Scalable Top-K Retrieval with Sparta

Gali Sheffi
galish@cs.technion.ac.il
Technion
Haifa, Israel

Dmitry Basin
dbasin@verizonmedia.com
Yahoo Research
Haifa, Israel

Edward Bortnikov
ebortnik@verizonmedia.com
Yahoo Research
Haifa, Israel

David Carmel
david.carmel@gmail.com
Amazon
Haifa, Israel

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

Abstract

Many big data processing applications rely on a *top-k retrieval* building block, which selects (or approximates) the k highest-scoring data items based on an aggregation of features. In web search, for instance, a document's score is the sum of its scores for all query terms. Top-k retrieval is often used to sift through massive data and identify a smaller subset of it for further analysis. Because it filters out the bulk of the data, it often constitutes the main performance bottleneck.

Beyond the rise in data sizes, today's data processing scenarios also increase the number of features contributing to the overall score. In web search, for example, verbose queries are becoming mainstream, while state-of-the-art algorithms fail to process long queries in real-time.

We present Sparta, a practical parallel algorithm that exploits multi-core hardware for fast (approximate) top-k retrieval. Thanks to lightweight coordination and judicious context sharing among threads, Sparta scales both in the number of features and in the searched index size. In our web search case study on 50M documents, Sparta processes 12-term queries more than twice as fast as the state-of-the-art. On a tenfold bigger index, Sparta processes queries at the same speed, whereas the average latency of existing algorithms soars to be an order-of-magnitude larger than Sparta's.

CCS Concepts • **Software and its engineering** **Multiprocessing / multiprogramming / multitasking**; • **Information systems** **Web search engines**; **Top-k retrieval in databases**; **Distributed retrieval**; • **Computing methodologies** **Parallel algorithms**; **Concurrent algorithms**; **Distributed algorithms**.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374522

Keywords parallel computing, multi-threading, performance, information retrieval, web search, top-k search

ACM Reference Format:
Gali Sheffi, Dmitry Basin, Edward Bortnikov, David Carmel, and Idit Keidar. 2020. Scalable Top-K Retrieval with Sparta. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3332466.3374522

1 Introduction

Interactive big data processing is proliferating with applications like information retrieval, web search, data mining, data analytics, and more [22]. Such services often need to identify relevant data based on multiple features, or query *terms*. For instance, a real-time analytics engine (e.g., [5]) might keep daily lists of application access statistics – the number of users accessing every application on a given day. A query may then retrieve the popular applications over a ten-day period by aggregating over ten lists. Real-time analytics databases offer a TopN search primitive to facilitate such queries [6].

In modern use cases, datasets are becoming larger and queries exceedingly involve more features. A case in point is web search: While early days web queries were short (2.4 terms on average [32]), modern search experiences such as query suggestion, reformulation, and conversational interfaces, stimulate users to submit much longer queries. E.g., more than 5% of voice search queries exceed 10 terms [20].

Interactive data processing usually involves two stages [38]. The first is *top-k retrieval*, roughly matching the top-k documents most relevant to the query (e.g., $k = 1000$) based on a simple multi-feature score. It is followed by a more elaborate analysis. The first stage sifts through huge volumes of data and therefore dominates the execution time.

Yet obtaining the exact top-k matches out of a large corpus is typically too slow to meet real-time latency requirements, especially as the number of searched features becomes large. Luckily, perfect results are usually not essential, as later processing stages can work with approximate results [6, 16, 25, 38]. Based on this observation, we focus on

approximate (sometimes called *non-safe*) query evaluation, tuned to achieve a certain high recall (e.g., 97% or more).

In this paper, we accelerate approximate top-k retrieval on multi-core hardware. We design and implement *Sparta* – Scalable PARallel Threshold Algorithm. Sparta's design is inspired by the seminal *Threshold Algorithm (TA)* by Fagin et al. [18], which retrieves the top-k objects from a database based on an aggregation of features that may reside in multiple nodes. Transforming TA into an efficient concurrent algorithm is challenging because coordination around shared state can become a major bottleneck. On the one hand, sharing state among threads is essential in order to benefit from TA's early stopping feature. Indeed, we show that a shared-nothing (partitioned) parallelization performs two times worse than even a single-threaded implementation. On the other hand, a naïve attempt to parallelize TA using shared memory also results in even worse performance than the sequential algorithm. Sparta instead judiciously shares pertinent information among threads, thus keeping the synchronization overhead and memory overlap low.

