Dual-Sorted Inverted Lists *

Gonzalo Navarro[1] and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Chile, Santiago, Chile.
gnavarro@dcc.uchile.cl
[2] School of Comp. Sci. & Inf. Tech., Royal Melbourne Institute of Technology, Melbourne, Australia. simon.puglisi@rmit.edu.au

Abstract. Several IR tasks rely, to achieve high efficiency, on a single pervasive data structure called the *inverted index*. This is a mapping from the terms in a text collection to the documents where they appear, plus some supplementary data. Different orderings in the list of documents associated to a term, and different supplementary data, fit widely different IR tasks. Index designers have to choose the right order for one such task, rendering the index difficult to use for others.
In this paper we introduce a general technique, based on *wavelet trees*, to maintain a single data structure that offers the combined functionality of two independent orderings for an inverted index, with competitive efficiency and within the space of one *compressed* inverted index. We show in particular that the technique allows combining an ordering by decreasing term frequency (useful for ranked document retrieval) with an ordering by increasing document identifier (useful for phrase and Boolean queries). We show that we can support not only the primitives required by the different search paradigms (e.g., in order to implement any intersection algorithm on top of our data structure), but also that the data structure offers novel ways of carrying out many operations of interest, including space-free treatment of stemming and hierarchical documents.

1 Introduction

The last decade has been witness to tremendous progress in the field of compact data strucutres. These data structures mimic the operations of their classical counterparts within much less space and sometimes, surprisingly, offer much wider functionality. Recently, several authors have brought compact data structures to bear on problems in Information Retrieval (IR), in particular ranked document retrieval [14, 20]. Although quite different in their details, the common vision of these works is to use breakthroughs in compressed pattern matching as an efficient algorithmic base on which the more sophisticated operations required by IR systems can be built. Our work in this invited paper is complementary to these efforts, applying compact data structures to gain a new perspective on a tool already widely adopted in IR: the inverted index.

* Funded in part by Fondecyt Grant 1-080019, Chile (first author) and by the Australian Research Council (second author).

Inverted indexes are an old and simple data structure, yet one of the most successful in IR. They play a central role in any book on the topic [6, 31, 12, 22, 11], and are also at the heart of most modern Web search engines.

Given a *text collection* regarded as a set of *documents*, an inverted index is an array of *lists*. Each array entry corresponds to a different *word* or *term* of the collection, and its list points to the documents where that word appears in the text collection. The set of different words is called the *vocabulary*. Empirical laws well accepted in IR [19] establish that the vocabulary is much smaller than the collection size n, more precisely of size $O(n^\beta)$, for some constant $0 < \beta < 1$ that depends on the text type.

Two main variants of inverted indexes exist [5, 35]. *Ranked retrieval* is aimed at retrieving documents which are "relevant" to a query, under some criterion. Documents are regarded as vectors, where terms are the dimensions, and the values of the vectors correspond to the relevance of the terms in the documents. The lists point to the documents where each term appears, storing also the weight of the term in that document (i.e., the coordinate value). The query is seen as a set of words, so that retrieval consists of processing the lists of the query words and finding the documents which, considering the weights the query terms have in the document, are predicted to be relevant. Query processing usually involves somehow merging the involved lists, so that documents can be assigned the combined weights over the different terms. Algorithms and different data organizations for this type of query have been intensively studied [25, 31, 35, 3, 30]. List entries are usually sorted into order of *descending weight* of the term in the documents.

The second variant is the inverted index for so-called *full-text retrieval* (also known as *boolean retrieval*). These simply find all the documents where the query terms appear. The lists point to the documents where each term appears, usually in *increasing document* order. Queries can be single words, in which case the retrieval consists simply of fetching the list of the word; or disjunctive queries, where one has to fetch the lists of all the query words and merge the sorted lists; or conjunctive queries, where one has to intersect the lists. While intersection can be done also by scanning all the lists in synchronization, it is usually the case that some lists are much shorter than the others [34], and so faster intersection algorithms are possible. These algorithms are especially relevant when many words have to be intersected.

Intersection queries have become extremely popular as Google-like default policies to handle multiword queries. Another important query where intersection is essential is the phrase query, where intersecting the documents where the words appear is the first step. The amount of recent research on intersection of inverted lists witnesses the importance of the problem [15, 8, 4, 7, 27, 28, 13, 9]. In particular, in-memory algorithms have received much attention recently, as large main memories and distributed systems make it feasible to hold the inverted index entirely in RAM.

