Dual-Sorted Inverted Lists in Practice*

Roberto Konow[1,2] and Gonzalo Navarro[1]

[1] Dept. of Computer Science, Univ. of Chile.
[2] Escuela de Informática y Telecomunicaciones (EIT), Univ. Diego Portales, Chile
{rkonow,gnavarro}@dcc.uchile.cl

Abstract. We implement a recent theoretical proposal to represent inverted lists in memory, in a way that docid-sorted and weight-sorted lists are simultaneously represented in a single wavelet tree data structure. We compare our implementation with classical representations, where the ordering favors either bag-of-word queries or Boolean and weighted conjunctive queries, and demonstrate that the new data structure is faster than the state of the art for conjunctive queries, while it offers an attractive space/time tradeoff when both kinds of queries are of interest.

1 Introduction

The inverted index is an old and simple, yet efficient, data structure that allows us to search within a set of documents for queries q formed by sets of words. It plays a central role in the Information Retrieval (IR) field [4, 6, 23, 27, 29] and in Web search engines. Given a text collection containing a set of D documents, where each document has a unique *document identifier* (docid), an inverted index is an array of *lists* or *postings*. Each entry of the array corresponds to a unique *word* or *term* that appears in the collection. The list corresponding to each term points to the different docids where the term appears. Variants of this data structure are used to support various ways to retrieve the documents relevant to a query, mainly *Ranked retrieval*, *Boolean retrieval*, and *Full-text retrieval* [5, 29]

The goal of ranked retrieval is to retrieve the documents considered most "relevant" to the query, under some criterion. In the popular vector-space model, documents are represented as vectors $\overrightarrow{d_i} = \langle w(t_1, d_i), w(t_2, d_i), \ldots, w(t_V, d_i)\rangle$, where $\{t_1, t_2, \ldots, t_V\}$ is the *vocabulary* of the distinct terms in the collection, and the value $w(t_j, d_i)$ in each dimension corresponds to the relevance of the term t_j in document d_i. The classical interpretation of a query is the so-called *bag-of-words* model, where documents are scored according to the sum of the weights of the individual query words inside them, $w(q, d) = \sum_{t \in q} w(q, d)$. Answering such queries implies processing the lists of each query word and retrieving the documents with highest score. For these kind of queries operations the lists are preferably sorted in *descending weight* order [4, 23, 29].

There are many relevance formulas, being variants of the *tf-idf* model the most widely used. In the basic formula, the weight of a document d for a term t

* Partially funded by Fondecyt grant 1-110066 and by the Conicyt PhD Scholarship Program, Chile.

is given by $w(t,d) = tf_{t,d} \times idf_t$. Here $tf_{t,d}$ is the *term frequency* of t in d, that is, the number of times t occurs in d. The second term is $idf_t = \log \frac{D}{df_t}$, where df_t is the *document frequency*, that is, number of documents where the term t appears. The variables idf_t or df_t can be stored in the vocabulary as they depend only on t, whereas the $tf_{t,d}$ values are stored together with each document d in the corresponding inverted list for term t.

Boolean retrieval, instead, retrieves all the documents where the query terms appear. If the query is a single term ($|q| = 1$), the retrieval process just fetches the list of the term. Multi-word queries are interpreted using (a variant of) the Boolean model. For example, for disjunctive queries (OR) all the corresponding lists have to be fetched and merged. In conjunctive queries (AND), the lists must be intersected. With the advent of large Web search engines where precision is a more serious concern than recall, intersection queries have become more popular, as witnessed by the amount of recent research on this problem [7, 8, 14, 24]. For Boolean queries, the lists are preferably sorted in *ascending docid* order.

A popular combination of the above queries are *Ranked AND* queries, where we must retrieve the highest ranked documents among those containing all the terms. For these queries, both of the above orderings may be competitive, and there are also special inverted list formats to support them [15].

Finally, full-text retrieval aims at finding the exact text positions where the query appears, and is useful for example to display snippets around occurrences and solve phrase queries. The inverted indexes to solve them are called *positional* and are larger than the previous ones, as the lists have an entry for every occurrence of every term. Full-text retrieval is out of the scope of this paper.