We evaluate Sparta via a web search case study, comparing Sparta to various TA variants and state-of-the-art web search algorithms. Our results show that Sparta scales well with both corpus size and query length. E.g., on the 50M-document TREC ClueWeb09B dataset [15], Sparta can serve 12-term queries within less than 200 ms, whereas today's best algorithms take at least twice as long. On a 500M-document index, Sparta's average latency is virtually unchanged, whereas the next best algorithm (one of TA variants) takes over a second. Sparta's throughput on a production-grade query mix (with the query length distribution from [20]) is 20% higher than that achieved by the best previous algorithm on the small corpus, and 5x higher on the large one.

2 Problem Statement: Top-k Retrieval

We focus on the fundamental primitive of top-k retrieval, widely used in information retrieval, web search, data mining, data analytics, and more [22]. This primitive is commonly used for the initial selection of documents over which a more refined search is performed [38].

Given a query q, the primitive retrieves the top k scored *documents* (data items) in a given corpus according to a scoring function $score(D, q)$. A *query* is given as a list of terms (features). Given an m-term query $q = t_1, \ldots t_m$ 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 the k documents with the highest scores for the query. An *approximate* solution returns k results that approximate the top k. Let L be the exact list of top-k documents, and let A be an approximate result. The quality (or accuracy) of A is measured by its *recall*, which is the fraction of L included in A.

3 Background

We provide background on state-of-the-art top-k algorithms used in web search and on Fagin et al.'s TA [18].

3.1 State-of-the-art Web Search Algorithms

Search algorithms use a preprocessed inverted index of the corpus. The index is organized according to terms and holds a *posting list* of all documents associated with each term. Top-k retrieval algorithms typically traverse posting lists sequentially. They track the top-k documents among those scored so far 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.

For big datasets, only a small portion of the index can reside in RAM at a given time. The I/O overhead is kept low by fetching contiguous chunks of the lists from disk. Additionally, algorithms use various heuristics to reduce the number of documents whose score needs to be evaluated.

Popular production top-k algorithms, e.g., *MaxScore* [33, 37], *WAND* [14], and *Block-Max WAND (BMW)* [17], simultaneously scan all relevant posting lists in order of document id, fully scoring each document before moving to the next one. We refer to these as *document-order* algorithms.

The disadvantage of document-order algorithms is that high-scoring documents may be discovered late in the traversal because document ids are not correlated with query relevance, and so the algorithm does not always produce useful partial results before it completes. This is mitigated by *score-order* algorithms (sometimes called *impact-order*), e.g., JASS [25], which traverse posting lists in decreasing order of term score. These algorithms accumulate the score of each document throughout the traversal, and thus document scores may be inaccurate at first and improve over time. Score-order algorithms have been shown to be slower but have more predictable performance than document-based ones [16].

3.2 The Threshold Algorithm

TA [18] 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.

TA is a score-order algorithm. To evaluate a query $q = t_1, \ldots t_m$, it traverses the posting lists of the m 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 traversal with $m = 3$. The scores of the last traversed items in each list are $UB[1] = 38$, $UB[2] = 32$, and $UB[3] = 41$.

TA also maintains a threshold Θ – the score of the k-th document in the heap. It stops when no candidate's score can

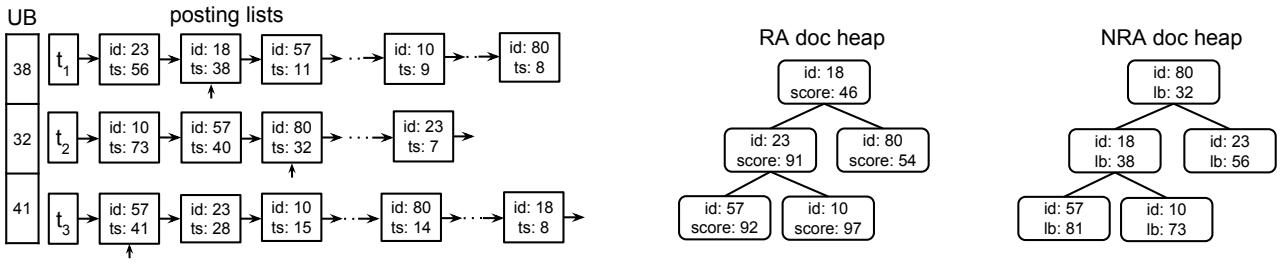


Figure 1. Traversal example in RA and NRA variants of the Threshold Algorithm (TA). Posting lists are sorted by decreasing term scores. Vertical arrows depict iterator positions. UB holds upper bounds on the terms' contributions to scores of untraversed documents. The RA document heap is ordered by full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

exceed Θ. We implement approximate variants of TA by stopping whenever the heap does not change for some parameter Δ ms. Either way, TA's output is the set of documents in the heap. TA has two flavors, which we now describe.

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 and 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:

$$UBStop \triangleq \sum_{i=1}^{m} UB[i] \leq \Theta. \quad (1)$$

At this point, no non-traversed document can achieve a high enough score to be included in the heap.

