Faster and Smaller Inverted Indices with Treaps *

Roberto Konow , Gonzalo Navarro
Department of Computer Science
University of Chile, Chile

Charles L.A Clarke ,
Alejandro López-Ortíz
School of Computer Science
University of Waterloo, Canada

ABSTRACT

We introduce a new representation of the inverted index that performs faster ranked unions and intersections while using less space. Our index is based on the treap data structure, which allows us to intersect/merge the document identifiers while simultaneously thresholding by frequency, instead of the costlier two-step classical processing methods. To achieve compression we represent the treap topology using compact data structures. Further, the treap invariants allow us to elegantly encode differentially both document identifiers and frequencies. Results show that the space consumption is below 10% of the size of the corpus and the index performs queries up to twice as fast than previous compact representations, which in addition require more space. Modern two-stage (massive filtering / detailed ranking) information retrieval systems would benefit from this boosting of the filtration stage of the query resolution process, which would free more resources for the ranking stage, thus enabling more precise results within a given time budget.

1. INTRODUCTION

Modern Web search engines, and other information retrieval systems, face two competing challenges. On the one hand, they have to manage huge amounts of data. On the other hand, they have to provide very precise results in response to user queries, often identifying a few relevant documents among increasingly larger collections. These requirements can be addressed via a two-stage ranking process [39, 17]. In the first stage, a fast and simple filtration procedure extracts a subset of a few hundreds or thousands of candidates from the possibly billions of documents forming the collection. In the second stage, more complex learned ranking algorithms are applied to the reduced candidate set in order to obtain a handful of high-quality results. In this paper, we focus on improving the efficiency of the first stage, freeing more resources for the second stage and increasing the overall performance. In contexts where traditional ranking methods are sufficient, the goal of the first stage is to directly convey a few top-quality results to the final user.

The first stage aims to return either a set of the highest ranked documents containing all the query terms (a ranked intersection) or some of the most important query terms (a ranked union). In most cases, ranked intersections are solved via a Boolean intersection, followed by the computation of scores for the resulting documents. Ranked unions are generally solved only in approximate form, avoiding a costly Boolean union. However, Ding and Suel [22] showed that ranked intersections can be processed faster than Boolean intersections. They also obtained the best known performance for ranked unions, giving exact, rather than approximate results, and demonstrating the feasibility of their approach.

In this paper, we introduce a new compressed representation for posting lists that performs ranked intersections and (exact) unions directly. This representation is based on the *treap* data structure [36], a binary tree that simultaneously represents a left-to-right and a top-to-bottom ordering. We use the left-to-right ordering for document identifiers (which supports fast Boolean intersections) and the top-to-bottom ordering for term weights (which supports the thresholding of results simultaneously with the intersection process). Using this data structure, we can obtain the top-k results for a ranked intersection/union without having to first produce the full Boolean intersection/union.

Additionally, the treap representation allows us to differentially encode both document identifiers and weights, which is crucial for the space-efficient representation of inverted indexes. Posting lists have been compressed for decades [40] to handle very large collections within minimal space. With a few exceptions, using other data structures we must choose one ordering for differential encoding, and cannot differentially encode the other ordering. Our doubly differential scheme achieves noticeable space reductions with respect to competing representations.

Our experiments show that the space usage of our treap-based inverted index representation is below 10% of the size of the corpus, and below that of the best previous representations. For example, our representation requires 18% less space than the Block-Max representation [22] and 13% less space than the Dual-Sorted [26] representation. As for the time, treaps outperform previous techniques for k up to 20 on intersections, and up to 100 on unions, being up to twice as fast as the alternatives in some cases. Those ranges of k values make this result of particular interest both in appli-

*Partially funded by Fondecyt grant 1-110066 , by the Conicyt PhD Scholarship Program, Chile and by the Emerging Leaders in the Americas Program, Government of Canada.

SIGIR '13

cations where a limited result set is of interest, and in large-scale distributed systems in which each node contributes a limited set to the global result.

2. BASIC CONCEPTS

The *inverted index* plays a central role in the efficient processing of ranked and Boolean queries [40, 43, 19, 17, 5]. It can be seen as an array of *lists* or *postings*, where each entry of the array corresponds to a different *term* or *word* in the collection, and the lists contain one element per distinct document where the term appears. For each document, the index stores the *document identifier (docid)* and the *weight* of the term in the document. The set of terms is called the *vocabulary* of the collection, which is comparatively small in most cases [25].

In the first stage of query processing, a simple metric is used to assign a *score* to a document with respect to a query. In the classical *bag-of-words* model, the query Q is seen as a set of q terms $t \in Q$, and the score of a document d is computed as $score(Q,d) = \sum_{t \in Q} w(t,d)$, where $w(t,d)$ is the weight of term t in document d. For example, in the well-known tf-idf scoring scheme, this weight is computed as $w(t,d) = tf_{t,d} \cdot idf_t$. 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, the number of documents where the term t appears, and D is the total number of documents. Since idf_t (or df_t) depend only on t, an efficient way to store $w(t,d)$ in an inverted index is to store idf_t or df_t together with each distinct vocabulary term, and store the values $tf_{t,d}$ in the posting list of term t, together with each docid d. In this paper we will assume that term frequencies are stored in the posting lists, but any other integer measure, such as impacts [2] could be used.

In the *bag-of-words* model we are given Q and k, and asked to retrieve k documents d with the highest $score(Q,d)$ values. In the two-stage model, typical values of k are hundreds to thousands, as discussed earlier. In simpler one-stage systems, typical values of k are below 20. Note that it is not necessary for all the terms of Q to appear in a returned document d; a missing term t simply implies that $w(t,d) = 0$. This problem is frequently called *ranked union*. A more recent variant of the problem, popularized by Web search engines to favor precision over recall, is the *ranked intersection*, where only documents containing all the terms are returned. Nowadays, ranked intersections are more common than unions.

The Boolean intersection problem, without ranking, aims at retrieving all the documents d where all the terms of Q appear. A typical way to solve a ranked intersection is to first compute a Boolean intersection, then compute the scores of all the resulting documents, and finally keep the documents with the k highest scores. This approach has triggered much research on the Boolean intersection problem [21, 6, 34, 8, 26]. This approach is, of course, suboptimal, since in principle one could use weight information to filter out documents that belong to the intersection but one can ensure will not make it to the top-k list. Only recently some schemes specifically aimed at solving ranked intersections have appeared [22]. All these schemes store the posting lists in increasing docid order, which is convenient for skipping documents during intersections.