While traditionally the lists of the terms were stored on disk, a recent trend triggered by the availability of large main memories is to store the whole inverted index in the main memory of one or several machines [12,25,26]. In the secondary memory context, reducing index space was a mean to save disk space and reduce transfer time. In the more modern context, saving space is still very important in order to reduce the number of machines needed to hold the index, their use of energy, and the amount of communication. In the case of a single machine, saving space allows us to handle larger collections in main memory. This is especially important in limited-memory devices such as hand-helds, and also in general because the disk is orders of magnitude slower than main memory.

Compressing the inverted index, while supporting different types of retrieval, has been an active research topic for decades [22, 25, 27, 29]. Most compression techniques exploit the fact that the inverted lists are sorted somehow, by storing differences between consecutive values rather than absolute ones. The direct access needed by the query algorithms, especially intersections, is supported by various sampling mechanisms [12, 24].

Most IR systems support both types of retrieval, ranked and Boolean, and combinations. Since each type is favored by a different sorting of the inverted lists, and doubling the space is undesirable, one must choose one ordering in detriment of the queries of the other type. Some schemes enrich an inverted index stored in one order with data to speed up queries of the other type [15].

Recently, Navarro and Puglisi [21] proposed a new compact in-memory representation of the inverted index using *wavelet trees* [19]. This representation allows one to handle both types of retrieval, ranked and Boolean, via a *dual sorting* of the inverted lists. That is, the representation can simulate that lists are sorted by docid (useful for Boolean and ranked AND queries) and by term weight (useful for bag-of-words), as desired, without increasing the space. This is an important theoretical promise for inverted indexes with rich functionality and reduced space, yet its practical value remained unclear.

Our contributions are as follows: (1) We implement the dual-sorted inverted index and describe the practical considerations that have been made. (2) We compare its performance with the state of the art. (3) We demonstrate that the technique has an important practical appeal: within 10%-15% of extra space (on top of the plain collection size), which is the state-of-the-art for solving one type of query, it handles both types of queries. (4) We show that our implementation is faster than a standard docid-sorted inverted index for intersection queries. For bag-of-word queries, it is slower than a frequency-sorted index. However, a frequency-sorted index alone is not competitive for conjunctive queries. If both types of queries have to be supported, the sum of a docid-sorted and a frequency-sorted index doubles the space of the dual-sorted index.

2 Related Work

2.1 Query Processing Strategies

Boolean queries aim at retrieving all the documents where some (OR) or all (AND) of the query words appear. Ranked or top-k queries, instead, retrieve only the k most "relevant" documents. When combined with OR-queries, the result is called a *bag-of-words* query, where all the documents containing some query word qualify. When combined with AND-queries, the result is called a *weighted conjunctive* or *ranked AND* query, where we look for the highest ranked among the documents containing all the query words. An IR system may have to provide support for all, or most, of these queries simultaneously.

Depending on how we traverse the lists to solve queries, the algorithms can be categorized as Term-at-a-time (TAAT) or Document-at-a-time (DAAT) [23].

Term-At-A-Time (TAAT) Query Processing. This technique is mostly preferred for bag-of-word queries [15]. The query is processed term by term. For each term posting, we choose the candidate documents that could be among the top-k most relevant ones for the given query. A set of active candidate documents is maintained, while their weights are increased by the contribution of each successive term. At the end, the top-k are chosen among the candidates.

Persin Algorithm [22]. This is one of the most famous TAAT query processing algorithms. The idea is to solve bag-of-word queries without scanning all of the lists. The algorithm requires the lists of each term to be sorted by decreasing weight. While the algorithm is described for the *tf-idf* model, it can be easily adapted to many variants, with lists sorted by so-called impact [2].

The first step of the algorithm creates an accumulator acc_d for each document d in the dataset (in practice, one can dynamically add a new accumulator when a candidate document is found). The second step will store into the corresponding accumulators acc_d the weights of the documents of the shortest among the lists of the query terms, that is, the one with the highest idf_t. The third step processes the rest of the lists in increasing length order, where the weight of each document is accumulated in its corresponding acc_d. In order to avoid processing the whole lists, they enforce a minimum threshold such that if the $w(t, d)$ values fall below it, the list is abandoned. Since the longer lists have a lower idf_t multiplying the term frequencies, it turns out that a lower proportion of the longer lists is traversed. They also apply a stricter threshold that limits the insertion of new documents as candidates. These thresholds provide a time/quality tradeoff.