Needless to say, space is an issue in inverted indexes, especially when combined with the goal of operating in main memory. Much research has been carried

out on compressing inverted lists [31, 24, 35, 13], and on the interaction of various compressed representation with different query algorithms, including list intersections. Most of the list compression algorithms for full-text indexes rely on the fact that the document identifiers are increasing, and that the differences between consecutive entries are smaller on the longer lists. The differences are thus represented with encodings that favor small numbers [31, 35]. Random access is supported by storing sampled absolute values. For lists sorted by decreasing weights, these techniques can still be adapted, by considering that most documents in a list have small weight values, and within the same weight one can still sort the documents by increasing identifier.

A problem with the current state of the art is that a serious IR system must support both types of retrieval: ranked and full-text. Yet, to maintain reasonable space efficiency, the list must be ordered either by decreasing weights or by increasing document number, but not both. Hence one type of search will be significantly slower than the other, if affordable at all.

In this paper we introduce a data structure that permits, within the same space required for a single compressed inverted index, retrieving the list of documents for any term in either decreasing-weight or increasing-identifier order, thus supporting both types of retrieval. Moreover, we can efficiently support the operations needed to implement any of the intersection algorithms, namely: retrieve the i-th element of a list, retrieve the first element larger than x, retrieve the next element, and several more complex ones. In addition, our structure offers novel ways of carrying out several operations of interest. These include, among others, the support for stemming and for structured document retrieval without any extra space cost. Indeed, the data structure can be generalized to support a combination of any two orderings, not only the two most popular ones.

2 Related work

2.1 Intersection algorithms for inverted lists

The intersection of two inverted lists can be done in a merge-wise fashion (which is the best choice if both lists are of similar length), or using a set-versus-set (*svs*) approach where the longer list is searched for each of the elements of the shortest, to check if they should appear in the result. Either binary or exponential (also called galloping or doubling) search are typically used for *svs*. The latter checks the list at positions $i + 2^j$ for increasing j, to find an element known to be after position i (but probably close).

Algorithm *bys* [7] is based on binary searching the longer list N for the median of the smallest list M. If the median is found, it is added to the result set. Then the algorithm proceeds recursively on the left and right parts of each list. At each new step the longest sublist is searched for the median of the shortest sublist. It has been shown that *bys* performs about the same number of comparisons as *svs* with binary search. As expected, both *svs* and *bys* improve upon the *merge* algorithm when $|N| >> |M|$ (actually from $|N| \approx 20|M|$).

Multiple lists can be intersected using any pairwise *svs* approach (iteratively intersecting the two shortest lists, and then intersecting the result against the next shortest one, and so on). Other algorithms are based on choosing the first element of the smallest list as an *eliminator* that is searched for in the other lists (usually keeping track of the position where the search ended). If the eliminator is found, it becomes a part of the result. In any case, a new eliminator is chosen. Barbay et al. [9] compared four multi-set intersection algorithms: *i)* a pairwise *svs*-based algorithm; *ii)* an eliminator-based algorithm [8] (called *sequential*) that chooses the eliminator cyclically among all the lists and exponentially searches for it; *iii)* a multi-set version of *bys*; and *iv)* a hybrid algorithm (called *small-adaptive*) based on *svs* and on the so-called *adaptive algorithm* [15], which at each step recomputes the list ordering according to the elements not yet processed, chooses the eliminator from the shortest list, and tries the others in order. In their experimental results [9] the simplest pairwise *svs*-based approach (coupled with exponential search) performed best.

2.2 Data structures for inverted lists

The previous algorithms require that lists can be accessed at any given element (for example those using binary or exponential search) and/or that, given a value, its smallest successor from a list can be obtained. Those needs interact with the methods employed for inverted list compression.

The compression of inverted lists usually represents each list $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ as a sequence of d-gaps $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, and uses a variable-length encoding for these differences, for example γ-codes, δ-codes or Golomb codes [31]. More recent proposals use byte-aligned [29, 10, 13] or word-aligned [2, 33] codes, which lose little compression and are faster at decoding.

