Inverted Treaps

ROBERTO KONOW, Universidad Diego Portales and University of Chile
GONZALO NAVARRO, University of Chile
CHARLES L.A CLARKE, University of Waterloo
ALEJANDRO LÓPEZ-ORTÍZ, University of Waterloo

We introduce a new representation of the inverted index that performs faster ranked unions and intersections while using similar 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 different alternative compact data structures. Further, the treap invariants allow us to elegantly encode differentially both document identifiers and frequencies. We also show how to extend this representation to support incremental updates over the index. Results show that, under the tf-idf scoring scheme, our index uses about the same space as state-of-the-art compact representations, while performing up to 2–20 times faster on ranked single-word, union, or intersection queries. Under the BM25 scoring scheme, our index may use up to 40% more space than the others and outperforms them less frequently, but still reaches improvement factors of 2–20 in the best cases. The index supporting incremental updates poses an overhead of 50%–100% over the static variants in terms of space, construction and query time.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Compact Data Structure, Top-k document retrieval

1. INTRODUCTION

The central goal of modern Web search engines, as well as most other information retrieval systems, is to provide very precise results in response to user queries, by identifying a few relevant documents from usually huge text collections. They then face the two competing challenges of *quality* and *efficiency*: to provide a few documents best matching the users' needs and to find them within tenths of seconds.

In the most sophisticated information retrieval systems, these requirements are handled via a two-stage ranking process [Wang et al. 2011; Büttcher et al. 2010]. 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. These complex algorithms are too slow to be applied on the whole collection, and they obtain better quality as they have more time to run. Current systems are actually multi-stage, but the first is still the one that uses indexing to perform the most massive filtration.

In this article, we focus on improving the efficiency of the first stage, thus freeing more resources for the second stage to increase the quality of results. In contexts where

Funded with Fondecyt Grant 1-140796, Chile, with a Conicyt PhD Scholarship, Chile, and by the Emerging Leaders in the Americas Program, Government of Canada. Author's addresses: Roberto Konow and Gonzalo Navarro, Department of Computer Science, University of Chile, Santiago, Chile, rkonow@dcc.uchile.cl, gnavarro@dcc.uchile.cl; Charles L. A. Clarke and Alejandro López-Ortíz, David R. Cheriton School of Computer Science, University of Waterloo, Canada, claclark@gmail.com, alopez-o@uwaterloo.ca.
A preliminary partial version of this work appeared in *Proc. SIGIR 2013* [Konow et al. 2013].
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2015 Copyright held by the owner/author(s). 1046-8188/2015/01-ART1 $15.00
DOI: 0000001.0000001

simple 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 a set of the highest ranked documents containing either 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 [2011] showed that ranked intersections can be processed faster than Boolean intersections. They also obtained the best known performance for ranked unions, giving exact results and thus showing that ranked unions can be efficiently solved without resorting to approximations.

The *inverted index* is the central data structure in information retrieval systems. It stores a list per vocabulary word (or term) storing the documents where the term appears, plus a weight associated with the term in each document. This index can be stored on disk or in main memory, and in both cases reducing its size is crucial. On disk, it reduces transfer time when reading the lists of the query terms. In main memory, it increases the size of the collections that can be managed within a given RAM budget, or alternatively reduces the number of servers that must be allocated in a cluster to hold the index, the energy they consume, and the amount of communication.

Inverted indexes are possibly the oldest successfully compressed data structures (e.g., see Witten et al. [1999]). The main idea to achieve compression is to differentially encode either the document identifiers (docids) or the weights stored in the inverted lists, 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 [Witten et al. 1999; Baeza-Yates et al. 2002; Konow and Navarro 2012].

In this article we introduce a new compressed representation for the lists of the inverted index, which performs ranked intersections and (exact) ranked unions directly. Our representation is based on the *treap* data structure [Seidel and Aragon 1996], 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 operations) 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 produce the full Boolean result first.

We explore different alternatives to engineer the new list representation using state-of-the-art compact data structures to represent the treap topology. The classical differential representation of docids becomes less efficient on the treap, but in exchange the treap representation allows us to differentially encode *both* docids and weights, which compensates the loss. We also present novel algorithms for processing the queries on the treap structure, and compare their performance against well-known approaches such as WAND [Broder et al. 2003], Block-Max [Ding and Suel 2011], and Dual-Sorted [Konow and Navarro 2012]. Our experiments under the classical tf-idf scoring scheme show that the space usage of our treap-based inverted index representation is competitive with the state-of-the-art compressed representations: our faster variant is around 25% larger than WAND, as large as Block-Max, and 15% smaller than Dual-Sorted. In terms of time, the fastest inverted treap variant outperforms previous techniques in many cases: on ranked one-word queries it is 20 times faster than Dual-Sorted and 25–200 times faster than Block-Max; WAND is even slower. On ranked unions it is from 10 times (for $k = 10$) to 3 times (for $k = 1000$) faster than Block-Max, and similarly from 45–50 times to 6–7 times faster than WAND and Dual-Sorted. It is always the fastest index for one- and two-word queries, which are the most popular. On ranked

intersections, our fastest treap alternative is twice as fast as Block-Max for $k = 10$, converging to similar times for $k = 1000$. In the same range of k, it goes from twice as fast to 40% faster than WAND, and from 80% faster to 10% slower than Dual-Sorted. Our inverted treap is always the fastest index up to $k = 100$. We also experimented under a quantized BM25 scoring scheme, where the fastest inverted treap uses 35%–40% more space than Block-Max or Dual-Sorted. It is still slightly better than all the alternatives on ranked unions. For ranked intersections, it is from twice as fast (for $k = 10$) to 15% slower (for $k = 1000$) on ranked intersections, still being the fastest for $k = 100$. On one-word queries, the inverted treap is 20 times faster than Dual-Sorted and 40–300 times faster than Block-Max.

Those ranges of k values make this result very relevant both for a first stage retrieving a few hundreds or thousands of documents, or for directly conveying a handful of final results to the user. The technique can also be used in large-scale distributed systems where each node contributes a small set of documents to the global result. We also show how to support incremental updates on the treap, making this representation a useful alternative for scenarios where new documents must be available immediately. The overhead of allowing for incremental updates compared to our static alternatives, in terms of space, construction and query times, ranges from 50% to 100%.

This article is structured as follows. Section 2 presents basic concepts. Section 3 provides a discussion on the related work and Section 4 describes the new representation of lists using treaps. Section 5 describes in detail the top-k query processing algorithms. Section 6 shows how extend the treap representation to support insertions of documents (incremental updates). Section 7 evaluates the performance of our structure and compares it with the state of the art. We discuss the results in Section 8.

2. BASIC CONCEPTS

2.1. Inverted Index

The inverted index is an old and simple, yet efficient, data structure that is at the heart of every modern information retrieval system, and plays a central role in any book on the topic [Witten et al. 1999; Büttcher et al. 2010; Baeza-Yates and Ribeiro-Neto 2011; Croft et al. 2009]. Let a text collection contain a set of D documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$. A document d_i can be regarded as a sequence of terms or words and the number of words in the document is denoted by $|d_i|$. The total length of the collection is then $\sum_{i=1}^{D} |d_i| = n$. Each document has a unique *document identifier* ($docid$) $\in [1, D]$. The set of distinct terms in the collection is called the *vocabulary*, which is comparatively small in most cases [Heaps 1978], more precisely of size $O(n^\beta)$, for some constant $0 < \beta < 1$ that depends on the text type. The inverted index 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 vocabulary of the collection, and the lists contain one element per distinct document where the term appears. For each term, the index stores in the list the *document identifier* ($docid$), the *weight* of the term in the document and, if needed, the positions where the term occurs in the document. The weight of the term in the document is a utility function that represents the importance of that word inside the document. The main components of the inverted index are then defined as the *vocabulary* and the *inverted lists*:

— **The Vocabulary.** The vocabulary stores all distinct terms contained in the collection of documents $\mathcal{D}$. This is commonly implemented with a dictionary data structure such as a hash table or a digital tree (also called a trie). In practice, the vocabulary stores two elements associated to each term: an integer value df_t called the *document frequency*, which is the number of documents that contain the term t, and a pointer to the start of its corresponding *inverted list*.

d_1 =	a	long	time	ago	in	a	galaxy	far	far	away
d_2 =	try	not	do	or	do	not	there	is	no	try
d_3 =	that	is	not	true						

Vocabulary / Inverted Lists $\langle d, w(t,d) \rangle$

df_t	Term t	Inverted List
1	a	⟨1,2⟩
1	ago	⟨1,1⟩
1	away	⟨1,1⟩
1	do	⟨2,2⟩
1	far	⟨1,2⟩
1	galaxy	⟨1,1⟩
1	in	⟨1,1⟩
2	is	⟨2,1⟩,⟨3,1⟩
1	long	⟨1,1⟩
1	no	⟨2,1⟩
2	not	⟨2,1⟩, ⟨3,1⟩
1	or	⟨2,1⟩
1	that	⟨3,1⟩
1	there	⟨2,1⟩
1	time	⟨1,1⟩
1	true	⟨3,1⟩
1	try	⟨2,2⟩

Fig. 1. An example non-positional inverted index built over the collection of documents at the top. The vocabulary and the postings lists built over the collection are shown on the bottom.

— **Inverted Lists.** A *non-positional inverted list* stores a list of elements containing pairs $\langle d, w(t, d)\rangle$, where d is the document identifier ($docid$) and $w(t, d)$ is a relevance measure of the term t in document d. A *positional inverted list* contains a list of triples $\langle d, w(t, d), \langle p_1, p_2, \cdots, p_k\rangle\rangle$, where the third component is a vector of the positions where the occurrences of term t are located in the document d.

Figure 1 shows an example of an inverted index for an example collection consisting of three documents. The inverted index from the example shows a *docid-sorted* organization, where each posting list is in increasing docid order. The postings lists could also follow a *weight-sorted* organization. Both docid-sorted and weight-sorted offers different alternatives in terms of compression techniques and query processing strategies that are discussed in Section 3.1.

2.2. Scoring

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

$$score(Q, d) = \sum_{t \in Q} w(t, d), \tag{1}$$

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* defined above. As explained, the

terms df_t are stored in the vocabulary, whereas $tf_{t,d}$ must be stored in the posting list of term t, together with the docid d. This is an efficient way to record $w(t, d)$ in the tf-ifd model. However, state-of-the-art information retrieval systems employ more complex ranking formulas such as Okapi BM25, which does not require much more space than a tf-idf scheme but requires more complex calculations.

Computing the ranking score can be a major bottleneck of the system's query processing routines. Pre-computing the scores and storing them as a float number would require between 24 and 32 bits. This alternative is usually unfeasible, since it increases the size of index significantly. To speed up the score calculation process and also maintain a reasonable size of the index we can *discretize* the range of possible scores into a predefined set of buckets. Anh et al. [2001] proposed an uniform quantization method which is a index-wide linear scaling of the term weights, in other words, the idea is to precompute and store the *impact score* for the term weight $I(d, t)$ using the following formula:

$$I(t, d) = \left\lfloor \frac{w(t, d) - min(w(t, d))}{max(w(t, d)) - min(w(t, d))} \times 2^b \right\rfloor, \quad (2)$$

where b is the number of bits that we are willing to reserve for each score contribution. Crane et al. [2013] shows that by setting $b = 8$ this quantization method achieves an effectiveness that is indistinguishable from using exact term weights.

2.3. Query Processing

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 for the first stage are hundreds to thousands, as discussed earlier. In simpler one-stage systems, typical values of k are below 20. In the *ranked intersection* model, all the terms in Q must appear in returned documents. In the *ranked union* model, instead, a missing term t simply implies that $w(t, d) = 0$. Ranked intersection was popularized by Web search engines to favor precision over recall, and is nowadays more common than ranked union.

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 [Demaine et al. 2000; Baeza-Yates and Salinger 2005; Sanders and Transier 2007; Barbay et al. 2009; Konow and Navarro 2012]. 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 [Ding and Suel 2011]. 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 [Persin et al. 1996; Anh and Moffat 2006]. Most of these techniques order the posting lists by decreasing weight values, not by docids. Recently, it has been shown that ranked unions can be solved in exact form within reasonable time [Broder et al. 2003; Strohman and Croft 2007; Ding and Suel 2011] by using increasing docid order for the posting lists in the best solution [Ding and Suel 2011].

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 [Demaine et al. 2000; Scholer et al. 2002; Baeza-Yates and Salinger 2005; Sanders and Transier 2007; Culpepper and Moffat 2007; Strohman and Croft 2007; Barbay et al. 2009; Konow and Navarro 2012]. 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. In this case, it is most interesting that an individual inverted index solves top-k queries efficiently for k values in the range 10–100 in the two-stage model [Büttcher et al. 2010].

2.4. Compact Data Structures

A *compact data structure* is a data structure that is represented within little space, ideally close to the compressed data size, and still offers the desired functionality. We describe the main compact data structures employed in our work.

2.4.1. Rank and Select on binary sequences. Binary sequences or bitvectors are a fundamental part of many compact data structures. A bitvector $B[1, n]$ stores a sequence of n bits and provides efficient solutions for three basic operations:

— ACCESS (B, k) returns the bit at position k of the sequence.
— RANK_b (B, i) returns the number of bits equal to b up to position i in B.
— SELECT_b (B, j) returns the position of the j-th occurrence of bit b in B.

All of these operations can be solved in constant time using $n + o(n)$ bits [Munro 1996]. In this article we use an implementation spending, in practice, 5% extra space on top of the original bitvector size and providing fast query processing [González et al. 2005].

2.4.2. Compact trees. There are $\Theta(4^n/n^{3/2})$ general trees of n nodes, and thus one needs $\log(4^n/n^{3/2}) = 2n - \Theta(\log n)$ bits to represent any such tree. There are various compact tree representations using $2n + o(n)$ bits that can in addition carry out many tree operations efficiently, including retrieving the first child, next sibling, computing postorder of a node, lowest common ancestor, and so on. We describe two compact tree representations: Balanced Parentheses (BP) and Level-Ordered Unary Degree Sequence (LOUDS)

BP. Balanced Parentheses were introduced by Jacobson [Jacobson 1989] and later improved to achieve constant time operations [Munro and Raman 2002]. It represents a tree by doing a depth-first traversal: an *opening* parenthesis is written when arriving at a node for the first time and a *closing* parenthesis is written after traversing the subtree of the node, therefore each node will generate two parentheses. The parentheses sequence is represented using a bitvector by assigning a '1' to the opening parenthesis and a '0' to the closing one. The representation then requires $2n$ bits. In practice [Arroyuelo et al. 2010], this representation requires $2.37n$ bits, since the bitvector requires additional data structures to process RANK/SELECT and other queries.

LOUDS. The Level Ordered Unary Degree Sequence [Jacobson 1989] is a simpler, yet efficient mechanism to represent ordinal trees. We start with an empty bitvector T and traverse the tree in a level-order fashion starting from the root. As we visit a node v with $d \geq 0$ children we append 1^d0 to T. We need to augment T only with RANK and SELECT operations to support basic navigation operations such as PARENT and CHILD in $O(1)$ time. However, the repertoire of LOUDS is more limited than that of by BP, as it excludes more complex operations such as SUBTREE_SIZE or LOW-

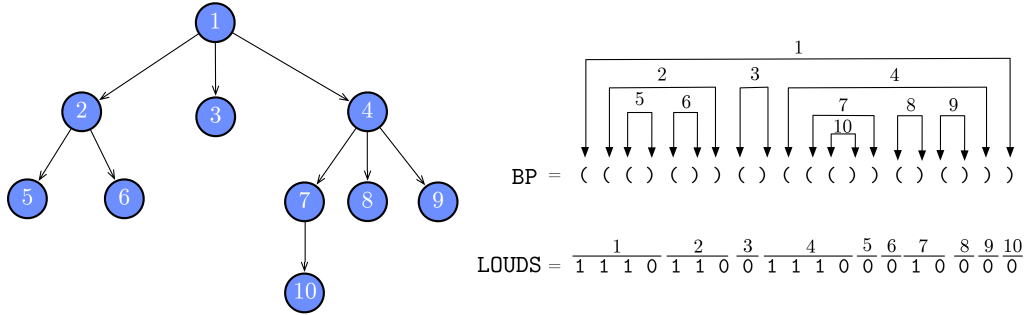

Fig. 2. Example of compact representations of a tree using BP and LOUDS.

EST_COMMON_ANCESTOR. In practice, since we can give RANK/SELECT support using 5% of extra space, a LOUDS representation uses $2.10n$ bits.

Figure 2 shows an example of these schemes. The left part shows a tree with nodes numbered levelwise. The right part shows the corresponding compact representations of the topology and where each node is located.

