Evaluation Strategies for Top-k Queries over Memory-Resident Inverted Indexes

Marcus Fontoura[1]*, Vanja Josifovski[2], Jinhui Liu[2], Srihari Venkatesan[2], Xiangfei Zhu[2], Jason Zien[2]
1. Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043
2. Yahoo! Research, 701 First Ave., Sunnyvale, CA 94089
marcusf@google.com, {vanjaj, jliu, venkates, xiangfei, jasonyz}@yahoo-inc.com

ABSTRACT

Top-k retrieval over main-memory inverted indexes is at the core of many modern applications: from large scale web search and advertising platforms, to text extenders and content management systems. In these systems, queries are evaluated using two major families of algorithms: *document-at-a-time* (DAAT) and *term-at-a-time* (TAAT). DAAT and TAAT algorithms have been studied extensively in the research literature, but mostly in disk-based settings. In this paper, we present an analysis and comparison of several DAAT and TAAT algorithms used in Yahoo!'s production platform for online advertising. The low-latency requirements of online advertising systems mandate memory-resident indexes. We compare the performance of several query evaluation algorithms using two real-world ad selection datasets and query workloads. We show how some adaptations of the original algorithms for main memory setting have yielded significant performance improvement, reducing running time and cost of serving by 60% in some cases. In these results both the original and the adapted algorithms have been evaluated over memory-resident indexes, so the improvements are algorithmic and not due to the fact that the experiments used main memory indexes.

1. INTRODUCTION

Top-k retrieval is at the core of many applications today. The most familiar application is web search, where the web is crawled and searched by massive search engines that execute top-k queries over large distributed inverted indexes [10]. Lately, a few results have been reported on the use of top-k retrieval in ad selection for online advertising [5, 6, 12, 24] where the query is evaluated over a corpus of available ads. Top-k retrieval is also present in enterprise domains where top-k queries are evaluated over emails, patents, memos and documents retrieved from content management systems.

In top-k retrieval, given a query Q and a document corpus D, the system returns the k documents that have the highest score according to some scoring function $score(d, Q)$, $d \in D$. Scoring is usually performed based on overlapping query and document *terms,* which are the atomic units of the scoring process and represent individual words, phrases and any document and query meta-data. The document corpus can be viewed as a matrix where the rows represent individual documents and the columns represent terms. Each cell in this matrix is called a *payload* and encodes information about the term occurrence within the document that is used during scoring (e.g., the term frequency).

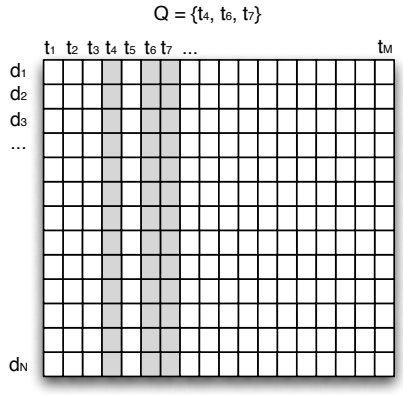

Figure 1: Document corpus as a matrix.

The document matrix is sparse, as most of the documents contain only a small subset of all the unique terms. Figure 1 shows the matrix representation of a document corpus with M unique terms and N documents. Given a query with a set of terms, finding the k documents with the highest score requires a search among the documents that contain at least one of the query terms. In Figure 1 this is illustrated by the shaded portion of the matrix.

There are two natural ways to search the documents that have non zero weights in the shaded portion of the matrix. The first way is to evaluate row by row, i.e., to process one document-at-a-time (DAAT). In this approach the score for each document is completely computed before advancing to a new row. The second way is to evaluate column by column, i.e., to process one term-at-a-time (TAAT). In this approach we must accumulate the score of multiple documents simultaneously and the contributions of each term to the score of each document are completely processed before moving to the next term. DAAT and TAAT strategies have been the cornerstone of the top-k evaluation in the last two

*Work done while the author was at Yahoo!.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

decades. Many versions of these algorithms have been proposed in the research literature [7, 8, 14, 19, 22, 23]. Several known systems in production today, from large scale search engines as Google and Yahoo!, to open source text indexing packages as Lucene [2] and Lemur [20], use some variation of these strategies.

While DAAT and TAAT algorithms have been prevalent in the research literature and the practice for a while, the parameters of their use have been changing. Today's commodity server machines have main memory that can exceed the disk capacities of a decade ago: machines with 32GB of memory are now commonplace in the service centers of the larger Internet search and online advertising engines. This, combined with the requirements for very high-throughput and low-latency, makes disk-based indexes obsolete even for large scale applications such as web search and online advertising [10]. However, most of the published work on top-k document retrieval still report performance numbers for disk-resident indexes [9, 14, 16].

We present performance results for several state-of-the-art DAAT [7, 23] and TAAT [8, 23] algorithms used in Yahoo!'s production platform for online advertising. The stringent latency requirements of online advertising applications make disk-based indexes unusable – a single disk seek could cause the query to time out. Therefore, our platform is completely based on memory-resident indexes. To the best of our knowledge, this is the first study that compares the performance of DAAT and TAAT algorithms in a production setting using main memory indexes. As these algorithms were originally presented for disk-based indexes, they implement optimizations to minimize index access (I/O) as much as possible. We find that some of these optimizations are not suitable for memory-resident indexes. Based on these observations, we explore variations of the original algorithms that greatly improve performance (by over 60% in some cases).

One of the key observation we make about the DAAT algorithms is that, as the index access cost is lower in main memory indexes, the relative cost of score evaluation is higher than in the case of disk-based indexes. To address this issue, we examine a technique that greatly reduces the number of score evaluations for DAAT algorithms without sacrificing result quality. This technique is similar in spirit to the term bounded *max_score* algorithm [22], however, it requires no modification to the underlying index structures. The key idea is to split the query into two parts, a "short query" that can be quickly evaluated and a "long query." We then use the results obtained by processing the short query to speed up the evaluation of the long query. By applying this technique we are able to improve the performance of all DAAT algorithms by an additional 20% in many instances.

The main contributions of this paper are:

- We present the first evaluation of DAAT and TAAT algorithms over main memory indexes in a production setting. We implemented several of the existing state-of-the-art algorithms in the same production framework. In this study, we evaluate the effectiveness of the different algorithms over different query workloads and ad corpora and present conclusions on which types of algorithms are the most effective depending on the workload characteristics.

- We describe adaptations of the TAAT and DAAT algorithms for main memory indexes. These adaptations try to minimize CPU usage at the expense of index access, which is the right tradeoff for memory-resident indexes. These adapted algorithms achieved around 60% improvement in performance over their original versions. In these results both the original and the adapted algorithms were evaluated over memory resident indexes. We describe two main adaptations: one for TAAT (Section 5.3) and one for the DAAT (Section 4.2) family of algorithms.

- We propose a new technique to speed up DAAT algorithms by splitting the query into a short query and a long query. This technique produced additional 20% performance gains without sacrificing result quality (Section 7).