Ranked unions, instead, cannot be efficiently solved through a Boolean union, as this returns too many results. In this case, most research has aimed at returning an *approximate* answer within good time bounds [32, 2]. Most of these techniques order the posting lists by by decreasing weight values, not by docids. Recently, it has been shown that ranked unions can be solved in exact form within reasonable time [16, 37, 22] by using increasing docid order for the posting lists in the best solution [22].

Traditionally, the posting lists were stored on disk. With the availability of large amounts of main memory, this trend has changed to use the main memory of a cluster of machines, and many intersection algorithms have been designed for random access [21, 6, 34, 20, 35, 37, 8, 26]. In distributed main-memory systems, usually documents are distributed across independent inverted indexes, and each index contributes with a few results to the final top-k list. Therefore, it is most interesting that an individual inverted index solves top-k queries efficiently for k values in the range 10–100 [17].

Both when stored on disk and in main memory, reducing the size of the inverted index representation is crucial. On disk, it reduces transfer time. In main memory, it increases the size of the collections that can be managed within a given RAM budget, or alternatively reduces the amount of servers that must be allocated in a cluster to hold the index, the energy they consume, and the amount of communication. Compression of inverted indexes is possibly the oldest and most successful application of compressed data structures (e.g., see [40]). The main idea to achieve compression is to differentially encode either the document identifers or the weights (depending on how the lists are sorted), whereas the other value (weight or docid, respectively) becomes harder to compress. The problem of this duality in the sorting, and how it affects compression and query algorithms, has been discussed in past work [40, 4, 26].

In this context, our contribution is a new in-memory posting list representation that, on the one hand, achieves improved compression because it allows differential encoding of both docids and frequencies, and on the other hand, performs exact ranked intersections and unions directly and natively without having to first intersect/merge and then rank.

3. RELATED WORK

3.1 Query Processing Strategies

Two kinds of approaches are used for unions and intersections (ranked or Boolean): Term-at-a-time (TAAT) and Document-at-a-time (DAAT) [17].

TAAT processes one posting list after the other. The lists are considered from shortest to longest, starting with the first one as a candidate answer set, and refining it as we consider the next lists. TAAT is especially popular for processing ranked unions [32, 2, 37], as the successive lists have decreasing idf_t value and thus a decreasing impact on the result, not only for the tf-idf model, but also for BM25 and other models. The documents in each list are also sorted by decreasing weight. Thus heuristic thresholds can be used to obtain an approximate ranked union efficiently, by pruning the processing of lists earlier, or avoiding lists completely, as we reach less relevant documents and our candidate set becomes stronger [32, 2]. A more sophisticated approach based on similar ideas can be used to guarantee that the answer is exact [37].

DAAT processing is more popular for Boolean intersec-

tions and unions. Here the q lists are processed in parallel, looking for the same document in all of them. Posting lists must be sorted by increasing docid, and we keep a pointer to the current position in each of the q lists. Once a document is processed, the pointers move forward. Much research has been carried out on Boolean intersections [21, 6, 34, 20, 8]. While a DAAT processing is always used to intersect two lists, experimental results suggest that the most efficient way to handle more lists is to intersect the two shortest ones, then the result with the third, and so on. This can be seen as a TAAT strategy.

Many ranked intersection strategies employ a full Boolean intersection followed by a postprocessing step for ranking. However, recent work has shown that it is possible to do better [22]. The advantage of DAAT processing is that, once we have processed a document, we have complete information about its score, and thus we can maintain a current set of top-k candidates whose final scores are known. This set can be used to set a threshold on the scores other documents need to surpass to become relevant for the current query. Thus the emphasis on ranked DAAT is not on terminating early but on skipping documents. This same idea has been successfully used to solve exact (not approximate) ranked unions [16, 22].

The strategies we use to solve ranked union and intersection queries in this paper are best classified as DAAT. We use sophisticated mechanisms to skip documents using the current threshold given by the current top-k candidate set.

3.2 Compressed Posting List Representations

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 [40], the latter usually giving the best compression. Recent proposals make use of byte-aligned [35, 20] or word-aligned [41, 1] codes, which are faster at decoding at a small loss in 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, cutting them into *blocks* that are differentially encoded, while storing in a separate sequence the absolute values of the block headers and pointers to the encoded blocks. Different sampling strategies have been used [20, 34] and the intersection algorithms have been tailored to them.

When lists are sorted by decreasing weight (for approximate ranked unions), 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 [4, 43].

3.3 State of the Art for Exact Ranked Queries

The following two approaches have recently displayed the best performance for exact ranked intersections and unions.

3.3.1 Block-Max

Block-Max [22] is a special-purpose structure for ranked intersections and unions. It sorts the lists by increasing docid, cuts the lists into blocks, and stores the maximum weight for each block. This enables them to skip large parts of the lists whose maximum possible contribution is very low, by comparing the contribution of a block with a threshold given by the current candidate set. This solution produces considerable performance gains over the best previous techniques for exact ranked unions [16, 37], and also over the techniques that perform ranked intersections via a Boolean preprocessing.

The basic concept is as follows: Suppose the next document of interest belongs to blocks $b_1, \ldots, b_q$ in the q lists. Compute an upper bound to $score(Q, d)$ using the block maxima instead of the weights $w(t, d)$. If even this upper bound does not surpass the kth best score known up to now, no document inside the current blocks can make it to the top-k list. So we can safely skip some blocks of the list.

Our technique can be seen as a generalization of the Block-Max idea, in which we use the treap concept to naturally define a hierarchical blocking scheme. The generalization is algorithmically nontrivial, but we demonstrate its practicality over the flat Block-Max scheme. In addition, the treap structure allows us to differentially encode both docids and weights, which translates into space savings with respect to Block-Max.