2.5. Direct Access Codes (DACs)

Directly Addressable Codes (DACs) [Brisaboa et al. 2013] is a variable-length encoding of integers. Given a chunk length b and a sequence of positive integers $\mathcal{S}[1,n] = x_1, x_2, \ldots, x_n$, DACs divide each integer x_i into $\lceil |x_i|/b \rceil$ chunks. Similar to vByte encoding [Williams and J.Zobel 1999], DACs employ a 'continuation' bit to indicate whether the code continues in further chunks. Instead of concatenating the encoding of x_{i+1} after that of x_i, however, they build a multi-layer data structure. Let M be the maximum element in the sequence; a DACs encoding will contain $\ell = \lceil \lfloor \log(M) \rfloor + 1/b \rceil$ layers consisting in two parts: (1) the lowest b bits of each chunk, which are stored contiguously in an array A_k with $1 \leq k \leq \ell$, and (2) the 'continuation' bits, which are concatenated into a bitvector B_k. An integer x_i will require exactly $\lceil |x_i|/b \rceil$ layers. For example, say we want to encode the integer $x = 6$ and we set the chunk length $b = 2$. The lowest b bits of 6 are 10, so we append them to array A_1. Since we still require more bits to represent the number, we append the continuation bit 1 to B_1. The next b lowest bits for representing x, 01 are now appended to A_2, and since we do not require more bits to represent x, we append the continuation bit 0 to B_2.

Any integer x_i can be extracted from the successive arrays A_k and bitvectors B_k by using RANK on the bitvectors to track the positions of the chunks in the arrays. Then the extraction time is $O(1 + |x_i|/b)$, and the number of wasted bits used to represent it is at most $\lceil |x_i|/b \rceil + b - 1$. A further improvement of DACs encoding is to choose different chunk lengths for each layer. The authors present a dynamic programming algorithm that computes the optimal chunk length and the optimal number of layers to achieve the smallest representation of the sequence of numbers.

3. RELATED WORK

3.1. Query Processing Strategies

There are different query evaluation mechanisms that exhibit affinities with different index organizations. They are classified in three different categories: Document-at-a-time (DAAT), Term-at-a-time (TAAT) and Score-at-a-time (SAAT).

Document-at-a-time. DAAT processing is more popular for Boolean intersections 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 [Demaine et al. 2000; Baeza-Yates and Salinger 2005; Sanders and Transier 2007; Culpepper and Moffat 2007; Barbay et al. 2009]. 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.

Term-at-a-time. 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. The documents in each list are sorted by decreasing weight. TAAT is especially popular for processing ranked unions, 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. 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 [Persin et al. 1996; Anh and Moffat 2006]. A more sophisticated approach based on similar ideas can be used to guarantee that the answer is exact [Strohman and Croft 2007]. TAAT approaches usually make use of an *accumulator* data structure that holds the intermediate accumulated results for each document.

Score-at-a-time. SAAT mechanism can be seen as a hybrid between document-at-a-time and term-at-a-time in which multiple index lists are open. This is usually employed with *impact-sorted* indexes [Anh and Moffat 2006]. In impact-ordered indexes the actual score contributions of each term are precomputed and quantized into what are known as *impact scores* [Lin and Trotman 2015]. The idea is that a complete impact block is processed at each step, and the results are obtained using a set of accumulator variables.

Many ranked intersection strategies employ a full Boolean intersection followed by a post processing ranking step. However, recent work has shown that it is possible to do better using DAAT strategies [Ding and Suel 2011]. 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 establish 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 [Broder et al. 2003; Ding and Suel 2011]. The strategies we use for solving ranked union and intersection queries in this article 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

Compression of the inverted lists is essential for efficient retrieval, on disk and in main memory [Witten et al. 1999; Büttcher and Clarke 2007]. The main idea to achieve compression is to differentially encode the docids, whereas the weights are harder to compress. A list of docids $\langle d_1, d_2, d_3, \ldots d_n \rangle$ is represented as a sequence of d-gaps $\langle d_1, d_2 - d_1, d_3 - d_2, \ldots, d_n - d_{n-1} \rangle$, and variable-length encoding is used to encode the

differences. Extracting a single list or merging lists is done optimally by traversing lists from the beginning, but query processing schemes can be done much faster if random access to the sequences is possible. This can be obtained by cutting the lists 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 [Culpepper and Moffat 2007; Sanders and Transier 2007].

Bit-aligned codes can be inefficient to decode, since they require several bitwise operations. Byte-aligned [Scholer et al. 2002] or word-aligned codes [Yan et al. 2009; Culpepper and Moffat 2005] are preferred when speed is the main concern. Examples of these techniques are Variable Byte (VByte) and Restricted-Prefix Byte Codes.

Another approach is to encode blocks of integers together, aiming to improve both compression and decoding speed. One popular block-based encoding mechanism is Simple9 or Simple16 [Anh and Moffat 2005], which encodes as many as possible of the next values using fixed-width fields in a 32-bit word. Another is PforDelta, which encodes the next, say, 128 numbers using fixed-width cells, while encoding separately the largest 10% of the numbers as outliers. In practice, PforDelta is one of the fastest for decoding and achieves excellent compression ratios.

A recent trend exploits the advantages of SIMD operations available on modern CPUs. For example, Variant-G8IU [Stepanov et al. 2011], encodes as many integers as possible into 8 consecutive bytes preceded by a one-byte descriptor, and uses SIMD instructions to encode/decode. Another example is SIMD-BP128 [Lemire and Boystov 2015], which can be seen as an adaptation of Simple9 to use 128-bit SIMD words.

Recently, Trotman [2014] introduced a new mechanism to encode integers, called *QMX*. This encoding scheme combines word-aligned, SIMD and run-length techniques, resulting in a highly space-efficient and fast-decoding scheme.

Vigna [2013] explored an *Elias-Fano* representation of monotone sequences to encode the docids of the posting lists. The representation allows for direct random access to any docid, and also for efficient skipping operations that are useful for performing intersections. His experiments showed that Elias-Fano achieves space and time competitive with state-of-the-art methods, even including PforDelta. Ottaviano and Venturini [2014] extended this idea by performing an $(1+\epsilon)$-optimal partitioning of the list into chunks, for any $\epsilon > 0$, and then encoding all the chunks and their endpoints with Elias-Fano. They show that this partitioned approach offers significantly better compression and similar query time performance, compared to representing the sequence as a whole.

When the 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 values) are frequent, and thus not only run-length encode them, but also sort the corresponding docids increasingly, so as to encode them differentially [Baeza-Yates et al. 2002; Zobel and Moffat 2006].

3.3. State of the Art for Exact Ranked Queries

The following approaches display the best performance to date for exact ranked intersections and unions. We also highlight the advantages of our work with respect to them.

3.3.1. Weak-And (WAND) . WAND [Broder et al. 2003] is a query processing algorithm that performs weighted unions following a DAAT strategy. Each term t_i in the query is assigned a weight w_i, and a threshold θ is established. The method returns the documents containing query terms t_i whose sum of weights w_i reaches θ.

The WAND algorithm traverses the lists of all the query terms in parallel, carrying out three phases in each iteration: pivot selection, alignment check, and evaluation. The procedure starts by selecting a pivoting term. This is done by sorting the lists by their current docid and summing the corresponding weights w_i until the sum reaches θ. The term where this happens is selected as the pivot. The key observation is that there is no docid smaller than the pivot's current docid that could reach a sum reaching θ, unless it was already considered. Next, WAND tries to align all the posting lists to the pivot's current docid, by moving forward the pointers in these lists. If the sum of the weights w_i of the entries where the pivot's docid appears reaches θ, the docid is reported. The process then continues by moving forward the pointer in the pivot list and then starting a new iteration.

In this context, a Boolean union query with q terms can be obtained by setting all the weights to $w_i = 1$ and $\theta = 1$, and a Boolean intersection by raising the bar to $\theta = q$.

In the context of top-k ranked unions, we can maintain the k docids with maximum score we have seen, and set θ dynamically as the kth largest score known. The weights w_i are also dynamic: they are the weight of the term in the current docid. To choose the pivot, we have a precomputed upper bound on w_i in the inverted index, associated with the term. Therefore, if a pivot is chosen accordingly to upper bounds on weights, we are not losing any relevant previous pivot. To obtain ranked intersections, we must also enforce that all the terms appear in the docid; note that the result is not different from performing a Boolean intersection, except that we may prune the consideration of a docid if it is clear that it will not make it to the top-k list.

In this article we use WAND to perform ranked unions and intersections, as described. For this purpose, WAND has been superseded by the more complex Block-Max, described next.

3.3.2. Block-Max. Block-Max [Ding and Suel 2011] 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 whole blocks whose maximum possible contribution is very low, by comparing its maximum weight with a threshold given by the current candidate set. Block-Max obtains considerable performance gains over the previous techniques for exact ranked unions [Broder et al. 2003; Strohman and Croft 2007], 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, d, 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, then no document inside the current blocks can make it to the top-k list. So we can safely skip the least advanced block.

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 it is practical and beats the flat Block-Max. In addition, the treap structure allows us to differentially encode both docids and weights, which translates into space savings.

3.3.3. Dual-Sorted inverted lists. Dual-Sorted inverted lists [Navarro and Puglisi 2010; Konow and Navarro 2012] represent the posting lists sorted by decreasing frequency, using a wavelet tree data structure [Grossi et al. 2003; Navarro 2012]. The wavelet tree efficiently simulates ordering by increasing docids as well. 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. INVERTED TREAPS

We describe our data structure in this section. First, we survey the treap data structure and show that it can be used to represent a posting list. Then we describe how we represent the resulting data structure using little space. At the end, we describe some practical improvements on the basic idea.

4.1. The Treap Data Structure

A *treap* [Seidel and Aragon 1996] 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 key of a node is larger than those of its left subtree and smaller than those of its right subtree. Furthermore, the treap satisfies the invariants of a binary heap with respect to the priorities: the priority of the parent is equal to or 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 [Seidel and Aragon 1996; Martínez and Roura 1997; Blelloch and Reid-Miller 1998] to ensure logarithmic expected height independently of the order of insertions, a treap can also be seen as the *Cartesian tree* [Vuillemin 1980] of the sequence of priorities once the values are sorted by keys.

Treaps are a particular case of *priority search trees* [McCreight 1985], 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 [Bialynicka-Birula and Grossi 2005; Bialynicka-Birula 2008] but, as far as we know, our use 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, then all the docids below it in the treap can be discarded as well.

Figure 3 illustrates a treap representation of a posting list. This treap will be used as a running example. Ignore for now the differential arrays on the bottom.

4.3. Construction

A treap on a list $\langle (d_1, w_1), \ldots, (d_n, w_n) \rangle$ of documents and weights can be built in $O(n)$ time in a left-to-right traversal [Berkman and Vishkin 1993; Bender and Farach-Colton 2000; Fischer and Heun 2011]. Initially, the treap is just a root node (d_1, w_1).

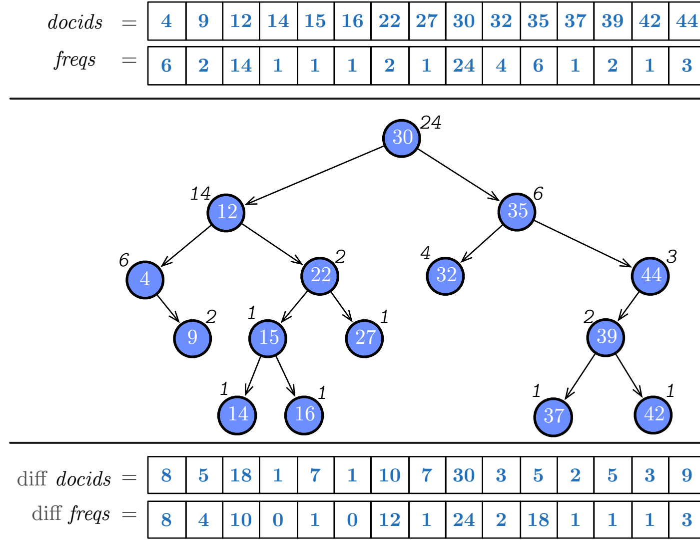

Fig. 3. An example posting list with docids and frequencies and the corresponding treap representation in our scheme. Note that docids (inside the nodes) are sorted inorder and frequencies (outside the nodes) are sorted top to bottom. The differentially encoded docids and frequencies are shown below the tree.

Now, assume we have processed $(d_1, w_1), \ldots, (d_{i-1}, w_{i-1})$ and the rightmost path of the treap, root to leaf, is $v'_1, \ldots, v'_\ell$, each v'_j representing the posting (d'_j, w'_j). Then we traverse the path from v'_ℓ to v'_1, until finding the first node v'_j with $w'_j \geq w_i$ (assume the treap is the right child of a fake root with weight $w'_0 = +\infty$, to avoid special cases). Then, (d_i, w_i) is set as the right child of v'_j and the former right child of v'_j becomes the left child of (d_i, w_i), which becomes the lowest node in the rightmost path.

Since for every step in this traversal the rightmost path decreases in length, and it cannot increase in length by more than 1 per posting, the total number of steps is $O(n)$. Under reasonable assumptions (i.e., the weight and the docid are statistically independent) the height of a treap is $O(\log n)$ [Martínez and Roura 1997], and so is the length of its rightmost path. Therefore, the maximum time per insertion is $O(\log n)$ expected (but $O(1)$ worst-case when amortized over all the insertions on the treap).

4.4. Compact Treap Representation

To represent this treap compactly we must encode the tree topology, the docids, and the term frequencies. We discuss only the docids and frequencies in this subsection. Our plan is not to access the posting lists in sequential form as in classical schemes, thus a differential encoding for 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 $id(v) - id(u)$ [Claude et al. 2012]. 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 show on the bottom of Figure 3. 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.

To do this we need to randomly access a differential value in the sequence, given a node. We store those values in an array indexed by node inorders and use the Direct Addressable Codes (DACs) described in Section 2.5 to directly access the values while taking advantage of their small size.

4.5. Representing the Treap Topology

We have shown that if we use a treap to represent posting lists we can differentially encode both docids and frequencies, however, we still need to represent the topology. A pointer-based representation of a treap topology of n nodes requires $O(n \log n)$ bits, which is impractical for large-scale data. Still, space is not the only concern: we need a compact topology representation that supports fast navigation in order to implement the complex algorithms deriving from ranked intersections and unions. In this subsection we introduce three representations designed to be space-efficient and to provide fast navigation over the topology.

4.5.1. Compact Treap using BP. This representation uses the balanced parentheses (BP) described in Section 2.4.2. However, this representation is designed for general ordinal trees, not for binary trees. For example, if a node has only one child, general trees cannot distinguish between it being the "left" or the "right" child.

A well-known isomorphism [Munro and Raman 2002] allows us represent a binary tree of n nodes using a general tree of $n+1$ nodes: First, a fake root node v_{root} for the general tree is created. The children of v_{root} are the nodes in the rightmost path of the treap, from the root to the leaf. Then each of those nodes is converted recursively. The general tree is then represented as a BP sequence $\mathcal{S}_{\text{BP}}[1, 2n+2]$. With this transformation, the original treap root is the first child of v_{root}. The left child of a treap node v is its first child in the general tree and the right child of v is its next sibling in the general tree. Moreover, the inorder in the original treap, which we use to access docids and frequencies, corresponds to the preorder in the general tree, which is easy to compute with parentheses. Figure 4 shows the transformed treap for our running example.

Each treap node i is identified with its corresponding opening parenthesis. Thus the root of the treap is node 2. The inorder of a node i is simply $\text{RANK}_1(\mathcal{S}_{\text{BP}}, i) - 1$, the number of opening parentheses up to i excluding the fake root. The left child of i in the treap, or its first child in the general tree, is simply $\text{FIRST_CHILD}(i) = i+1$. If, however, $\mathcal{S}_{\text{BP}}[i+1] = 0$, this means that i has no first child. For the right child of node i in the treap, or its next sibling in the general tree, we use $\text{NEXT_SIBLING}(i) = \text{CLOSE}(i) + 1$, where CLOSE returns the closing parenthesis that matches a given opening parenthesis. If $\mathcal{S}_{\text{BP}}$ has a 0 in the resulting position, this means that i has no right child.

Therefore, in addition to RANK, we need to implement operation CLOSE in constant time. This is achieved with a few additional data structures of size $o(n)$, as described in Section 2.4.2.

As an example, we demonstrate this procedure using the treap from Figure 4. We see that v_1 (the treap's original root) starts at position $i = 2$ in $\mathcal{S}_{\text{BP}}$. If we want to retrieve the index of its left child, which is node v_2, we perform $\text{FIRST_CHILD}(2)$. The procedure checks that $S_{\text{BP}}[3]$ is an opening parenthesis, so we can return 3 as the answer for the starting position of the left node of v_1. On the other hand, if want to obtain the position where the right child of v_1 begins, which is v_3, we perform $\text{CLOSE}(2) + 1$ and obtain position 20, which contains a 1 and thus corresponds to the starting position of v_3.