The rest of this paper is organized as follows. In Section 2 we provide the necessary technical background. In Section 3 we describe the ad data sets we used for evaluation and our implementation framework. We then overview DAAT algorithms in Section 4 and TAAT algorithms in Section 5. In Section 6 we show experimental results comparing the DAAT and TAAT algorithms for different index and query configurations. We then discuss the modifications reducing the number of score evaluations for DAAT algorithms and report the results of this technique in Section 7. We discuss related work in Section 8 and conclude in Section 9. More detailed information about the data sets we used is provided in the Appendix.

2. PRELIMINARIES

In this section we provide some background on inverted indexes and top-k retrieval.

Inverted indexes. Most IR systems use inverted indexes as their main data structure for both DAAT and TAAT algorithms [26]. In inverted indexes the occurrence of a term t within a document d is called a *posting*. The set of postings associated to a term t is stored in a *postings list*. A posting has the form <*docid, payload*>, where *docid* is the document ID of d and where the *payload* is used to store arbitrary information about each occurrence of t within d. Each postings list is sorted in increasing order of *docid*. Often, B-trees or skip lists are used to index the postings lists [26]. This facilitates searching for a particular *docid* within a postings list. Large postings lits are normally split into blocks (e.g. each block corresponding to a disk page). This allows entire blocks to be skipped in the search for a given *docid*. Each postings list in the inverted index corresponds to column in our matrix representation (Figure 1).

During query evaluation, a *cursor* C_t is created for each term t in the query, and is used to access t's postings list. $C_t.docid$ and $C_t.payload$ access the *docid* and *payload* of the posting on which C_t is currently positioned. DAAT and TAAT algorithms work by moving cursors in a coordinated way to find the documents that satisfy the query. Two basic methods on a cursor C_t are required to do this efficiently:

- $C_t.next()$ advances C_t to the next posting in its posting list.

- $C_t.fwdBeyond(docid\ d)$ advances C_t to the first posting in its posting list whose *docid* is greater than or equal to d. Since posting lists are ordered by *docid*, this operation can be done efficiently.

In disk-based indexes, both of these methods can incur in I/O if the desired *docid* is not in the same disk page as the current position of the cursor. DAAT and TAAT algorithms try to minimize this I/O cost by skipping parts of the index that are guaranteed not to contribute to the final response.

Scoring. In the vector space model, each document d is represented as a vector of weights:

$$d = \{d_1 \dots d_M\}$$

where M is the number of unique terms in the document corpus (i.e., the number of postings lists in the index). Each dimension of the vector corresponds to a separate term. If a term does not occur in the document its value in the vector is zero.

Similarly, each query Q is also represented as a vector of weights:

$$Q = \{q_1 \dots q_M\}$$

These document and query weights are used for scoring and can be derived using standard IR techniques, such as term frequency and inverse document frequency (tf-idf) or language modeling (LM) [18]. The document weights are computed prior to index construction and are stored in the inverted index as the document payloads. For simplicity, scoring function used in this paper is the dot product between the document and query vectors:

$$score(d, Q) = \overrightarrow{d} \bullet \overrightarrow{Q} = \sum_{1 \le i \le M} d_i q_i$$

We use this scoring function without loss of generality and the algorithms presented in this paper can be used with more elaborate scoring functions as well.

During TAAT and DAAT evaluation, for every candidate document d, the scoring function must be evaluated to determine if d belongs to the set of top-k documents. In this paper, we focus on algorithms for exact top-k computation. In these algorithms the retrieved documents are guaranteed to be the k documents with the highest score. There are several approximate algorithms for top-k retrieval, where result quality is sacrificed in order to achieve better performance [15, 16, 19]. Evaluations of these approaches are out of scope for this paper and left for future work.

3. DATA SETS AND IMPLEMENTATION FRAMEWORK

In this section we describe the data sets used throughout the paper in detail and present our implementation framework.

Data sets. Our work is focused on the application of information retrieval algorithms to contextual and sponsored search advertising [5, 6]. In these applications the document collection is a set of textual ads. Textual ads typically include a title, short description, and URL. So, unlike web documents, advertising documents are typically very small (on the order of tens of terms).

In contextual advertising, a query is derived from a target web page that is visited by a user. Therefore, unlike web queries that are quite small, contextual advertising queries can be quite large (over a hundred terms). In sponsored search advertising, ads are triggered based on user search queries on a search engine. These queries are typically quite small (around ten terms or less even after query expansion).

Parameter	Index	
	SI	LI
Number of documents	283,438	3,485,597
Number of terms	7,760,649	69,593,249
Size (MB)	269	3,277
Avg. document size	84.59	130.33
Std. deviation	149.15	103.86
Avg. postings list size	3.09	6.53
Std. deviation	272.80	1,212.95
Avg. document weight	118.05	855.00
Std. deviation	5.44	1.73

Table 1: Index parameters.

Parameter	Query set	
	SQ	LQ
Number of queries	16,181	11,203
Avg. number of terms	4.26	57.76
Std. deviation	0.77	3.32
Avg. query weight	622.91	147.74
Std. deviation	30.68	17.92

Table 2: Query parameters.

We used two test indexes (SI and LI) and two query sets (SQ and LQ). Index SI is a small index of textual ads, while LI is a much larger index of textual ads. In the query side, SQ is a query set with short queries (the sponsored search use case) based on real search queries while LQ has long queries (the contextual advertising use case) based on real web pages.

Table 1 shows the main parameters for our test indexes while Table 2 does the same for the query sets. In our performance experiments we tested several combinations of index/query set pairs. Table 3 shows the average number of postings for these index/query set combinations, i.e., the average number of entries in the intersection of the document and query terms. Extra information about these data sets is provided in the Appendix.

Implementation framework. All the algorithms described in the paper were implemented in the context of the RISE indexing framework. RISE is an inverted index platform that has been widely used within Yahoo! both for research [4, 5, 6, 12, 24] and production over the last three years. RISE is a C++ indexer and its performance has been heavily tuned. Our indexes are compressed using delta encoding [3, 25] for the *docids* and our implementation of postings list access operations is optimized. In RISE, posting lists are stored in several (contiguous) blocks and skip lists are used to index these blocks. This allows us to skip entire blocks during query evaluation, when we conclude that these blocks cannot contribute to the final top-k results.

The algorithms described in this paper were developed in C++ and compiled with g++-4.1.2 with option -O3. All experiments were executed in a Xeon L5420 2.50GHz, with 8GB of RAM and L2 cache of 12MB running RHEL4. In every experiment where we report running time, the index was preloaded into memory and the numbers are averaged over three independent runs. The latency numbers are always in microseconds. The number of desired results (k) was set to

Parameter	Query set	
	SQ	LQ
Avg. number of postings for SI	3,540.95	52,560.38
Std. deviation for SI	7,429.81	44,190.83
Avg. number of postings for LI	79,615.76	378,026.59
Std. deviation for LI	168,724.48	398,347.04

Table 3: Number of postings for different index/query set combinations.