Intersection of compressed inverted lists is still possible using a merge-type algorithm. However, approaches that require direct access are not possible as sequential decoding of the d-gaps values is mandatory. This problem can be overcome by sampling the sequence of codes [13, 27]. The result is a two-level structure composed of a top-level array storing the absolute values of, and pointers to, the sampled values in the sequence, and the encoded sequence itself.

Assuming $1 \leq p_1 < p_2 < \ldots < p_\ell \leq u$, Culpepper and Moffat [13] extract a sample every $k' = k \log \ell$ values[3] from the compressed list, where k is a parameter. Each of those samples and its corresponding offset in the compressed sequence is stored in the top-level array of pairs $\langle \textit{value}, \textit{offset} \rangle$ needing $\lceil \log u \rceil$ and $\lceil \log(\ell \log(u/\ell)) \rceil$ bits, respectively, while retaining random access to the top-level array. Accessing the v-th value of the compressed structure implies accessing the sample $\lceil v/k' \rceil$ and decoding at most k' codes. We call this "(a)-sampling". Results [13] show that intersection using *svs* coupled with exponential search in the samples performs just slightly worse than *svs* over uncompressed lists.

Sanders and Transier [27], instead of sampling at regular intervals of the list, propose sampling regularly at the domain values. We call this a "(b)-sampling".

[3] Our logarithms are in base 2 unless otherwise stated.

The idea is to create buckets of values identified by their most significant bits and then build a top-level array of pointers to them. Given a parameter B (typically $B = 8$), and the value $k = \lceil \log(uB/\ell) \rceil$, bucket b_i stores the values $x_j = p_j \bmod 2^k$ such that $(i-1)2^k \leq p_j < i\,2^k$. Values x_j can also be compressed (typically using variable-length encoding of d-gaps). Comparing with the previous approach [13], this structure keeps only pointers in the top-level array, and avoids the need of searching it (in sequential, binary, or exponential fashion), as $\lceil p_j/2^k \rceil$ indicates the bucket where p_j appears. In exchange, the blocks are of varying length and more values might have to be scanned on average for a given number of samples. The authors also keep track of up to where they have decompressed a given block in order to avoid repeated decompressions.

2.3 Algorithms for ranked retrieval

Persin et al. [25] proposed heuristics to solve ranked retrieval problems without scanning all of the lists, and assuming they are sorted by decreasing weight. To fix ideas we will assume, as in their work, that the weight is simply the *term frequency*, that is, the number of times the term appears in the document. This supports various *tf-idf*-like formulas, yet other weights that have been proposed (for example the so-called impacts [3]) can be acommodated as well.

In the *tf-idf* model, the final weight of a document d for a query q is $w(d) = \sum_{t \in q} tf_{t,d} \times idf_t$ summed over all the query terms t. The query retrieves the documents with highest $w(d)$. The term $tf_{t,d}$ is the term frequency of t in d, whereas $idf_t = \log \frac{D}{df_t}$, where D is the total number of documents and df_t is the number of those where t appears. While idf_t (or df_t) is stored in the vocabulary, a term's $tf_{t,d}$ values are stored together with each document d in the inverted list of each term t, and the documents d are sorted by decreasing $tf_{t,d}$ value.

The algorithm retrieves first the shortest list (i.e., with highest idf_t) and stores the documents as candidates for the final answer. Now the other lists are processed in increasing length order. The documents of each list are sought in the set of candidates, and their weight accumulated if found; otherwise they are inserted as new candidates. There is a threshold for continuing processing each list: if the $tf_{t,d}$ values fall below it, the list is abandoned (see Ahn et al. [1] and references therein). There is also a stricter threshold for inserting new elements as candidates. These heuristic thresholds provide a time/quality tradeoff.

3 Wavelet trees

Let $L[1, N]$ be a sequence of N symbols, where each symbol is in the range $[1, D]$. The wavelet tree of L is a perfect binary tree with D leaves. The leaves are labeled left-to-right with the symbols $[1, D]$ in increasing order. For a given internal node v of the tree, let S_v be the subsequence of L consisting of only the symbols on the leaves in the subtree rooted at v. We store at v a bitvector B_v of $|S_v|$ bits, setting $B_v[i] = 1$ if symbol $S_v[i]$ is appears below the right child of v, and $B_v[i] = 0$ otherwise. Note that S_v is not actually stored, only B_v. Finally, each bitvector

B_v is preprocessed for $O(1)$ rank and select queries [26]: $rank_b(B_v, i)$ returns the number of occurrences of bit b in $B_v[1, i]$; and $select_b(B_v, i)$ returns the position in B_v of the ith occurence of bit b. As we shall see, this preprocessing allows for efficient navigation of the tree when resolving certain range queries on L.