Note that being a score-order algorithm, TA is amenable to such early stopping: high scoring documents are likely to be discovered early. This is in contrast with document-order algorithms, in which finding high scoring documents remains equally likely throughout the execution.

Unfortunately, random access is costly, in particular for large datasets 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. An additional drawback is RA is the need to maintain a secondary index by document id in addition to the one sorted by term score, which doubles its footprint.

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:

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

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.

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.

4 Sparta

Sparta Like our aforementioned implementations of TA, it can be safe or can be configured to provide approximate results by stopping after the heap does not change for some Δ time.

4.1 Sparta Data Structures

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

Table 1. Sparta's data structures and initial values.

Table 1 defines the data structures used by Sparta and Figure 2 illustrates them. As in NRA, the algorithm maintains the

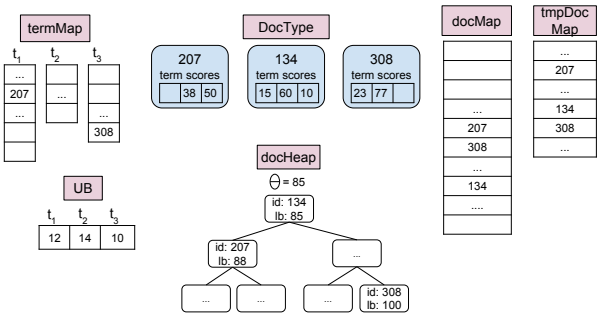


Figure 2. Sparta data structures. The *docMap* keeps track of partially scored documents; *tmpDocMap* and *termMap* are local partial copies of *docMap*.

current top-k results in a heap, *docHeap*, and its lowest value in Θ. It keeps 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 *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 according to Equation 1 and the second is checked as follows:

$$\forall D \in docMap \setminus docHeap : UB(D) \leq \Theta. \quad (2)$$

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.

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

4.2 Splitting the Work

When multiple threads evaluate documents in parallel, there is high contention around *docMap*. To reduce it, 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 [29]). On the other hand, it is possible to shrink *docMap* by removing documents whose upper bounds have been exceeded by Θ. We exploit this and stop sharing *docMap* among the threads once it is sufficiently small, thereby eliminating the synchronization overhead altogether.

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 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.

We allocate an additional task to the CLEANER function (lines 39– 48). This task is invoked once Equation 1 holds and so *docMap* no longer grows. It 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 [19]). Second, it determines when the algorithm can stop (line 46). It 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. In addition, it checks whether the heap has not changed for Δ time (the exact version is obtained by setting $\Delta = \infty$). 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. In order to reduce cache misses, instead of updating UB after each document evaluation, the workers update it at the end of a segment traversal (line 24). Note that delaying updates does not affect correctness, it only slows down convergence.

Updates of *docHeap* and Θ are protected by a shared lock (lines 27 and 38), 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 30-32).

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

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 ∧ |docMap| < Φ ∧ termMap[i] = docMap then
        ▷ docMap is shrinking and small – create a local copy for term i
10:         termMap[i] ← new hash map
11:         for all D ∈ docMap s.t. D.score[i] = 0 do
12:             add D to termMap[i]
13:     for j = 1 to segSize do
14:         if done then return
15:         ⟨id, score⟩ ← next entry in ith posting list
16:         D ← termMap[i](id)
17:         if D = ⊥ then                               ▷ document missing
18:             if ¬UBStop then                         ▷ hash incomplete
19:                 create new document object D
20:                 add D to termMap[i](id)
21:             else continue
22:         D.score[i] ← score                          ▷ update term score
23:         if Σ_{j=1}^{m} D.score[j] > Θ then UPDATE_HEAP(D)
24:     UB[i] ← score                                   ▷ update term's upper bound
25:     add PROCESSTERM(i) to job queue

26: procedure UPDATE_HEAP(D)
27:     lock docHeap
28:     if D ∉docHeap then
29:         insert D to docHeap
30:         for all d ∈ docHeap do
31:             d.LB ← Σ_{j=1}^{m} d.score[j]
32:             move d to correct place in heap
33:         if |docHeap| > k then
34:             remove lowest scored doc
35:         if |docHeap| = k then                       ▷ set Θ to k^th lowest score
36:             Θ ← lowest score in docHeap
37:         HeapUpdTime ← current time
38:     unlock docHeap