3.3.2 Dual-sorted inverted lists

Dual-Sorted inverted lists [30, 26] represent the posting lists sorted by decreasing frequency, using a wavelet tree data structure [24, 29]. The wavelet tree efficiently simulates ordering by increasing docids. TAAT processing is used for approximate ranked unions and DAAT-like processing for (exact) ranked intersections. The latter, although building on Boolean intersections, is implemented in native form on wavelet trees, which makes it particularly fast, even faster than Block-Max. Basically, the wavelet tree can recursively subdivide the universe of docids and efficiently determine that some list has no documents in the current interval.

Our technique shares with Dual-Sorted the ability to maintain the lists sorted by both docids and weights simultaneously, and is able to perform a similar kind of native intersection, that is, determine that in an interval of documents there is a list with no elements. In contrast, Dual-Sorted does not know the frequencies until reaching the individual documents, whereas our treaps give an upper bound to the frequencies in the current interval. This allows us to perform ranked intersections faster than the Boolean intersections of Dual-Sorted. In addition, the treap uses less space, since Dual-Sorted cannot use differential encoding on docids.

4. DIFFERENTIALLY ENCODED TREAPS

We describe our data structure in this section. First, we survey the treap data structure and show it can be used to represent a posting list. Then we describe how we represent the resulting data structure using little space. In addition, we describe some practical improvements on the basic idea. Finally we describe how query processing is carried out on the final representation.

4.1 The Treap Data Structure

A *treap* [36] is a binary tree where nodes have two attributes: a *key* and a *priority*. The treap satisfies the invariants of a binary search tree with respect to the keys: the root key is larger than those of its left subtree and smaller than those of its right subtree. Furthermore, the treap sat-

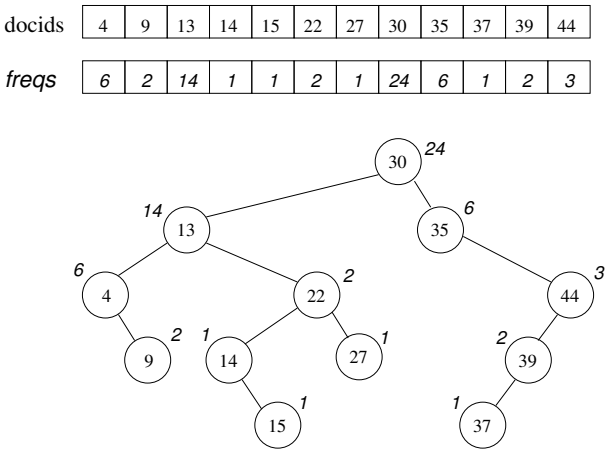


Figure 1: An example posting list (with docids and frequencies) and the corresponding treap representation in our scheme. Note that docids (inside the nodes) are sorted inorder and frequencies (outside the nodes) are sorted top to bottom.

isfies the invariants of a binary heap with respect to the priority: the priority of the parent is larger than those of its descendants.

Given its invariants, a treap can be searched for a key just as a binary search tree, and it can be simultaneously used as a binary heap. While in the literature it has mostly been used with randomly assigned priorities [36, 27, 14] to ensure logarithmic expected height independently of the order of insertions, a treap can also be seen as the *Cartesian tree* [38] of the sequence of priorities once the values are sorted by keys. Such Cartesian tree can be built in $O(n)$ time from a sequence of n elements already sorted by key, even in compressed form [10, 9, 23].

We note that treaps are a particular case of *priority search trees* [28], which can guarantee balancedness but are unlikely to be as compressible as Cartesian trees. There has been some work on using priority search trees for returning top-k elements from suffix trees and geometric range searches [12, 11] but, as far as we know, our usage of treaps for ranked queries on inverted indexes, plus their differential compression, is novel.

4.2 Inverted Index Representation

We consider the posting list of each term as a sequence sorted by docids (which act as keys), each with its own term frequency (which act as priorities). Term impacts, or any other term weights, may also be used as priorities. We then use a treap to represent this sequence. Therefore the treap will be binary searchable by docid, whereas it will satisfy a heap ordering on the frequencies. This means, in particular, that if a given treap node has a frequency below a desired threshold, all the docids below it in the treap can be discarded as well.

Figure 1 illustrates a treap representation of a posting list. This treap will be used as a running example.

4.3 Compressing the Treap

In order to compete with existing compressed representations of posting lists, we represent the treap data (topology, docids, and frequencies) in compact form. The key issue is that we choose a representation where all the treap operations can be carried out efficiently, so as to exploit the treap properties at query time.

4.3.1 Compact topology representation

Given a posting list of n documents, the treap will be a binary tree of n nodes. We represent it as a general tree using a well-known isomorphism: First, a fake root node v_r is created. The children of v_r become the nodes in the rightmost path of the treap, from the root to the leaf. Then each of those nodes are converted recursively.

With this transformation, the treap root is the first child of v_r. The left child of a treap node v is its first child in the general tree. The right child of v is its next sibling in the general tree. An inorder traversal of the treap corresponds to a postorder traversal of the general tree.

There are $\Theta(4^n/n^{3/2})$ general trees of n nodes, and thus one needs $\log_2(4^n/n^{3/2}) = 2n - \Theta(\log n)$ bits to represent any such tree. There exist various compact tree representations using $2n+o(n)$ bits that can in addition carry out many tree operations efficiently, including taking the first child, next sibling, computing postorder of a node, and so on. We will use a recent representation that has proven to be efficient in practice [33, 3]. It is based on a balanced parentheses representation of the tree, obtained by a preorder traversal where we append an opening parenthesis when reaching a node and a closing parenthesis when leaving it.

4.3.2 Differentially encoded trees

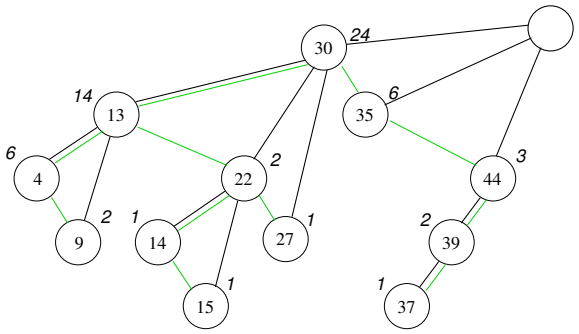
In addition to the tree topology, we must represent docids and term frequencies. Our plan is not to access the posting lists in sequential form as in classical schemes, thus differentially encoding each docid with respect to the previous one is not directly applicable. Instead, we make use of the invariants of the treap data structure.