4.5.2. Compact treap using LOUDS. The LOUDS representation (recall Section 2.4.2) can be adapted to support binary trees efficiently: for every node in a level-order traversal we append two bits to the bit sequence, setting the first bit to '1' iff the node contains

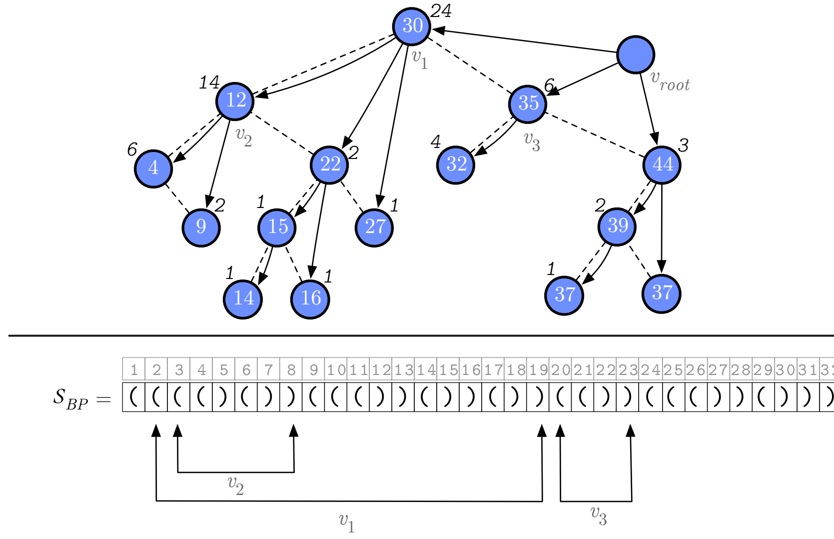

Fig. 4. The original binary tree edges (dashed) are replaced by a general tree, whose topology is represented with parentheses. The opening and closing parentheses of nodes v_1, v_2 and v_3 in $\mathcal{S}_{\text{BP}}$ are shown on the bottom part.

a left child and setting the second to '1' iff the node contains a right child. For example, a leaf node will be represented as 00, while a binary tree node containing both children is represented as 11. The concatenation of these bits builds a bit sequence $\mathcal{S}_{\text{LOUDS}}[1, 2n]$. Since every node adds two bits to this sequence, we can use the levelwise order i of a node as its identifier, knowing that its two bits are at $\mathcal{S}_{\text{LOUDS}}[2i-1, 2i]$.

The LOUDS representation is simpler than BP, as it can be navigated downwards using only RANK operations. In addition, it does not require the tree isomorphism. Given the sequence $\mathcal{S}_{\text{LOUDS}}$ representing the treap topology as a binary tree and a node i, we navigate the tree as follows: if i has a left child (i.e., if $\mathcal{S}_{\text{LOUDS}}[2i-1] = 1$) then the child is the node (with levelwise order) $\text{RANK}_1\,(\mathcal{S}_{\text{LOUDS}}, 2i-1)+1$. Analogously, the right child exists if $\mathcal{S}_{\text{LOUDS}}[2i] = 1$, and it is $\text{RANK}_1\,(\mathcal{S}_{\text{LOUDS}}, 2i) + 1$.

Figure 5 shows an example of a binary tree using a LOUDS representation. We demonstrate how to navigate the tree with an example: say that we are at node v_4 (meaning its LOUDS identifier is 4), which is represented by the bits located at positions 7 and 8. We know that v_4 has no left child because the first bit is 0, but it has a right child since the second bit is a 1. The right child of v_4 is the node with identifier (or levelwise order) $\text{RANK}_1\,(\mathcal{S}_{\text{LOUDS}}, 2\cdot 4) + 1 = 8$, which we draw as v_8. The two bits of this node are at positions 15 and 16. Since both bits are set to 0, this node is a leaf.

Compared to the BP representation, the LOUDS-based solution requires less space in practice (2.10 bits per node instead of 2.37) and simpler operations. On the other hand, the BP representation has more locality of reference when traversing subtrees.

For this representation, we store the sequences of differentially encoded docids and frequencies (sequences "diff *docids*" and "diff *freqs*" of Figure 3) following the level-order traversal of the binary tree. This ordering is shown as "Node" at the bottom part of Figure 5.

4.5.3. Compact treap using Heaps. Even if the topology representations using LOUDS or BP support constant-time tree navigation, in practice they are 10 to 100 times slower

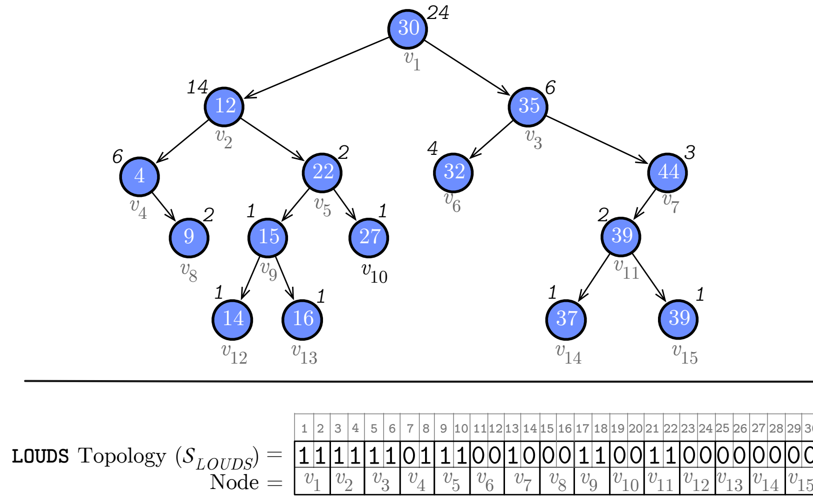

Fig. 5. The LOUDS representation of an example treap. $\mathcal{S}_{\text{LOUDS}}$ denotes the bit sequence that describes the topology.

than a direct access to a memory address (i.e., accessing any position in an array), as shown in the preliminary version of this work [Konow et al. 2013]. In order to avoid these costly operations, we designed a new compact binary tree representation that is inspired on binary heaps. The main idea is to take advantage of the fact that a *complete* binary tree of n nodes does not require any extra information to represent its topology, since the values can be represented using just an array of size n. In order to navigate the tree, we can use traditional binary heap operations: the left child of node i is located at position $2i$ and the right child at position $2i+1$. However, a treap posting list representation will rarely be a complete binary tree. Therefore, we take the maximal top part of the tree that is complete and represent it as a heap. We then recursively represent the subtrees that sprout from the bottom of the complete part. The motivation is to avoid the use of RANK or more complex operations every time we need to navigate down the tree.

We start at the root of the treap and traverse it in levelwise order, looking for the first node that does not contain a left or a right child. Say that this happens at level ℓ of the tree, so we know that all the nodes up to level ℓ have both left and right children. In other words, the subtree formed by all the nodes starting from the root up to level ℓ forms a complete tree $\mathcal{T}_1$ that contains $2^\ell - 1$ nodes. Figure 6 shows an example, where the first node that does not have a left or right child (v_4) is located at level $\ell = 3$, therefore $|\mathcal{T}_1| = 2^3 - 1 = 7$. We then append the differential values of the nodes (docid and frequency) of the complete subtree to the sequences "diff *docids*" and "diff *freqs*" (see Figure 3), in levelwise order.

For each of the $2^{\ell-1}$ leaves of $\mathcal{T}_1$ we write in a bit sequence $\mathcal{S}_{\text{HEAP}}$ two bits, indicating if the corresponding leaf contains a left or a right child, similarly to LOUDS.

We continue this procedure by considering each node located at level $\ell+1$ as a new root, for further trees $\mathcal{T}_j$. The trees yet to process are appended to a queue, so the trees $\mathcal{T}_j$ are also deployed levelwise. Note that any of these new roots (including the original root) could lack a left or a right node, in which case we will have a complete subtree of only one node.

We also need a sequence P where $P[j]$ is the starting point of $\mathcal{T}_j$ in the sequences of docids and frequencies. This also serves to compute $|\mathcal{T}_j| = P[j+1] - P[j]$. Since there may be a considerable number of small complete trees, P may require up to $n \log n$ bits. To reduce its size, and considering that $P[j+1] - P[j]$ is of the form $2^\ell - 1$, we store another array instead: $P'[0] = 0$ and $P'[j+1] = \ell - 1 = \log_2(|\mathcal{T}_j| + 1) - 1$. This reduces the space to at most $n \log \log n$ bits, and the starting position of the sequences of $\mathcal{T}_j$ can be obtained as $P[j] = \sum_{i=0}^{j-1}(2^{P'[i]+1} - 1)$. To compute this sum faster, we divide P' into blocks of fixed size b and store in a separate sequence the sums up to the beginning of each block. This way, we limit to b the number of elements that are summed up. A similar trick, computing $-1 + \sum_{i=0}^{j-1} 2^{P'[i]+1} = P[j] + j - 1$, gives the starting position of $\mathcal{T}_j$ in the bitvector $\mathcal{S}_{\text{HEAP}}$.

Figure 6 shows our representation. The grey triangles represent the complete trees $\mathcal{T}_1$ to $\mathcal{T}_5$. On the bottom of the figure we show the extra structures discussed.

In order to navigate the tree we proceed as follows: we represent a node v as a pair $\langle j, pos \rangle$, so that v is the node at levelwise-order position pos inside the subtree $\mathcal{T}_j$. To move to the left child, we just set $pos' = 2 \cdot pos$, and to move to the right child we set $pos' = 2 \cdot pos + 1$. If $pos' \leq |\mathcal{T}_j|$ we are within the same complete subtree $\mathcal{T}_j$, so we are done. On the other hand, if $pos' > |\mathcal{T}_j|$, we know two things: first, node v is a leaf within its complete subtree $\mathcal{T}_j$, and second, we need to move to another complete subtree. Before moving to another subtree we first need to check if the leaf node has the desired (left or right) child. Thus we map the position of the leaf within its subtree, pos, to the sequence $\mathcal{S}_{\text{HEAP}}$. This can be done with $pos_map = P[j] + j - 1 + 2 \cdot (pos - 1 - \lfloor |\mathcal{T}_j|/2 \rfloor) = P[j] + j - 2 + 2 \cdot pos - |\mathcal{T}_j|$, adding 1 if we descend to the right child. Now, we check in $\mathcal{S}_{\text{HEAP}}[pos_map]$ if the corresponding bit is set. In the case the leaf node in the subtree $\mathcal{T}_j$ has the desired left or right child, we calculate the new node subtree index with $j' = \text{RANK}_1(\mathcal{S}_{\text{HEAP}}, pos_norm) + 1$, and set $pos' = 1$.

We demonstrate this process with an example based on Figure 6: Let us begin at node v_7, which is represented by the pair $\langle 1, 7 \rangle$ and let us say that we want to move to the left. We set $pos' = 2 \cdot pos = 14$. Since $|\mathcal{T}_1| = 7 < 14$, we realize that we are located at a leaf node. Thus we map pos to the sequence $\mathcal{S}_{\text{HEAP}}$ with $pos_map = 1 + 1 - 2 + 2 \cdot 7 - 7 = 7$. Note that the 7-th bit in $\mathcal{S}_{\text{HEAP}}$ tells if v_7 has a left child or not. Since $\mathcal{S}_{\text{HEAP}}[7] = 1$ we proceed to figure out which tree we must go to. This is computed with $\text{RANK}_1(\mathcal{S}_{\text{HEAP}}, 7) + 1 = 5$. Our new node is then represented as the pair $\langle 5, 1 \rangle$.

For this representation, we maintain the sequences of docids and frequencies following the level-order traversal of the nodes within each complete subtree. This traversal is denoted “Node” in the bottom part of Figure 6).

The idea of separating the treap into complete trees is inspired by the level-compressed tries of Andersson and Nilsson [1994]. Under reasonable models for tries they show that the expected number of complete subtrees traversed in a root-to-leaf traversal is $O(\log \log n)$ and even $O(\log^* n)$. While we are not aware of an analogous result for random binary trees, it is reasonable to expect that similar results hold. Note that this is the number of RANK operations needed in a traversal, instead of the $O(\log n)$ that we can expect using BP or LOUDS.

4.6. Practical Improvements

The scheme detailed above would not be so successful without three 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. The overhead

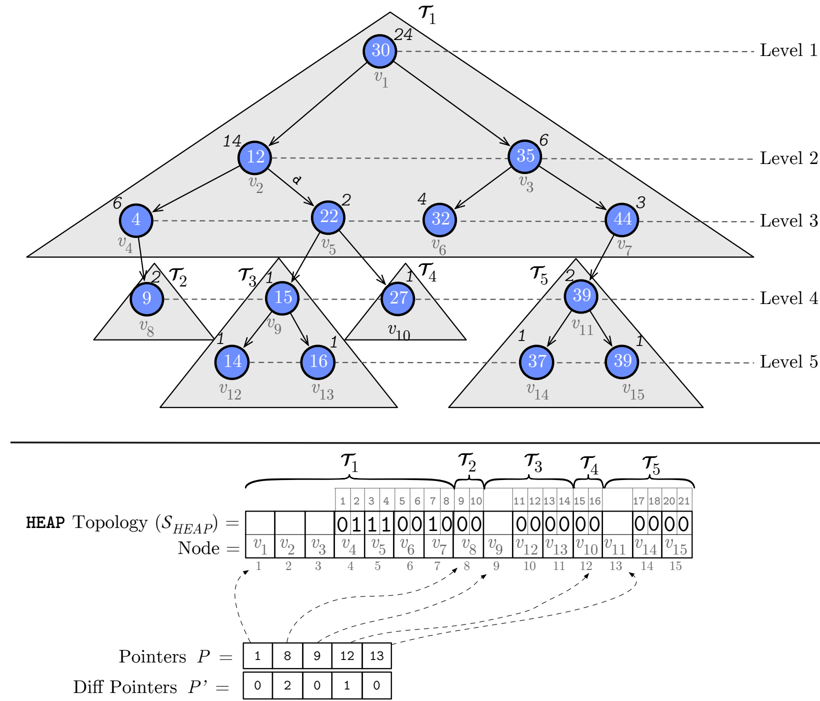

Fig. 6. An example HEAP treap topology representation. At the top, we draw each complete tree with a grey background. The levels of the tree are shown on the right. At the bottom, we show the resulting $\mathcal{S}_{\text{HEAP}}$ sequence (the holes are not represented) and mark the area of each complete treap $\mathcal{T}_j$. The starting positions of these areas correspond to the sequences of docids and frequencies, as they consider the holes and thus account for the internal nodes as well (see array Node). Those starting positions are written below, in array P. Note that the corresponding starting position in $\mathcal{S}_{\text{HEAP}}$ is simply $P[j] + j - 1$. Instead of P, we store the logarithms of the sizes, in P'.

of storing the chunk lengths and other administrative data outweighs the benefits for short sequences.

A second improvement is to break ties in frequencies so as to make the treap as balanced as possible, by choosing the root as the maximum that is closest to the center of each interval (in every subtree). This improves the binary searches for docids and the tree traversal for the HEAP representation. While it is still possible to build the treap in linear time with this restriction, a simple brute-force approach to find the centered maximum performs better in most practical cases.

The third, 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 [Zipf 1949; Croft et al. 2009; Büttcher et al. 2010; Baeza-Yates and Ribeiro-Neto 2011], 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 store a treap at all, as they never occur more than f_0 times in any document. We represent the gap-encoded lists using PforDelta and take an absolute sample every 128 values (which form a block). Samples

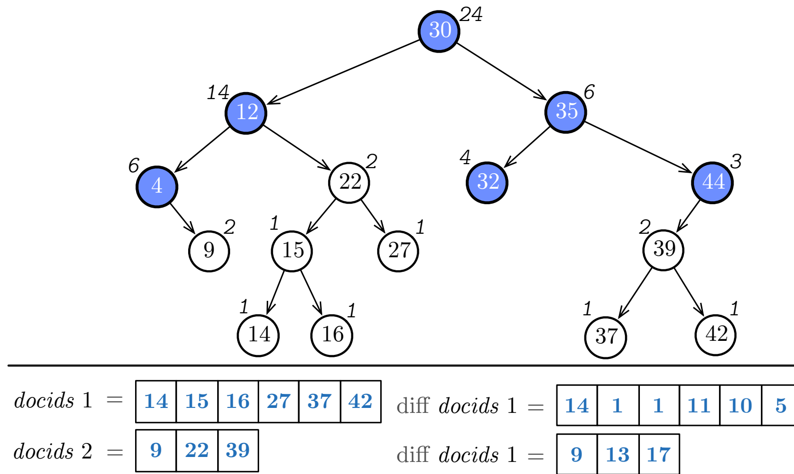

Fig. 7. Separating frequencies below $f_0 = 2$ in our example treap. The nodes that are removed from the treap are on white background. For the documents with frequencies 1 and 2, we show the absolute docids on the left and their differential version on the right.

are stored separately and explicitly in an array, with pointers to the block [Culpepper and Moffat 2007]. Searches in these lists will ask for consecutively larger values, so we remember the last element found and exponentially search for the next query starting from there. Figure 7 illustrates the separation of low-frequency elements from our example treap.