39: procedure CLEANER
40:     if |docMap| > Φ then                            ▷ shrink docMap
41:         tmpDocMap ← new hash map
42:         for every doc D ∈ docMap do
43:             if UB(D) > Θ ∨ D ∈ docHeap then
44:                 add D to tmpDocMap                  ▷ D is still relevant
45:         replace docMap by tmpDocMap
            ▷ check stopping conditions; in exact version Δ = ∞
46:     if |docMap| = |docHeap| ∨ HeapUpdTime+Δ < now then
47:         done ← true
48:     else add CLEANER() to job queue
```

The cleaner task starts removing elements from *docMap* after it is guaranteed that no new entries are added to it. This reduces the memory footprint and improves performance. Nevertheless, constantly updating *docMap* would lead to frequent cache invalidations at the tasks that read the map. To avoid this, 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*, retaining *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). 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 workers. 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, Φ = 10K entries. In lines 11-12, we scan *docMap* and copy 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 at most one 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 (where $\Delta = \infty$) is safe and returns the top-k results.

In terms of performance, NRA was shown to be *instance-optimal* when random access is impossible [18], namely, its number of accesses to posting list entries is asymptotically close the optimum for every problem instance. This property holds for NRA as long as the rates in which different threads access different posting lists are within constant multiples of each other [18], 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. Sparta differs from NRA 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 also asymptotically instance-optimal under the same assumptions as NRA.

5 Case Study: Web Search

We conduct a case study of top-k query processing on a web dataset. We compare the performance of Sparta to the following multiprocessor algorithms:

- **pBMW** [31] – parallel BMW, the best-in-class parallelization of a document-order top-k algorithm;
- **pJASS** [28] – a recent parallelization of the JASS [25] score-order retrieval algorithm;
- **pRA** – a parallel implementation of RA;
- **sNRA** – a shared-nothing parallelization of NRA; and
- **pNRA** – a naïve shared-state parallel implementation of NRA.

In order to crystallize the comparison among the core algorithms, we abstract away other contributors to the wall-clock latency, e.g., index compression. As recent studies show, given state-of-the-art compression techniques, the impact of decompression on end-to-end performance is marginal (e.g., up to 6% with QMX-D4 compression [26]).

We focus on disk-resident search indexes. We also experimented with RAM-resident indexes, and in all cases, all algorithms except pRA got similar results, which is not surprising given that the algorithms traverse posting lists sequentially. These results are omitted for lack of space.

5.1 Methodology

We study 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 entire 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 document order or in score order) is pre-built offline and stored on disk uncompressed as a collection of binary files, each storing a shard of data partitioned by term. The benchmark environment memory-maps the contents of these files via the MappedByteBuffer API [2]. 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 dataset, Category B (ClueWeb09B), which is widely used for information retrieval research [15]. 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 [3] for preprocessing the index to generate posting lists; this includes text tokenization, posting list maintenance, and term statistics retrieval. We score documents using a standard tf-idf score function with document length normalization [10].

We draw queries from the public AOL search log [4]. 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 $k = 1000$, based on the assumption that simple tf-idf retrieval is the first phase of multi-stage ranking, which may require large values of k for effectiveness [38]. (Indeed, Crane et al. [16] report that the classical AP and RBP quality metrics are close to optimal with this choice of k, for multiple datasets.) Experiments with $k = 100$ produced qualitatively similar results, which we omit for lack of space.

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 (namely, tf-idf) are stored in the posting lists as integers, scaled by 10^6 and rounded as in [13]. Using integer arithmetic instead of floating-point significantly speeds up document evaluation.