Let $id(v)$ be the docid of a treap node v, and $f(v)$ its frequency. We represent $id(v)$ and $f(v)$ for the root in plain form, and then represent those of its left and right children recursively. For each node v that is the left child of its parent u, we represent $id(u) - id(v)$ instead of $id(v)$. If, on the other hand, v is the right child of its parent u, we represent $id(v) - id(u)$ [18]. In both cases, we represent $f(u) - f(v)$ instead of $f(v)$. Those numbers get smaller as we move downwards in the treap.

The sequence of differentially encoded $id(v)$ and $f(v)$ values is represented according to an inorder traversal of the treap. As we move down the treap, we can easily maintain the correct $id(v)$ and $f(v)$ values for any node arrived at, and use it to compute the values of the children as we descend.

For this sake we need to be able to randomly access a differential value in the sequence, given the inorder of a node. While we can compute the inorder of any node from our topology representation, we need a storage mechanism for the differences that: *i*) can access any value in the sequence, while *ii*) using fewer bits to represent smaller numbers. We use Direct Addressable Codes (DACs) [15], which are designed precisely with this aim.

DACs encode a sequence of numbers $x_1, \ldots, x_n$ as follows. The $\lceil \log_2(\max\{x_i\}+1) \rceil$ bits needed to represent any x_i are divided into chunks of varying size. Then the first chunk of lowest bits of all the numbers are represented in a first sequence, the second chunks in a second sequence,



Topology ((((()))(()()))(()))()((()))))

diff ids	9	5	17	8	1	9	27	30	5	2	5	9
diff freqs	*8*	*4*	*10*	*1*	*0*	*12*	*1*	*24*	*18*	*1*	*1*	*3*

Figure 2: The compressed representation of the example treap. The original binary tree edges (light color) are replaced by a general tree, whose topology is represented with parentheses. Docids and frequencies are sorted inorder and represented in differential form with respect to their parent.

and so on. Some numbers x_i will only participate in the first sequences because they are smaller than others. Compact bitmap representations are used to drive the extraction process for any x_i through the different sequences where its chunks are represented. DACs can tune the block sizes so as to use minimum space, given the sequence of x_i values.

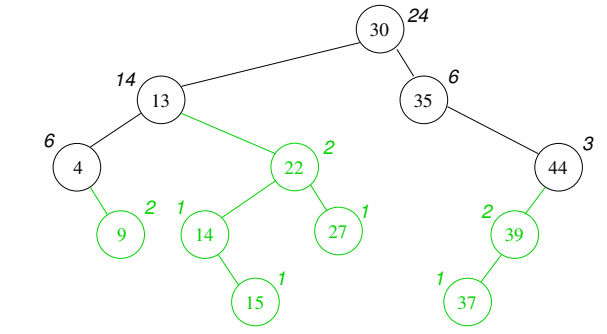
Figure 2 illustrates our compressed treap representation.

4.4 Practical Improvements

The scheme detailed above would not be so successful without two important improvements. First, because many posting lists are very short, it turns out to be more efficient to store two single DAC sequences, with all the differential docids and all the differential frequencies for all the lists together, even if using individual DACs would have allowed us to optimize their space for each sequence separately. This is because the overhead of storing the chunk lengths and other administrative data overweights the benefits for short sequences.

The second, and more important, improvement is to omit from the treap representation all the elements of the lists where the frequency is below some threshold f_0. According to Zipf's law [42, 19, 17, 5], a large number of elements will have low frequencies, and thus using a separate posting list for each frequency below f_0 will save us from storing those frequencies wherever those elements would have appeared in the treap. Further, the docids of each list can be differentially encoded in classical sequential form, which is more efficient than in treap order.

It turns out that many terms do not have to be stored in a treap at all, as they never occur more than f_0 times in any document. We represent the gap-encoded lists using Rice codes and taking an absolute sample every 128 values (which form a block). Samples are stored separately and explicitly in an array, with pointers to the block [20]. Searches in these lists will ask for consecutively larger values, so we remember



ids1	14	15	27	37
ids2	9	22	39	

diff ids1	14	1	12	10
diff ids2	9	13	17	

Figure 3: Separating frequencies below $f_0 = 2$ in our example treap. The part that is removed from the treap is in light color. For the documents with frequencies 1 and 2, we show the absolute docids on the left and their differential version on the right.

the last element found and exponentially search for the next query starting from there. Figure 3 illustrates the separation of low-frequency elements from our example treap.

4.5 Query Processing

4.5.1 General procedure

Let Q be a query composed of q terms $t \in Q$. To obtain the top-k documents from the intersection or union of q posting lists we proceed in DAAT fashion: We traverse the q posting lists in synchronization, identifying the documents that appear in all or some of them, and accumulating their weights $w(t,d)$ into a final $score(Q,d) = \sum_t w(t,d) = \sum_t tf_{t,f} \cdot idf_t$. Those documents are inserted in a min-priority queue limited to k elements, where the priority is the score. Each time we insert a new element and the queue size reaches $k+1$, we extract and remove the minimum. At the end of the process, the priority queue contains the top-k results. Furthermore, at any stage of the process, if the queue has reached size k, then its minimum score L is a lower bound to the scores we are interested in for the rest of the documents.

4.5.2 Intersections

Let d be the smallest docid not yet considered (initially $d = 1$). All the treaps t maintain a stack of nodes (initially holding just a sentinel value element u_t with $id(u_t) = +\infty$ and $f(u_t) = +\infty$), and a cursor v_t (initially the treap root). The stack will contain the nodes in the path from the root to v_t where we descend by the left child. We will always call u_t the top of the stack; u_t is an ancestor of v_t and it holds $id(u_t) > id(v_t)$.

We advance in all the treaps simultaneously towards a node v with docid $id(v) = d$, while skipping nodes using the current lower bound L. In all the treaps t we maintain the invariant that, if v is in the treap, it must appear in the subtree rooted at v_t. In particular, this implies $d < id(u_t)$.