A neat feature of these lists is that often we will not need to access them at all during queries, since ranked queries aim at the highest frequencies.

5. QUERY PROCESSING

In this section we describe the procedure to perform efficient top-k query processing using the inverted treaps.

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 $score(Q,d) = \sum_t w(t,d) = \sum_t tf_{t,d} \cdot idf_t$. 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 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.

5.2. Intersections

Let d be the smallest docid not yet considered (initially $d = 1$). Every treap t involved in the query Q maintains 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, thus 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: While $id(u_t) \leq d$, we assign $v_t \leftarrow u_t$ and remove u_t from the stack. 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 obtained the best results by choosing the treap t of the shortest list. 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).

In order to move towards $d \neq id(v_t)$ in a 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.

5.2.1. 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, we look 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

[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 most weighting formulas, including tf-idf and BM25.

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 ← treap of shortest list such that 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
REPORT(d, s)
  results ← results ∪ (d, s)
  if |results| > k then
    remove minimum from results, L ← minimum priority in results
  end if
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
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
```

before proceeding in the actual treap. Finding this new range $[l, r]$ in the list may imply seeking and decompressing a new block.

The cleanest 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.

5.3. Unions

The algorithm for ranked unions requires a few changes on the algorithm for intersections. First, in the two lines that call $\text{CHANGED}(id(v_t))$, 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 $\text{CHANGED}(\min_{t \in Q} nextd_t)$. Finally, $\text{CHANGED}(newd)$ should not only update d but also update, for all the treaps t, $nextd_t$ to $\max(nextd_t, newd)$.

Algorithm 2 gives the detailed pseudocode.

5.4. Supporting Different Score Schemes

Arguably the simplest scoring scheme is to use the sum of term frequencies $tf_{t,d}$ of the words involved in a bag-of-words union or intersection query. This case is easy to implement using inverted treaps, since the topology is constructed employing the term frequency as the priority and the term frequencies are represented differentially. A trivial extension is tf-idf scoring: every time we need to calculate U, we multiply the term frequency by the corresponding idf_t, as shown in Algorithms 1 and 2. However, in order to support more complex scoring schemes, such as BM25, additional information is required (i.e., document length) and the resulting relative order of documents inside a list may be different from *tf*. In these cases, creating the treap topology based on the the term frequency $tf_{t,d}$ is not useful. Moreover, if we actually use the exact score, we would require float or double precision numbers, thus increasing the size of the index.

An alternative to cope with BM25 is to compute each score at construction time, and build the topology according to the computed score, but still store the term frequency at each node. The query processing algorithm is still valid since we are able to compute the complete score at query time. However, the treap cannot encode term frequencies differentially anymore, since it is possible that a term frequency stored in a node's child is greater than the one of the node itself. If we represent the absolute frequencies, the resulting inverted treap approach is not competitive in terms of space.

In this work we use another approach to this problem. We employ impact-scoring (Section 2.2) instead of term frequencies. This enables the inverted treaps to support any type of scoring scheme. The procedure is to construct the treap topology using the pre-calculated impacts, and store them as if they were the term frequencies in the nodes. This allows for differential encoding of impacts, and we can use the same query algorithms without any change.

ALGORITHM 2: Top-k of union using treaps.

```
UNION(Q, k)
  results ← ∅ // priority queue of pairs (key, priority)
  for t ∈ Q do
    stack_t ← ⟨⊥⟩ // stack of treap t, id(⊥) = f(⊥) = +∞
    nextd_t ← 1 // next possible value in treap t
    v_t ← root of treap t
  end for
  compute U as a score using all f(v_t) values
  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) ∨ nextd_t > d then
      if ∃t ∈ Q, d = id(v_t) then
        REPORT(d, Σ_{t∈Q,d=id(v_t)} w(t, d))
        CHANGED(d + 1)
      else
        CHANGED(min_{t∈Q} nextd_t)
      end if
    else
      t ← choose where to advance, d = nextd_t ≠ 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
          nextd_t ← 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))
          nextd_t ← id(v_t))
        end if
      end if
    end if
  end while
  return results

