's contents.

4.3 Synchronization

Note that, *docHeap*, UB, *docMap*, and *DocType* objects referenced by them are accessed concurrently by multiple threads. We need to protect such access to avoid inconsistencies. On the other hand, reducing contention is crucial for performance. Moreover, Sparta is a memory-intensive algorithm, and in order to keep the memory access latencies low, it is paramount to exploit the CPU hardware cache, in particular the core-private L1 caches. We now explain how we synchronize access to each of the shared variables in a way that reduces contention and improves cache utilization.

Since at most one thread processes each term, no races arise around updating UB entries, and no lock is needed. However, all threads read all UB entries, and therefore frequent updates can lead to frequent cache misses, and in turn, poor performance. In order to reduce the number of cache misses, instead of updating UB after each document evaluation, the workers update it lazily, at the end of a segment traversal (line 24). Since upper bounds can only decrease whereas Θ can only increase, such lazy updates do not affect correctness.

Updates of *docHeap* and Θ are protected by a shared lock (lines 31 and 42), which serializes all updates. To avoid races around evaluating a *DocType*'s lower bound and inserting it into *docHeap*, we update the lower bound in a lazy manner while holding the global lock on *docHeap*: Every thread that adds a document to the heap updates the lower bounds of all heap documents (lines 34-36).

Before the first stopping condition (Equation 1) holds, multiple workers update *docMap* concurrently. We therefore protected each hash bucket by a granular lock, which performed better than the generic Java concurrent hashmap [1].

The cleaner task starts removing elements from *docMap* after it is guaranteed that no new entries are added to it; such removals substantially improve the term processing performance. Nevertheless, allowing the cleaner to constantly update *docMap* would lead to frequent cache invalidations at the tasks that read the map. To avoid frequent cache misses, the global map is kept read-only most of the time, while the cleaner works on a local copy: it repeatedly builds a new map *tmpDocMap*, holding *docMap* entries whose upper bounds are higher than Θ as well as ones that are included in *docHeap* (whose upper bounds may be exactly Θ); recall that other *docMap* entries no longer need to be kept. Once *tmpDocMap* is ready, the cleaner replaces *docMap* with it via a single pointer swing (flipping the global reference).

Access to *docMap* is a principal performance bottleneck, since it is frequently read by all worker threads. Initially, it is too large to fit into local caches, and so the parallel execution inherently requires global memory accesses. But thanks to the cleaner's work, *docMap* shrinks in the course of the execution. Moreover, not all *docMap* entries are relevant for all terms – if D's term score for t_i has already been computed, then a thread handling term t_i does not need to access D. Thus, the relevant subset of *docMap* for each term eventually becomes small enough to fit in its local cache. As long as the thread continues to access the global *docMap*, it experiences massive cache misses every time the cleaner replaces the global *docMap*. But once it becomes small enough to fully fit in the local cache, there is no need to keep using the global copy.

To this end, Sparta associates a local map replica, *termMap*, with each posting list. *termMap* is created by the worker that currently owns that posting list once *docMap*'s size drops below a threshold Φ, in our implementation, $\Phi = 10$K entries. The INITMAP function scans *docMap*, and copies to *termMap* the references to those *DocType* objects that do not contain the score for the worker's term yet. Once a *termMap* has been created, every worker that handles its posting list uses it. Note that since every posting list is accessed by a single worker at any given time, no synchronization is required.

4.4 Analysis

Sparta accesses posting lists in the same manner as NRA does, and stops only when NRA's stopping conditions hold. Thus, like NRA, its exact version ($\Delta = \infty$) is safe, and returns the top-k results.

In terms of performance, NRA was shown to be *instance-optimal* when random access is impossible [15], namely, its number of accesses to posting list entries is asymptotically close the optimum for every problem instance. This property holds for pNRA as long as the rates in which different threads access different posting lists are within constant multiples of each other [15], because in this case a thread that "runs ahead" without knowing it should stop only accesses a constant factor more entries than the algorithm needs to, which the asymptotic analysis ignores. Sparta differs from pNRA in deferring updates to UB until the end of the segment, which may further delay stopping for *segSize* additional posting list accesses. Since *segSize* is constant, Sparta is asymptotically instance-optimal under the same assumptions as pNRA.