Document-At-A-Time (DAAT) Query Processing. This scheme is convenient for the other queries, Boolean and ranked AND. All the $|q|$ lists are traversed in parallel, looking for the same document in all of them. Posting lists are sorted by increasing docid. Each posting has a pointer to the current document that is being evaluated. Once a document is processed, the pointers move forward. For Boolean disjunctive queries (OR), one moves to the closest document across all the lists, as all the documents have to be processed. The problem is more interesting for conjunctive queries (AND and ranked AND), where there are various techniques to try to skip as much as possible from the lists [7, 8, 14, 24]. DAAT techniques are be very fast on conjunctive queries, and are considered the state of the art. Ranked disjunctive queries, however, are not efficiently implemented on this representation [15].

Block-Max Index [15]. This is a special-purpose structure for ranked AND queries. It sorts the lists by increasing docid, but cuts the lists into blocks and stores the maximum relevance score for each block. This enables them to skip large parts of the lists whose maximum possible contribution is very low by comparing the the contribution of a block with a threshold θ. This solution led to considerable performance gains over other approaches [10, 26]. Needless to say, the same data structure solves efficiently Boolean queries using DAAT traversals.

2.2 Data Structures for Inverted Lists

A list $\langle p_1, p_2, p_3, \ldots p_i \rangle$ is usually represented as a sequence of d-gaps $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_l - p_{l-i} \rangle$, and uses a variable-length encoding for these differences, for example δ-codes, γ-codes or Rice/Golomb codes [27], the latter usually giving the best compression. Recent proposals make use of byte-aligned [12,25] or word-aligned [1,28] codes, which are faster at decoding while losing little compression. Extracting a single list or merging lists is done optimally by traversing the lists from the beginning, but intersections can be done much faster if random access to the sequences is possible. A typical solution to provide random access is to perform a sampling of the sequences, by storing the absolute values and pointers. The result is a two-level structure: the first contains the sampled values

and the second stores the encoded sequence itself. For example, Culpepper and Moffat [12] extract a sample every $p' = p \log l$ values from the compressed list, where p is a space/time tradeoff parameter (our logarithms are in base 2). Direct access requires a binary search on the samples list plus the decompression of a within-samples block. Sanders and Transier [24], instead, sample regularly the domain values: all the values differing only in their $p = \log(B/l)$ lowest bits (for a parameter B, typically 8), form a block. The advantages are that binary searches on the top structure, and storing absolute values in it, are not necessary. A disadvantage is that the blocks are of varying length and more values might have to be scanned on average for a given number of samples.

Various list intersection algorithms exist [7, 8, 14, 24], some of them tailored to specific representations. In general, the best approach to intersect various lists is the so-called *set-vs-set (svs)* [8]: the two shortests lists are intersected, then the result is intersected with the next shortest list, and so on. For a pair of lists, one typically searches the longer list for the values of the shorter one.

When list are sorted by decreasing weight (for bag-of-word queries), the differential compression of docids is not possible, in principle. Instead, term weights can be stored differentially. When storing *tf* values, one can take advantage of the fact that long runs of equal *tf* values (typically low ones) are frequent, and sort the corresponding docids increasingly, to encode them differentially [5, 29].

2.3 Wavelet Trees versus Inverted Indexes

The *wavelet tree* [19] is a data structure that stores a sequence S in a particular (compressed) form that enables various queries over the sequence. Wavelet trees have been applied to various IR problems, leading to diverse solutions. Brisaboa et al. [9] used a variant of wavelet trees to represent the sequence S of words in a text collection. As a result, they represent the collection in compressed form and in addition simulate a positional inverted index. Arroyuelo et al. [3] extended the representation to support Boolean document retrieval operations (single-word, AND and OR queries). The interest of these structures is that they can operate within very little extra space on top of that of the compressed text (say, 3%, as opposed to 15%–20% of inverted indexes). Within this niche they are unbeaten, but in absolute terms their query times are orders of magnitude slower than using explicit inverted indexes. On the other extreme, Culpepper et al. [13] use a wavelet tree to represent the sequence of documents to which each text position belongs, after lexicographically sorting the suffixes starting at those positions. This wavelet tree uses much more space than classical inverted indexes, say 200%-400% of the text size, but it can search for arbitrary substrings and handle non-natural-language texts. If restricted to indexing only words, the space would drop to about 30%–60% and still would be able to solve some complex queries such as ranking phrases and do stemming on the fly and prefix searches, but it would lose in time (and space) to inverted indexes on the typical Boolean and ranked retrieval queries.