CHANGED(newd)
  d ← newd
  for t ∈ Q do
    nextd_t ← max(nextd_t, newd)
    v ← v_t
    while d ≥ id(TOP(stack_t)) do
      v ←TOP(stack_t)
      POP(stack_t)
    end while
    CHANGEV(t, v)
  end for
```

6. INCREMENTAL TREAPS

So far, we described a *static* representation of posting list using treaps. In this section we show how to extend the inverted treap representation to support incremental updates, that is, to allow the addition of new documents to the collection while the index is loaded in memory.

Incremental in-memory inverted indexes have been developed to cope with the efficiency challenges in Tweeter [Busch et al. 2012] and for indexing microblogs [Wu et al. 2013]. In these two cases, the more recent posts are generally more relevant, thus appending them at the end of the inverted lists, just as in the indexes designed for Boolean intersections, allows for efficient query processing. In main memory, the problem of maintaining such an inverted index up to date is simpler, because the price for non-contiguous storage of the inverted lists is not so high. In a thorough recent study, Asadi and Lin [2013] show that the difference in query performance between lists cut into many short isolated blocks versus fully contiguous lists is only 10%–20% for Boolean intersections and 3%–6% for ranked intersections.

However, there are cases where we require immediate updating of the index but have no preference for the most recent posts. Obvious examples are online stores like Ebay or Amazon, where new products must be immediately available but they are not necessarily better than previous ones. In those cases, we are interested in ranked retrieval using traditional relevance measures, which are mostly independent of the insertion time.

While this form of dynamization is simple for the WAND and Block-Max formats [Asadi and Lin 2013], it is much more challenging for the treaps, because postings are not physically stored in increasing document identifier order, and therefore one cannot simply append the inserted postings at the end of the inverted lists.

6.1. Supporting Insertions

Our solution is inspired by the linear-time algorithms for building treaps offline (recall Section 4.3). We maintain the rightmost path of the tree in uncompressed form, and their left subtrees are organized into progressively larger compressed structures. This allows for smooth insertion times without large sudden reconstructions, reasonable compression performance and search times.

The main idea is to maintain a treap for each inverted list, as in the static case. However, this treap is only gradually converted into a compressed static structure, and never completely. Some nodes are represented with classical pointers (we call those *free* nodes), whereas some subtrees are represented in the form of static treaps.

The rightmost path is always composed of free nodes. Some nodes descending from the left children of those nodes may also be free, but not many. Each free node v stores the number $\mathcal{F}(v)$ of free nodes in its subtree; these always form a connected subtree rooted at the node. We use a blocking parameter b, so that when a left child v of a rightmost path node has b free nodes or more, all those free nodes are converted into a static treap.

Precisely, when a rightmost node v'_j is converted into the left child of a new incoming node v_i, we check if $\mathcal{F}(v'_j) \geq b$ (since v'_j belonged to the rightmost path, the limit to free nodes did not apply to it, so if v'_j has ℓ rightmost descendants, it could have up to $f(v'_j) = b\ell$ free nodes descending from it). If $\mathcal{F}(v'_j) \geq b$, all those $\mathcal{F}(v'_j)$ free nodes are converted into a static treap and we set $\mathcal{F}(v_i) \leftarrow 1$ for the new node. Otherwise, we set $\mathcal{F}(v_i) \leftarrow \mathcal{F}(v'_j) + 1$.

Hence, the maximum time per insertion is $O(b\ell)$, which is $O(b \log n)$ in expectation. This, however, does not add up to more than $O(n)$, since static treaps are built in linear time and each node becomes part of a static tree only once.

The static treaps we create are not completely identical to the static ones of Section 4. In this case, some free nodes may be parents of static treaps. Therefore, the process results in a tree of static treaps (with some free nodes in the top part). To accommodate this extension, the static structure is expanded with a bitvector B that has one bit per leaf of the static treap, indicating whether the leaf is actually a leaf or it contains a pointer to another static treap. Those pointers are packed in an array inside the structure, so that the ith 1 in B corresponds to the ith pointer in the array. Its position is computed with $\text{RANK}_1(B, i)$.

6.2. Gradual Growth

Note that there will be $O(b \log n)$ free nodes *per inverted list* in expectation. Therefore, b must be reasonably small to avoid their pointers blow up the space (in practice, b should not exceed a few thousands). On the other hand, a small b implies that the static treaps may contain as little as b nodes. Thus b should be large enough for the static structures to use little space (otherwise, the constant number of integers and pointers they use may be significant). It may be impossible to satisfy both requirements simultaneously.

To cope with this problem, we enforce a gradual increase of the treap sizes. Static treaps will be classified in *generations*. A static treap T is of generation $g(T) = 1$ when it is first created. No treap can have descendants of lower generations. Each static treap T stores its generation number $g(T)$ and the number $d(T)$ of descendant treaps of its same generation, including itself. It is easy to compute $d(T) = 1 + \sum_{T', g(T')=g(T)} d(T')$ for a newly created static treap T that points to several other existing treaps T'. Given a parameter c, we establish that, when a new static treap T is created and $d(T) \geq c$, that is, its subtree has c or more treaps of its generation, then all those are collected and recompressed into a larger static treap S, which now belongs to generation $g(S) = g(T) + 1$. The same formula above is used to compute $d(S)$ for the new treap.

This technique creates larger and larger static treaps towards the bottom and left part of the tree. Now a node can be reprocessed $\log_c(n/b)$ times along its life, to make it part of larger and larger static treaps; therefore the total construction time becomes $O(n \log n)$ (albeit with a very low constant). Parameter c should be a small constant.

Figure 8 shows a normal left-to-right construction process (as described in Section 4.3), but it also illustrates how the incremental version is built. We have enclosed in gray sets the nodes that are grouped into static treaps, for $b = c = 2$. In particular, observe the situation in the rightmost cell of the second row. A static treap of generation 1 is created for the nodes with docid 13, 22, and 27. But now this node is of generation 1 and its subtree has $d = 3 \geq c$ treaps of its same generation. Thus, a new static treap, of generation 2, is created with all those generation-1 descendants. This is shown in the leftmost cell of the third row.

7. EXPERIMENTAL SETUP AND RESULTS

In this section we describe the experimental setup, in terms of the collections used and the environment employed for the experiments. We also explain the engineering details required to implement the indexes and the baselines, and discuss the space/time results obtained.

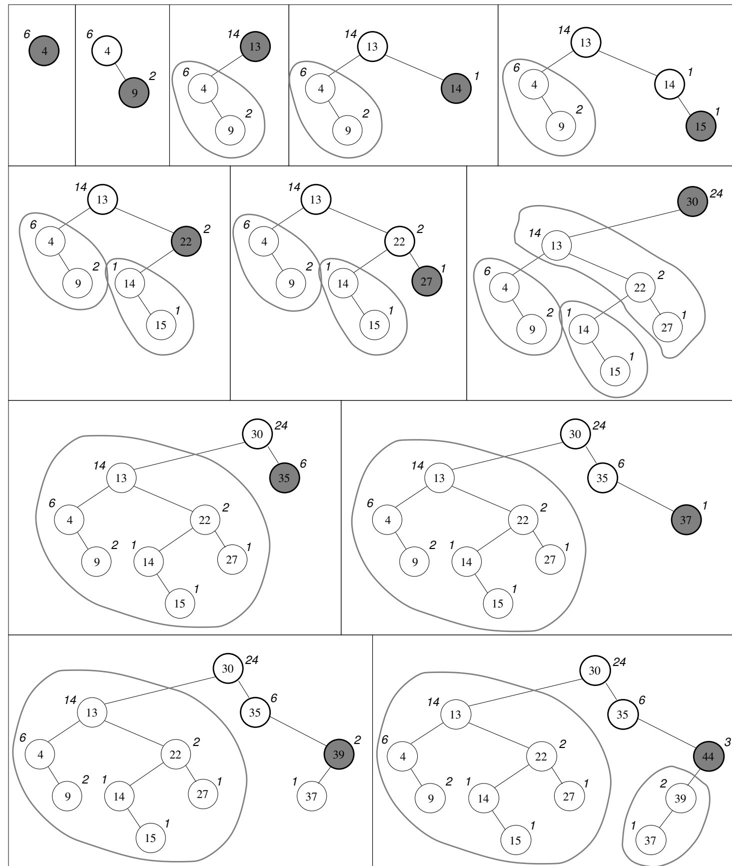

Fig. 8. The left-to-right construction of an example treap. We show in bold the rightmost path and shade the node that is added in each iteration. Sets of nodes with gray borders indicate static treaps that are built.

7.1. Collections

We use the TREC GOV2 collection, parsed using the Indri search engine[2], and Porter's stemming algorithm. The collection contains about 25.2 million documents and about 39.8 million terms in the vocabulary. The inverted lists contain about 4.9 billion postings in total. After pre-processing and filtering, a plain representation of the GOV2 collection requires about 72GB. The average posting list length is 76 and the average document contains 932 words.

We also performed experiments using other collections, such as the English Wikipedia dump, containing about 5 million documents and 6 million terms. An average Wikipedia article contains 352 words and the average number of elements in a posting list is 132. We also performed experiments on the Weblogs collection[3], containing about 50 million documents and requiring 120GB of space. For queries, we used the 50,000 TREC2005 and TREC2006 Efficiency Queries dataset with distinct numbers of terms, from $q = 2$ to 5. We omitted those not appearing in our collection, thus we actually have 48,583 queries.

[2]http://www.lemurproject.org/indri/
[3]http://www.icwsm.org/data/

In this article we only show the results obtained using the TREC GOV2 collection, since the results over the others do not change significantly.

7.2. Baselines and Setup

We compare our results with five baselines: (1) Elias Fano WAND implementation [Ottaviano and Venturini 2014], (2) Block-compressed WAND Implementation [Broder et al. 2003], (3) Block-Max [Ding and Suel 2011], (4) Dual-Sorted [Konow and Navarro 2013] and (5) ATIRE [Trotman et al. 2012].

For Elias Fano WAND, we use the implementation[4] provided by Ottaviano and Venturini [2014]; we denote this implementation EF in the charts. For block-compressed WAND, we use the implementation obtained from the `SURF` framework[5]; we denote this implementation WAND in the charts. For Block-Max, we adapted the implementation of Petri et al. [2013] by extending it to support ranked intersections, and included it into the `SURF` framework; we call it BMAX in the charts. For both WAND and BMAX we use the optimal PForDelta encoding for the docids and Simple9 encoding for the frequencies, as these gave the smallest indexes. In both cases the posting lists were encoded using blocks of 128 values. In the case of BMAX, we also store the maximum value for every block. The implementations of those encodings were obtained from the FastPFor library[6].

We use ATIRE[7] as our baseline for impact-sorted indexes. ATIRE is an open-source search engine that supports different early termination algorithms based on impact or frequency sorted posting lists. We constructed both, frequency-sorted and quantized BM25 ($q = 8$) impact-sorted indexes for our experiments.

In the case of Dual-Sorted, we use the original implementation of Konow and Navarro [2012], using compressed bit sequences representation.

All baselines were modified, when needed, to support both quantized impact BM25 scores (Section 2.2) and tf-idf scoring. Our experiments were run on a dedicated server with 16 processors Intel Xeon E5-2609 at 2.4GHz, with 256 GB of RAM and 10 MB of cache. The operating system is Linux with kernel 3.11.0-15 64 bits. We used GNU `g++` compiler version 4.8.1 with full optimizations (`-O3`) flags.