5 Evaluation

We compare the performance of Sparta to the following algorithms: parallel BMW (pBMW) [26], the best-in-class multiprocessor implementation we are aware of, a parallel implementation of RA (pRA), a shared-nothing (partitioned) parallelization of NRA (sNRA), and a naïve shared-state parallel implementation of NRA (pNRA). Although our primary focus is on approximate algorithms, for completeness, we also experiment with their exact counterparts.

In what follows, Section 5.1 describes the experiment setup, Section 5.2 explains how the algorithms are implemented, and Section 5.3 presents our results.

5.1 Experiment Setup

We study the algorithms in terms of query latency and throughput attainable at a single multi-core server. We use mid-tier industry-standard hardware – a 12-core Intel Xeon E5620 with 24GB RAM and 1TB SSD drive.

The benchmarking environment and the algorithms are implemented in Java. A *benchmark driver* draws queries from an input queue and submits them to the algorithm being tested, which uses a thread pool for intra-query parallelism. The driver controls the pool size. When testing latency, the thread pool is used by a single query. In the throughput evaluation mode, queries are scheduled first-come-first-served, and a new query is scheduled for execution (i.e., assigned threads) once there are idle threads with no outstanding work from currently executing queries. All queries scheduled for execution equally share the thread pool.

In all experiments, the appropriate index (either in id order or in score order) is pre-built offline and stored uncompressed on disk as a collection of binary files, each storing a shard of data partitioned by term. The benchmark environment memory-maps the content of these files via the MappedByteBuffer API [1]. Prior to each experiment, we flush the file system's page cache so all pages are physically read from disk during the experiment.

We experiment with two document corpora. The first is the TREC ClueWeb dataset, Category B (ClueWeb09B) [12], which is widely used for information retrieval research. This dataset includes approximately 50M web documents and takes up roughly 30GB of original content, uncompressed. The second corpus is a synthetic 10x scale-up of ClueWeb, named ClueWebX10, which we created to explore the algorithms' scalability with the dataset size. The 450M synthetic documents in ClueWebX10 are generated as follows. Each document is a bag of words drawn from the original ClueWeb dictionary (the order is immaterial for our document scoring function) so that the number of occurrences of a term t_i with an original global frequency rate of $F(t_i)$ is drawn from a geometric distribution with a stopping probability of $1 - F(t_i)$. This process preserves the term frequency distribution of ClueWeb in ClueWebX10.

We use the popular Lucene open-source search engine [2] for text tokenization, posting list maintenance, and term statistics retrieval. We score documents using a standard tf-idf score function with document length normalization [7].

We draw queries from the public AOL search log [3]. For each number of terms from 1 to 12, we independently sample 100 queries of this length uniformly at random from the AOL log. We also experimented with a query log of another commercial web search engine; this experiment produced statistically similar results, and so we omit them here.

We use $k = 1000$. (Experiments with $k = 100$ produced qualitatively similar results).

5.2 Implementation

Posting lists are stored as contiguous uncompressed arrays; pRA also stores its secondary index (document id to position in the posting list mapping) in the same form. Term scores are stored in the posting lists as integers, scaled by 10^6 and rounded as in [10]. Using integer arithmetics instead of floating-point significantly speeds up document evaluation.

The specific algorithm implementations are as follows.

5.2.1 pBMW

Our implementation of pBMW closely follows the description in [25]. The algorithm partitions the execution of the sequential BMW [14] among multiple threads. Each thread handles a distinct subset of documents, and computes a local top-k result. The algorithm then merges the partial results to obtain the final top-k.

Similarly to Sparta, pBMW's threads obtain jobs from a common job queue. Here, a job defines a range of document ids to scan. We set the number of jobs to be twice the number of worker threads, and assign equal-size ranges to all threads. This partition results in well-balanced execution in which the whole worker pool is utilized most of the time.

Each thread maintains a thread-local heap with the current top-k documents. (We also experimented with a shared heap and got inferior results; a similar finding was reported in [25].) Similarly, each thread T maintains a local threshold Θ_T for filtering heap insertions; Θ_T is *at least* the lowest score in the local heap, but may be higher thanks to the progress of other threads. Thread T periodically compares Θ to its local Θ_T, and promotes the smaller of the two to $\max(\Theta_T, \Theta)$. This way, slower workers catch up with faster ones.

pBMW splits posting list segments into blocks, and uses block-level statistics to prune the search [14]. We experimented with multiple block sizes and selected 64, which yielded the best performance. The approximate version's pruning aggressiveness is controlled by a parameter $f \geq 1$, which multiplies Θ to obtain a higher threshold for document score upper bounds [11]. For $f = 1$, the algorithm is exact.