30 in all experiments we run. When reporting results for a query set (SQ or LQ) we average the results over all of its queries.

4. DAAT ALGORITHMS

The DAAT algorithms simultaneously traverse the postings lists for all terms in the query. The naive implementation of DAAT simply merges the involved postings lists (which are already sorted by *docid* in the index) and examines all the documents in the union. A min-heap is normally used to store the top-k documents during evaluation. Whenever a new candidate document is identified it must be scored. The computed score is then compared to the minimum score in the heap, and if it is higher, the candidate document is added to the heap. At the end of processing the top-k documents are guaranteed to be in the heap.

There are two main factors in evaluating the performance of the DAAT algorithms: the index access cost and the scoring cost[1]. In the case of the naive DAAT algorithm, every posting for every query term must be accessed. The index access cost is then proportional to the sum of the sizes of the postings list for all query terms. The scoring cost includes computing the scoring function and updating the result heap. As most of the DAAT (and TAAT) algorithms have been designed for disk-based indexes, they try to minimize the index access cost, e.g., skipping parts of the postings list.

In the next subsections we present two DAAT algorithms: WAND [7] and *max_score* [23]. We analyze the performance of these algorithms and propose optimizations to WAND to make it more suitable for memory-resident indexes.

4.1 WAND

The main intuition behind WAND [7] is to use upper bounds on score contributions to improve query performance. For each postings list in the index, we can pre-compute and store the maximum payload value, i.e., the maximum document weight for that term. Given a query Q, during initialization we compute the maximum upper-bound UB_t for each term $t \in Q$ as:

$$UB_t = D_t q_t$$

where D_t is the maximum payload value, i.e., the maximum value of d_t for every $d \in D$. WAND works by keeping a cursor for each of the query terms and sorting these cursors by *docid*. After the cursors are sorted a *pivot term* is identified. The pivot term p is the minimum index on the

[1]In this paper we are counting the postings list decompression cost as part of the scoring cost, as we use the index access cost to model the operations that could result in I/O for disk-resident indexes.

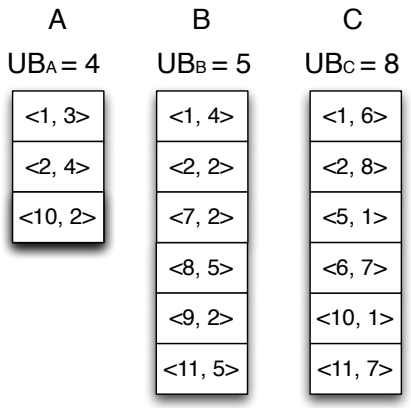

Figure 2: Postings list and upper bounds for query terms A, B and C.

array of sorted cursors for which:

$$\sum_{1 \le t \le p} UB_t > \theta$$

where θ is the minimum document score in the top-k result heap. The document pointed by the pivot term is the minimum document (i.e., the document with the smallest *docid*) that can possibly be a valid candidate. This is called the pivot document. Once the pivot is identified, WAND checks if the *docids* pointed by cursors 1 and p are the same – if this is true the document is scored, otherwise WAND selects a term between 1 and p and tries to move the cursor for that term to the pivot document. This operation is normally done using the index skipping mechanism (B-trees or skip lists) and can reduce the index access time if large portions of postings list are skipped. After each cursor move, the cursor array is resorted and a new pivot term is identified. The full WAND algorithm is described in [7].

Let us consider query $Q = \{A, B, C\}$ with all query weights $q_A = q_B = q_C = 1$ (so the document scores are the sum of the document weights). Figure 2 shows the postings list and their upper bounds for terms A, B and C. Each posting is represented as a <*docid, payload*> pair. In this example let us consider $k = 2$, i.e., we want to retrieve the two documents with the highest scores. After *docids* 1 and 2 have been processed we have a heap with two documents:

Heap	
docid	$score(d, Q)$
1	13 (θ)
2	14

At this point the cursors for terms A, B and C point to documents 10, 7 and 5, respectively. WAND then sorts the cursors by *docid* in order to identify the pivot. After the sort, we have:

	C	B	A
p	1	2	3
docid	5	7	10

where p is the index of the cursor array. At this point WAND starts scanning the array of sorted cursors to select

the pivot. For $p = 1$, we have:

$$UB_C = 8 < \theta = 13$$

For $p = 2$ we have:

$$UB_C + UB_B = 8 + 5 = \theta = 13$$

For $p = 3$ we have:

$$UB_C + UB_B + UB_A = 8 + 5 + 4 > \theta = 13$$

The pivot is then set to term A ($p = 3$) and the pivot document is $docid = 10$. This means that the minimum *docid* that can potentially be in the top-k results is document 10. Therefore WAND will move either B's or C's cursor to document 10 in order to continue processing. When compared to the naive DAAT algorithm, it is clear that WAND may reduce both the index access and the scoring costs. In this simple example *docids* 6, 8 and 9 are completely skipped in postings list for terms B and C and are not scored.

At the implementation level we try to optimize the cursor sort for the pivot selection by noticing that part of the array (all entries beyond the pivot term) are not affected by the cursor moves. Therefore, that part of the array is already sorted and we can only sort the initial part of array with index $i \leq p$.

4.2 Memory-resident WAND

WAND was originally designed for disk-resident indexes, therefore it tries to reduce index access cost as much as possible. Let us consider again the example from Figure 2. At the point we identify $docid = 10$ as the pivot document, WAND will select either cursor B or C to move to document 10 (or beyond). The reason for moving only one of these cursors per time is that in disk-resident indexes each cursor movement is a potential new I/O (if the desired position for the cursor lies in a different page than its current position).

In this example, WAND may choose to move cursor B to *docid* 10 and reevaluate the pivot. In that case, we would have:

	C	A	B
p	1	2	3
docid	5	10	11

The pivot term for this new configuration would be B and the pivot document would be 11. In that case, WAND would be able to skip over the posting for document 10 in C's postings list.

In order to minimize index access WAND performs extra pivot finding operations. This is a good tradeoff for disk-based indexes, but not for main memory indexes where the cost of index access is normally smaller than the cost of pivot find operations (remember that for finding the pivots the array of cursors must be sorted by *docid*).

Based on this observation we propose a variation of WAND that we call Memory-Resident WAND (mWAND). The main difference between mWAND and the original algorithm is that after a pivot term p is selected, we move all terms between 1 and p beyond the pivot document. By doing that we are increasing the cost of index access in order to reduce the number of pivot selections (and the associated cost of sorting the cursor array). In our running example, if both cursors B and C had been simultaneously moved to document 10, the posting entry for document 10 in C's list would not have been skipped.