7.3. Inverted Treaps Implementation

We implemented our indexes based on the `sdsl-lite` library [Gog et al. 2014]. The document ids are stored in a `dac_vector<6>` of fixed width 6, which gave the best results at parameter tuning time. The weights are stored in a `dac_vector<2>` of fixed width 2. The f_0 list are represented using PForDelta using a similar implementation to the ones used in WAND and BMAX. We do not use inverted treaps to represent every posting list, but only those containing at least 1024 elements. The other posting lists are represented using WAND. At query time, if necessary, they are fully decompressed and handled by maintaining a pointer to the current docid being evaluated. The BP topology is implemented using the `bp_tree<>` class, while the LOUDS topology is implemented using the `bit_vector<>` class enhanced with rank operations, for which we use the alternative dubbed `rank_support_v5<>`, which requires 5% extra space. The HEAP implementation uses an integer vector `int_vector` requiring $\lceil \log(X+1) \rceil$ bits, where X is the maximum element in the array P'. The implementation of the bit sequence for the topology is the same one as the one employed in LOUDS. We perform all

[4]http://github.com/ot/partitioned_elias_fano
[5]http://github.com/simongog/surf
[6]https://github.com/lemire/FastPFor
[7]http://atire.org/index.php?title=Main_Page

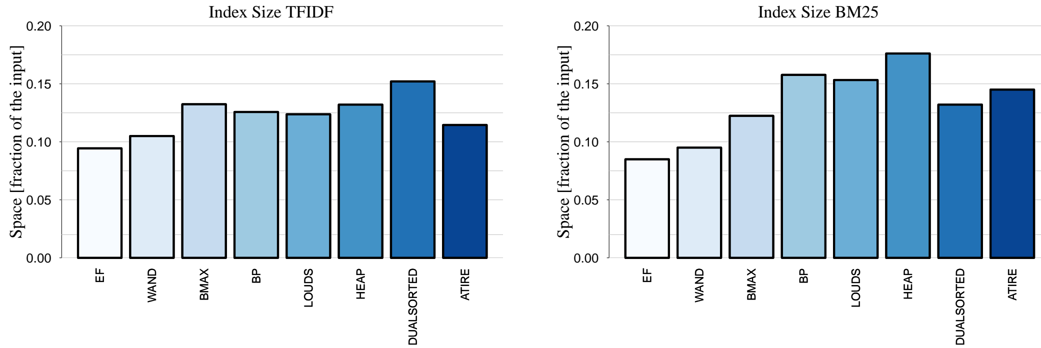

Fig. 9. Total sizes of the indexes depending on the scoring scheme.

the experiments for tf-idf score and for quantized BM25 impact scoring, as described in Section 5.4, using 8 bits for each impact.

7.4. Index Size

We start by showing the size required of each index in Figure 9, separated by the scoring scheme used. The left part of the figure shows the case of tf-idf scoring, whereas the right part shows BM25 scoring. In both cases, EF is clearly the smallest alternative. The second alternative, WAND, is about 10% larger.

In the tf-idf indexes, BMAX requires more space than any of the treap alternatives, LOUDS being about 10% smaller than BMAX and HEAP almost equal. Dual-Sorted, on the other hand, is the most space-consuming alternative. With BM25 scoring, all the inverted treap alternatives require more space than the baselines, climbing from about 13% of the text space under tf-idf scoring to up to 18%. This is mainly because, when using the quantized BM25 score, the number of posting lists elements having score 1, which is efficiently represented using the f_0 lists, is considerably reduced: for the tf-idf score scheme there are about 3 billion posting list elements with frequency 1, but this decreases to 800 million under BM25. The space of ATIRE also increases when moving from tf-idf, where it is the third smallest index, to BM25, where it is only smaller than our treap alternatives.

Figure 10 shows the space breakdown of the inverted treap components, using tf-idf scoring on the left and BM25 on the right. This figure is based on our smallest case, which is the LOUDS alternative. The component *Small Lists* represents all the posting lists that have less than 1024 elements and are represented using WAND. For the tf-idf case, we see that the topology requires 7% of the total index size, and the biggest component is the f_0 lists. However, in the BM25 case, the f_0 lists use a negligible amount of space, and the document ids is the heaviest component.

The only component that changes between our three alternatives is how we represent the treap topology. Figure 11 shows the difference in space requirements, depending on the scoring scheme. We see that LOUDS is always the smallest alternative, and HEAP is the biggest, requiring about twice the size of LOUDS.

7.5. Construction Time

Figure 12 shows the time required to build each index. We see that the construction times of the inverted treap alternatives are not so distant from those of the baseline inverted index representations. This holds except for the HEAP alternative, which is up to twice as slow to build than the fastest baseline. It is interesting to note that the Elias-Fano WAND index builds 1.4 to 1.8 times slower than the block-compressed

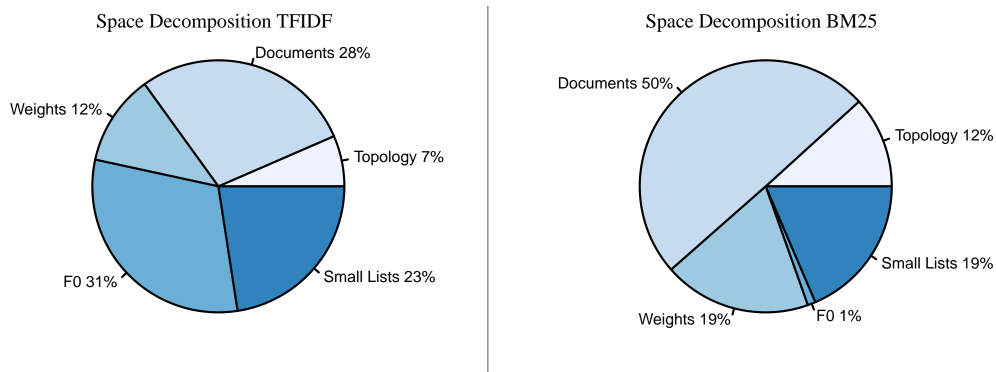

Fig. 10. Total sizes of the indexes depending on the scoring scheme.

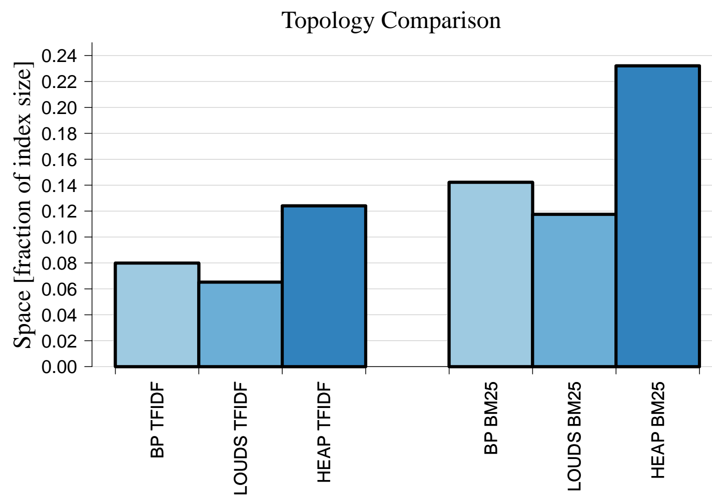

Fig. 11. Sizes of the topology components depending on the scoring scheme.

WAND implementation. ATIRE was the slowest alternative to build in both cases. We do not include the time required to build the Dual-Sorted index, since the construction is not optimized and was above 200 minutes.

7.6. Ranked Union Query Processing

We describe the time results for the processing of ranked union queries. We first discuss the results globally and then consider how they evolve as a function of k or the number of words in the query.

Global analysis. Figure 13 (left) shows the average time per query, for distinct values of k, using the tf-idf scoring scheme. These times average all the queries of all the lengths (2 to 5 terms) together.

The results show that EF, WAND and Dual-Sorted are not competitive for these queries, as they are sharply outperformed by BMAX. In turn, all our inverted treap alternatives outperform BMAX by a wide margin. The differences become less drastic as we increase k, but still for $k = 1000$ the HEAP alternative is more than 3 times faster than BMAX. Our LOUDS alternative is always slower than HEAPS, and BP is

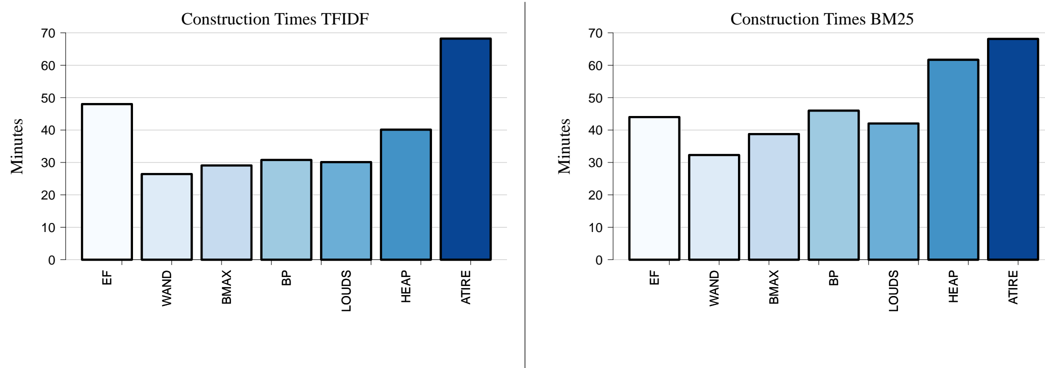

Fig. 12. Construction time of the indexes depending on the scoring scheme, in minutes.

slower than LOUDS. Still, BP is almost twice as fast as BMAX even for $k = 1000$. We discarded results from ATIRE in most of the following figures because it required more than 500 milliseconds on average.

Figure 13 (right) shows the distribution of the results using the BM25 quantized score scheme. The differences are much smaller in this case, and in particular our LOUDS and BP variant are the slowest. Our HEAP alternative, instead, is still the fastest or on par with the fastest.

The worse performance of our variants under BM25 owes to the fact that most of the lists are stored as treaps, whereas under tf-idf many of them are stored as f_0 lists. The union algorithm performs a significant amount of sequential traversal, and the simple f_0 lists are faster at this than the treaps. Instead, the considerable improvement obtained by EF and WAND owes to the narrower universe of impacts, which increases the chances that the lower bound θ is not reached along the process and enables more frequent skipping (recall Section 3.3.1). Up to a lesser, extent, BMAX also improves thanks to more frequent skipping, as on a narrower universe it is also less likely to outperform the current kth highest score. Finally, the significant improvement of Dual-Sorted owes to the fact that it implements the method of [Persin et al. 1996] (which does not give exact results, so the comparison is not totally fair) and this method is also favored by the BM25 quantized scores: it reaches sooner a stable situation where the upcoming scores are no better than those already obtained. The optimal-partitioned Elias-Fano implementation was consistently slightly slower than the block-compressed WAND implementation, so we will only consider the latter alternative for the rest of the comparisons.

We show more detailed results grouped by percentiles in Table I. In the case of tf-idf, the best alternative by far is the LOUDS treap for all percentiles. For the quantized BM25 case, HEAP is the best alternative. In general terms, the table shows that our improved time results are consistently better, and are not blurred by a high variance.

Analysis as a function of k and separated by query length. Figure 14 separates the times according to the number of words in the query, and shows how times evolve with k, using tf-idf scoring. Note that the times are independent of k for some techniques, or grow very slowly with k in the others.

For 2 query terms, all the inverted treap alternatives are an order of magnitude faster than WAND and Dual-Sorted. For small k values, BMAX is about twice as slow as the fastest treap alternative, HEAP. For large k values, instead, HEAP is about 5 times faster than BMAX. The LOUDS and BP alternatives are also faster or on par with BMAX. As the number of query terms increases, however, BMAX starts to

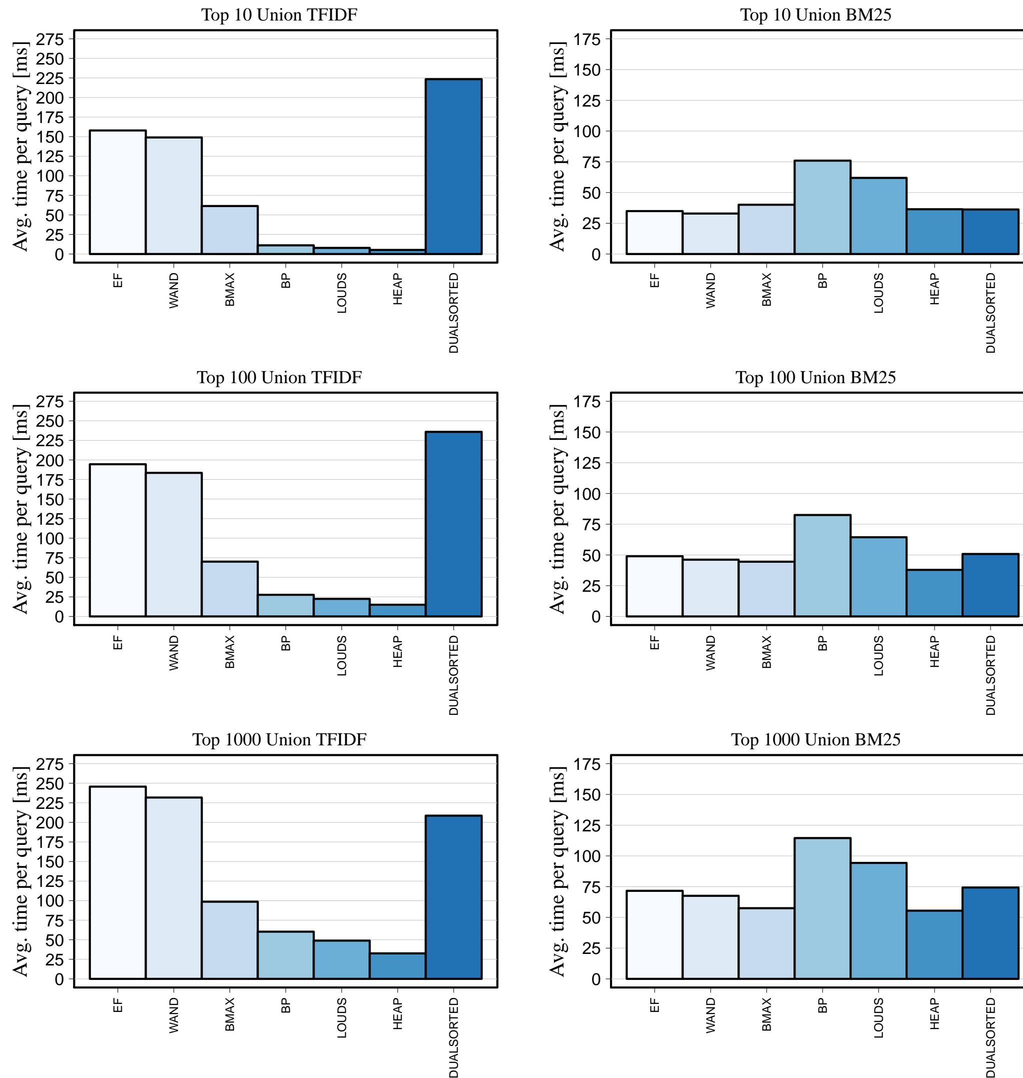

Fig. 13. Ranked union times for distinct k values, in milliseconds. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

Table I. Ranked union results grouped by percentiles for $k = 10$. The numbers indicate the maximum time reached by $X\%$ of the fastest queries.

	Union TFIDF					Union BM25				
Index/Percentile	**50%**	**80%**	**90%**	**95%**	**99%**	**50%**	**80%**	**90%**	**95%**	**99%**
EF	170	342	724	781	976	35	102	144	181	230
WAND	177	462	701	787	918	30	77	116	141	168
BMAX	28	139	195	246	332	20	74	122	169	222
BP	10	27	47	73	94	60	80	156	229	294
LOUDS	7	21	39	61	80	40	62	103	123	231
HEAP	40	177	289	457	617	20	49	76	96	156
DualSorted	151	416	736	866	1101	25	70	121	155	202

outperform the slower inverted treap alternatives. Still, HEAP is always faster than BMAX and 3–4 times faster than WAND and Dual-Sorted.

Figure 15 shows the same experiment on the quantized BM25 scoring scheme. In this case, the treap alternatives are competitive only on queries of 2 and 3 words. In particular, our fastest approach, HEAP, is twice as fast as BMAX for 2 words, but up to 50% slower on the longest queries. Still, we note that most real queries are short. For example, the average query length has been measured in 2.4 words [Spink et al. 2001], and in our dataset of real queries, 44% of the multi-word queries have indeed 2 words.

Analysis as a function of query length and separated by k value. Figure 16 shows the ranked union times as a function of the query length, for distinct k values. As mentioned before, in the case of tf-idf (left side of the figure), our fastest approach HEAP is consistently faster than all the other alternatives, for all query lengths and up to $k = 1000$. In the case of BM25 (right side of the figure), our HEAP alternative is competitive when 2 or 3 query terms are involved. In general, the costs grow linearly with the number of query terms, but the growth rate of WAND and Dual-Sorted is higher on tf-idf and lower on BM25.

7.7. Ranked Intersection Query Processing

We proceed to describe the time results for processing ranked intersection queries. As before, we first discuss the results globally and then consider how they evolve as a function of k or the number of words in the query. We do not include the results from ATIRE since it does not support a native mechanism to perform top-k ranked intersection.

Global analysis. Figure 17 (left) shows the average time per query, for distinct values of k, using the tf-idf scoring scheme. These times average all the queries of all the lengths (2 to 5 terms) together.