The wavelet tree was originally designed [17] to allow accessing any $S[i]$, as well as computing queries $rank_d(L, i)$ and $select_d(L, i)$ on L for any value d, all in $O(\log D)$ time.

4 Our data representation

Let D be the total number of documents in the collection and V the number of different terms. Let $L_t[1, df_t]$ be the list of document identifiers in which term t appears, in decreasing *tf* order. Let $N = \sum_t df_t$ be the total number of occurrences of *distinct* terms in the documents[4], and $n = \sum_{t,d} tf_{t,d}$ be the total length, in words, of the collection. (Thus $D \leq N \leq \min(DV, n)$.) Finally, let $|q|$ be the number of terms in query q.

We propose to concatenate all the lists L_t into a unique list $L[1, N]$, and store for each term t the starting position s_t of list L_t within L. The sequence L of document identifiers is then represented with a wavelet tree.

The *tf* values themselves are stored in differential and run-length compressed form in a separate sequence. More precisely, we mark the v_t different $tf_{t,d}$ values of each list in a bitmap $T_t[1, m_t]$, where $m_t = \max_d tf_{t,d}$, and the points in $L_t[1, df_t]$ where value $tf_{d,t}$ changes, in a bitmap $R_t[1, df_t]$. With T_t and R_t preprocessed for *rank* and *select* queries we can obtain $tf_{t,L_t[i]} = select_1(T_t, v_t - rank_1(R_t, i) + 1)$.

Finally, the s_t sequence is represented using a bitmap $S[1, N]$, also preprocessed for *rank* and *select* queries. Thus $s_t = select_1(S, t)$, and also $rank_1(S, i)$ tells the list $L[i]$ belongs to.

The analysis of wavelet trees [17, 23] shows that the space occupied by that of L is $NH_0(L) + o(N \log D)$ bits. Here $NH_0(L) = \sum_d dt_d \log \frac{N}{dt_d} \leq N \log D$, where dt_d is the number of distinct terms in document d. The classical differential encoding of inverted files produces a set of N numbers. If they are sorted by increasing document identifier, these numbers can be represented using $\sum_t df_t \log \frac{N}{df_t} \leq N \log V$ bits plus lower-order terms, by using Elias δ-encoding. If, however, they are sorted by decreasing *tf*, the analysis is not so clean.

In general the measures are not comparable, yet we remind that our wavelet tree representation will offer the combined functionality of *both* inverted indexes, and more.

The other structures are the *tf* and the s_t values. The former is encoded with T_t and R_t, which are compressible as they have only v_t bits set. We use a bitmap representation [18, BSGAP, Section 4.3] supporting *rank* and *select* in time $O(\log v_t)$ and requiring $v_t \log \frac{m_t}{v_t} + O(v_t \log \log \frac{m_t}{v_t})$ bits for T_t and $v_t \log \frac{df_t}{v_t} +$

[4] $N = \sum_t df_t$ counts each distinct term once for each document it appears in. This is also the total length of the inverted lists.

$O(df_t \log\log \frac{df_t}{v_t})$ for R_t. This is asymptotically similar to the space needed to represent, in a traditional *tf*-sorted index, each new $tf_{t,d}$ value and the number of entries that share it. The s_t values are represented so that they support constant-time *rank* and *select* [26], requiring $V \log \frac{N}{V} + O(V) + o(N)$ bits, which is less than the usual pointers from the vocabulary to the list of each term. In the worst case the bitmaps add up to $O(N \log \frac{N}{V})$ bits and the time to compute *tf* is $O(\log D)$.