Thus, wavelet trees have been used as a *replacement* of the inverted index data structure, leading to structures that excell in other niches. In this work

we use them to *emulate* inverted indexes, so as to compete with them within the same space range, and for the same queries inverted indexes are designed to solve. We present the basic concepts in the next section.

3 Engineering Dual-Sorted Inverted Lists

Our main data structure is a wavelet tree [19] storing a sequence $L[1,n]$ containing symbols from an alphabet $[1,D]$. In its basic form, it uses $n \log D(1+o(1))$ bits for representing L, while supporting a set of useful operations. The structure is a complete balanced binary tree with D leaves labeled with the different symbols appearing on L, in increasing order. For any internal node v of the wavelet tree, let L_v be a subsequence of L containing only the symbols on the leaves in the subtree with root v. Every node v stores not L_v but rather a bitvector B_v with $|L_v|$ bits, where $B_v[i] = 1$ if symbol $L_v[i]$ appears below the right child of v, and $B_v[i] = 0$ if the symbol appears below the left child. All bitvectors are processed to handle binary *rank* and *select* queries in $O(1)$ time [20]: $rank_b(B,i)$ is the number of occurrences of bit b in $B[1,i]$, and $select_b(B,j)$ is the position of the j-th b in B. The following primitives, supported by wavelet trees, are relevant for this work [17, 21].

Retrieve all values a range $L[i,j]$: We start from the root node v and map the range $B_v[i,j]$ to the left and to the right child. The new interval is $[i,j] \leftarrow [rank_b(B_v, i-1)+1, rank_b(B_v,j)]$, where $b=0$ when descending to the left child, and $b=1$ on the right child. Nodes where the interval becomes empty are abandoned. Whenever we reach a leaf labeled d, we know there are $j-i+1$ occurrences of d in the original range. Note the symbols and their occurrences are delivered in increasing order. The time complexity is $O(m \log(D/m))$, where m is the number of distinct symbols reported.

Retrieve the k-th value in a range $L[i,j]$: We start from the root node v and the range $B_v[i,j]$. If $k' \leftarrow rank_0(B_v,j) - rank_0(B_v,i-1) \leq k$, we descend to the left child, mapping the range as above (with b=0). Else, we descend to the right child, mapping with b=1 and setting $k \leftarrow k-k'$. When we arrive at a leaf d, then the k-th value in the range is a d. The time is $O(\log D)$.

Navarro and Puglisi [21] propose the following representation of inverted lists. Regard the list of absolute docids associated to term t as a sequence L_t over an alphabet $[1,D]$, where the docids are sorted by decreasing weight. Concatenate all the lists L_t into a unique sequence $L[1,n]$, which is represented with a wavelet tree, and mark the starting positions of the lists in a bitvector $s[1,n]$. The weights (actually, the $tf_{t,d}$ values) are stored in a sequence $W[1,n]$ aligned to L. They use a theoretically appealing (but practically doomed) representation of W.

We make the following practical considerations to implement this data structure. The wavelet tree is represented using a pointerless version [11] because D is considerably large compared to n. As L is not expected to be compressible, it is better to strive for time efficiency and represent the bitmaps of the wavelet tree using a fast implementation [18] that uses 37.5% extra space on top of the

bitmap. Bitmap s is replaced by V pointers from the terms t to the starting positions of the lists L_t in L. The weights within the range of each list L_t, which are decreasing, are represented in differential form (using Rice codes). Absolute values are sampled every K positions, where K is a space/time tradeoff parameter. In fact, as there are many runs of equal weights, we store only the nonzero differences, and store a bitmap $W'[1,n]$ indicating which differences are nonzero. So we actually access position $W[rank_1(W',i)]$ instead of $W[i]$, using a representation that requires 5% on top of the bitmap W' to support $rank$ [18].

