New Algorithms on Wavelet Trees and Applications to Information Retrieval [1]

Travis Gagie[a], Gonzalo Navarro[b], Simon J. Puglisi[c,d]

[a] *Department of Computer Science*
Aalto University, Finland.
[b] *Department of Computer Science*
University of Chile, Chile.
[c] *School of Computer Science and Information Technology*
Royal Melbourne Institute of Technology, Australia
[d] *Newton Fellow, Department of Informatics*
King's College London, United Kingdom

Abstract

Wavelet trees are widely used in the representation of sequences, permutations, text collections, binary relations, discrete points, and other succinct data structures. We show, however, that this still falls short of exploiting all of the virtues of this versatile data structure. In particular we show how to use wavelet trees to solve fundamental algorithmic problems such as *range quantile* queries, *range next value* queries, and *range intersection* queries. We explore several applications of these queries in Information Retrieval, in particular *document retrieval* in hierarchical and temporal documents, and in the representation of *inverted lists*.

Keywords: Information retrieval, Document retrieval, Data structures, 1D range queries, Wavelet trees

1. Introduction

The *wavelet tree* [3] is a versatile data structure that stores a sequence $S[1,n]$ of elements from a symbol universe $[1,\sigma]$ within asymptotically the same space required by a plain representation of the sequence, $n\log\sigma\,(1+o(1))$ bits.[2] Within that space, the wavelet tree is able to return any sequence element $S[i]$, and also to answer two queries on S that are fundamental in compressed data structures for text retrieval:

$$\begin{aligned} rank_c(S,i) &= \text{number of occurrences of symbol } c \text{ in } S[1,i],\\ select_c(S,j) &= \text{position of the } j\text{th occurrence of symbol } c \text{ in } S. \end{aligned}$$

The time for these three queries is $O(\log\sigma)$.[3] Originally designed for compressing suffix arrays [3], the usefulness of the wavelet tree for many other scenarios was quickly realized. For example, it was soon adopted as a fundamental component of a large class of compressed text indexes — the FM-index family [5] — giving birth to most of its modern variants [6, 7, 4, 8].

The connection between the wavelet tree and an old geometric structure by Chazelle [9] made it evident that wavelet trees could be used for *range counting and reporting* points in the plane. More formally, given a set of t points $P = \{(x_i, y_i),\ 1 \le i \le t\}$ on a discrete grid $[1,n]\times[1,\sigma]$, wavelet trees answer the following basic queries:

$$\begin{aligned} range\_count(P, x^s, x^e, y^s, y^e) &= \text{number of pairs } (x_i,y_i)\in P \text{ such that } x^s \le x_i \le x^e,\ y^s \le y_i \le y^e,\\ range\_report(P, x^s, x^e, y^s, y^e) &= \text{list of those pairs } (x_i,y_i)\in P \text{ in some order.} \end{aligned}$$

[1] Early parts of this work appeared in SPIRE 2009 [1] and SPIRE 2010 [2]. The second author was partially supported by Fondecyt Grant 1-110066, Chile. The third author was partially supported by the Australian Research Council.

Email addresses: travis.gagie@gmail.com (Travis Gagie), gnavarro@dcc.uchile.cl (Gonzalo Navarro), simon.puglisi@rmit.edu.au (Simon J. Puglisi)

[2] Our logarithms are in base 2 unless otherwise stated. Moreover, within a time complexity, $\log x$ should be understood as $\max(1,\log x)$.

[3] This can be reduced to $O\big(1+\frac{\log\sigma}{\log\log n}\big)$ [4] using multiary wavelet trees, but, as we will explain later, these do not merge well with the new algorithms we develop in this article.

Query *range_count* is solved in time $O(\log \sigma)$, whereas *range_report* takes time $O((1 + occ) \log \sigma)$ to report *occ* points [10].[4] These new capabilities were subsequently used to design powerful succinct representations of two-dimensional point grids [10, 11, 12], permutations [13], and binary relations [14], with applications to other compressed text indexes [15, 16, 17], document retrieval problems [18] and many others.

In this paper we show, by uncovering new capabilities, that the full potential of wavelet trees is far from realized. In particular, we show that the wavelet tree allows us to solve the following fundamental queries:

$$\begin{aligned} \mathit{range\_quantile}(S, i, j, k) &= k\text{th smallest value in } S[i, j], \\ \mathit{range\_next\_value}(S, i, j, x) &= \text{smallest } S[r] \geq x \text{ such that } i \leq r \leq j, \\ \mathit{range\_intersect}(S, i_1, j_1, \ldots, i_k, j_k) &= \text{distinct common values in } S[i_1, j_1], S[i_2, j_2], \ldots, S[i_k, j_k]. \end{aligned}$$

The first two are solved in time $O(\log \sigma)$, whereas the cost of the latter is $O(\log \sigma)$ per delivered value plus the size of the intersection of the k tries built on the binary representations the values in $S[i_1, j_1], \ldots, S[i_k, j_k]$. A crude upper bound for the latter is $O(\min(\sigma, j_1 - i_1 + 1, \ldots, j_k - i_k + 1))$. However, we give an adaptive analysis of our method, showing it requires $O\left(\alpha k \log \frac{\sigma}{\alpha}\right)$ time, where α is the so-called *alternation complexity* of the problem [19].