As expected, BMAX always outperforms WAND by a significant margin. Among our alternatives, as before, HEAP is always faster than LOUDS and this is faster than BP. Our results are better for small k, where HEAP outperforms all the other indexes by a factor of 2 or more. The difference narrows down for larger k, but still for $k = 1000$ we have that HEAP is faster than BMAX. Dual-Sorted performs a Boolean intersection and then computes the score of all the qualifying documents. The experiment shows that, for $k = 1000$, this becomes (slightly) better than the more sophisticated alternatives that try to filter the documents on the fly. As for unions, the optimal-partitioned Elias-Fano implementation was consistently slightly slower than the block-compressed WAND implementation, and so we did not include it in further experiments on time performance.

Figure 17 (right) shows the distribution of the results using the BM25 quantized score scheme. Unlike in the case of unions, the powerful filtration enabled by the treaps outweighs its slowness compared to traversing an f_0 list. As a result, the inverted treaps are faster on BM25 than on tf-idf scores. The methods WAND and BMAX also improve thanks to the quantized scores. Dual-Sorted also improves: even if it always performs the same Boolean intersection and then computes the scores of the surviving candidates, this computation is faster because it uses the stored quantized scores, whereas for tf-idf it must multiply each stored tf by the idf associated with the query term. The comparisons between all the alternatives stay, overall, similar as in the case of tf-idf scoring.

We show detailed results grouped by percentiles in Table II for all the alternatives considered. In both cases, tf-idf and BM25, the best alternative is the treap using the

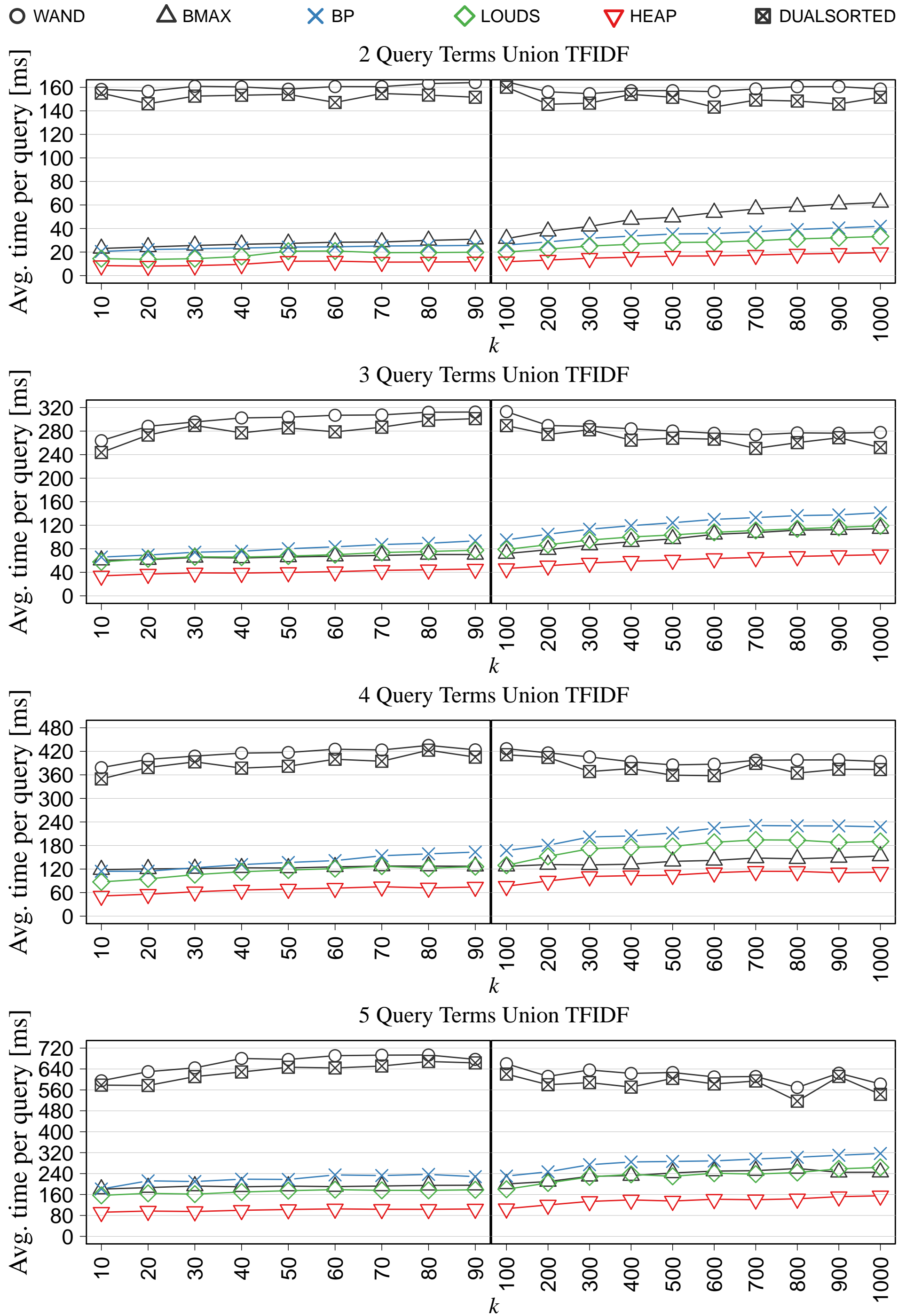

Fig. 14. Ranked union times as a function of k, grouped by number of terms per query, using tf-idf.

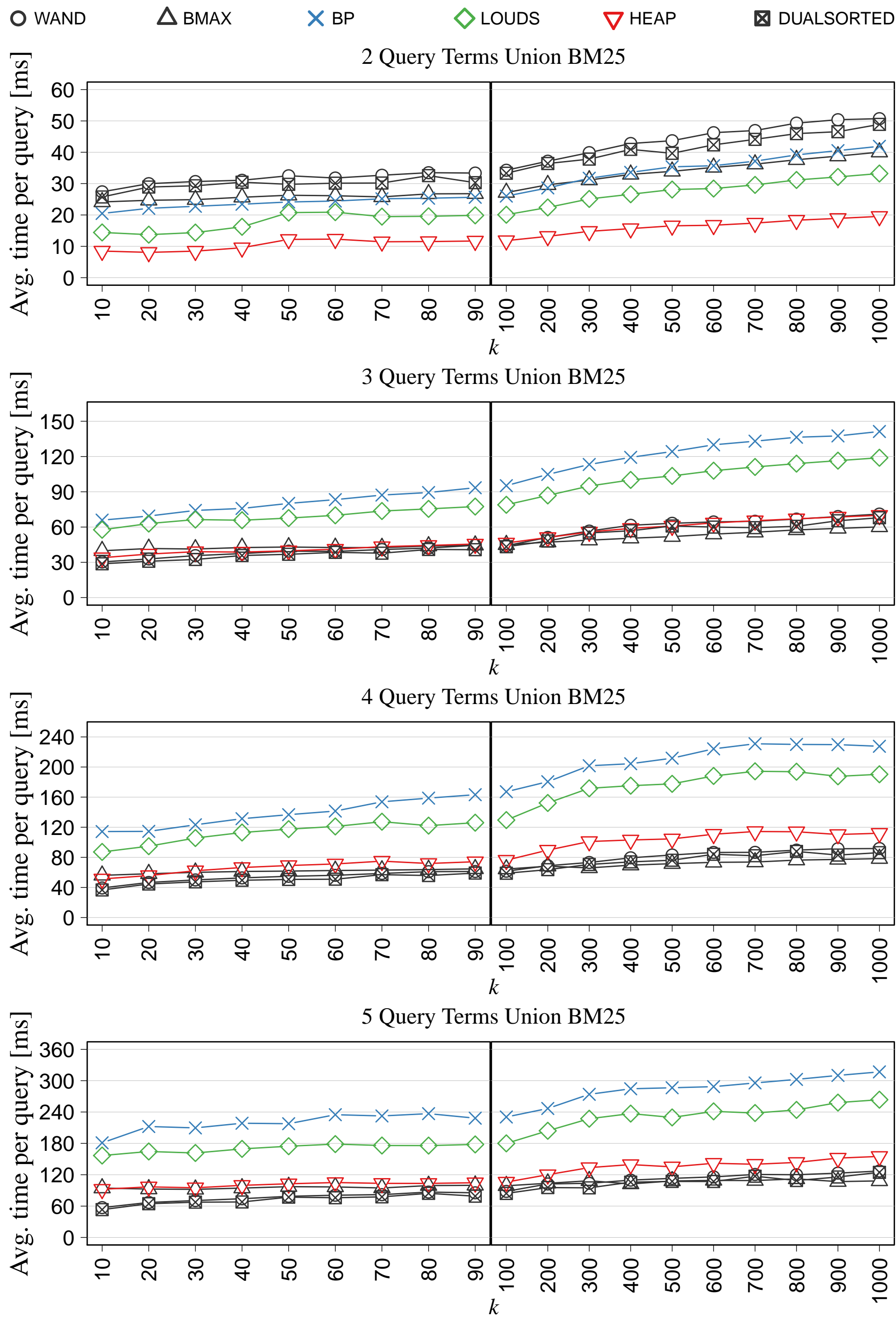

Fig. 15. Ranked union times as a function of k, grouped by number of terms per query, using BM25.

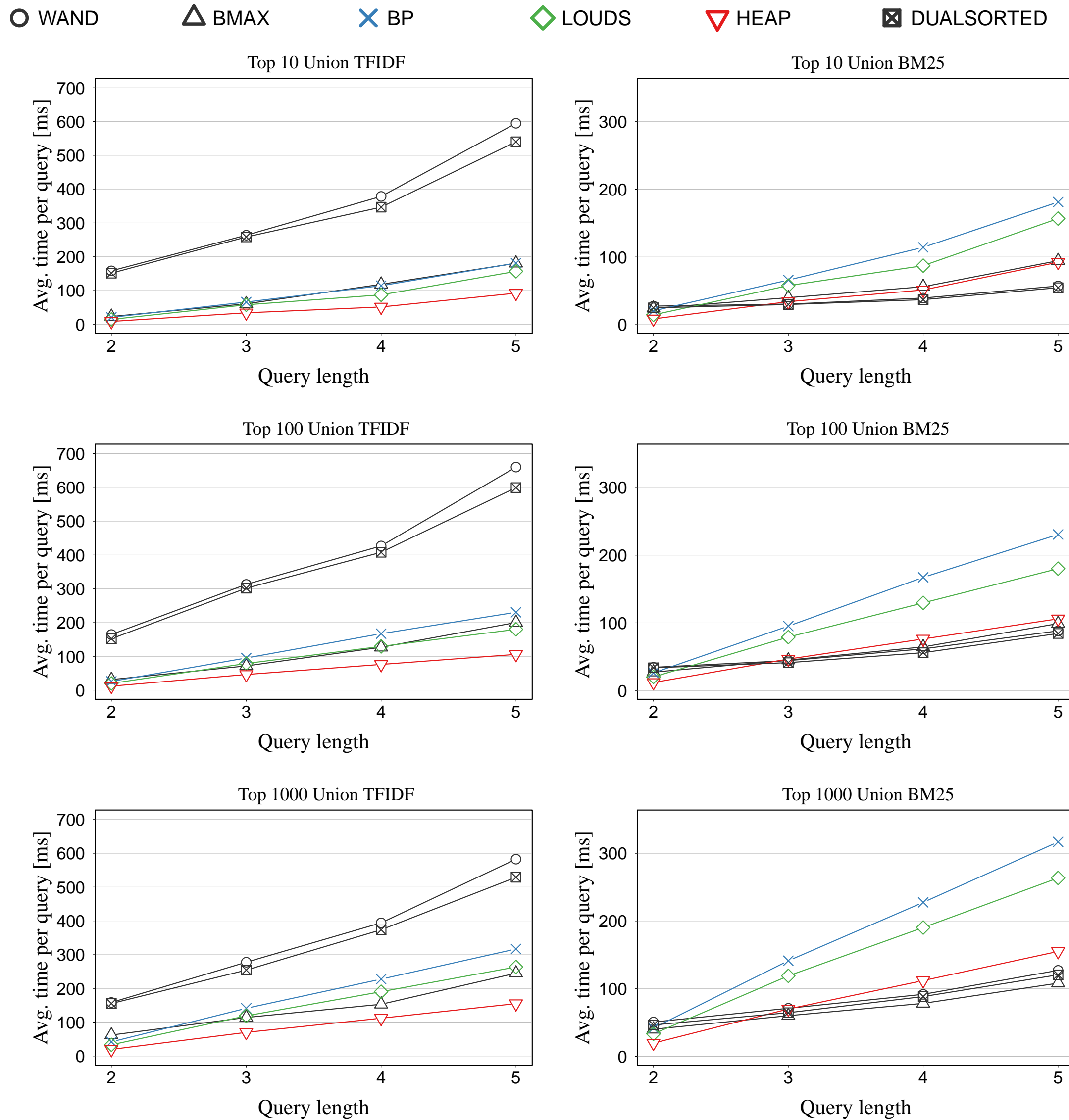

Fig. 16. Ranked union times for distinct k values as a function of the query query length. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

HEAP topology. Again, the table shows that our improved time results are consistently better, and are not blurred by a high variance.

Analysis as a function of k and separated by query length. Figure 18 shows how times evolve with k, using tf-idf scoring. As in the ranked unions, some techniques are independent of k and others (in this case, the inverted treaps with 2 or 3 words) grow slowly with k.

For 2 query terms, the HEAP alternative is the fastest up to $k = 300$, from where BMAX takes over. For 3 and 4 words, HEAP is either the fastest choice or very close to

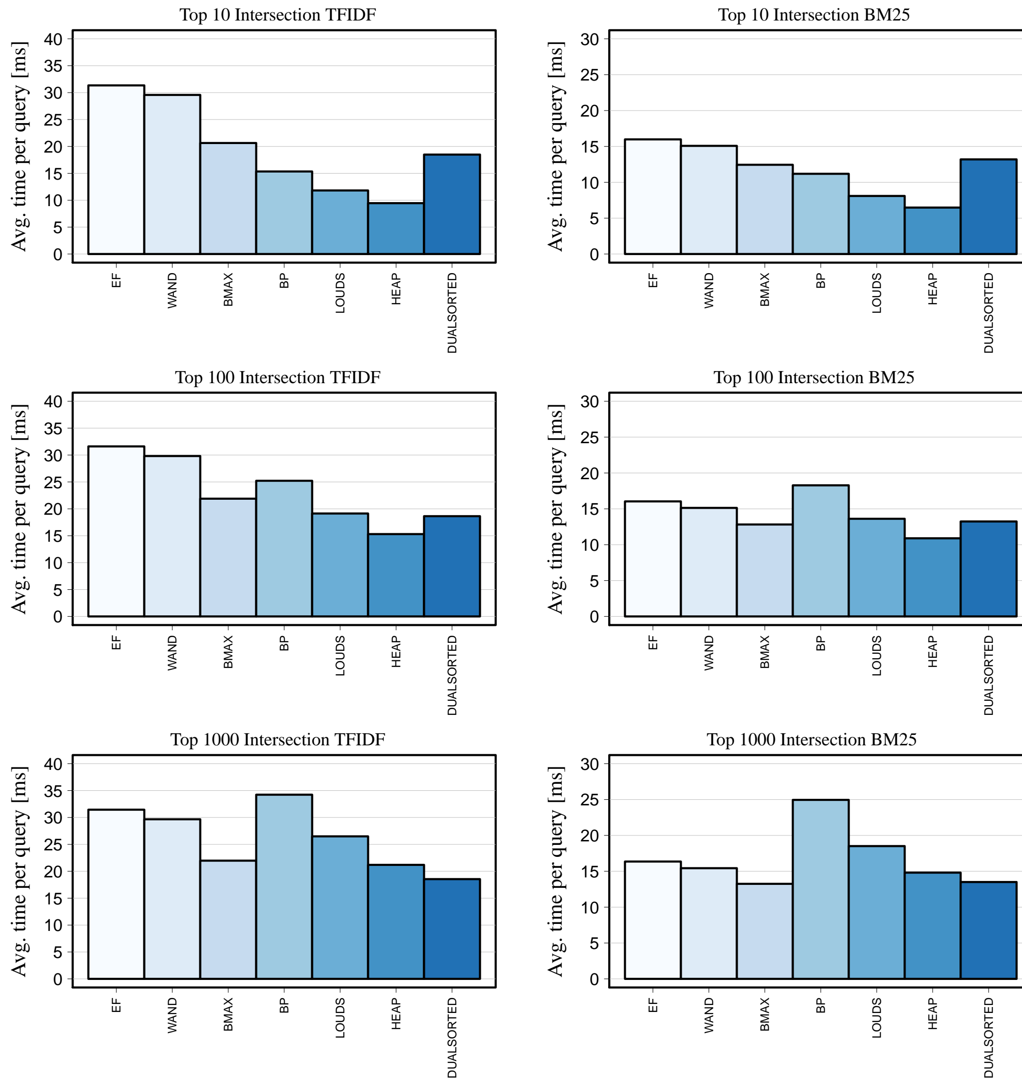

Fig. 17. Ranked intersection times for distinct k values, in milliseconds. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

Table II. Ranked intersection results grouped by percentiles for $k = 10$. The numbers indicate the maximum time reached by $X\%$ of the fastest queries.

	Intersection TFIDF					Intersection BM25				
Index/Percentile	**50%**	**80%**	**90%**	**95%**	**99%**	**50%**	**80%**	**90%**	**95%**	**99%**
EF	4	35	85	195	526	9	49	91	146	304
WAND	4	43	93	178	451	9	35	89	128	223
BMAX	29	146	209	260	321	3	29	62	93	156
BP	17	42	58	68	77	18	43	67	85	101
LOUDS	13	22	34	47	60	12	29	44	56	67
HEAP	3	12	18	23	34	9	21	32	42	51
DualSorted	24	132	230	287	386	9	33	84	122	212

it, and for 5-word queries it takes over again. As mentioned by Ding and Suel [2011], the performance of BMAX is considerably affected by the number of terms participating in the query. For 4 query terms, it is one of the slowest alternatives, together with the BP inverted treap variant. For 5 query terms, it is definitely the slowest. This explains the poor performance of BMAX compared to WAND when considering all the queries together.

Figure 19 shows the results for ranked intersections on the quantized BM25 scoring scheme. While the general picture is similar to the case of tf-idf, HEAP is overcomed more frequently. For few-word queries it is outperformed sooner by BMAX, for $k = 200$ on 2 words and for $k = 80$ on 3 words. On 4-word queries, it is almost always outperformed by a small margin. It is again the fastest alternative on 5 words, but by a smaller margin than for tf-idf. BMAX is also heavily affected as the number of words increases.

Analysis as a function of query length and separated by k value. Figure 20 shows the ranked intersection times as a function of the query length, for distinct k values. On the left side of the figure we show the results of the tf-idf scoring scheme. We can see more clearly how BMAX is heavily affected by the query length. The others stay unaltered or fluctuate as a function of the number of words. This is because, as this number increases, more lists have to be handled, but it is also more likely to filter out portions of the lists. The interaction of the two effects produces increments and decrements in the query times. Recall that WAND and Dual-Sorted perform a Boolean intersection followed by the evaluation of all the resulting scores, so their behavior is very similar. Note that our HEAP alternative is generally the best on tf-idf, whereas on BM25 it is the best for $k = 10$ and in some cases for larger k.

7.8. One-word Queries

We have not yet considered the simplest one-word queries, which account for a significant percentage of typical queries (almost 24% in our query set). For these queries, we must obtain the k highest-ranked documents from a single inverted list. In the case of WAND, this requires traversing the whole list and retaining the k highest scores. BMAX speeds this up by skipping blocks where the maximum score is not higher than the kth score we already know. Dual-Sorted, instead, simply requires to extract the first k elements from the list of the query term, as its lists are sorted by decreasing frequency. Therefore the Dual-Sorted time is bounded by $O(k \log D)$, as it is implemented on a wavelet tree. This is the best scenario for ATIRE, since the posting lists are sorted by either frequencies or quantized scores, so returning the k best documents is done simply by traversing the first k postings.

For our inverted treaps, we use a simplification of the procedures for ranked unions and intersections. We insert the root of the treap in a heap that sorts by decreasing score. We then iteratively extract the top of the heap, report its document, and insert its two children. Therefore we require $O(k)$ operations on the treap and the heap, leading to total time $O(k \log k)$.

Figure 21 shows the time performance. The time differences are so significant that we have used logarithmic scale. Our fastest variant, HEAP, requires from 5–10 microseconds per query with $k = 10$ to 100–200 with $k = 1000$, whereas our slower variant, BP, requires 10–20 to 200–500 microseconds. Dual-Sorted, instead, goes from 100–200 to 2000–5000 microseconds, that is, around 20 times slower than HEAP. The slowest technique in the plots is BMAX, which requires 1–5 milliseconds per query, that is, 25–200 times slower than HEAPS. This is because its time is not bounded in terms of k. Still, the time of BMAX increases with k because its ability to filter blocks decreases as k grows. On the other hand, the times of the other indexes grows more or

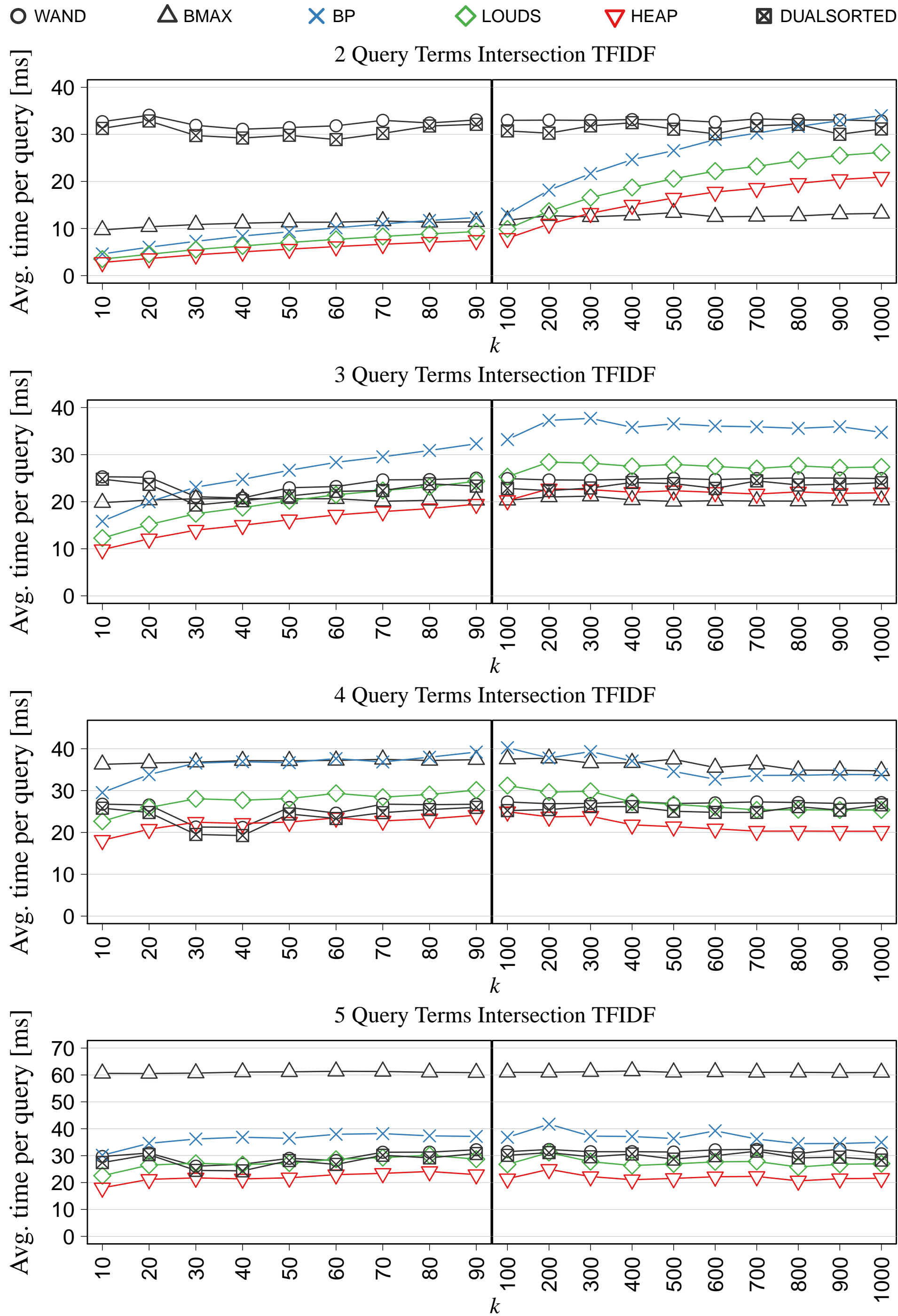

Fig. 18. Ranked intersection times as a function of k, grouped by number of terms per query, using tf-idf.

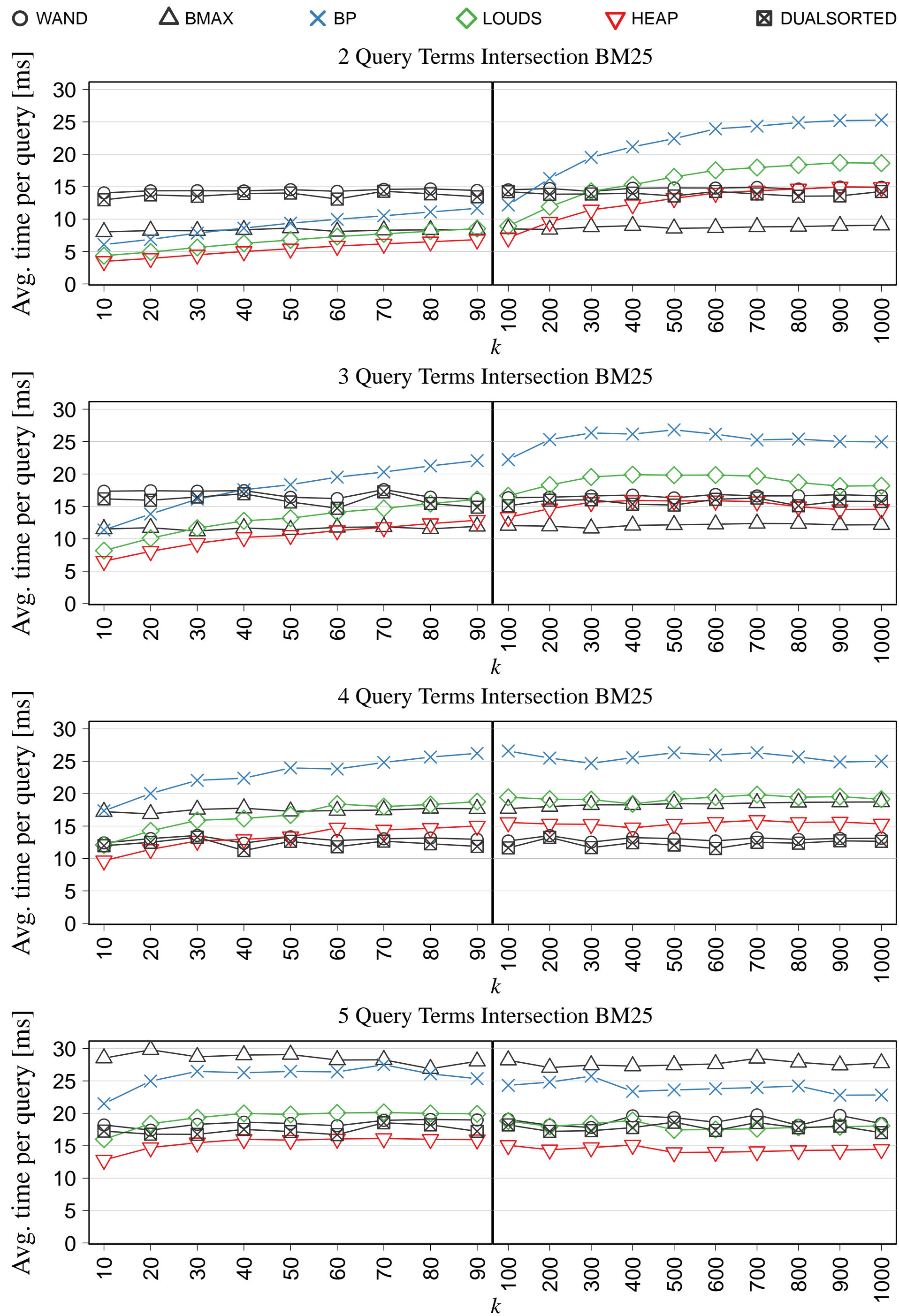

Fig. 19. Ranked intersection times as a function of k, grouped by number of terms per query, using BM25.

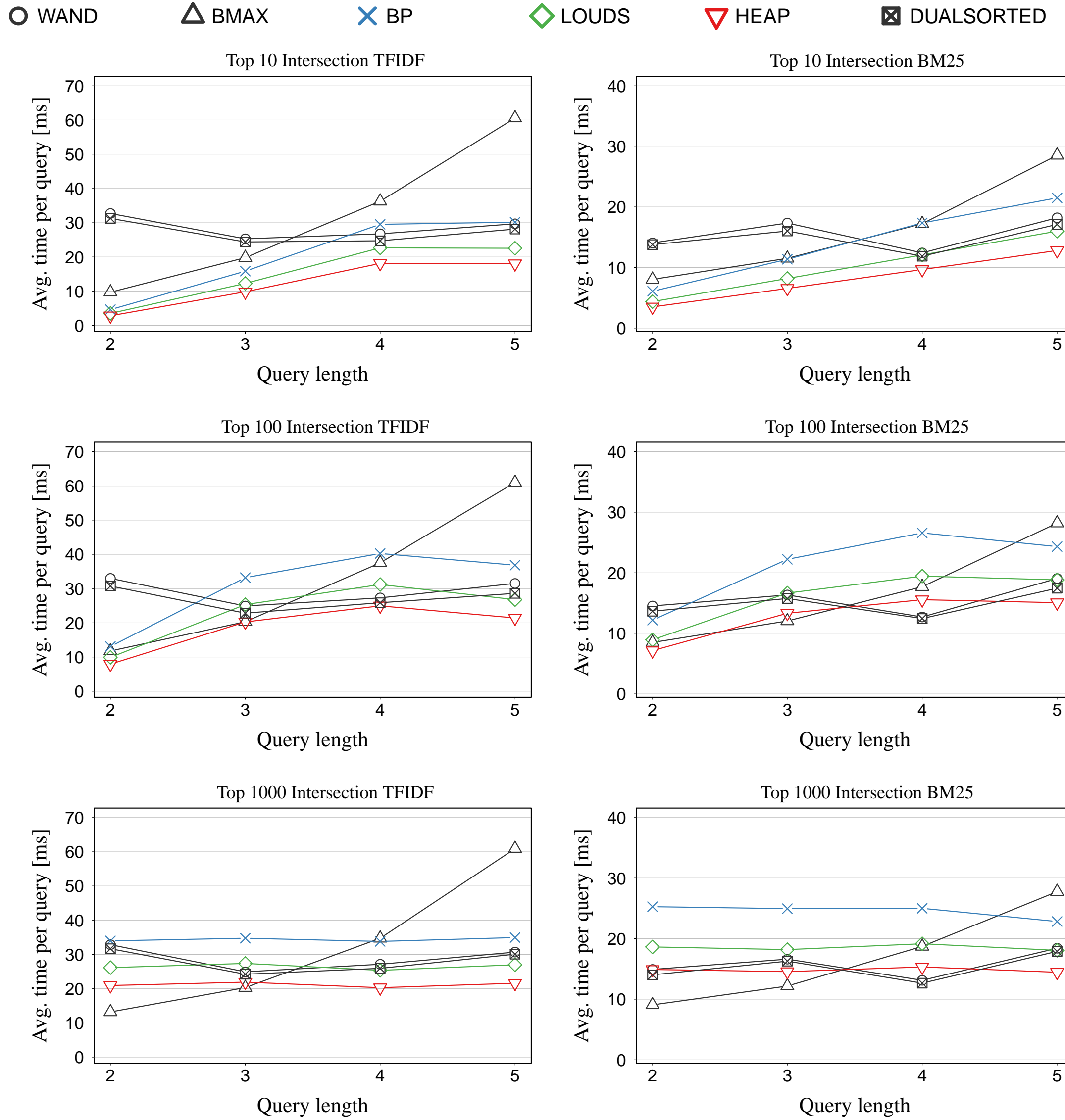

Fig. 20. Ranked intersection times for distinct k values as a function of the query query length. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

less linearly with k after a query initialization time: A rough fitting gives, for HEAP, $0.15k + 6.5$ microseconds on tf-idf and $0.10k + 5$ on BM25; for BP it gives $0.5k + 10$ on tf-idf and $0.3k + 5$ on BM25; and for DualSorted it climbs to $4k + 100$ on tf-idf and $2k + 80$ on BM25. ATIRE is clearly the fastest alternative for single-term queries, taking $0.05k + 1$ microseconds in both cases. We have not included the times of WAND as they are much higher, 15,000 microseconds almost independently of k (note that WAND needs to decompress the whole list, independently of k, and then find the k largest scores).

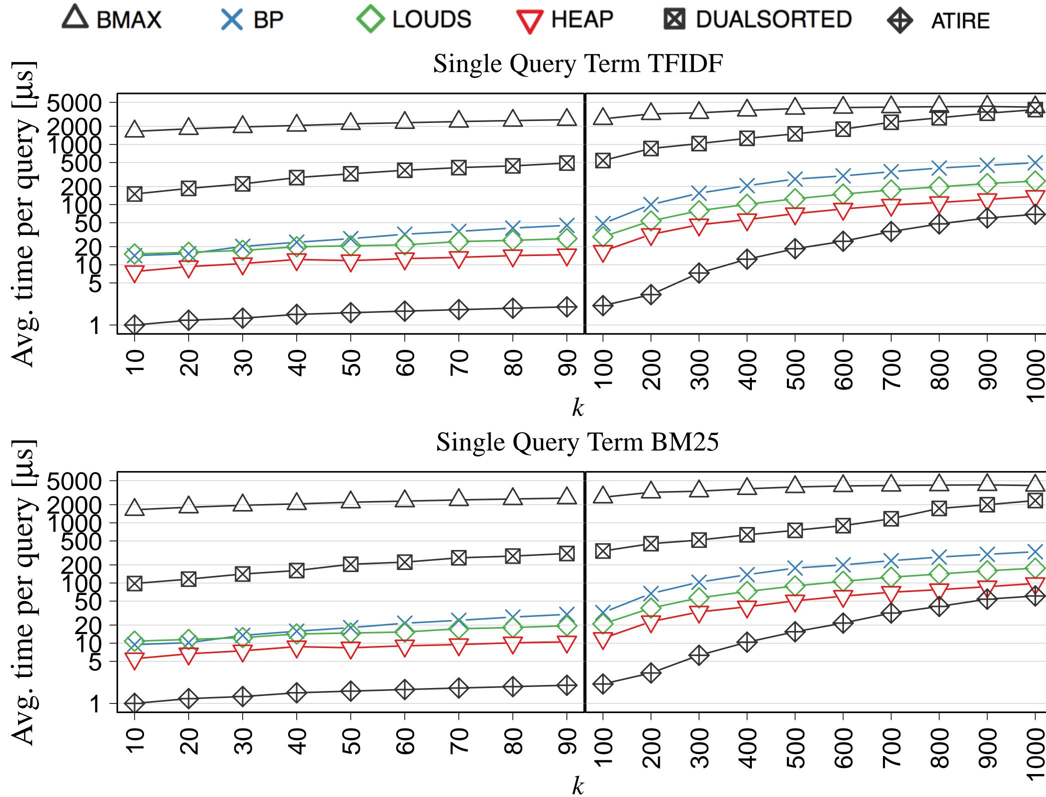

Fig. 21. One-word query times as a function of k, using tf-idf (top) and BM25 (bottom) scoring schemes. Note the logscale.

7.9. Incremental Treaps

Figure 22 shows the time required to insert increasing prefixes of the GOV2 collection on our incremental inverted treaps. For these experiments, we use the block parameter $b = 1024$ and recompress any subtree that has generation $c = 16$ into a new larger static treap (recall Section 6.1). These values were chosen by parameter tuning experiments. As a baseline, we consider the static inverted treap construction, for LOUDS and HEAP, on the same prefixes. The figure shows that the time for incremental treap construction grows slightly superlinearly, as expected from its $O(n \log n)$ time complexity. The static construction, instead, displays linear-time performance. Still, after inserting 24 million documents, the incremental construction is only twice as slow as the fastest static construction (LOUDS) and 40%–60% slower than the static construction giving the best query times (HEAPS).

In terms of memory usage the incremental treap requires an additional bitvector, causing an increase in the overall size of about 7%. However, the size occupied by the *free* nodes is considerably larger, using about 40% more space. This is because the free nodes are not compressed in any way, that is, we are using 64-bit pointers, and 32-bit integers for the the docid and the weight. In addition, we need to keep track of counters for the block parameter b and the generations for parameter c. For these variables we use integers of 16 and 8 bits, respectively. In total, the incremental variant is about 50% larger than the static LOUDS variant.

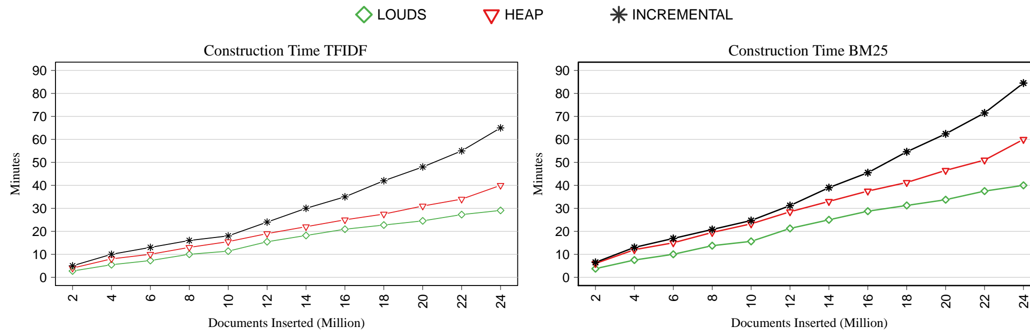

Fig. 22. Incremental versus static construction of the inverted treaps, for tf-idf (left) and BM25 (right) scoring schemes.

The incremental treap is also about 45% slower than the LOUDS implementation in all cases of ranked unions and intersections, and more than twice as slow as HEAPS. We included the times of BP, which show that the incremental treap is still slightly slower than it. There are two main reasons for such a degradation in the performance: First, the free nodes are not located in contiguous memory, leading to cache misses. Cache misses are also caused because each static tree has its own dac_vector and topology bitmap, isolated from those of other static treaps. Second, the incremental treap requires an additional RANK_1 operation each time we move from a free node to a static tree, or from a static tree to another static tree.