SI	SQ	LQ
Pivot selections (WAND)	2,843.44	17,636.18
Pivot selections (mWAND)	2,840.13	12,798.87
Skipped postings (WAND)	532.56	28,581.22
Skipped postings (mWAND)	531.20	27,214.16
Latency (WAND)	206.0	5,519.0
Latency (mWAND)	200.0	2,104.6
LI	**SQ**	**LQ**
Pivot selections (WAND)	28,007.55	282,356.02
Pivot selections (mWAND)	27,814.06	275,164.82
Skipped postings (WAND)	48,089.58	82,511.85
Skipped postings (mWAND)	47,985.65	66,997.41
Latency (WAND)	1896.6	14,082.6
Latency (mWAND)	1867.0	7,556.3

Table 4: Comparison between WAND and mWAND.

In Table 4 we compare the performance of WAND and mWAND. We show the number of pivot selections, number of skips and query latency (running time) for different index/query set combinations. As expected, mWAND performs less pivot selection operations at the expense of less skipping. As shown in the table, this tradeoff is always beneficial and mWAND performs better than WAND for all cases we tested. The improvement is more noticeable in the case of long queries, where the number of pivot selection operations is dramatically reduced. In the (SI, LQ) combination mWAND performs 60% better than WAND.

4.3 DAAT *max_score*

Both DAAT and TAAT implementations of *max_score* have been proposed by Turtle and Flood [23]. We present the DAAT version here and the TAAT version in Section 5.2. Like WAND, the idea of *max_score* is to use upper bounds to reduce index access and scoring costs.

DAAT *max_score* starts by sorting the query cursors by the size of their postings list. This cursor order is fixed throughout evaluation. Before the first k documents have been evaluated, *max_score* works as the naive DAAT algorithm. After that point, when the heap is full, *max_score* uses the minimum score of the heap (θ) as the lower bound for the next documents. This lower bound allows *max_score* to skip over postings (and therefore reduce both index access and scoring costs).

The intuition for skipping in *max_score* is to identify which set of cursors must be present in the document in order to have a potential candidate[2]. This operation is done by considering the upper bounds of the query terms and comparing them to the current value of the lower bound θ. As in the case of WAND, θ increases and as the evaluation proceeds and it becomes harder to identify new candidates as the algorithm progresses.

Let us again consider the example of Figure 2. After docu-

[2]The skipping strategy is not clearly specified in the original paper [23]. This approach is based on our interpretation of how skipping may be achieved in DAAT *max_score*.

ments 1 and 2 have been added to the heap, we have $\theta = 13$. In *max_score* cursors are only sorted in the beginning of processing (by their postings list sizes), and we have the following order for this example:

	A	B	C
Postings list size	3	6	6

We then split the sorted array of terms into two groups: required and optional terms. The split property is that in order for a new document to be a valid candidate it must have at least one term from the required set. We identify the optional terms by processing the array of cursors from the end to the beginning.

In this example we start from cursor C and check if it can (by itself) be pointing to the next valid document:

$$UB_C = 8 < \theta = 13$$

This means that C by itself is not enough. We then try to add the next term:

$$UB_C + UB_B = 8 + 5 = \theta = 13$$

This is still not enough to qualify a document. Then we have:

$$UB_C + UB_B + UB_A = 8 + 5 + 4 > \theta = 13$$

We then split the array to mark A as a required term and B and C as optional terms. Intuitively, this means that, to be considered a valid candidate, documents must contain term A. Once we identify the split into required and optional terms, DAAT *max_score* behaves like naive DAAT for the terms in the required set. Once a candidate document d is identified from the required terms, we must move the cursors from the optional set to d's *docid* for scoring. Whenever new candidates are identified, the split between optional and required terms must be recomputed.

In our example, since only A belongs to the required set it will produce document 10 as a potential candidate. At that point we try to move cursors B and C to document 10 for scoring. In this example, postings 6, 8 and 9 from cursors B and C would be skipped. It is clear that *max_score* improves over the naive DAAT by reducing the index access and scoring costs. When compared to WAND, DAAT *max_score* has the advantage of avoiding the sort operations to compute the pivot, at the expense of less optimized skipping.

4.4 Comparing the DAAT algorithms

Table 5 reports the query latency for naive DAAT, mWAND and DAAT *max_score* for all index/query set combinations. For both indexes, *max_score* performs better than mWAND for short queries while the opposite happens for long queries. The reason is that for long queries the pivot selection procedure in mWAND can greatly improve skipping. However, for short queries, the gains in skipping are not that much when compared to *max_score* to justify the overhead of pivot selection. Table 6 compares the skipping between mWAND and DAAT *max_score*.

SI	SQ	LQ
Naive DAAT	193.0	4,554.6
mWAND	200.0	2,104.6
DAAT *max_score*	169.0	2,685.6
LI	SQ	LQ
Naive DAAT	3,581.3	26,778.3
mWAND	1,867.0	7,556.3
DAAT *max_score*	1,572.6	9,321.3

Table 5: Latency results for naive DAAT, mWAND and DAAT *max_score*.

SI	SQ	LQ
mWAND	531.20	27,214.16
DAAT *max_score*	505.12	22,013.45
LI	SQ	LQ
mWAND	47,985.65	275,164.82
DAAT *max_score*	45,709.97	235,740.23

Table 6: Number of skipped postings for mWAND and DAAT *max_score*.

5. TAAT ALGORITHMS

TAAT algorithms traverse one postings list at-a-time. The contributions from each query term to the final score of each document must be stored in an array of accumulators A. The size of the accumulator array[3] is the number of documents in the index (N). For dot product scoring, the score contributions for each term can be independently computed and added to the appropriate documents in the accumulator array. In the naive implementation of TAAT we must access every posting for every term. For each posting, we compute its score contribution and add it to the accumulator array. When processing term t, we compute:

$$A[d] \leftarrow A[d] + d_t q_t$$

for every posting in t's postings list, where d_t is the document weight in the posting for document d and q_t is the query weight for term t. The accumulator array must be initialized to zero in the beginning of the query execution. After processing all terms in the query, the k entries in A with the highest value are top-k documents that should be returned as the query results.

As in the case of the naive DAAT, this naive implementation of TAAT must access every posting for every query term and compute the full score for every document. In the next subsections we present two TAAT algorithms: Buckley and Lewit [8] and *max_score* [23]. We also evaluate the performance of these algorithms and propose optimizations that make TAAT *max_score* more suitable for memory-resident indexes.

5.1 Buckley and Lewit

The algorithm proposed by Buckely and Lewit [8] is one of the first optimizations for top-k ranked retrieval. The main intuition is to evaluate one term-at-a-time, in the decreasing order of their upper bounds, and stop when we realize that further processing will not be able to add new documents to set of top-k documents. We maintain a heap of size $k + 1$,

[3]Different implementations for accumulators have been proposed, such as hash tables or sorted arrays [21].

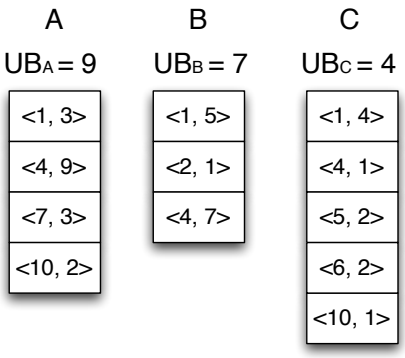

Figure 3: Another example of postings list and upper bounds.

which contains the $k+1$ documents with the highest partial scores we have seen so far, i.e., the $k+1$ highest scores from the accumulator array.