The representation supports a wealth of traditional and not so traditional operations [21]. Next we describe how we use it to solve the queries of interest in this paper, which are the basic ones in IR.

Bag-of-words (ranked OR): We implement Persin et al.'s algorithm [22]. We use the primitive just described to extract a range $L[i,j]$ in order to obtain the whole shortest list, and to extract a prefix of the next lists. The extension of the prefix to be extracted (according to the threshold given on $w(t,d)$ given by the algorithm) is computed by exponential search on W. Note that the primitive obtains the documents of the lists sorted by docid, which makes it convenient to merge into our set of accumulators acc_d if they are also sorted by docid. Note that W stores *tf* values; these are multiplied by idf_t before accumulating them.

Weighted conjunctive queries (Ranked AND): We find the $|q|$ intervals $[s_t, e_t]$ of the query words using the pointers from the vocabulary to the inverted lists L_t, sort them by increasing lengths, and use the primitive for tracking ranges. We track all the $|q|$ ranges simultaneously, stopping as soon as any of those becomes empty. The leaves arrived at correspond to documents d that participate in the intersection. Their term frequencies are available (as $j-i+1$, as described in the primitive), so we can immediately compute the document score. We retain the k highest scoring documents.

Boolean queries: The data structure supports boolean conjunctive and disjunctive queries by easily adapting the previous algorithms.

4 Experiments

Setup and implementations. We implemented a document-sorted inverted index (*IX Doc Sort*) and a frequency-sorted inverted index (*IX Freq Sort*) using Rice encoding for compressing d-gaps (i.e., docid gaps for IX Doc Sort and *tf* gaps for IX Freq Sort). Both inverted indexes allow random access by storing sampled absolute values at fixed intervals l (the other data, i.e., frequencies of IX Doc Sort and docids for IX Doc Sort, are stored in plain form). On IX Doc Sort, intersections are done using *svs* for AND queries, whereas OR queries are done by merging lists. Ranked AND queries are implemented with a fast postprocessing after the AND query. On IX Freq Sort, Persin's algorithm is used for bag-of-word queries. We set Persin's algorithms parameters for processing an existing term to 1.2 and to process a new term to 1.2. If we use one of these indexes to run the

queries supported by the other, the times are very high, comparable to those of a Boolean OR query (i.e., a few queries per second are processed).

We also implemented *Block-Max* [15] on our docid-sorted structure, using blocks of size l. This solves Boolean queries just as IX Doc Sort, but for ranked AND queries it skips blocks whose max-score is too low. Bag-of-word queries are solved by traversing the shortest list in decreasing score order (using max-score to guide the search), and finding the documents in the other lists (using max-score to avoid decompressing uninteresting blocks). As the precise criterion is not specified in the original paper [15], we use Persin's threshold.

As an external implementation to compare we chose *Zettair*, a publicly available and open-source search engine engineered for efficiency (`www.seg.rmit.edu.au/zettair`). Zettair supports both disjunctive and conjunctive queries and implements the *tf-idf* ranking formula (among others). It also implements two index organizations: docid-sorted and impact-sorted. We will show the results achieved with the best organization, although docid-sorting is generally better and impact-sorting is only slightly better for bag-of-word queries. All implementations have been set to run exclusively on main memory.

Our machine is an Intel(r) Xeon(r) model E5620 running at 2.40GHz with 96 Gb of RAM and 12288 Kb of cache. The operating system is Linux with kernel 2.6.32-41 64 bits and we used GCC version 4.4.3 with -O3 optimization.

Experimental data. We used a random sample of the TREC GOV2 Collection (`http://trec.nist.gov`) containing 165GB of text and 14,415,482 documents, having V = 45,092,117 different words. We also obtained all English articles from Wikipedia (`http://www.wikipedia.com`) retrieved on August 2011. The English Wikipedia corpus has about 33.2GB of text, distributed in 11,846,040 documents, with V = 19,231,312 different terms. Both collections have been parsed using Porter's stemming algorithm. For both collections, we constructed query logs based on the efficiency queries from TREC with distinct amount of terms, ranging from $q = 2$ to 10. For every q value, we filtered 2,000 queries that appeared in at least 1,000 documents.