5.2.2 Parallel TA variants

sNRA is a shared-nothing parallelization of NRA, where the index is partitioned to 12 shards by document id. Each thread finds the top-k documents in its shard by running NRA independently with thread-local data structures. When all threads complete, their lists are merged and the global top-k documents are kept.

pNRA is a naïve shared-state parallelization of NRA that does not employ Sparta's optimizations. Namely, it uses a shared document map, which it does not clean, and updates the term upper bounds

	Sparta	pNRA	sNRA	pRA	pBMW
ClueWeb	860	13,291	5,553	480	750
ClueWebX10	12,010	90,000+	56,223	7,410	10,210

Table 2 Average query latency (in ms) of 12-term queries with exact algorithms using 12 threads. None of the algorithms meets real-time SLAs.

upon every document evaluation. As in Sparta, a dedicated task checks the stopping condition. (Distributed stopping detection yielded worse results).

pRA maintains its results in a shared heap (experiments did not show any advantage to using local heaps). Note that the algorithm's multiple worker threads may encounter postings of the same document independently, and consequently score that document and try to insert it into the heap multiple times. The implementation guarantees the uniqueness of insertion (only the first one takes effect).

Since RA's stopping detection is lightweight, we do not dedicate a task to it. Instead, all workers check the *UBStop* condition and monitor the time elapsed since the last heap update, and notify each other if they decide to stop.

5.3 Results

For an algorithm A, we consider the following variants: exact (denoted A-exact), high-recall (denoted A-high), and low-recall (denoted A-low). Note that the approximate algorithms' parameters (Δ and f) affect the recall but do not directly control it; our high recall instances are ones that empirically achieve a recall of 97% or higher for both datasets.

5.3.1 Exact Algorithms

Our first experiment shows that none of the exact algorithms meets a real-time SLA for verbose queries. Table 2 depicts the mean processing latencies of 12-term queries with 12 worker threads (i.e., a single query fully exploits the multi-core CPU). Most of the algorithms complete within 1 second for on the ClueWeb dataset, but run for many seconds on ClueWebX10. pRA is the fastest algorithm in this setting, whereas pNRA is much worse than the other algorithms. Its average latency exceeds 13 seconds on ClueWeb and 1.5 minutes on ClueWebX10.

We will revisit the algorithms' execution dynamics – namely, how fast they accrue their results – as we study approximate algorithms in the sequel.

5.3.2 Approximate Algorithms

With exact instances of all algorithms failing to match real-time requirements, we turn to focus on the approximate instances. We parameterize Sparta, pRA, pNRA, and sNRA with $\Delta = 10$ ms. This yields high recall in all four algorithms, and satisfactory performance (meeting SLA requirements) in Sparta. We instantiate pBMW with $f = 5$ for high recall and $f = 10$ for low recall.

Accuracy. Table 3 depicts the empirical accuracy results for 12-term queries.

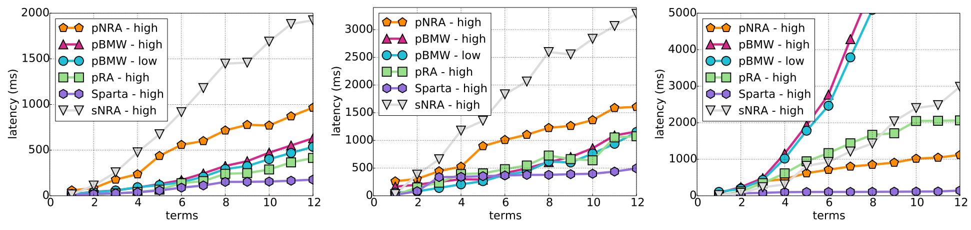
Latency. Figure 3 depicts the scaling of single-query latency of the approximate algorithms as the number of terms scales from 1 to 12. The number of workers in each test is equal to the number of terms (for Sparta, pRA, and pNRA, this allows maximal parallelism).