Because of the decreasing frequency property of treaps, if d is in a node v within the subtree rooted at v_t, then $f(v) \leq f(v_t)$. Therefore, we can compute an *upper bound* U

to the score of document d by using values $f(v_t)$ instead of $f(v)$, for example $U = \sum_{t \in Q} f(v_t) \cdot idf_t$ for a tf-idf scoring[1]. If this upper bound is $U \leq L$, then there is a valid top-k answer where d does not participate, so we can discard d. Further, no node that is below all the current v_t nodes can qualify. Therefore, we can safely compute a new target $d \leftarrow \min_t(id(u_t))$. Each time the value of d changes (it always increases), we must update the stack of all the treaps t to restore the invariants: We assign $v_t \leftarrow u_t$ and remove u_t from the stack until $id(u_t) > d$. We then resume the global intersection process with this new target d. The upper bound U is recomputed incrementally each time any v_t value changes (U may increase or decrease).

When $U > L$, it is still feasible to find d with sufficiently high score. In this case we have to advance towards the node containing d in some treap. We use a round-robin scheme to choose the treap t (more complex ideas, like choosing the treap with maximum contribution to the value of U, did not make a noticeable difference). We must choose a treap where we have not yet reached d; if we have reached d in all the treaps then we can output d as an element of the intersection, with a known score (the current U value is the actual score of d), insert it in the priority queue of top-k results as explained (which may increase the lower bound L), and resume the global intersection process with $d \leftarrow d + 1$ (we must update stacks, as d has changed).

Once we have decided to move towards $d \neq id(v_t)$ in some treap t, we proceed as follows. If $d < id(v_t)$, we move to the left child of v_t, l_t, push v_t in the stack, and make $v_t \leftarrow l_t$. Instead, if $d > id(v_t)$, we move to the right child of v_t, r_t, and make $v_t \leftarrow r_t$. We then recompute U with the new v_t value.

If we have to move to the left and there is no left child of v_t, then d does not belong to the intersection. We stay at node v_t and redefine a new target $d \leftarrow id(v_t)$. If we have to move to the right and there is no right child of v_t, then again d is not in the intersection. We make $v_t \leftarrow u_t$, remove u_t from the stack, and redefine $d \leftarrow id(u_t)$. In both cases we adjust the stacks of the other treaps to the new value of d, as before, and resume the intersection process.

Algorithm 1 gives pseudocode for the intersection.

4.5.3 Handling low-frequency lists

We have not yet considered the lists of documents with frequencies up to f_0, which are stored separately, one per frequency, outside the treap. While a general solution is feasible (but complicated), we describe a simple strategy for the case $f_0 = 1$, which is the case we implemented.

Recall that we store the posting lists in gap-encoded blocks. Together with the treap cursor, we will maintain a *list cursor*, which points inside some block that has been previously decompressed. Each time there is no left or right child in the treap, we must search the list for potential elements omitted in the treap. More precisely, for elements in the range $[d, id(v_t)-1]$ if we cannot go left, or in the range $[d, id(u_t)-1]$ if we cannot go right. Those elements must be processed as if they belonged to the treap before proceeding in the actual treap. Finding this new range $[l, r]$ in the list may imply seeking and decompressing a new block.

[1]Replacing $f(v)$ by $f(v_t)$ will yield an upper bound whenever the scoring function is monotonic with the frequencies. This is a reasonable assumption and holds for all the formulas we use.

Algorithm 1 Top-k of intersection using treaps.

```
Intersect(Q, k)
  results ← ∅ // priority queue of pairs (key, priority)
  for t ∈ Q do
    stack_t ← ⟨⊥⟩ // stack of treap t, id(⊥) = f(⊥) = +∞
    v_t ← root of treap t
  end for
  compute score U using f(v_t) values, e.g. Σ_{t∈Q} f(v_t) · idf_t
  d ← 1
  L ← −∞
  while d < +∞ do
    while U ≤ L do
      changed(min_{t∈Q} id(top(stack_t)))
    end while
    if ∀t ∈ Q, d = id(v_t) then
      report(d, U)
      changed(d + 1)
    else
      t ← choose where to advance, d ≠ id(v_t)
      if d < id(v_t) then
        l_t ← left child of v_t
        if l_t is not null then
          push(stack_t, v_t)
          changev(t, l_t)
        else
          changed(id(v_t))
        end if
      else
        r_t ← right child of v_t
        if r_t is not null then
          changev(t, r_t)
        else
          changev(t, pop(stack_t))
          changed(id(v_t))
        end if
      end if
    end if
  end while
  return results

changed(newd)
  d ← newd
  for t ∈ Q do
    v ← v_t
    while d ≥ id(top(stack_t)) do
      v ← top(stack_t)
      pop(stack_t)
    end while
    changev(t, v)
  end for

report(d, s)
  results ← results ∪ (d, s)
  if |results| > k then
    remove minimum from results
    L ← minimum priority in results
  end if

changev(t, v)
  remove contribution of f(v_t) from U, e.g. U − f(v_t) · idf_t
  v_t ← v
  add contribution of f(v_t) to U, e.g. U + f(v_t) · idf_t
```

The best way to process range $[l, r]$ is to search as if it formed a subtree fully skewed to the right, descending from v_t. If we descended to the left of v_t towards the range, we push v_t into the stack. Since all the elements in the list have the same frequency, when we are required to advance towards (a new) d we simply scan the interval until reaching or exceeding d, and the docid found acts as our new $id(v_t)$ value. When the interval $[l, r]$ is exhausted, we return to the treap. Note that the interval $[l, r]$ may span several physical list blocks, which may be subsequently decompressed.

4.5.4 Unions

The algorithm for ranked unions requires a few changes on the algorithm for intersections. First, in the two lines that call **changed**$(id(v_t))$, we do not change the d for all the treaps when the current treap does not find it. Rather, we keep values $nextd_t$ where each treap stores the minimum $d' \geq d$ it contains, thus those lines are changed by $nextd_t \leftarrow id(v_t)$. Second, we will choose the treap t to advance only among those where $id(v_t) \neq d$ and $nextd_t = d$, as if $nextd_t > d$ we cannot find d in treap t. Third, when all the treaps t where $id(v_t) \neq d$ satisfy $nextd_t > d$, we have found exactly the treaps where d appears. We add up $score(Q, d)$ over those treaps where $id(v_t) = d$, report d, and advance to $d+1$. If, however, this happens but no treap t satisfies $id(v_t) = d$, we know that d is not in the union and we can advance d with **changed**$(\min_{t \in Q} nextd_t)$. Finally, **changed**$(newd)$ should not only update d but also update, for all the treaps t, $nextd_t$ to $\max(nextd_t, newd)$.