Time performance. For timing results, we set the space/time tradeoff parameter to $m = 16$ in Dualsorted and to $l = 16$ in Block-Max, IX Freq Sort, and IX Doc Sort. Figure 1 shows bag-of-word queries per second solved by the different indexes, retrieving the top-20 and top-1000 documents. As expected, IX Freq Sort is the fastest method for this query, but our implementation of Dualsorted is not too far away, and in turn it performs better than our implementation of Block-Max. Zettair is the slowest alternative, as it computes as many top-ranked document as memory permits (thus it is more competitive for $k = 1000$).

Figure 2 shows the times for conjunctive queries, returning the top-20 results. While Zettair's performance worsens as more words are interesected, IX Doc Sort stays similar or improves (since the shortest list is shorter). Block-max is always close to, and slightly better than, IX Doc Sort. Interestingly, Dualsorted is the fastest alternative. It first improves (with more words, some interval becomes empty sooner) and then finally degrades (as we track more intervals through the

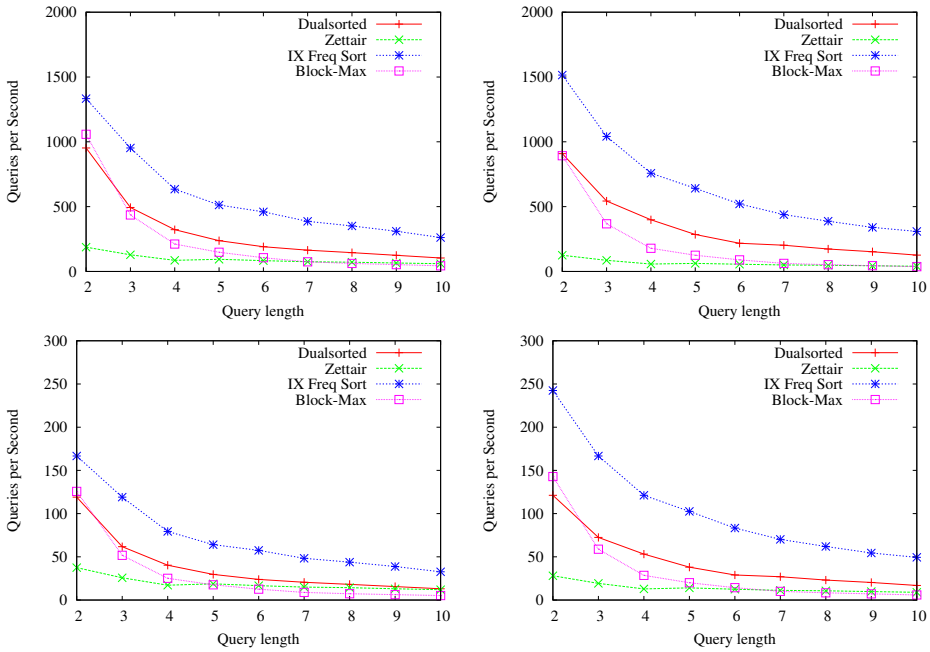

Fig. 1. Queries per second for top-20 (top) and top-1000 (bottom) bag-of-word queries (higher is better). On the left, on the TREC dataset; on the right, on Wikipedia.

tree). These queries are much less affected by a larger k: for top-1000 results, the throughput is about one third; the relative performances do not change.

We omit for lack of space Boolean AND queries, which perform very similarly to ranked AND (with large k), and OR queries, which are very slow for all the structures (1–4 queries per second), and not very interesting for IR.

Space/time tradeoffs. To evaluate space usage we vary the compression parameters in each method. For Dualsorted, we tried sampling values $m = 16, 32, 64, 128$. For IX indexes and Block-Max, we tried values $l = 16, 32, 64, 128$ for the list sampling parameter. We consider the time to solve 3-word ranked queries (conjunctive and disjunctive), which are representative of the other times. In our results, the *2x InvList* represents the union of the IX Doc Sorted and IX Freq Sorted inverted lists, where we display the best of the two for ranked conjunctive and disjuncitve queries.