After processing each postings list, we compare the scores of the kth and $(k+1)$th documents to decide if we can early terminate. If the sum of the upper bound contributions of the remaining lists cannot bring the $(k+1)$th document score over the kth score we can safely stop processing. Formally, we must check if:

$$A[k] \geq A[k+1] + \sum_{t>i} UB_t$$

where i is the current term being processed. If this check passes, we know that the k documents with the highest partial scores so far are the top-k documents we must retrieve (however, their actual rank from 1 to k may not correspond to the final rank had we added the contributions of the remaining terms).

Figure 3 shows another example of postings list and their upper bounds. Let us again consider that the weights for all query terms A, B and C are 1 and that we want to retrieve the two documents with the highest scores.

Table 7 shows the state of the accumulators after each iteration of the algorithm. Column i indicates which term is being processed ($i = 1$ is A, the term with the highest upper bound). When we finish processing term B (the last row in the table), the second (kth) document with the highest score is 1 and the third is document 7. At this point the check:

$$A[1] = 8 \geq A[7] + \sum_{t>2} UB_t = 3 + 4$$

succeeds and and we can early terminate. It is clear from this example that the Buckley and Lewit pruning procedure helps in reducing index access and scoring costs when compared to the naive TAAT algorithm.

5.2 TAAT *max_score*

We now describe the TAAT variation of *max_score* [23]. This algorithm has two main phases. In the first phase, we maintain a heap of size k that contains the k documents with the highest partial scores so far. Like in the DAAT version of *max_score*, we process terms in the decreasing order of their postings list sizes. After processing each term, we check if the partial score for the kth document is greater than the sum of the upper bounds for the remaining postings list:

i	*docid*	A[1]	A[2]	A[4]	A[5]	A[6]	A[7]	A[10]
1	1	3	0	0	0	0	0	0
1	4	3	0	9	0	0	0	0
1	7	3	0	9	0	0	3	0
1	10	3	0	9	0	0	3	2
2	1	8	0	9	0	0	3	2
2	2	8	1	9	0	0	3	2
2	4	8	1	16	0	0	3	2

Table 7: The term (i), *docid*, and accumulator values after each iteration of Buckley and Lewit.

i	A[1]	A[2]	A[4]	A[5]	A[6]	A[7]	A[10]
1	5	1	7	0	0	0	0
2	8	1	16	0	0	3	2

Table 8: The term (i) and accumulator values after processing each term in phase I of TAAT *max_score*.

$$A[k] > \sum_{t>i} UB_t$$

If this condition holds, we know that no documents that are not already present in the accumulator array (i.e, that have 0 partial score so far) can be in the top-k documents. We can then break phase I of the algorithm and start phase II. In the second phase, we only need to score the documents that we have seen in phase I. Therefore, we can use list of documents processed in phase I to skip parts of the remaining postings list. To do this we must maintain an ordered list of the documents processed in phase I – we call this list the *candidate list*.

Table 8 shows the accumulator values for our running example after processing each term in phase I. As the cursors are processed by postings list size (instead of upper bound contributions), the processing order for *max_score* differs from Buckley and Lewit. When we are done processing term A ($i = 2$), we have:

$$A[1] = 8 > \sum_{t>2} UB_t = 4$$

At this point we can stop phase I and our candidate list is $\langle 1, 2, 4, 7, 10 \rangle$. We can then use this list to skip when processing term C. This means that we would not need to look in the postings for documents 5 and 6 while processing term C.

Another optimization proposed in TAAT *max_score* is to prune the candidate list during phase I. This can be done by checking if:

$$A[k] > A[d] + \sum_{t>i} UB_t$$

If this holds, document d can never be part of the top-k candidates and it can be safely removed from the candidate list. By applying this optimization in our example, we can remove documents 2, 7 and 10 from the candidate list before starting phase II.

5.3 Memory-resident TAAT *max_score*

TAAT *max_score* was originally designed for disk-resident indexes, where each cursor movement may correspond to an

SI	SQ	LQ
Number of terms left for phase II	0.13	3.44
Latency (TAAT *max_score*)	193.3	3,109.0
Latency (mTAAT *max_score*)	129.3	1,385.3
LI	SQ	LQ
Number of terms left for phase II	0.48	3.66
Latency (TAAT *max_score*)	3,139.3	17,260.6
Latency (mTAAT *max_score*)	2,520.6	11,839.6

Table 9: Comparison between TAAT *max_score* and mTAAT *max_score*.

extra I/O depending if the requested *docid* is in the same disk page or not. Therefore, its second phase tries to minimize cursor movements by using the candidate list to drive skips in the remaining postings list. In order to use skipping in phase II, TAAT *max_score* has to maintain an ordered list of candidate documents. Please note that the accumulator array is a sparse array of size N (N is the number of documents in the index) and therefore it is usually much larger than the candidate list. Given this, it is usually more efficient to keep an extra candidate list than to use the accumulator array to drive skipping in phase II.

However, this means that this candidate list must be updated during phase I and sorted before phase II starts. We observed that since index access is not as expensive in memory-resident indexes, in many cases it is more efficient to not skip during phase II, but to scan the postings list sequentially only scoring the documents that have a positive value in the accumulator array. We also do not try to prune the candidate list during phase I. We call this variation of *max_score* memory-resident TAAT *max_score* (mTAAT *max_score*).

In Table 9 we compare TAAT *max_score* with its memory-resident variant. We show the number of query terms left to be evaluated during phase II and query latency (running time) for different index/query set combinations. In all cases the performance of mTAAT *max_score* is superior. The main reason is the fact that the number of query terms left for phase II is always very small – therefore it is not worthwhile to compute and maintain the candidate list to drive skips over a small number of postings list. In the case of small indexes, where the size of the postings list are smaller, the benefit of mTAAT *max_score* is higher. For the case of (SI, LQ), the improvement is around 58%.

5.4 Comparing the TAAT algorithms

Table 10 reports the query latency for naive TAAT, Buckley and Lewit and mTAAT *max_score*. In all cases mTAAT *max_score* performs better than the other approaches. The main reason is that it does much less scoring computations. Table 11 shows the number of postings that are not scored for the two algorithms (for the naive algorithm this number is always 0 since it does not skip any postings). Although Buckley and Lewit is able to skip a few scoring computations, it is not enough to justify the overhead of the algorithm for small indexes (when comparing it to naive TAAT).

6. COMPARING DAAT WITH TAAT

In this section we compare the best DAAT algorithms (mWAND and DAAT *max_score*) with the best TAAT algorithm (mTAAT *max_score*). Table 12 summarizes their latencies for the several index/query combinations. As shown

SI	SQ	LQ
Naive TAAT	141.0	1,694.6
Buckley and Lewit	143.6	1,744.6
mTAAT *max_score*	129.3	1,385.3
LI	SQ	LQ
Naive TAAT	3,777.6	18,913.0
Buckley and Lewit	3,643.6	17,690.3
mTAAT *max_score*	2,520.6	11,839.6

Table 10: Latency results for naive TAAT, Buckley and Lewit and mTAAT *max_score*.