5. EXPERIMENTS AND RESULTS

5.1 Experimental Setup

We use the TREC GOV2 collection, containig about 25.2 million documents and about 32.8 million terms in the vocabulary. We parsed the collection using Porter's stemming algorithm. We used the TREC2006 Efficiency Queries dataset using distinct amounts of terms, from $|q| = 2$ to 5.

We compare our results with two baselines: (1) Block-Max [22], using their implementation and modifying it to use tf-idf scoring, and (2) Dual-Sorted [26], using their implementation. As additional baselines, we implemented (3) our own version of a traditional docid-sorted inverted index using Rice encoding of the gaps, sampling values every 128 values to support random access via exponential search, and (4) our own version of a traditional frequency-sorted inverted index, using gap-encoding for the frequencies. An interesting question for our treap, and its particular way of storing both differential docids and frequencies, is how far it is from an idealized "optimal" ordering in which one could sort docids and represent them differentially, and also sort frequencies and represent them differentially. This is of course unfeasible since both orders are incompatible, but treaps offer some intermediate combination. We implemented this "optimal" (and unfeasible) space using Rice codes.

Our experiments were performed on an Intel(r) Xeon(r) model E5620 running at 2.40 GHz with 96GB of RAM and 12,288KB of cache, running version 2.6.31-41 64 bits of the Linux kernel. All solutions were implemented in C++, compiled with g++ version 4.4.3 and -O3 optimization.

5.2 Space Usage

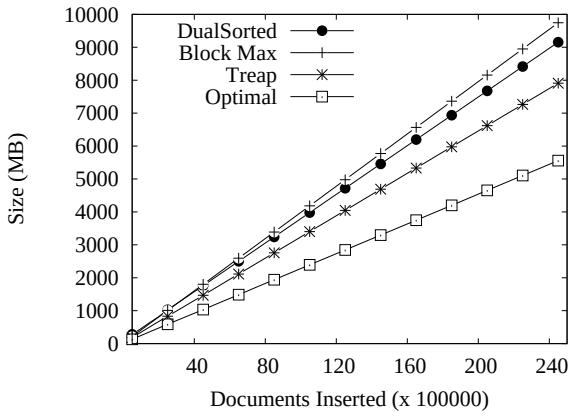


Figure 4: Space usage per document inserted.

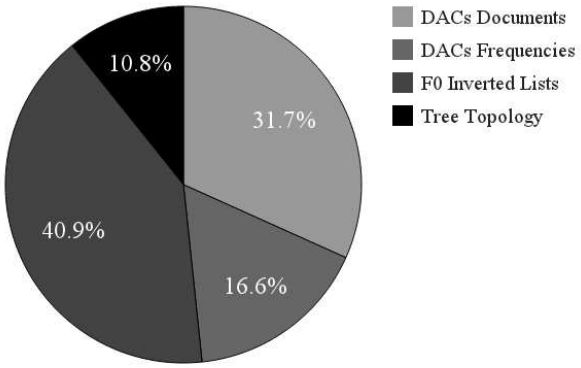


Figure 5: Fraction of space usage of the structures.

Figure 4 shows the space usage of the compared structures for increasing subsets of GOV2. Our docid-sorted index uses almost the same space as Block-Max, and our frequency-sorted index takes somewhat more space, so they are omitted for clarity. It can be seen that our compressed treap structures offer a space gain of 18% over Block-Max (which differentially encodes docids but not frequencies) and of 13% over Dual-Sorted (which differentially encodes frequencies but not docids), while using about 40% more space than the "optimal" (unfeasible) representation.

Figure 5 shows how the space distributes across our structures. Almost half of the space is used for the treap differential encodings, the docids using about twice the space of the frequencies. The docids for frequency $f_0 = 1$, differentially encoded, use 41% of the space, which shows how relevant it is to avoid explicitly representing their frequency. Finally, the compressed treap topologies require only 11% of the total space.

5.3 Ranked Intersection

Figure 6 gives ranked intersection times for varying k, averaging over all the queries, and for $k = 10$ and $k = 20$, separating the queries by number of words (q). As noted in previous work [26], Dual-Sorted is unique in that it improves for longer queries, taking over for queries of 4–5 words or

more. Averaged over all the queries, it performs similarly to Block-Max, and both are superior to a Boolean intersection followed by a ranking (labeled "Intersection") implemented over our docid-sorted inverted index. None of these methods is much affected by k (which is expected for Dual-Sorted and Intersection since they always produce the full intersection and then rank the resulting documents).

Our treaps are more affected by the value of k, and as a consequence they are competitive only for k up to 20. For $k = 10$, treaps are indeed the fastest choice by a wide margin (up twice as fast as Block-Max, the closest competitor) for queries up to 4 words. For longer queries, as described, Dual-Sorted takes over. Treaps are still the fastest when averaging over all the queries in our benchmark. For $k = 20$, they are still the fastest choice for queries formed by 2 and 3 words, losing to Dual-Sorted for more words.

5.4 Ranked Union

Figure 7 shows the results for ranked union queries. Using a Boolean union as a filter for these queries is ineffective, so we use our frequency-sorted inverted index to implement an approximate ranked union, Persin et al.'s [32] (labeled "Persin" in the plots). Dual-Sorted also implements Persin et al.'s algorithm, so both report only approximate results. Only Block-Max and our treaps give exact results.

It can be seen that all the times worsen as k and q increase, more than linearly on q and sublinearly on k. Our treaps outperform Block-Max and Dual-Sorted for all k values up to 100. They are up to twice as fast as them, for example for 3-word queries. Treaps are only outperformed by the native Persin implementation, which however is not exact.

6. EXTENSIONS

In this section, we explore some more complex scenarios our treap-based representation of inverted indexes could handle, often more conveniently than current representations.