Before considering the classical and extended operations that can be carried out with our data structure, let us raise a couple of issues:

1. *Stemming* is a useful tool to enhance recall [21, 32]. A way to provide it is by stemming the terms directly during the parsing, yet in this case the index is unable to provide at the same time non-stemmed searching. One can of course index the stemmed and non-stemmed occurrence of each term, thus increasing the space. We will be able to provide stemmed retrieval without any extra space. All we require is that all the variants of the same stemmed word be contiguous in the vocabulary (this is in many cases automatic as stemmed terms share the same root, or prefix).
2. Most IR systems support a *flat* set of documents, while in XML or file systems, for example, one has a hierarchy of documents and would like to choose, at query time, which level of the hierarchy to use (e.g., to retrieve relevant sections, or relevant chapters, or relevant books), or to carry out ranked IR on a certain subtree. In a temporal (e.g., news archives) or versioned (e.g., Wikipedia) text collection, one might want to search only a range of documents. Our data structure has also support for some queries of this kind without using any extra space.

4.1 Full-text retrieval

The full-text index, rather than L_t, requires a list F_t, where the same terms are sorted by increasing document identifier. Different kinds of access operations need to be carried out on F_t. We show now how all these can be carried in $O(\log D)$ time.

Direct retrieval. First, with our wavelet tree representation of L we can find any value $F_t[i]$. This is equivalent to finding the i-th smallest value in $L[st_t, st_{t+1} - 1]$. The algorithm, for a general range $L[l, r]$, is as follows [16]. Let v be the root of the wavelet tree and B_v its bitmap. We count with $n_1 = rank_1(B_v, r) - rank_1(B_v, l-1)$ the number of 1s in $B_v[l, r]$, and with $n_0 = (r-l+1) - n_1$ the number of 0s. If $i \leq n_0$, then there are at least i values d in $L[l, r]$ belonging to the smaller half of the document identifiers, thus we continue recursively on the left child of v, with $l = rank_0(B_v, l-1) + 1$ and $r = rank_0(B_v, r)$. Otherwise, we continue on the right child, with $l = rank_1(B_v, l-1) + 1$, $r = rank_1(B_v, r)$, and $i = i - n_0$. The symbol corresponding to the leaf arrived at is the answer.

We can also extract any segment $F_t[i, i']$, in order, in time $O((i' - i + 1)(1 + \log \frac{D}{i'-i+1}))$. The algorithm is as above, going just by one branch when both i and i' choose the same, and splitting the interval into two separate searches when they do not. At worst we arrive at $i' - i + 1$ leaves of the wavelet tree, but part of the paths to these leaves must be shared. At worst, their paths become all distinct at depth $\log(i' - i + 1)$, up to which point we work on all the $O(i' - i + 1)$ different wavelet tree nodes, and after then we work on a different path, of length $\log D - \log(i' - i + 1)$, for each value.

Another useful operation is to find $F_t[j]$ after having visited $F_t[i]$, for some $j > i$. We show this can be done in amortized time proportional to $\log(j - i + 1)$. We need to store $\log D$ numbers m_ℓ, d_ℓ, and b_ℓ, where $m_0 = \infty$ and $d_1 = 0$, and the others are computed as follows when we obtain $F_t[i]$: If, at wavelet tree depth ℓ (the root being depth 1), we must go left, then $m_\ell = \min(m_{\ell-1}, d_\ell + n_0 - i)$ and $d_{\ell+1} = d_\ell$, else $m_\ell = m_{\ell-1}$ and $d_{\ell+1} = d_\ell + n_0$. Here n_0 is the value local to the node. Therefore d_ℓ counts the values skipped to the left, and m_ℓ is the maximum $j - i$ value such that the downward paths to compute $F_t[i]$ and $F_t[j]$ coincide up to depth ℓ. We also set $b_\ell = i$. Now, to compute $F_t[j]$, we consider all the ℓ values, from largest to smallest, until finding the first one such that $j - b_\ell \leq m_\ell$. From there on we recompute the downward path, resetting d_ℓ and m_ℓ accordingly and setting $b_\ell = j$.

Overall, if we carry out this operation k times, across a range $[i, i']$, the cost is $O(\log D + k(1 + \log \frac{i'-i+1}{k}))$, as there can be only $O(1)$ different paths longer than $O(\log(i' - i + 1))$ arriving at $i' - i + 1$ consecutive leaves, and considering the argument above to analyze the retrieval of $F_t[i, i']$.

Boolean operations. The most important operation for intersecting lists is to be able to find the first j such that $F_t[j] \geq d$, given d. This is usually solved with a combination of sampling and linear, exponential, or binary search. We show now that our representation supports this operation in $O(\log D)$ time.