SI	SQ	LQ
Buckley and Lewit	0.06	624.39
mTAAT *max_score*	1,118.79	24,538.98
LI	SQ	LQ
Buckley and Lewit	0.88	7,394.00
mTAAT *max_score*	60,757.20	26,0882.20

Table 11: Number of postings that are not scored for Buckley and Lewit and mTAAT *max_score*.

in the table, for the small index TAAT performs better than DAAT while the opposite is true for the large index.

The main reasons are the fact that for small indexes the sequential behavior of TAAT is very beneficial. Moreover, although mTAAT *max_score* does not really skip postings, this does not have a major impact for small, memory-resident indexes. When we go to large indexes, on the other hand, the lack of skipping is also a bigger disadvantage. In addition to that, the number of cache misses for TAAT drastically increases, probably due to random access over the large array of accumulators. Please note that other implementations of accumulators are possible, e.g. based on dense sorted arrays [21] – these variations, however, make the TAAT algorithms much more inefficient in our settings. Figure 4 shows the relative number of cache misses for the different algorithms, highlighting its impact in the TAAT performance when we go from small to large indexes.

7. OPTIMIZING DAAT

We now propose a new technique that can be used to improve the performance of all DAAT algorithms. We first split the query terms into two groups: terms with short postings list and terms with long postings list. The split is based on a configurable threshold (T) as follows:

$$Q = Q_{t \leq T} \cup Q_{t > T}$$

where $t \leq T$ means that the size of the postings list for

SI	SQ	LQ
mWAND	200.0	2,104.6
DAAT *max_score*	169.0	2,685.6
mTAAT *max_score*	129.3	1,385.3
LI	SQ	LQ
mWAND	1,867.0	7,556.3
DAAT *max_score*	1,572.6	9,321.3
mTAAT *max_score*	2,520.6	11,839.6

Table 12: Latency results for mWAND, DAAT *max_score* and mTAAT *max_score*.

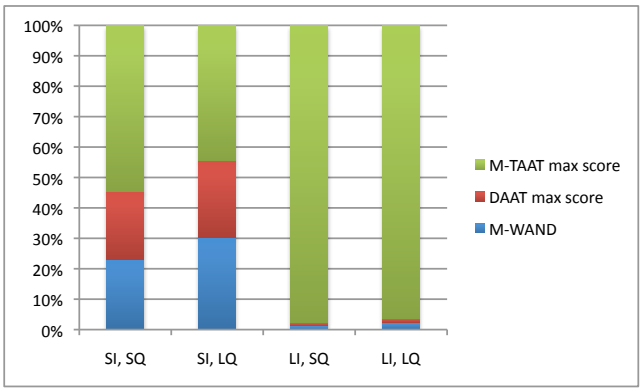

Figure 4: Relative number of cache misses for the different algorithms over different index and query set combinations.

term t is smaller than the threshold T. The threshold T must be defined during query initialization (or prior to that). Once we split the query, we start evaluation by processing the sub query with small postings list, $Q_{t \leq T}$. That can be done using any of the TAAT or DAAT algorithms described in this paper. When that processing completes, we have partial scores for all documents that were evaluated. We can then use the partial score for the kth element in the heap as the lower bound (θ), which will be used for processing the long sub query, $Q_{t>T}$. Another result obtained from processing $Q_{t \leq T}$ is a candidate list (cl), which is the list of all documents that were evaluated and have partial score greater than zero.

We now use a DAAT algorithm to evaluate a new query:

$$Q_{DAAT} = Q_{t>T} \cup \{cl\}$$

where the candidate list cl is viewed as another postings list, i.e., as an extra term. During this DAAT evaluation we use θ as the initial lower bound. This algorithm can be viewed as a hybrid TAAT-DAAT, since we are imposing some restriction on the order the terms are processed by doing the initial splitting of the query terms (which has a TAAT flavor).

The main intuition behind this algorithm is the belief that $Q_{t \leq T}$ can be quickly evaluated (since the postings list are small) and we can then set a good lower bound θ for processing the large postings list. With good lower bounds DAAT can do better skipping. The idea of doing some preprocessing work to set a better lower bound for DAAT was also successfully used in the term bounded *max_score* [22].

We have implemented several variations of this idea:

1. DAAT-mWAND: uses naive DAAT for $Q_{t \leq T}$ and mWAND for Q_{DAAT}
2. TAAT-mWAND: uses naive TAAT for $Q_{t \leq T}$ and mWAND for Q_{DAAT}
3. DAAT-DAAT *max_score*: uses naive DAAT for $Q_{t \leq T}$ and DAAT *max_score* for Q_{DAAT}
4. TAAT-DAAT *max_score*: uses naive TAAT for $Q_{t \leq T}$ and DAAT *max_score* for Q_{DAAT}

For each of these algorithms we did an offline search for the best threshold value T for each of our index and query set combinations. Procedures to automatically select the best threshold value T for each workload are outside the scope of this paper and left for future work.

SI	SQ	LQ
DAAT-mWAND	186.3	2,044.6
TAAT-mWAND	189.0	2,060.3
mWAND	200.0	2,104.6
DAAT-DAAT *max_score*	159.0	2,350.3
TAAT-DAAT *max_score*	160.3	2,354.0
DAAT *max_score*	169.3	2,685.6
LI	**SQ**	**LQ**
DAAT-mWAND	1,619.6	6,862.6
TAAT-mWAND	1,669.3	6,927.3
mWAND	1,867.0	7,556.3
DAAT-DAAT *max_score*	1,390.3	7,513.0
TAAT-DAAT *max_score*	1,433.3	7,604.3
DAAT *max_score*	1,572.6	9,321.3

Table 13: Latency results for the hybrid algorithms, mWAND and DAAT *max_score*.

SI	SQ	LQ
DAAT-mWAND	663.9	28,862.3
TAAT-mWAND	544.9	28,510.2
mWAND	531.2	27,214.2
DAAT-DAAT *max_score*	639.25	23,887.03
TAAT-DAAT *max_score*	639.25	23,887.03
DAAT *max_score*	505.12	22,013.45
LI	**SQ**	**LQ**
DAAT-mWAND	52,155.23	283,952.02
TAAT-mWAND	50,623.64	283,533.80
mWAND	47,985.65	275,164.82
DAAT-DAAT *max_score*	49,757.74	251,963.15
TAAT-DAAT *max_score*	48,833.30	250,527.67
DAAT *max_score*	45,709.97	235,740.23

Table 14: Number of skipped postings for the hybrid algorithms, mWAND and DAAT *max_score*.

Table 13 compares the performance of each of these algorithms, using the best threshold value T for each case, with mWAND and DAAT *max_score*. In all cases the hybrid algorithms perform better than mWAND and DAAT *max_score*, which means that these hybrid algorithms are the overall best for large indexes. Table 14 confirms our intuition that by setting a good initial lower bound θ the DAAT algorithms can skip more.