6.1 Beyond Exact Ranked Boolean Queries

It is not hard to adapt our algorithm for unions to the ranked version of the more general *thresholded* queries [7], which in addition to Q give a value $q' < q$, so that at least q' of the q query terms must appear in the reported documents. In this more general view, ranked unions correspond to $q' = 1$ and ranked intersections to $q' = q$. The more general Weak-AND operator [16] can also be easily supported. When the subset of the treaps t reaches value $id(v_t) = d$, we can evaluate document d and determine whether or not it qualifies.

On the other hand, approximate answers for ranked union queries have been the norm for decades, and our treap data structures can efficiently implement those as well. For example, we could easily implement Persin et al.'s [32] TAAT processing, even better than classical frequency-sorted lists. We could maintain the candidate set as a list sorted by docid. Each new treap that is processed is traversed in docid order, stopping at nodes where the threshold for considering documents is reached. As we produce the qualifying documents in docid order, we can simply merge them with the candidate set, without the need of more sophisticated structures. Furthermore, for subtrees of treap nodes whose frequency is below the threshold for inserting new documents in the candidate set, we can switch to a mode where the subtree

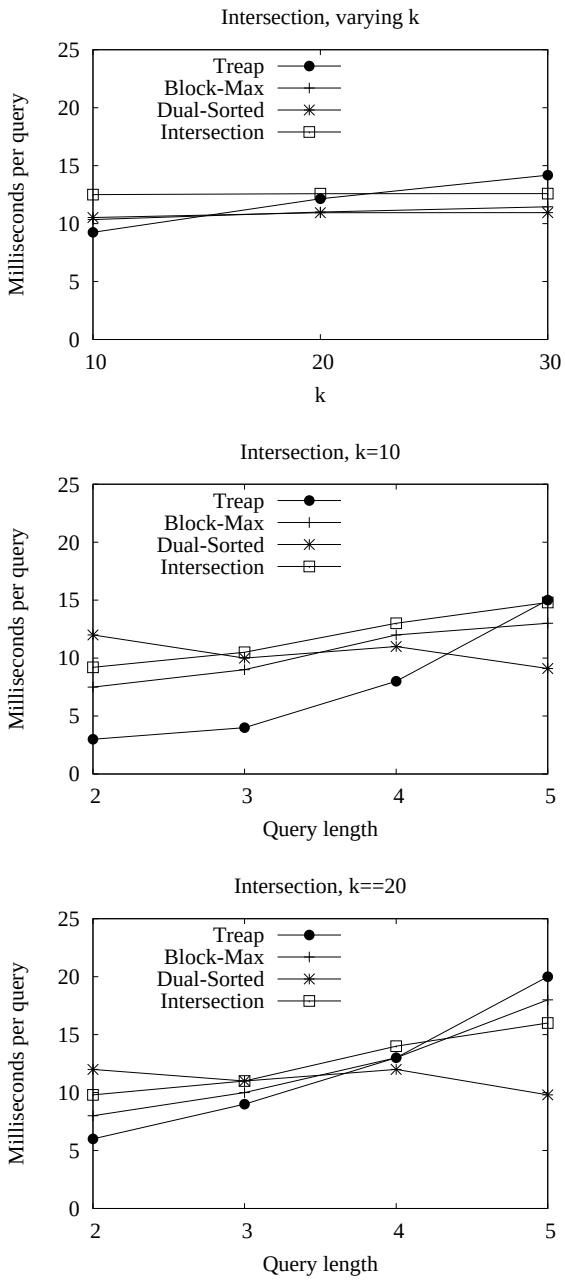


Figure 6: Time performance for ranked intersections. On top, for all the queries and increasing k. The other two discriminate by number of words in the query and use fixed $k = 10$ and $k = 20$.

is intersected with the candidates, using the next candidate docid to skip treap nodes.

6.2 Dynamic Representations

Our treap data structures can support insertions and deletions, so that the inverted index can be made dynamic. Such updates on treaps require logarithmic expected time [36], assuming that term frequencies are independent of docids. A deeper analysis [13, 11] gives somewhat stronger guarantees. Dynamic representations of trees and gap-encoded

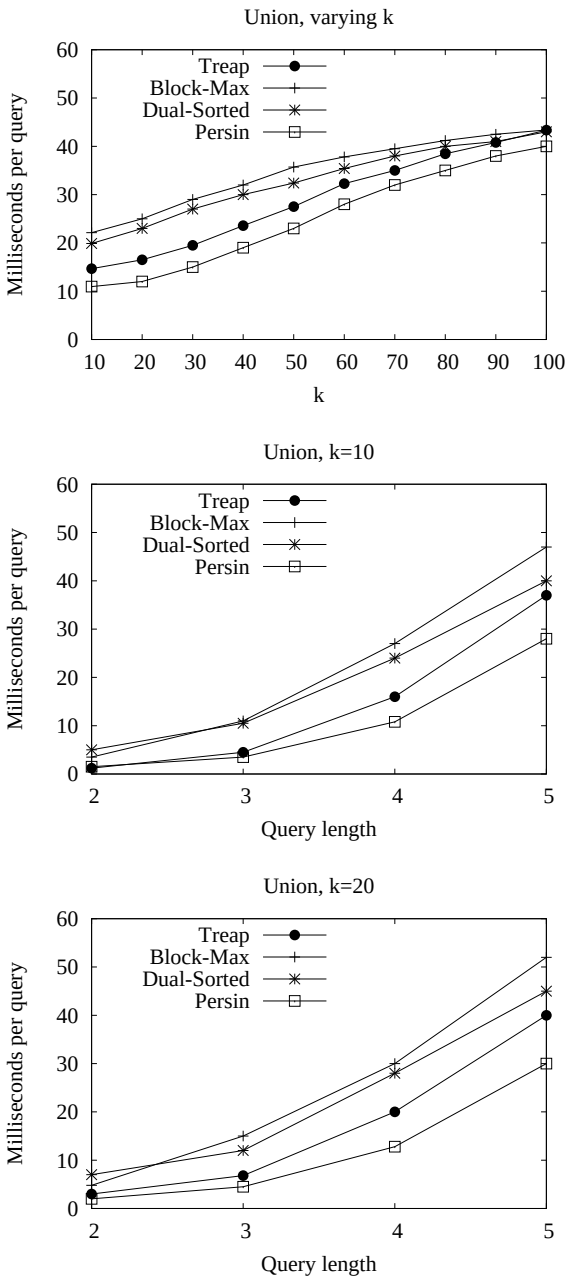


Figure 7: Time performance for ranked unions. On top, for all the queries and increasing k. The other two discriminate by number of words in the query and use fixed $k = 10$ and $k = 20$.

values [31] allow us to update those treaps in compressed form. However, more research on algorithm engineering is required to make those compressed dynamic structures efficient enough compared to their classical counterparts.