The specific algorithm implementations are as follows.

5.2.1 State-of-the-art parallel search algorithms

pBMW Our implementation of pBMW closely follows the description in [30]. The algorithm partitions the execution of the sequential BMW [17] 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 executions 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 [30].) 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 due 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 [17]. 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 relax the threshold for score upper bounds [14]. For $f = 1$, the algorithm is exact.

pJASS Our implementation of pJASS follows the description in [28]. It traverses all posting lists in parallel, in score order, and accumulates the encountered scores per-document in *docMap*. Each document is protected by a lock, and a thread that encounters a document locks it, adds the partial score from the term it traversed, and then unlocks it. The algorithm stops after scanning a predefined fraction, p, of postings. In the exact version, $p = 1$.

5.2.2 Parallel TA variants

sNRA 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 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 it updates the term upper bounds upon every document evaluation. As in Sparta, a dedicated task checks the stopping condition. (Distributed stopping detection yielded worse results).

pRA Our implementation of pRA maintains its results in a shared heap (experiments did not show any benefit 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 allows only the first to take effect.

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

	Sparta	pNRA	sNRA	pRA	pBMW	pJASS
CW	860	13,291	5,553	480	750	54,343
CWX10	12,010	N/A	56,223	7,410	10,210	N/A

Table 2. Average query latency (in ms) of 12-term queries with exact algorithms using 12 threads. N/A indicates the algorithm crashed due to lack of memory. None of the algorithms meets real-time SLAs.

5.3 Results

Although our focus is on approximate top-k, we experiment first with the exact variants of the algorithms. For an algorithm A, we denote its exact variant A-exact. We then consider approximate variants with high and low recall, denoted A-high and A-low, respectively. Note that the approximate algorithms' parameters (Δ, f, and p) affect the recall but do not control it directly; our high recall instances are ones that empirically achieve a recall of 96% or higher.

5.3.1 Exact Algorithms

Our first experiment looks into the feasibility of using exact algorithms. Usability studies (e.g., [9]) 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. Therefore, a typical SLA (service-level agreement) for a top-k service requires queries to be served within 250 ms on average.

Our experiment shows that none of the exact algorithms meets a real-time SLA for long 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). While on ClueWeb some algorithms complete within less than a second, on ClueWebX10, some algorithms crash, and others take between 7 seconds and nearly a minute, which is clearly unacceptable.

We 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 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. We instantiate pBMW with $f = 5$ for high recall and $f = 10$ for low recall. Finally, pJASS is instantiated with $p = 0.005$ for low recall

and $p = 0.02$ for high recall (using $p = 0.1$, as suggested, e.g., in [16], produced unacceptably high latencies). The recall achieved with these parameters for 12-term queries is summarized in Table 3.

Latency. Figures 3a– 3d depict the scaling of single-query latency with query length. The number of workers in each test is equal to the number of terms (for Sparta, pRA, and pNRA, this allows maximal parallelism). Figures 3a and 3b present, respectively, the mean and 95% latencies of queries on ClueWeb in the high-recall algorithms. The latter captures the so-called "tail latency" of the slowest 5% of the queries. Sparta outperforms all other algorithms in terms of both average latency and the 95th percentile, on all query lengths. The margin is noticeable for long queries.

Perhaps surprisingly, although we saw (in Table 2) that pRA-exact outperforms Sparta-exact, the trend is reversed in the approximate variants: pRA-high's latency for 12-term queries it is more than 2x slower than Sparta-high's. This means that 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. While Sparta collects the approximate results quickly, the performance of pRA suffers due to intensive access to the secondary index that cannot be sustained even with modern SSD hardware.