Figure 3 shows the space used, as a fraction of the dataset size, versus queries per second, for the different indexes on bag-of-word and ranked AND queries, returning the top-20 results. On bag-of-word queries, Dualsorted achieves better results within its space consumption range, while Block-Max is the closest alternative. The IX index can be faster but it needs significantly more space (e.g., roughly, it needs twice the space to be 50% faster). On ranked AND queries,

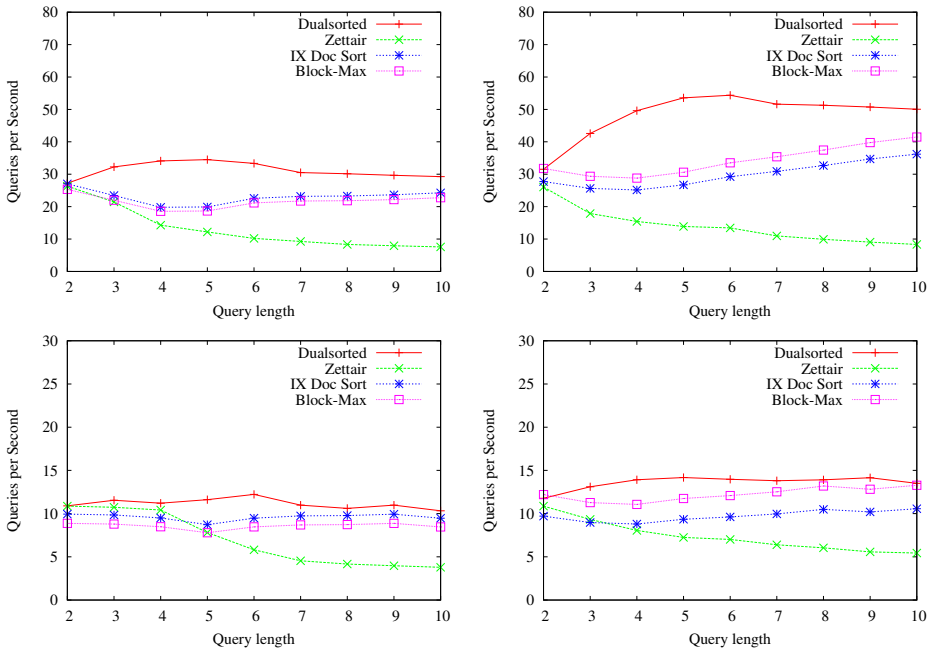

Fig. 2. Queries per second for top-20 (top) and top-1000 (bottom) ranked AND queries (higher is better). On the left, on the TREC dataset; on the right, on Wikipedia.

Dualsorted is not only the least space-consuming index, but also the fastest, dominating all the space/time tradeoff.

5 Conclusions and Future Work

We have demonostrated that an engineered implementation of dual-sorted inverted indexes [21] is an appealing data structure for conjunctive queries. The native list intersection it supports turns out to be faster than state-of-the-art implementations based on docid-sorted inverted indexes, for (ranked or Boolean) conjunctive queries. The dual-sorted index also supports bag-of-word queries. Despite in this case it is slower than a frequency-sorted index, the performance is still acceptable, and the index does not require further space. Thus, if we need to solve both kinds of queries, the dual-sorted index requires about half the space of the sum of a docid-sorted and a frequency-sorted index.

Adding both indexes is, of course, a simple alternative to settle a first baseline. In the future we plan to compare with other space/time tradeoffs, such as storing short lists (which are the most) in only one order, and/or store just a prefix of the frequency-sorted lists for the long ones.