A well-known intermediate solution is to maintain the recently inserted documents in an uncompressed data structure, which is merged with the compressed body of the index when it reaches some threshold size. Queries are carried out on both the compressed and the new parts and the results are merged. Deletions are managed by marking the deleted documents, which are excluded from the answers, and periodically purguing the inverted index.

6.3 Range Restricted Queries

Cartesian trees are useful for *range maximum query (RMQ)* operations [10, 9, 23]. Given an array of values, an RMQ gives two endpoints and asks for the maximum value in the range. If we build the Cartesian tree of the array values, the RMQ reduces to a *lowest common ancestor (LCA)* query on the tree, which asks for the lowest node that is an ancestor of the two nodes that represent the extremes of the range. Furthermore, if we use the described isomorphism with general trees (as we actually do for representing the treaps) the LCA can be computed in constant time and without accessing the array of values [23].

This allows for another interesting use of the treaps. Imagine we wish to carry out a ranked query only on a range of docids. This restriction makes sense, for example, if documents have a timestamp (e.g., in versioned collections, periodic publications, etc.) and we wish to retrieve only documents within a range of times. Or the documents may be Web pages in lexicographic URL order and ranges of docids may correspond to domains, sites, or subdirectories. Our *explicit* treap spans all the documents, but we can *simulate* a treap on the range of documents $[d_1, d_2]$: the root of the simulated treap is document $d_r = RMQ(d_1, d_2)$. The left child of that root is $RMQ(d_1, d_r - 1)$ and the right child is $RMQ(d_r + 1, d_2)$, and so on.

As LCA operations are carried out in constant time and efficiently in current compact tree implementations [3], this gives an efficient technique to carry out range-restricted ranked unions and intersections. One problem is that our differential encoding of docids and frequencies does not work anymore, as we do not traverse the treap in top-down form but jump across nodes.

7. CONCLUSIONS AND FUTURE WORK

We have introduced a new inverted index representation based on the treap data structure. Treaps turn out to be an elegant and flexible tool to represent simultaneously the docid and the weight ordering of a posting list. We use them to design efficient ranked intersection and union algorithms that simultaneously filter out document by docid and frequency. The treap also allows us to represent both docids and frequencies in differential form, thus enabling better compression of the posting lists. Our experiments show significant gains in space and time compared to the state of the art: not only our structure uses 13%–18% less space than previous ones, but also it is faster (sometimes as much as twice as fast) for up to $k = 20$ on ranked intersections, and up to $k = 100$ on ranked unions.

In addition, we have described various other problems treaps could naturally solve. Many other questions remain unanswered and are subject of future work:

- How would the scheme perform with other scoring schemes? We used tf-idf for simplicity, but we could use BM25, impacts, etc. Some require to adapt the way we compute the upper bound U, such as the consideration of document sizes in BM25 (but this has been solved already [16, 22]).
- What would be the impact of rearranging docids in a convenient form? There is much recent research on

this topic (see, e.g., [22]) that shows that rearrangement can significantly improve both space and processing time. How much would treaps improve with such schemes? Can we optimize the rearrangement for a treap layout?

- How could we efficiently separate lists with frequencies higher than $f_0 = 1$? How would the space and time performance be affected?

8. REFERENCES

[1] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[2] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th SIGIR*, pages 372–379, 2006.

[3] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th ALENEX*, pages 84–97, 2010.

[4] R. Baeza-Yates, A. Moffat, and G. Navarro. Searching large text collections. In *Handbook of Massive Data Sets*, pages 195–244. Kluwer, 2002.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.

[6] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th SPIRE*, pages 13–24, 2005.

[7] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. 13th SODA*, pages 390–399, 2002.

[8] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14:art. 7, 2009.

[9] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 9th LATIN*, pages 88–94, 2000.

[10] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[11] I. Bialynicka-Birula. *Ranked Queries in Index Data Structures*. PhD thesis, University of Pisa, 2008.

[12] I. Bialynicka-Birula and R. Grossi. Rank-sensitive data structures. In *Proc. 12th SPIRE*, pages 79–90, 2005.

[13] I. Bialynicka-Birula and R. Grossi. Amortized rigidness in dynamic cartesian trees. In *Proc. 23rd STACS*, pages 80–91, 2006.

[14] G. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proc. 10th SPAA*, pages 16–26, 1998.

[15] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013.

[16] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. 12th CIKM*, pages 426–434, 2003.

[17] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[18] F. Claude, P. Nicholson, and D. Seco. Differentially encoded search trees. In *Proc. 22nd DCC*, pages 357–366, 2012.

[19] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Pearson Education, 2009.

[20] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, pages 137–148, 2007.

[21] E. Demaine, A. López-Ortiz, and J. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th SODA*, pages 743–752, 2000.

[22] S. Ding and T.Suel. Faster top-k document retrieval using block-max indexes. In *Proc. 34th SIGIR*, pages 993–1002, 2011.

[23] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal of Computing*, 40(2):465–492, 2011.

[24] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[25] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, NY, 1978.

[26] R. Konow and G. Navarro. Dual-sorted inverted lists in practice. In *Proc. 19th SPIRE*, pages 295–306, 2012.

[27] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1997.

[28] E. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[29] G. Navarro. Wavelet trees for all. In *Proc. 23rd CPM*, pages 2–26, 2012.

[30] G. Navarro and S. Puglisi. Dual-sorted inverted lists. In *Proc. 17th SPIRE*, pages 309–321, 2010.

[31] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 2012. To appear. `http://arxiv.org/abs/0905.0768`.

[32] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[33] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st SODA*, pages 134–149, 2010.

[34] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, 2007.

[35] 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.

[36] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[37] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th SIGIR*, pages 175–182, 2007.

[38] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[39] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. 34th SIGIR*, pages 105–114, 2011.

[40] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.

[41] 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.

[42] G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[43] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):art. 6, 2006.