The operation is as follows. We start at the root v, with bitmap B_v, and the interval $L[l, r]$ with $l = st_t$ and $r = st_{t+1} - 1$. If number d belongs to the first half of the wavelet tree, we descend left, otherwise right. In both cases we update l and r as in the algorithm to retrieve $F_t[i]$. If, at some point, the interval $[l, r]$ becomes empty, then there is no value d in the subtree and we return without a value. If, instead, we arrive at a leaf with a nonempty $[l, r]$ (indeed, it must hold $l = r$ in this case), then the leaf arrived at is d and we return this value. If the recursive step returns no result, then we must look for the first result to the right: If the recursion was to the right child, or it was to the left but there is not any 1 in $B_v[l, r]$, we in turn return with no result. Otherwise we enter the right child looking for the smallest value. From there, we enter recursively the left child only if there is some 0 in $B_v[l, r]$, otherwise we go right. Thus in at most two root-to-leaf traversals we find out the first $d' \geq d$ value in F_t. To obtain j, the position of d' in list F_t, we must add up the n_0 values at all the nodes in the path to d' where we have gone to the right. Note $O(\log D)$ is not far from the time required by a binary search on F_t.

As before, if we know $F_t[j] = d$ and seek for the first value $F_t[j'] \geq d'$, where $d' \geq d$, we can do it in amortized time proportional to $\log(d' - d + 1)$. The reason is that, once again, we can redo the work for d' from the corresponding position of the path used for d (this position is now easier to calculate: it is the first bit at which d and d' differ). For the same reason as before, k searches covering a range $[d, d']$ will cost at most $O(\log D + k \log \frac{d'-d+1}{k})$ time. This is indeed the time required by k successive searches using exponential search.

Finally, our data structure allows us to carry out a particular intersection algorithm. Consider intersecting two lists F_t and $F_{t'}$. This is equivalent to finding the common document numbers in $L[l, r]$ and $L[l', r']$. We proceed as follows. Let v be the wavelet tree root and B_v its bitmap. We descend to the left with $l = rank_0(B_v, l-1) + 1$, $r = rank_0(B_v, r)$, $l' = rank_0(B_v, l'-1) + 1$, and $r' = rank_0(B_v, r')$. We also descend to the right using the same formulas replacing $rank_0$ with $rank_1$. If at any point range $[l, r]$ or range $[l', r']$ is empty, we abort that branch. If we arrive at a leaf d with $l = r$ and $l' = r'$, then we report d as an element in the intersection. This algorithm is indeed a materialization of *bys* [7], where the binary searches have been replaced by constant-time *rank* operations, hence it is an $O(\log D)$ factor faster!

Furthermore, this can be extended to intersecting k terms simultaneously, by maintaining k ranges instead of two, with an $O(k)$ time penalty factor. As soon as one such range disappears, the tree branch is abandoned. This can offer much better performance than the successive pairwise intersections that are currently the best choice in practice. Perhaps more importantly, this scheme can be relaxed to report any document where at least k' out of the k words appear, by abandoning the branches when there are less than k' nonempty intervals. Again, it is not easy to implement this type of search by, say, successive intersections.

We can proceed similarly in case of unions. We start with the k intervals and proceed recursively as long as one of the intervals is nonempty. The cost is $O(M(1 + \log \frac{D}{m}))$, where m is the size of the output and M is the sum, over the returned documents, of the number of intervals where they appeared. The reason is that each interval must be projected to all of its leaves, but again, we arrive at m different leaves overall, but the m paths of length $\log D$ cannot be all different. The classical algorithm is $O(M \log k)$ time, which can be slightly better or worse.

Other operations of interest. If the range of terms $[t, t']$ represent the derivatives of a single stemmed root, we might wish to act as if we had a single list $F_{t,t'}$ containing all the documents where they occur. Indeed, if we apply our previous algorithm to obtain $F_t[i]$ from $L[st_t, st_{t+1} - 1]$, on the range $L[st_t, st_{t'+1} - 1]$, we obtain precisely $F_{t,t'}[i]$, if we understand that a document d may repeat several times in the list if different terms in $[t, t']$ appear in d.