Figure 3c depicts the mean latency for ClueWebX10 in all algorithms. Sparta's average latency scales perfectly, remaining below 180 ms for all query lengths for both ClueWeb and ClueWebX10. In other words, Sparta's result set solidifies after processing a similar number of postings, even when the overall index size grows 10x. Its 95% latency is below 500 ms. None of the other algorithms scales as well to the big dataset.

Figures 3d and 3e compare Sparta-high against the low-recall variants of state-of-the-art web algorithms. By sacrificing recall, pBMW and pJASS do improve performance and bring their tail latencies for short queries below Sparta's. Note that this is in line with the use case for which they were developed – predictable performance on short queries. Yet neither algorithm is able to meet Sparta's average latency for any query length or its tail latency for long queries. Moreover, neither algorithm fairs well on the large data set. For example, pBMW-high processes 12-term queries 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 improves this latency by 15%.

On ClueWeb, the shared-nothing and unoptimized parallelizations 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. The results are presented in Figures 3f and 3g for the ClueWeb and ClueWebX10 datasets, respectively. Because the approximate versions of Sparta, pRA, and pJASS 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. Hence, we plot all three instances of pBMW.

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. The first two accrue results at similar rates until pBMW-low stops at approximately 80% recall. pJASS's behavior is similar to Sparta's, but it is a bit slower and fails to reach 100% recall within a minute of execution.

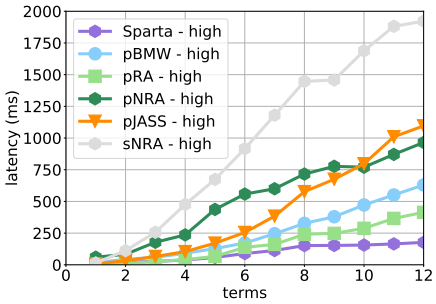
Parallelism. We next study latency scaling with intra-query parallelism. We consider 12-term queries with a number of threads varying from 1 to 12. The average latencies appear in Figure 3h (for ClueWeb) and Figure 3i (for ClueWebX10). The left-most data point in each curve corresponds to the performance of the respective sequential algorithm.

Sparta requires some level of parallelism in order to achieve real-time speed – its sequential latency is 840 ms, which is above typical SLA requirements. Most of the gain is achieved at low-parallelism levels (2 threads suffice). The same is true for pJASS, which hardly improves when afforded more threads, due to unequal thread workloads. 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 fewer resources, which benefits throughput as we next show.

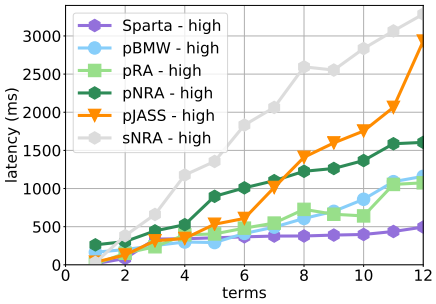
Throughput. Finally, we compare the throughput (in queries per second) provided by the different algorithms. First, we evaluate throughput on fixed query lengths. Figure 4 shows the throughput achieved for each query length. Next, we generate a workload with the query size distribution reported

	Sparta-high	pRA-high	pNRA-high	sNRA-high	pBMW-high	pBMW-low	pJASS-high	pJASS-low
ClueWeb	97.5%	98.5%	98.5%	99%	97.5%	80%	96%	93%
ClueWebX10	99%	99%	99%	99%	97%	79.9%	N/A	99%

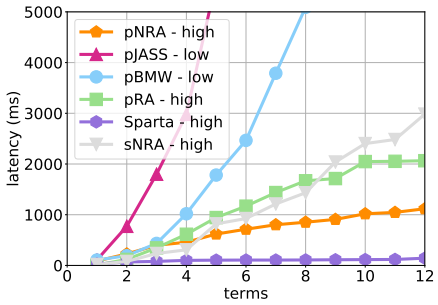
Table 3. Recall of approximate algorithms for 12-term queries.



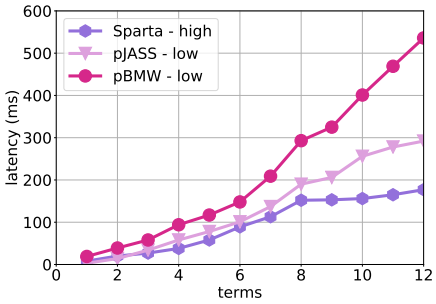
(a) Average latency, ClueWeb, high recall algorithms



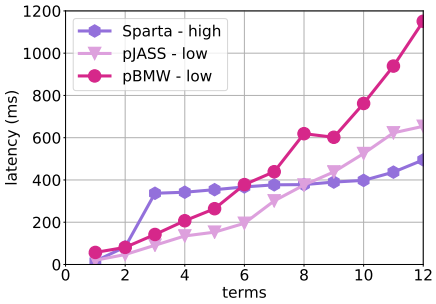
(b) 95% latency, ClueWeb, high recall algorithms



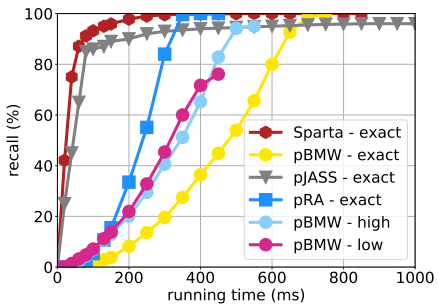
(c) Average latency, ClueWebX10



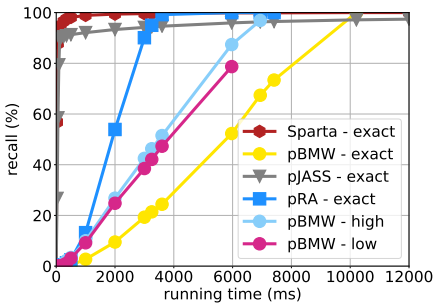
(d) Average latency, ClueWeb, Sparta vs. low recall algorithms



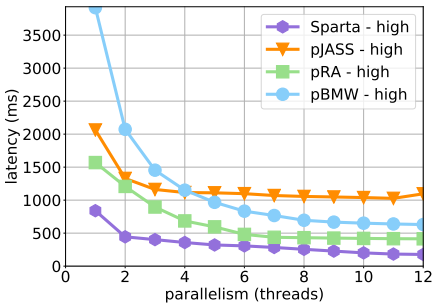
(e) 95%, ClueWeb – Sparta vs. low recall algorithms



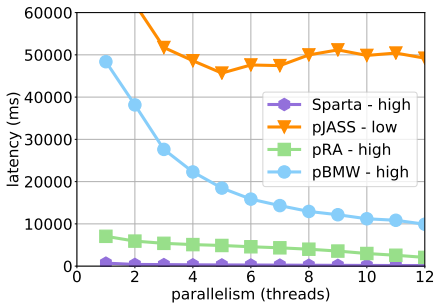
(f) Recall dynamics, ClueWeb



(g) Recall dynamics, ClueWebX10



(h) Average latency, ClueWeb, 12 terms



(i) Average latency, ClueWebX10, 12 terms

Figure 3. Top-k ($k = 1000$) query latency. Plots (a)–(e) show scaling with the number of query terms; plots (f)–(g) show recall dynamics with elapsed time for 12-term queries; plots (h)–(i) show scaling with intra-query parallelism.

in [20], where the average query length is 4.2 with a standard deviation of 2.96. More than 5% of the queries have 10 or more terms. The queries are generated as follows: we first sample a query length ℓ from the distribution in [20], 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, especially on the large dataset, where its throughput is 25x that of pBMW-high. Its pronounced advantage over pBMW 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 [21]

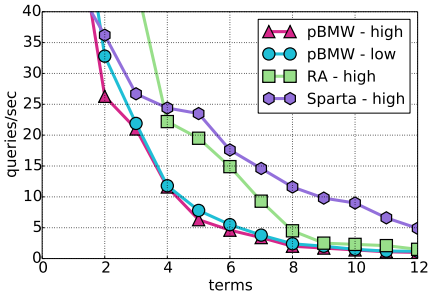


Figure 4. Top-k query throughput scaling with the number of query terms, ClueWeb. The intra-query parallelism is equal to the number of terms in all parallel algorithms.