Figure 3(a) and Figure 3(b) present, respectively, the mean and 95% latencies of queries on ClueWeb. The latter captures the so-called "tail latency" of the slowest 5% of the queries; Figure 3(c)

	Sparta-high	pRA-high	pNRA-high	sNRA-high	pBMW-high	pBMW-low
ClueWeb	97.5%	98.5%	98.5%	99%	97.5%	80%
ClueWebX10	99%	99%	99%	99%	97%	79.9%

Table 3 Recall of approximate algorithms for 12-term queries.

depicts the mean latency for ClueWebX10. Sparta outperforms its competitors on all query sizes. The margin is especially big for verbose queries (5+ terms). Sparta's average query latency scales perfectly with the dataset size, remaining below 180 ms for all query lengths for both ClueWeb and ClueWebX10. Its 95% latency is below 500 ms.



(a) Average, ClueWeb **(b)** 95%, ClueWeb **(c)** Average, ClueWebX10

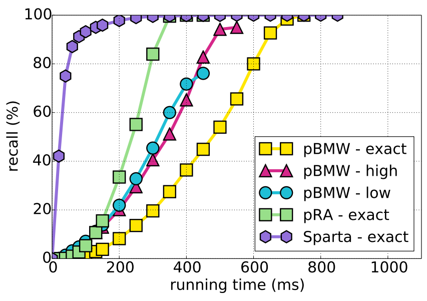
Figure 3 Top-k ($k = 1000$) query latency scaling with the number of query terms. The intra-query parallelism is equal to the number of terms in all algorithms.

In contrast, pRA exhibits much weaker scalability. For example, for 12-term queries it is slower than Sparta by more than 2x when applied to ClueWeb, and by more than 10x on ClueWebX10. We explain this by the cost of evaluating the complete document scores in pRA, which forces intensive access to the secondary index. This translates to volumes of random I/O that cannot be sustained even with modern SSD hardware. Note that this trend is the reverse of the one observed for pRA-exact, which outperforms Sparta-exact (Table 2). That is, Sparta spends much more work than pRA in order to collect the remaining 2.5% of the exact result set. We revisit this phenomenon below.

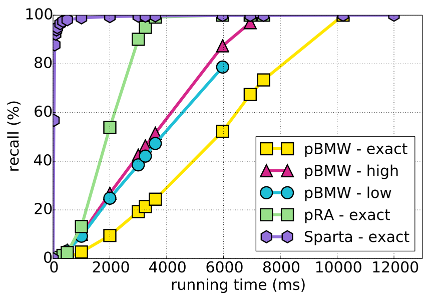
We see that pBMW, the best-in-class algorithm in the literature, fails to match Sparta's speed. For example, for 12-term queries, pBMW-high completes within 630 ms on average on ClueWeb, and within as long as 9.9 seconds on ClueWebX10. pBMW-low, which sacrifices 20% of the recall for performance, only succeeds to improve these latencies by 10% to 15%.

On ClueWeb, the shared-nothing and unoptimized parallelization of NRA are weaker than all other alternatives: pNRA's average latency for 12-term queries is 1 second, and sNRA's is 1.7 seconds. On the larger dataset, pNRA and sNRA are still poorer than Sparta, but perform better than pBMW. This is thanks to the high scalability and early-stopping nature of the approximate NRA approach. These results emphasize the necessity of sharing information among threads (unlike sNRA) on the one hand, and the importance of Sparta's locality optimizations, (which are missing in pNRA), on the other. Specifically, the background cleaning and local copies of *docMap* and the lazy updates of *UB* allow Sparta to benefit from local access to data that resides in hardware caches. We omit pNRA and sNRA from further discussion.

Recall dynamics. In order to understand how the top-k results get accrued by the different algorithms, we zoom in on the dynamics of query recall over the running time. We focus on 12-term queries in a 12-worker configuration. Figure 4(a) and Figure 4(b) present the results for the ClueWeb and ClueWebX10 datasets, respectively. Because the approximate versions of Sparta and pRA are identical to the respective exact versions until they stop, we show the dynamics of the exact versions only. The same is not true for pBMW, where f impacts the algorithm's results from the outset.