More than that, the algorithms to find the first j such that $F_t[j] \geq d$ can be applied verbatim to obtain the same result for $F_{t,t'}[j] \geq d$ (except that $l = r$ may not hold at the leaves, but rather $r - l + 1$ is the number of times the resulting document appears in $F_{t,t'}$). All the variants of these queries are directly

supported as well. Finally, the *bys*-like search can also be applied verbatim in order to intersect stemmed terms (again, at the leaves it may hold that $l \leq r$ and $l' \leq r'$, not necessarily the equality).

Note we can obtain the list of unique documents d for a range of terms $[t, t']$ by using the method that finds the first j and d such that $F_t[j] = d \geq 1$, then $F_t[j'] = d' \geq d + 1$, and so on.

Note also that we have a kind of *summarization* information available. In particular, we can obtain the *local vocabulary* of a document d, that is, the set of different terms that appear in d. By descending to a leaf d, and then locating back all of its occurrences $L[i]$ (via *select* as we move upwards in the wavelet tree), we can find all the i such that $L[i] = d$, and then $rank_1(S, i)$ gives the terms, all in time $O(\log D)$ per term. Moreover, as the occurrences of d within its leaf are a stable sort of the original order in L, we can retrieve the local vocabulary in lexicographic order. This allows, for example, merging in linear time the vocabularies of different documents, or binary searching for a particular term in a particular document (yet, the latter is easier via two *rank* operations on L: $rank_d(L, s_{t+1} - 1) - rank_d(L, s_t - 1)$; then the corresponding position can be obtained by $select_d(L, 1 + rank_d(L, s_t - 1))$).

The way to support hierarchical documents by mapping them to ranges $[d_l, d_r]$ of documents is relatively obvious. It is sufficient to restrict all our wavelet tree traversals to the nodes that contain leaves in this range, disregarding others.

4.2 Ranked retrieval

We focus now on the operations of interest for ranked retrieval, which are also simulated essentially in $O(\log D)$ time.

Direct access and Persin's algorithm. The L_t lists used for ranked retrieval are directly concatenated in L, so $L_t[i]$ is obtained by extracting symbol $L[s_t + i - 1]$ using the wavelet tree. Recall that the term frequencies *tf* are also available in time $O(\log D)$. Again, a range $L_t[i, i']$ is obtained in $O((i' - i + 1) \log \frac{D}{i' - i + 1})$ time, as follows. Start at the root v with bitmap B_v and let the range to extract be $[l, r]$. Compute the corresponding ranges $[l_0, r_0]$ and $[l_1, r_1]$ for the left and right child, as usual, and descend recursively to both. Stop the recursion when the range is empty. Upon arriving at a leaf d, report d. One can, for example, find with $select_1(R_t, v_t - rank_1(T_t, p) + 1) - 1$ the length of the prefix of L_t where the *tf* values are $> p$, which is useful for Persin's algorithm [25].

This algorithm is correct but it has the problem of retrieving the documents in document order, not in *tf* order as they are in L_t. To recover the correct ordering, we must merge the results at each internal wavelet tree node during the recursion, as they arrive. At node v with results returned by its left and right child, we use $select_0$ and $select_1$, respectively, in B_v to map their positions in the bitmap of v. We advance in both lists so as to build their union in the correct order in B_v prior to sending them upwards. Note that, due to this merging effort, the complexity is again $O((i' - i + 1) \log D)$, but in practice the method is faster than extracting each $L[i]$ one by one.

Note, nevertheless, that retrieving the highest-*tf* documents in document order is indeed beneficial for Persin's algorithm, where a difficulty is how to accumulate results across unordered document sets. One could use the threshold p of the algorithm to retrieve the relevant documents from the next list to consider, and gracefully merge the result with the current candidates, as all are automatically in increasing document order.

Other operations of interest. Any candidate document d in Persin's algorithm can be directly evaluated, obtaining its weight $w(d)$, by finding it within L_t for each $t \in q$ (with $rank_d$ and $select_d$ on L, as explained), and its *tf* obtained, all in $O(|q| \log D)$ time.