8. RELATED WORK

DAAT and TAAT algorithms have been compared in the past. In [9], the authors of the Juru search system performed experiments comparing DAAT and TAAT algorithms for the large TREC GOV2 document collection. They found that DAAT was clearly superior for short queries, showing over 55% performance improvement. In addition, the performance for DAAT for long queries was even better, often a factor of 3.9 times faster when compared to TAAT. Unlike our work, which focuses on memory-resident indexes, this work used the Juru disk-based index for the performance evaluations.

A large study of known DAAT and TAAT algorithms was conducted by [14] on the Terrier IR platform (with

disk-based postings list) using TREC WT2G, WT10G, and GOV2 document collections using both short and long queries (except that long queries were not used for GOV2). They found that in terms of the number of scoring computations, the Moffat TAAT algorithm [19] was the best, though it came at a tradeoff of loss of precision compared to naive TAAT and the TAAT and DAAT *max_score* algorithms [23]. In this paper we did not evaluate approximate algorithms such as Moffat TAAT [19]. We leave this study as future work.

A memory-efficient TAAT query evaluation algorithm was proposed in [15]. This algorithm addresses the fact that in TAAT strategies, a score accumulator is needed for every document. However, for top-k document retrieval, it is possible to dramatically reduce the size of the accumulator array without noticeable loss in precision. Using the TREC GOV2 document collection with short queries, they found that with even as few as accumulators for 0.4% of the documents, result precision was very effective.

A new DAAT algorithm, the term bounded *max_score*, was proposed in [22]. This algorithm improves upon the DAAT *max_score* [23] by using extra index structures to set a better initial threshold for DAAT *max_score*. These extra index structures are small lists that contain the top scoring documents for each term. These lists are processed before DAAT *max_score* starts, to set a tighter initial threshold (and therefore increase index skips). Term bounded *max_score* is an exact algorithm, as it produces the exact same results DAAT *max_score* would produce. Using the GOV2 document collection with short queries, they saw a 23% performance improvement gain over DAAT *max_score*. This algorithm is very similar in spirit to our DAAT optimization described in Section 7, except for the fact that we do not need extra index structures.

The problem of improving performance of queries by pruning documents was investigated in [16]. Instead of pruning documents based only on their term scores, they considered scoring functions that also contain a query independent component (such as the PageRank). They explored the performance impact of providing a global document order to the index based on the query independent components of the score. They used the *pingo* search engine (which uses disk-based postings list) and run experiments over large 120 million corpus of web documents. They proposed several pruning techniques based on the global document scores and showed that latency can be greatly reduced with little loss in precision. This work is complementary to term-based pruning techniques.

In [13], the authors indicate that two main ways of improving performance in information retrieval systems are: (1) early termination while evaluating postings list for top-k queries and (2) combining postings list together into one shorter lists using intersection (as proposed in [17], where intersection postings list were cached to speed up query evaluation). The contribution of this work is to combine these two techniques using a rigorous theoretical analysis. In addition, they performed empirical tests on the TREC GOV2 data set and on real web queries showing the performance gains of their early termination method.

In this paper we focus on inverted indexes that are sorted by *docid*, as these are pervasive in large search engines [10] and online advertising [5, 6, 12, 24]. Inverted indexes where posting lists are sorted by scores, however, have also been studied by the information retrieval and database communities [1, 11, 21]. Anh and Moffat [1] have proposed the idea of impact-sorted indexes, where posting lists are split into e.g. eight segments (of increasing sizes), with the documents with higher scores being stored in the initial segments of the index. The idea of the query evaluation algorithms is to access the minimum required number of segments, which is possible since the highest scoring documents for each term are stored in the beginning of the index. Please note that in this index organization it is hard to efficiently evaluate some operations that are very common in web search, such as phrase queries and scoring functions that are based on term proximity.

In [21] the authors propose improvements the query evaluation algorithms proposed by Anh and Moffat [1]. These improvements reduce the size of the accumulator array, in turn reducing the number of score computations. They also study the effect of different skipping strategies and verify that changing the skip lengths has little effect in performance in their setting. The proposed algorithms are evaluated in main memory, using the Galago search engine and the TREC GOV2 test collection.

Fagin et. al. [11] have proposed family of algorithms known as the Threshold Algorithms (TA). These algorithms are also based on the idea that the several lists are independently sorted in decreasing order of scores. The TA algorithms provide formal bounds on the minimum number of postings that need to be accessed to guarantee that the top-k document are correctly retrieved. The authors also prove that the TA algorithms are instance-optimal, i.e., they are optimal for every instance of index and query workloads. The fact that each of the postings list have a different order makes TA algorithms not directly applicable to web search and online adverting for the same reasons presented in the discussion of impact-based indexes: phrase and proximity queries, for instance, cannot be evaluated efficiently.

9. CONCLUSIONS

We presented a study of top-k retrieval algorithms using Yahoo!'s production platform for online advertising where the inverted indexes are memory-resident. Such setup is common in many applications that require low latency query evaluation as web search, email search, content management systems, etc. While many variants of the two main families of top-k algorithms have been proposed and studied in the literature, to the best of our knowledge, this is the first study that evaluates their performance for main memory indexes in a production setting.

We have also shown that the performance of the algorithms can be substantially improved with some modification to take in account the in-memory setting. mWAND, a new variation of the WAND algorithm [7], can improve the original algorithm by over 60%. An enhanced mTAAT *max_score* improves the performance of the original TAAT *max_score* [23] by 58% for the (SI, LQ) case. In these results both the original and the adapted algorithms were implemented over memory-resident indexes, so the improvements are algorithmic.

We have also experimented with improving the performance of DAAT algorithms by using multi-phase algorithms that split the query into two parts based on the size of the postings list. By doing so, we can first evaluate a "short query" and use the results of this computation to speed

up the processing of the remaining (long) terms. Our results showed that this technique improves the performance of the original DAAT algorithms for all index and query set combinations we tested. From these experiments we found, for instance, that performance over the already fast DAAT *max_score* algorithm can be further improved by an additional 20%.

Our conclusion is that variants of the traditional DAAT and TAAT algorithms that have been originally proposed and evaluated in disk-based settings can be more efficient for modern, main-memory settings.

10. REFERENCES

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

[2] Apache. Apache hadoop project. In *lucene.apache.org/hadoop*.

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

[4] M. Bendersky, E. Gabrilovich, V. Josifovski, and D. Metzler. The anatomy of an ad: Structured indexing and retrieval for sponsored search. In *WWW*, 2010.

[5] A. Broder, P. Ciccolo, M. Fontoura, E. Gabrilovich, V. Josifovski, and L. Riedel. Search advertising using Web relevance feedback. In *CIKM*, pages 1013–1022, 2008.

[6] A. Broder, M. Fontoura, V. Josifovski, and L. Riedel. A semantic approach to contextual advertising. In *SIGIR*, pages 559–566. ACM Press, 2007.