(a) ClueWeb



(b) ClueWebX10

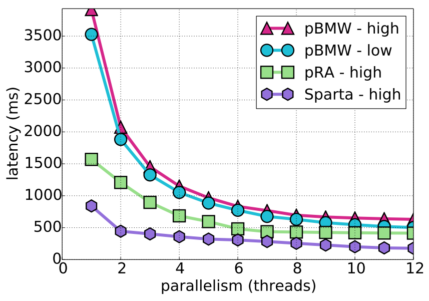
Figure 4 Recall dynamics with elapsed time, for 12-term queries, with 12 worker threads.

Hence, we show the dynamics of all three instances of pBMW. In the exact algorithms's curves, the rightmost data point corresponds to the exact algorithm's completion time.

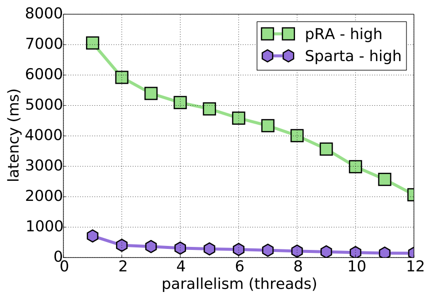
We see that Sparta's recall growth is the fastest. For instance, it surpasses 80% recall in less than 50 ms, and 90% recall in less than 100. But over time, its returns diminish, and most of the work becomes unproductive. Whereas pRA takes much longer to converge because it needs to fully score each encountered document, its concluding phase is faster because most relevant documents have complete scores. pBMW scans the postings in the order of document ids, which is unrelated to document scores, and hence accumulates the true hits at a near-linear rate. Obviously, the convergence rate of pBMW-high and pBMW-low is faster than that of pBMW-exact. For both datasets, the first two accrue results at similar rates until pBMW-low stops at approximately 80% recall.

Parallelism. We next consider 12-term queries with a number of threads varying from 1 to 12. The results appear in Figure 5(a) (for ClueWeb) and Figure 5(b) (for ClueWebX10). The latter depicts only Sparta and pRA because none of the pBMW instances scales close to real-time performance. The left-most data point in each curve refers to the performance of the respective sequential algorithm.

Sparta requires some level of parallelism in order to achieve real-time speed, e.g., its sequential latency on ClueWeb is 840 ms, which is above typical SLA requirements. Most of the gain is achieved at low-parallelism levels (2 threads suffice). On the other hand, for pBMW, much higher parallelism is essential – its latency is inversely proportional to the number of threads. Thus, Sparta is not only faster than pBMW, but also requires less resources, which benefits throughput as we next show.



(a) ClueWeb



(b) ClueWebX10

Figure 5 Top-k query latency scaling with intra-query parallelism, for 12-term queries.

Throughput. Finally, we compare the throughput (in queries per second) provided by the

	Sparta-high	pRA-high	pBMW-high	pBMW-low
ClueWeb	12.5	10.9	5.95	6.64
ClueWebX10	9.6	1.8	0.38	0.42

Table 4 Average throughput (in queries per second) of the approximate algorithms on a query distribution measured for voice queries in production.

different algorithms. To this end, we generate a workload with the query size distribution reported in [18], where the average query length is 4.2 (std: 2.96), and more than 5% of the queries are 10 terms or longer. The queries are generated as follows: we first sample a query length ℓ from the distribution in [18], and then select a query uniformly at random among all the length-ℓ queries in the complete set of 1200 AOL queries.

Table 4 depicts the results of running this query mix on a shared worker pool of 12 threads. Here too, Sparta improves over its competitors by a wide margin, especially on ClueWebX10, where its throughput is 25x that of pBMW-high. This advantage is thanks to a combination of Sparta's speed and lower resource utilization.

6 Related Work

Verbose queries challenge standard top-k processing techniques in terms of runtime latency. Huston and Croft [19] evaluated several sequential query processing techniques for verbose queries, concluding that the most effective one is to simply reduce the length of the query by omitting stop-words or "stop-structure" expressions. In this work we ignore the query pre-processing phase and consider the query as a bag of words given after textual analysis.

Crane et al. [13] showed that algorithms that traverse documents in order of id are susceptible to tail queries that may take orders of magnitude longer than the median query; approximate query evaluation in WAND and BMW does not significantly reduce the variance. Moreover, in agreement with our findings, they showed that algorithms that access posting lists in decreasing score order are less sensitive to tail queries due to their effective early termination capability.