Overall, compared with LOUDS, dynamism costs us about 50% overhead in both space and query time performance, and building from scratch by successive insertions requires twice the time of a static construction. Compared with HEAPS, dynamism poses a 50% overhead in both space and construction time, and it requires twice the time at queries. Of course, reconstruction from scratch is not an alternative when insertions are mixed with queries.

8. 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 union and intersection algorithms that simultaneously filter out documents by docid and frequency. The treap also allows us to represent both docids and frequencies in differential form, to improve the compression of the posting lists. Our experiments under the tf-idf scoring scheme show that inverted treaps use about the same space as competing alternatives like Block-Max and Dual-Sorted, but they are significantly faster: from 20 times faster on one-word queries to 3–10 times faster on ranked unions and 1–2 times faster on ranked intersections. On a quantized BM25 score, inverted treaps use about 40% more space than the best alternatives, but they are still 20 times faster on one-word queries, slightly faster on unions, and up to 2 times faster on intersections. Inverted treaps are generally the fastest alternative for $k \leq 100$, and on one- and two-word queries, which are the most popular ones. In addition, we have shown that treaps can handle insertions of new documents, with a 50%–100% degradation in space, construction and query time performance.

A future research line is to study the effect of reassigning docids. Some results [Ding and Suel 2011] show that reassignment can significantly improve both space and processing time. How much would treaps improve with such schemes? Can we optimize the reassignment for a treap layout?

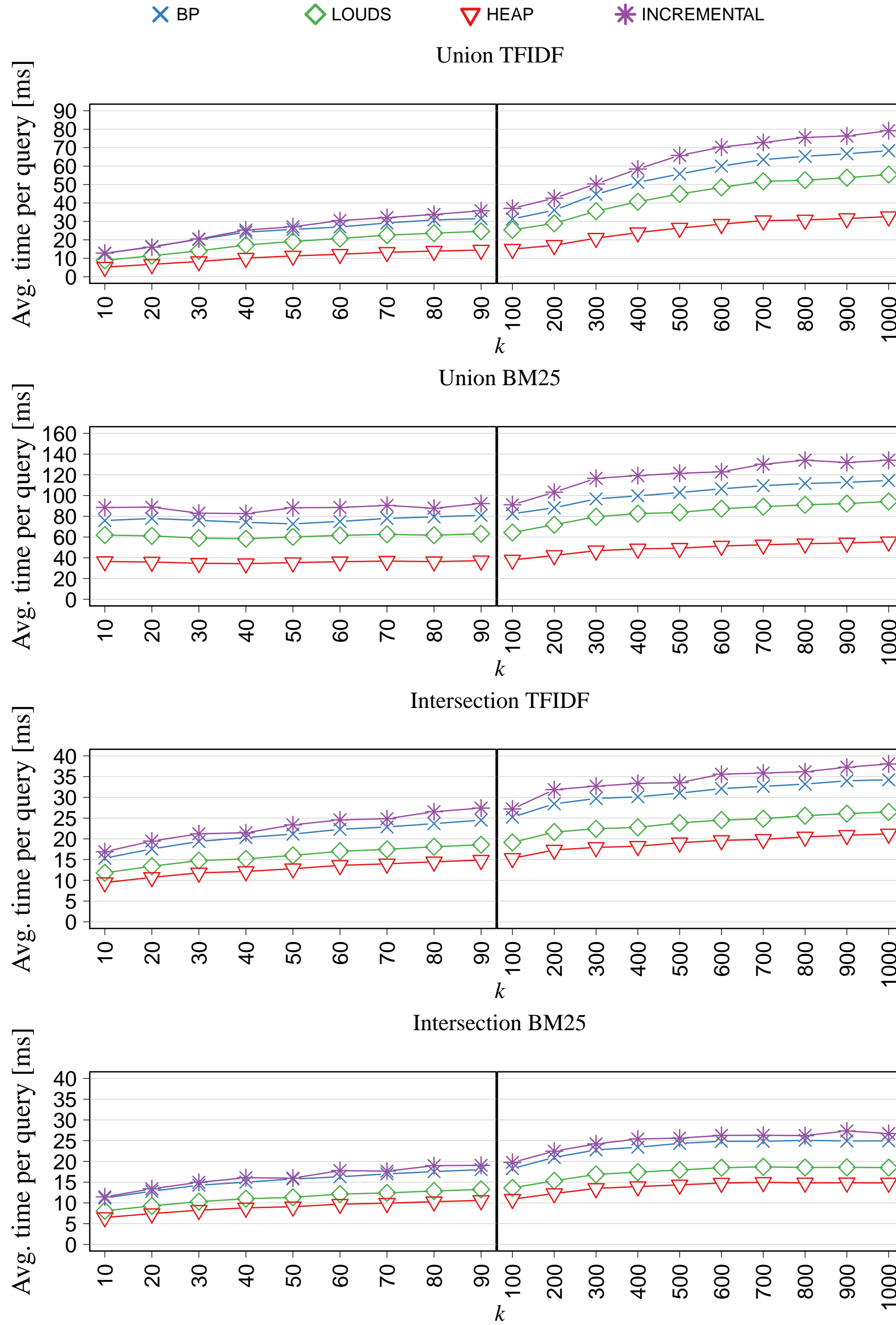

Fig. 23. Ranked union and intersection times as a function of k, using tf-idf and BM25, for our static and incremental variants.

An important part of our gain owed to separating lists with frequency $f_0 = 1$ (this is the main explanation why our scheme performs better on tf-idf than on quantized BM25). How to efficiently separate lists with higher frequencies or impacts is a challenge, and it can lead to important gains. It is also interesting to explore how this idea could impact on schemes like Block-Max and Dual-Sorted.

Third, we have used DAAT processing on our inverted treaps. Such an approach penalizes long queries, as already noted before in Block-Max [Ding and Suel 2011]. We believe the time would become almost nonincreasing with the query length if we used treaps under a TAAT scheme, where the longer lists were processed after determining good lower bounds with the shorter lists. This constitutes another interesting line of future work.

Finally, we plan to evaluate our inverted treaps in a multithreaded environment with queries arriving in batch and seeking to maximize throughput. The fact that, for example, Elias-Fano uses less space than our structures may give it a further advantage that could compensate its higher average time per query.

Acknowledgements

We thank the reviewers for their comments, which helped us improve the presentation significantly.

REFERENCES

A. Andersson and S. Nilsson. 1994. Faster searching in tries and quadtrees - an analysis of level compression. In *Proc. 2nd Annual European Symposium on Algorithms (ESA) (LNCS 855)*. 82–93.

V. Anh, O. Kretser, and A. Moffat. 2001. Vector-space ranking with effective early termination. In *Proc. 24th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 35–42.

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

V. Anh and A. Moffat. 2006. Pruned query evaluation using pre-computed impacts. In *Proc. 29th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 372–379.

D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. 2010. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 84–97.

N. Asadi and J. Lin. 2013. Fast, incremental inverted indexing in main memory for Web-scale collections. *CoRR* abs/1305.0699 (2013). `http://arxiv.org/abs/1305.0699`.

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

R. Baeza-Yates and B. Ribeiro-Neto. 2011. *Modern Information Retrieval* (2nd ed.). Addison-Wesley.

R. Baeza-Yates and A. Salinger. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*. 13–24.

J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. 2009. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics* 14 (2009), 128–140.

M. Bender and M. Farach-Colton. 2000. The LCA problem revisited. In *Proc. 9th Latin American Theoretical Informatics (LATIN) (LNCS 1776)*. 88–94.

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

I. Bialynicka-Birula. 2008. *Ranked Queries in Index Data Structures*. Ph.D. Dissertation. University of Pisa.

I. Bialynicka-Birula and R. Grossi. 2005. Rank-sensitive data structures. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE) (LNCS 3772)*. 79–90.

G. Blelloch and M. Reid-Miller. 1998. Fast set operations using treaps. In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 16–26.

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

A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proc. 12th ACM International Conference on Information and Knowledge Management (CIKM)*. 426–434.

M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. 2012. Earlybird: Real-time search at Twitter. In *Proc. 28th International Conference on Data Engineering (ICDE)*. 1360–1369.

S. Büttcher and C. L. A. Clarke. 2007. Index compression is good, especially for random access. In *Proc. 16th ACM International Conference on Information and Knowledge Management (CIKM)*. 761–770.

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

F. Claude, P. K. Nicholson, and D. Seco. 2012. Differentially encoded search trees. In *Proc. 22nd Data Compression Conference (DCC)*. 357–366.

M. Crane, A. Trotman, and R. O'Keefe. 2013. Maintaining discriminatory power in quantized indexes. In *Proc. 22nd ACM International Conference on Information and Knowledge management (CIKM)*. 1221–1224.

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

J. Culpepper and A. Moffat. 2007. Compact set representation for information retrieval. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*. 137–148.

S. Culpepper and A. Moffat. 2005. Enhanced byte codes with restricted prefix properties. In *Proc 12th International Symposium on String Processing and Information Retrieval (SPIRE) (LNCS 3772)*. 1–12.

E. Demaine, A. López-Ortiz, and J. I. Munro. 2000. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 743–752.

S. Ding and T. Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proc. 34th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 993–1002.

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

S. Gog, T. Beller, A. Moffat, and M. Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. 326–337.

R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. 2005. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 27–38.

R. Grossi, A. Gupta, and J. Vitter. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

H. Heaps. 1978. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press.

G. Jacobson. 1989. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*. 549–554.

R. Konow and G. Navarro. 2012. Dual-sorted inverted lists in practice. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE) (LNCS 7608)*. 295–306.

R. Konow and G. Navarro. 2013. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*. 351–360.

R. Konow, G. Navarro, C. L. A. Clarke, and A. López-Ortíz. 2013. Faster and smaller inverted indices with treaps. In *Proc. 36th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 193–202.

D. Lemire and L. Boystov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

J. Lin and A. Trotman. 2015. Anytime ranking for impact-ordered indexes. In *Proc. ACM International Conference on the Theory of Information Retrieval (ICTIR)*. 198–210.

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

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

J. I. Munro. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 37–42.

J. I. Munro and V. Raman. 2002. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing* 31, 3 (2002), 762–776.

G. Navarro. 2012. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM) (LNCS 7354)*. 2–26.

G. Navarro and S. Puglisi. 2010. Dual-sorted inverted lists. In *Proc. 17th International Conference on String Processing and Information Retrieval (SPIRE) (LNCS 6393)*. 309–321.

G. Ottaviano and R. Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282.

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

M. Petri, S. Culpepper, and A. Moffat. 2013. Exploring the magic of WAND. In *Proc. Australasian Document Computing Symposium (ADCS)*. 58–65.

P. Sanders and F. Transier. 2007. Intersection in integer inverted indices. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*.

F. Scholer, H. Williams, J. Yiannis, and J. Zobel. 2002. Compression of inverted indexes for fast query evaluation. In *Proc. 25th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 222–229.

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

A. Spink, D. Wolfram, , M.Jansen, and T. Saracevic. 2001. Searching the Web: The public and their queries. *Journal of American Society of Information Science and Technology* 52, 3 (2001), 226–234.

A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. 2011. SIMD-based decoding of posting lists. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*. 317–326.

T. Strohman and B. Croft. 2007. Efficient document retrieval in main memory. In *Proc. 30th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 175–182.

A. Trotman. 2014. Compression, SIMD, and postings lists. In *Proc. Australasian Document Computing Symposium (ADCS)*. Article 50.

A. Trotman, X. Jia, and M. Crane. 2012. Towards an efficient and effective search engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*. 40–47.

S. Vigna. 2013. Quasi-succinct indices. In *Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 83–92.

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

L. Wang, J. Lin, and D. Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In *Proc. 34th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 105–114.

H. Williams and J.Zobel. 1999. Compressing integers for fast file access. *SIAM Journal on Computing* 42, 3 (1999), 193–201.

I. Witten, A. Moffat, and T. Bell. 1999. *Managing Gigabytes* (2nd ed.). Morgan Kaufmann.

L. Wu, W. Lin, X. Xiao, and Y. Xu. 2013. LSII: An indexing structure for exact real-time search on microblogs. In *Proc. 29th International Conference on Data Engineering (ICDE)*. 482–493.

H. Yan, S. Ding, and T. Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web (WWW)*. 401–410.

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

J. Zobel and A. Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys* 38, 2 (2006), Article 6.

Received XXX; revised XXXXX; accepted XXXXX