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

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

[9] D. Carmel and E. Amitay. Juru at 2006: Taat versus daat in the terabyte track. In *TREC*, 2006.

[10] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, page 1, 2009.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[12] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, S. Venkatesan, and J. Y. Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD Conference*, pages 3–14, 2010.

[13] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top- aggregation using intersections of ranked inputs. In *WSDM*, pages 222–231, 2009.

[14] P. Lacour, C. Macdonald, and I. Ounis. Efficiency comparison of document matching techniques. In *Efficiency Issues in Information Retrieval Workshop; European Conference for Information Retrieval*, pages 37–46, 2008.

[15] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, pages 470–477, 2005.

[16] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, pages 129–140, 2003.

[17] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, pages 257–266, 2005.

[18] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[19] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[20] P. Ogilvie and J. Callan. Experiments using the lemur toolkit. In *TREC-10*, pages 103–108, 2002.

[21] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *SIGIR*, pages 175–182, 2007.

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

[23] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[24] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing boolean expressions. *PVLDB*, 2(1):37–48, 2009.

[25] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[26] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.

APPENDIX

A. DATA SETS CHARACTERISTICS

In this appendix we present more detailed statistics of our ad datasets as well as query workloads.

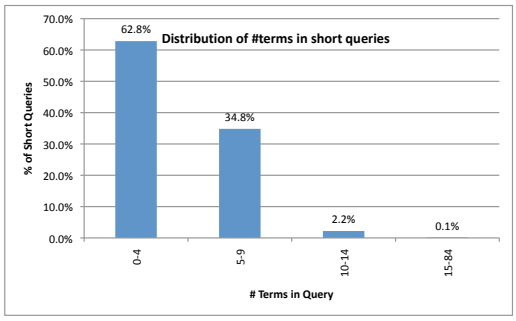

Figure 5: SQ term distribution.

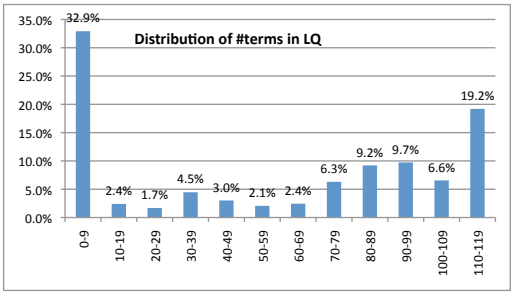

Figure 6: LQ term distribution.

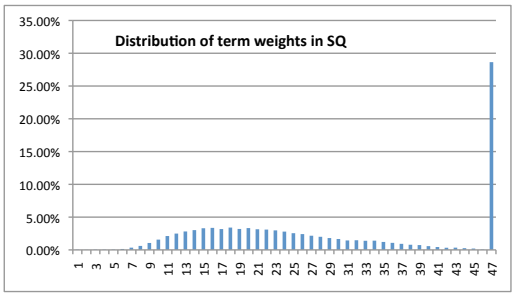

Figure 7: SQ term weight distribution.

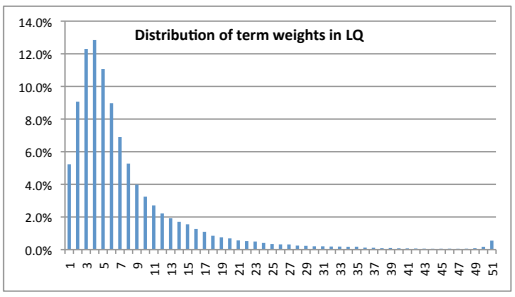

Figure 8: LQ term weight distribution.

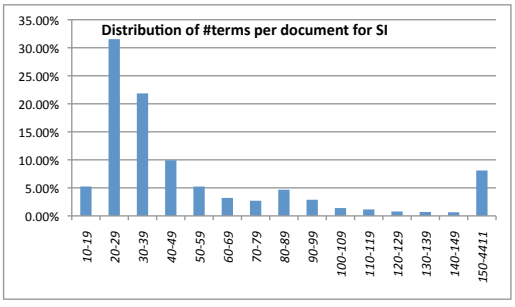

Figure 9: SI document size distribution.

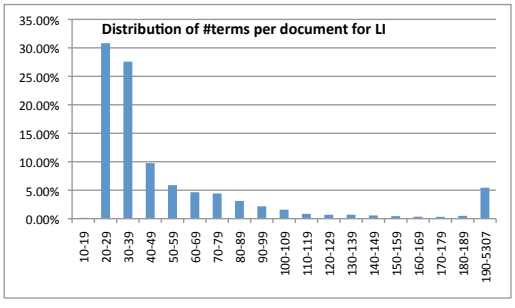

Figure 10: LI document size distribution.

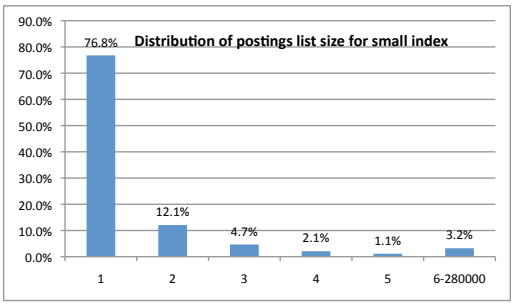

Figure 11: SI postings list size distribution.

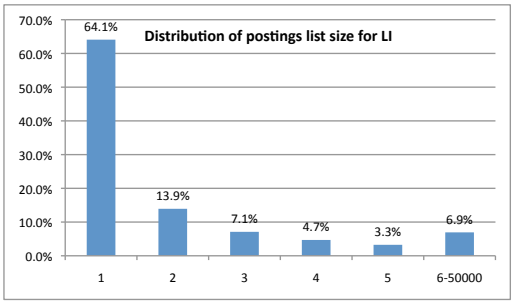

Figure 12: LI postings list size distribution.

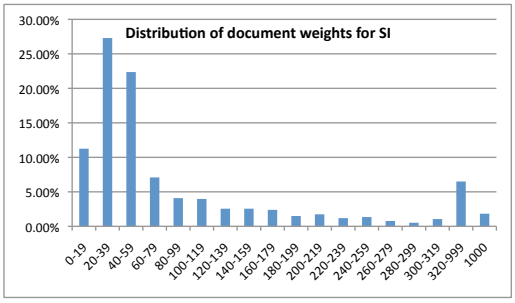

Figure 13: SI document weight distribution.

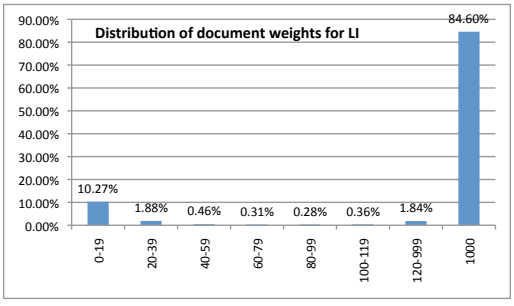

Figure 14: LI document weight distribution.