Some previous works, e.g., [29, 23], studied parallel computation of conjunctive queries via posting list intersection. Note that this problem is different from (and easier than) the problem considered in this paper, where a top-scored document does not necessarily include all query terms.

Other works [9, 26] have parallelized state-of-the-art sequential algorithms like WAND and BMW by sharding the document space, computing the top-k in each shard independently, and finally merging the results. Implementations differ in whether threads share a common heap and threshold Θ or not. A global threshold is tighter than the threads' local thresholds, hence less work is done by each of the threads as more documents can be safely skipped. On the other hand, additional overhead is induced by the synchronization (e.g., using locks) needed to guarantee exclusive updates of the shared heap. Experimental results [26] have shown the superiority of the local-heap approach. The pBMW implementation used in our experiments follows this approach, but periodically shares the Θ values among the threads for improved performance.

Jeon et al. [21] presented an adaptive resource management algorithm that chooses the degree of parallelism at runtime for each query, based on predicting high-latency queries. Such efforts are orthogonal to the performance improvement we achieve via parallelization of (long) queries. Other works [5, 23] have explored using GPU hardware for information retrieval; [23] focused on adaptively choosing whether to use a CPU or a GPU based on the query's difficulty, and [5] focused on optimizing throughput rather than latency. In contrast, our work leverages standard server-grade multi-threaded CPUs.

The Threshold Algorithm and its variants [15, 16, 4] have been extensively studied by the database community, and have been applied in many relational database systems (for a comprehensive survey see [20]). Mamoulis et al. [24] observed two main phases during NRA processing – the "growing phase", where the candidate list grows, and the "shrinking phase" where no new documents can end up in the top-k results, after the first stopping condition is met. They used different data structures for the two phases in order to minimize the number of accesses and the memory requirements. Gursky et al. [17] also noticed the bottleneck in NRA computation derived from NRA's needs to maintain an extremely large number of partially scored candidates. They proposed several optimization methods for candidate list maintenance to speed up the search. One of their suggested approaches is to periodically remove irrelevant candidates from the candidate list, which we also do in Sparta.

Yuan et al. [34] observed that the number of accesses to the sorted lists by NRA could be further reduced by selectively performing the sorted accesses to the different lists (instead of in parallel). They proposed a selection policy that prioritizes the accesses to the sorted lists and cuts down unnecessary accesses. They showed significant cutoff in the number of accesses with respect to the original NRA. However, as the authors pointed out, the effectiveness of this approach in terms of run-time latency still has to be explored.

In the IR setting, TA has received much less attention. A few exceptional examples are [31, 8, 30], which experimented with TA on web data using standard IR metrics. Bast et al. [8] optimized the TA scheduling method based on a cost model for sequential and random accesses. Theobald et al. [30] extended TA for XML query languages. Another work by Theobald et al. [31] introduced an approximate TA algorithm based on probabilistic arguments: When scanning the posting lists in descending order of local scores, various forms of derived bounds are employed to predict when it is safe, with high probability, to skip candidate items hence trading off accuracy for sorted access. Applying similar probabilistic pruning rules for Sparta may prove beneficial and is left for future work.

7 Conclusions

We presented Sparta – the first practical algorithm to provide approximate top-k retrieval of verbose queries within interactive latency bounds. Sparta leverages the efficiency and early-stopping properties of the Threshold Algorithm; it forgoes the need for random access and duplicate indices by using the "lazy" scoring approach of TA's NRA variant. We parallelized the algorithm on shared-memory multi-core hardware while optimizing memory footprints, memory access patterns, inter-thread data sharing, and synchronization.

Unlike previously suggested algorithms, Sparta yields sub-180 ms average latencies on standard hardware for queries of up to 12 terms when applied to datasets of up to 500M documents, and can therefore support modern search experiences – which induce long queries – within real-time SLA requirements. Sparta also produces a highly accurate approximation of the exact results (a recall of above 97.5%). For comparison, a state-of-the-art parallel algorithm (pBMW) providing similar accuracy required 640 ms on a 50M-document dataset, and 9.9 seconds on a 500M-document corpus in our experiments.

References

1 https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html.

2 https://lucene.apache.org.

3 http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection.

4 Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *Proceedings of VLDB*, pages 495–506. VLDB Endowment, 2007. URL: http://dl.acm.org/citation.cfm?id=1325851.1325909.