The dual-sorted index intersection algorithm is easily extended to other variants of conjunctive queries like WAND queries [10]. It also adapts easily to some

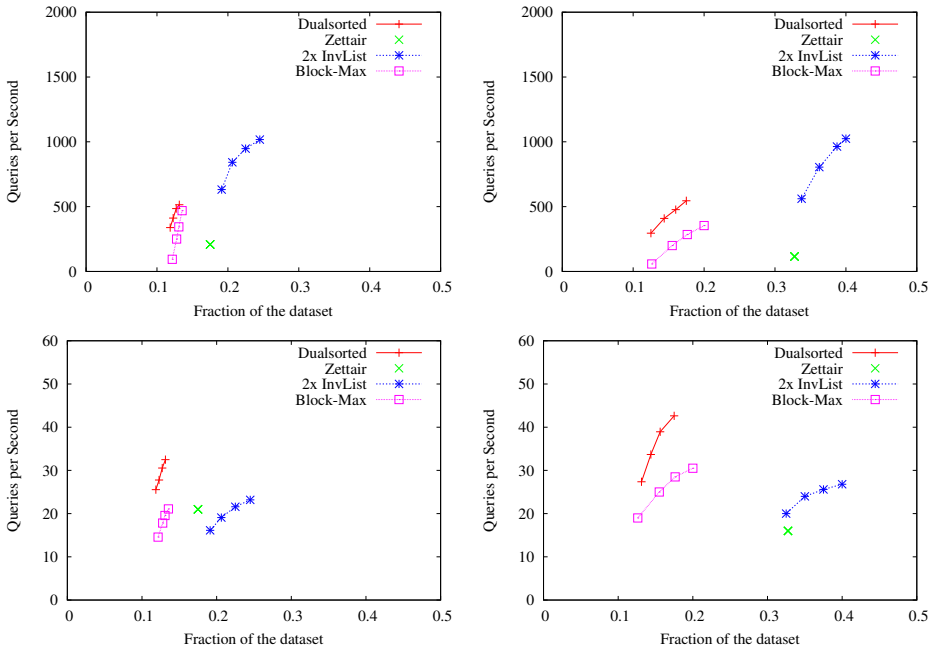

Fig. 3. The fraction of the dataset space (x-axis) used by the different indexes (leftward is better) compared to the queries per second solved (higher is better) for top-20 queries of 3 terms. On the left, the TREC dataset; on the right, the Wikipedia dataset. On top, bag-of-word queries; on the bottom, ranked AND queries.

variants of the *tf-idf* formula, yet others are more challenging. For example, a popular measure is Okapi BM25 [16]. This measure modifies the usual *tf* in a way that depends on the length of the document, so that the weight $w(t,d)$ is a real number. Even sorting the lists by decreasing $w(t,d)$, the values are much harder to compress than *tf*, which are integer values, most of them small. Reducing precision [2] is a promising direction we are pursuing.

References

1. V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.
2. V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, pages 372–379, 2006.
3. D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. SPIRE*, pages 43–54, 2010.
4. D. Metzler B. Croft and T.Strohman. *Search Engines: Information Retrieval in Practice*. Pearson Education, 2009.
5. R. Baeza-Yates, A. Moffat, and G. Navarro. *Searching Large Text Collections*, pages 195–244. Kluwer Academic Publishers, 2002.

6. R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.
7. R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. SPIRE*, pages 13–24, 2005.
8. J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM J. Exp. Alg.*, 14:art. 7, 2009.
9. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.
10. A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.
11. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. SPIRE*, 2008.
12. J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. SPIRE*, pages 137–148, 2007.
13. S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205 (part II), 2010.
14. E. Demaine and I. Munro. Adaptive set intersections, unions, and differences. In *Proc. SODA*, pages 743–752, 2000.
15. S. Ding and T.Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.
16. S. Robertson et al. Okapi at TREC-3. In *Proc. 3rd TREC*, pages 109–126, 1994.
17. T. Gagie, S.J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. SPIRE*, pages 1–6, 2009.
18. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. Posters WEA*, pages 27–38, 2005.
19. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
20. I. Munro. Tables. In *Proc. FSTTCS*, pages 37–42, 1996.
21. G. Navarro and S. Puglisi. Dual-sorted inverted lists. In *Proc. SPIRE*, pages 309–321, 2010.
22. M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Amer. Soc. Inf. Sci.*, 47(10):749–764, 1996.
23. C. Clarke S. Büttcher and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.
24. P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. ALENEX*, 2007.
25. F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, pages 222–229, 2002.
26. T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, pages 175–182, 2007.
27. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.
28. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
29. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):art. 6, 2006.