If we wish to use stemming, we might want to retrieve prefixes of several lists L_t to $L_{t'}$. We may carry out the previous algorithm to deliver all the distinct documents in these prefixes, now carrying on the $t'-t+1$ intervals as we descend in the wavelet tree. When we arrive at the relevant leaves d, the corresponding positions will be contiguous, thus we can naturally return just one occurrence of each d in the union. This is a simplification of the method sketched earlier to obtain the unique documents in $F_{t,t'}$. If we wish to obtain the sum of the *tf* values for all the stemmed terms in d, we can traverse the wavelet tree upwards for each interval element at leaf d, and obtain its *tf* upon finding its position i in L. Alternatively, we could store the *tf* values aligned to the leaves and mark their cumulative values on a compressed bitmap, so as to obtain the sum as the difference of two $select_1$ queries on that bitmap. The space for *tf* becomes now $O(N \log \frac{n}{N})$ bits. In any case, this delivers the results in document order.

There is also some basic support for hierarchical documents: If we wish to know the total *tf* of t in a range $[d, d']$ of documents, this range is exactly covered by $O(\log D)$ wavelet tree nodes. We can descend, projecting the range of L_t in L, until those nodes, and then move upwards again to find their positions and individual *tf* values, to add them all.

More interesting is the fact that we can carry out ranked retrieval restricted to any range of documents $[d, d']$ (e.g., within a particular XML subtree or filesystem directory or range of document versions). Once again, it is sufficient to restrict any of the operations described above so that they do not descend to a node whose range does not intersect $[d, d']$. This automatically yields, for example, Persin's algorithm over a range of documents.

5 Conclusions and future work

In this paper we have shown how wavelet trees can be used to achieve dual-ordered inverted lists, that is, lists that are simultaneously sorted by document id (useful for intersections and document retrieval) and by term impact or frequency (useful for ranked retrieval). We are in the process of translating these data structures into practice and verifying them experimentally.

Finally, we emphasize that our approach can be applied to any ordering on the documents. A very different and interesting ordering from the one considered

here is that induced by the suffix array (the D array of Culpepper et al. [14]). Applying our data structure and *bys*-like intersection algorithm over this ordering immediately yields efficient "bag-of-strings" queries from suffix arrays, further bridging the gap between IR problems and optimal pattern matching data structures.

References

1. V. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. 24th SIGIR*, pages 35–42, 2001.
2. V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.
3. V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th SIGIR*, pages 372–379, 2006.
4. R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. 15th CPM*, LNCS 3109, pages 400–408, 2004.
5. R. Baeza-Yates, A. Moffat, and G. Navarro. *Searching Large Text Collections*, pages 195–244. Kluwer Academic, 2002.
6. R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 1999.
7. R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th SPIRE*, LNCS 3772, pages 13–24, 2005.
8. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. 13th SODA*, pages 390–399, 2002.
9. 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:article 7, 2009.
10. N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. S,C-dense coding: an optimized compression code for natural language text databases. In *Proc. 10th SPIRE*, LNCS 2857, pages 122–136, 2003.
11. S. Buettcher, C. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, Cambridge, MA, 2010.
12. B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Pearson Education, 2009.
13. J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, LNCS 4726, pages 137–148, 2007.
14. J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th ESA*, 2010. To appear.
15. E. Demaine and I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th SODA*, pages 743–752, 2000.
16. T. Gagie, S. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th SPIRE*, LNCS 5721, pages 1–6, 2009.
17. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
18. A. Gupta. *Succinct Data Structures*. PhD thesis, Duke University, USA, 2007.
19. H. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, 1978.

20. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE FOCS*, pages 713–722, 2009.
21. D. A. Hull. Stemming algorithms: A case study for detailed evaluation. *J. Amer. Soc. Inf. Sci.*, 47(1):70–84, 1996.
22. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
23. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):art. 2, 2007.
24. G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retr.*, 3(1):49–77, 2000.
25. 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.
26. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
27. P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, 2007.
28. P. Sanders and F. Transier. Compressed inverted indexes for in-memory search engines. In *Proc. 10th ALENEX*, pages 3–12, 2008.
29. F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th SIGIR*, pages 222–229, 2002.
30. T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th SIGIR*, pages 175–182, 2007.
31. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.
32. J. Xu and W. B. Croft. Corpus-based stemming using cooccurrence of word variants. *ACM Trans. Inf. Sys.*, 16(1):61–81, 1998.
33. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th WWW*, pages 401–410, 2009.
34. G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.
35. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):article 6, 2006.