	Sparta	pRA	pBMW	pJASS
ClueWeb	12.5	10.9	5.95	10.8
ClueWebX10	9.6	1.8	0.38	N/A

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

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. [16] showed that document-order algorithms 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, they showed that score-order algorithms are less sensitive to tail queries due to their effective early termination capability.

Mackenzie et al. [28] suggest pJASS– a parallel version of the score-order algorithm JASS [25]. JASS's virtue is its simplicity – it performs very little processing per-posting. In order to achieve early termination, JASS applies a heuristic to limit the fraction of processed terms. Similarly to Sparta, pJASS focuses on approximate top-k retrieval (its exact variant is inefficient). We show that Sparta achieves better performance and better recall thanks to multiple optimizations – e.g., continuous result-set pruning that leads to better locality, fine-tuned termination, and careful synchronization. Sparta's additional advantage is its small RAM footprint, whereas pJASS intentionally avoids pruning and maintains a huge in-memory document map throughout the query evaluation.

Some previous works, e.g., [27, 34], 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 [12, 31] 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 [31] 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. [24] 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 [8, 27] have explored using GPU hardware for information retrieval; [27] focused on adaptively choosing whether to use a CPU or a GPU based on the query's difficulty, and [8] focused on optimizing throughput rather than latency. In contrast, our work leverages standard server-grade multi-threaded CPUs.

The Threshold Algorithm and its variants [7, 18] have been extensively studied by the database community, and have been applied in many relational database systems (for a comprehensive survey see [23]). Mamoulis et al. [29] 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. [19] 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. [39] 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, a few works [11, 35, 36] have experimented with TA on web data using standard IR metrics. Bast et al. [11] optimized the TA scheduling method based on a cost model for sequential and random accesses. Theobald et al. [35] extended TA for XML query languages. Another work by Theobald et al. [36] 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, a practical algorithm for approximate top-k retrieval on multi-core hardware. Sparta can support modern analytics and search experiences, which induce long queries, within real-time requirements. To our knowledge, Sparta is the first algorithm capable of serving long queries (10 or more terms) on server-grade hardware within interactive latency bounds.

Sparta leverages the efficiency and early-stopping properties of the seminal Threshold Algorithm. It forgoes the need for random access and duplicate indexes by relying on TA's NRA variant. It achieves high performance by optimizing memory footprints, memory access patterns, inter-thread data sharing, and synchronization.

Sparta scales perfectly with dataset size. In a web search use case, Sparta yields average latencies of 180 ms on standard hardware for queries of up to 12 terms, when applied to datasets of both 50M and 500M documents. It does so while producing a highly accurate approximation of the exact results (a recall of above 97.5%). For comparison, its state-of-the-art parallel competitors, pBMW and pJASS, require 640 ms and above 1 second, respectively, to provide similar accuracy on the 50M document dataset; on the larger dataset, pBMW's latency soars to almost 10 seconds, while pJASS crashes.

References

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

[2] [n. d.]. https://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html.

[3] [n. d.]. https://lucene.apache.org.

[4] [n. d.]. http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection.

[5] [n. d.]. Flurry. https://www.flurry.com/.

[6] [n. d.]. TopN queries. http://druid.io/docs/latest/querying/topnquery.html.

[7] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2007. Best Position Algorithms for Top-k Queries. In *Proceedings of VLDB*. VLDB Endowment, 495–506. http://dl.acm.org/citation.cfm?id=1325851.1325909

[8] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *Proc. VLDB Endow.* 4, 8 (May 2011), 470–481. https://doi.org/10.14778/2002974.2002975

[9] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. 2014. Impact of Response Latency on User Behavior in Web Search. In *Proceedings of SIGIR*. ACM, 103–112.