5 Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proc. VLDB Endow.*, 4(8):470–481, May 2011. URL: http://dx.doi.org/10.14778/2002974.2002975, doi:10.14778/2002974.2002975.

6 Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of SIGIR*, pages 103–112. ACM, 2014.

7 Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

8 Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of VLDB*, pages 475–486. VLDB Endowment, 2006.

9 Carolina Bonacic, Carlos García, Mauricio Marin, Manuel Prieto-Matias, and Francisco Tirado. Building efficient multi-threaded search nodes. In *Proceedings of CIKM*, pages 1249–1258. ACM, 2010.

10 Edward Bortnikov, David Carmel, and Guy Golan-Gueta. Top-k query processing with conditional skips. In *Proceedings of WWW Companion*, pages 653–661, 2017.

11 Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of CIKM*, pages 426–434. ACM, 2003.

12 Jamie Callan, Mark Hoy, Changkuk Yoo, and Le Zhao. Clueweb09 data set, 2009.

13 Matt Crane, J. Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. A comparison of document-at-a-time and score-at-a-time query evaluation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 201–210, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3018661.3018726, doi:10.1145/3018661.3018726.

14 Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of SIGIR*, pages 993–1002. ACM, 2011.

15 Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of PODS*, pages 102–113, New York, NY, USA, 2001. ACM. URL: http://doi.acm.org/10.1145/375551.375567, doi:10.1145/375551.375567.

16 Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

17 Peter Gurský and Peter Vojtáš. Speeding up the nra algorithm. In *Proceedings of the 2Nd International Conference on Scalable Uncertainty Management*, SUM '08, pages 243–255. Springer-Verlag, 2008. URL: http://dx.doi.org/10.1007/978-3-540-87993-0_20, doi:10.1007/978-3-540-87993-0_20.

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

19 Samuel Huston and W. Bruce Croft. Evaluating verbose query processing techniques. In *Proceedings of SIGIR '10*, pages 291–298. ACM, 2010. URL: http://doi.acm.org/10.1145/1835449.1835499, doi:10.1145/1835449.1835499.

20 Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

21 Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of SIGIR*, pages 253–262. ACM, 2014. URL: `http://doi.acm.org/10.1145/2600428.2609572`, `doi:10.1145/2600428.2609572`.

22 Jimmy Lin and Andrew Trotman. Anytime ranking for impact-ordered indexes. In *Proceedings ICTIR*, pages 301–304. ACM, 2015.

23 Yang Liu, Jianguo Wang, and Steven Swanson. Griffin: Uniting cpu and gpu in information retrieval systems for intra-query parallelism. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’18, pages 327–337, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3178487.3178512`, `doi:10.1145/3178487.3178512`.

24 Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W. Cheung. Efficient top-k aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3), August 2007. URL: `http://doi.acm.org/10.1145/1272743.1272749`, `doi:10.1145/1272743.1272749`.

25 Oscar Rojas, Veronica Gil-Costa, and Mauricio Marin. Distributing efficiently the block-max wand algorithm. *Procedia Computer Science*, 18:120–129, 2013.

26 Oscar Rojas, Veronica Gil-Costa, and Mauricio Marin. Efficient parallel block-max wand algorithm. In *European Conference on Parallel Processing*, pages 394–405. Springer, 2013.

27 Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. Searching the web: The public and their queries. *J. Am. Soc. Inf. Sci. Technol.*, 52(3):226–234, February 2001.

28 Trevor Strohman, Howard Turtle, and W. Bruce Croft. Optimization strategies for complex queries. In *Proceedings of SIGIR*, pages 219–225. ACM, 2005.

29 Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of SIGIR*, pages 963–972. ACM, 2011.

30 Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. Topx: Efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 17(1):81–115, January 2008. URL: `http://dx.doi.org/10.1007/s00778-007-0072-z`, `doi:10.1007/s00778-007-0072-z`.

31 Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of VLDB*, VLDB ’04, pages 648–659. VLDB Endowment, 2004. URL: `http://dl.acm.org/citation.cfm?id=1316689.1316746`.

32 Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, November 1995.

33 Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of SIGIR*, pages 105–114. ACM, 2011.

34 Jing Yuan, Guangzhong Sun, Tao Luo, Defu Lian, and Guoliang Chen. Efficient processing of top-k queries: selective nra algorithms. *Journal of Intelligent Information Systems*, 39(3):687–710, 2012.