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

[11] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. 2006. IO-Top-k: Index-access Optimized Top-k Query Processing. In *Proceedings of VLDB*. VLDB Endowment, 475–486.

[12] Carolina Bonacic, Carlos García, Mauricio Marin, Manuel Prieto-Matias, and Francisco Tirado. 2010. Building Efficient Multi-threaded Search Nodes. In *Proceedings of CIKM*. ACM, 1249–1258.

[13] Edward Bortnikov, David Carmel, and Guy Golan-Gueta. 2017. Top-k Query Processing with Conditional Skips. In *Proceedings of WWW Companion*. 653–661.

[14] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. 2003. Efficient Query Evaluation Using a Two-level Retrieval Process. In *Proceedings of CIKM*. ACM, 426–434.

[15] Jamie Callan, Mark Hoy, Changkuk Yoo, and Le Zhao. 2009. Clueweb09 data set.

[16] Matt Crane, J. Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. 2017. 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)*. ACM, New York, NY, USA, 201–210. https://doi.org/10.1145/3018661.3018726

[17] Shuai Ding and Torsten Suel. 2011. Faster Top-k Document Retrieval Using Block-max Indexes. In *Proceedings of SIGIR*. ACM, 993–1002.

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

[19] Peter Gurský and Peter Vojtáš. 2008. Speeding Up the NRA Algorithm. In *Proceedings of the 2Nd International Conference on Scalable Uncertainty Management (SUM '08)*. Springer-Verlag, 243–255. https://doi.org/10.1007/978-3-540-87993-0_20

[20] Ido Guy. 2016. Searching by Talking: Analysis of Voice Queries on Mobile Web Search. In *Proceedings of SIGIR*. ACM, 35–44.

[21] Samuel Huston and W. Bruce Croft. 2010. Evaluating Verbose Query Processing Techniques. In *Proceedings of SIGIR '10*. ACM, 291–298. https://doi.org/10.1145/1835449.1835499

[22] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008), 11:1–11:58. https://doi.org/10.1145/1391729.1391730

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

[24] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive Parallelization: Taming Tail Latencies in Web Search. In *Proceedings of SIGIR*. ACM, 253–262. https://doi.org/10.1145/2600428.2609572

[25] Jimmy Lin and Andrew Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *Proceedings ICTIR*. ACM, 301–304.

[26] Jimmy Lin and Andrew Trotman. 2017. The Role of Index Compression in Score-at-a-time Query Evaluation. *Inf. Retr.* 20, 3 (June 2017), 199–220. https://doi.org/10.1007/s10791-016-9291-5

[27] Yang Liu, Jianguo Wang, and Steven Swanson. 2018. 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)*. ACM,

New York, NY, USA, 327–337. https://doi.org/10.1145/3178487.3178512

[28] Joel Mackenzie, Falk Scholer, and J. Shane Culpepper. 2017. Early Termination Heuristics for Score-at-a-Time Index Traversal. In *Proceedings of the 22Nd Australasian Document Computing Symposium (ADCS 2017)*. ACM, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3166072.3166073

[29] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W. Cheung. 2007. Efficient Top-k Aggregation of Ranked Inputs. *ACM Trans. Database Syst.* 32, 3, Article 19 (Aug. 2007). https://doi.org/10.1145/1272743.1272749

[30] Oscar Rojas, Veronica Gil-Costa, and Mauricio Marin. 2013. Distributing efficiently the Block-Max WAND algorithm. *Procedia Computer Science* 18 (2013), 120–129.

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

[32] Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. 2001. Searching the Web: The Public and Their Queries. *J. Am. Soc. Inf. Sci. Technol.* 52, 3 (Feb. 2001), 226–234.

[33] Trevor Strohman, Howard Turtle, and W. Bruce Croft. 2005. Optimization Strategies for Complex Queries. In *Proceedings of SIGIR*. ACM, 219–225.

[34] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. 2011. Posting List Intersection on Multicore Architectures. In *Proceedings of SIGIR*. ACM, 963–972.

[35] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. 2008. TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. *The VLDB Journal* 17, 1 (Jan. 2008), 81–115. https://doi.org/10.1007/s00778-007-0072-z

[36] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. 2004. Top-k Query Evaluation with Probabilistic Guarantees. In *Proceedings of VLDB (VLDB '04)*. VLDB Endowment, 648–659. http://dl.acm.org/citation.cfm?id=1316689.1316746

[37] Howard Turtle and James Flood. 1995. Query Evaluation: Strategies and Optimizations. *Inf. Process. Manage.* 31, 6 (Nov. 1995), 831–850.

[38] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A Cascade Ranking Model for Efficient Ranked Retrieval. In *Proceedings of SIGIR*. ACM, 105–114.

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