All these algorithmic problems are well known. Har-Peled and Muthukrishnan [20] describe applications of range median queries (a special case of *range_quantile*) to the analysis of Web advertising logs. Stolinski et al. [21] use them for noise reduction in grey scale images. Similarly, Crochemore et al. [22] use *range_next_value* queries for interval-restricted pattern matching, and Keller et al. [23] and Crochemore et al. [24] use them for many other sophisticated pattern matching problems. Hon et al. [25] use *range_intersect* queries for generalized document retrieval, and in a simplified form the problem also appears when processing conjunctive queries in inverted indexes.

We further illustrate the importance of these fundamental algorithmic problems by uncovering new applications in several Information Retrieval (IR) activities. We first consider *document retrieval* problems on general sequences. This generalizes the classical IR problems usually dealt with on Natural Language (NL), and defines them in a more general setting where one has a collection C of strings (i.e., the documents), and queries are strings as well. Then one is interested in any substring of the collection that matches the query, and the following IR problems are defined (among several others):

$$\begin{aligned} \mathit{doc\_listing}(C, q) &= \text{distinct documents in } C \text{ where query } q \text{ appears}, \\ \mathit{doc\_frequency}(C, q, d) &= \text{number of occurrences of query } q \text{ in document } d \in C, \\ \mathit{doc\_intersect}(C, q_1, \ldots, q_k) &= \text{distinct documents in } C \text{ where all queries } q_1, \ldots, q_k \text{ appear}. \end{aligned}$$

These generalized IR problems have applications in text databases where the concept of *words* does not exist or is difficult to define, such as in Oriental languages, DNA and protein sequences, program code, music and other multimedia sequences, and numeric streams in general. The interest in carrying out IR tasks on, say, Chinese or Korean is obvious despite the difficulty of automatically delimiting the words. In those cases one resorts to a model where the text is seen as a sequence of symbols and one must be able to retrieve any substring. Agglutinating languages such as Finnish or German present similar problems to a certain degree. While indexes for plain string matching are well known, supporting more sophisticated IR tasks such as ranked document retrieval is a very recent research area. It is not hard to imagine that similar capabilities would be of interest in other types of sequences: for example listing the functions where two given variables are used simultaneously in a large software development system, or ranking a set of gene sequences by the number of times a given substring marker occurs.

By constructing a *suffix array* A [26] on the text collection, one can obtain in time $O(|q| \log |C|)$ (where $|C|$ denotes the sum of document lengths in C) the range of A where all the occurrence positions of q in C are listed. The classical solution to document retrieval problems [27] starts by defining a *document array* D giving the document to which each suffix of A belongs. Then problems like document listing reduce to listing the distinct values in a range of D, and intersection of documents becomes the intersection of values in a range of D. Both are solved with our new fundamental algorithms (the former with range quantile queries). Other queries such as computing frequencies reduce to a pair of $rank_d$ queries on D.

Second, we generalize document retrieval problems to other scenarios. The first scenario is *temporal* documents, where the document numbers are consistent with increasing version numbers of the document set. Then one is interested in restricting the above queries to a given interval of time (i.e., of document numbers). A similar case is that of *hierarchical* documents, which contain each other as in the case of an XML collection or a

[4] Again, this can be reduced to $O\left(1 + \frac{\log \sigma}{\log \log n}\right)$ using multiary wavelet trees [11].

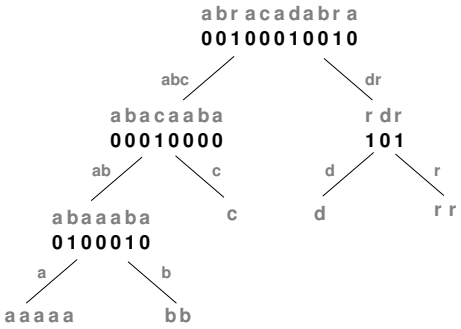

Figure 1: A balanced wavelet tree for the sequence S = "abracadabra". The grayed texts at each node v correspond to subsequence S_v of S, whereas the black bitmaps refer to B_v. The symbols on the edge arriving at a node list the labels of the node. Only the structure and the bitmaps are actually represented.

file system. Here, restricting the query to a subtree of the hiearchy can be reduced to restricting it to a range of document numbers. However, one can consider more complex queries in the hierarchical case, such as marking a set of *retrievable* nodes at query time and carrying out the operations with respect to those nodes. We show how to generalize our algorithms to handle this case as well.

Finally, we show that variants of our new fundamental algorithms are useful to enhance the functionality of *inverted lists*, the favorite data structures for both *ranked* and *full-text retrieval* in NL. Each of these retrieval paradigms requires a different variant of the inverted list, and one has to maintain both in order to support all the activities usually required in an IR system. We show that a wavelet tree representation of the inverted lists supports not only the basic functionality of both representations within essentially the space of one, but also several enhanced functionalities such as on-the-fly stemming and restriction of documents, and most list intersection algorithms.

The article is structured as follows. In Section 2 we review the wavelet tree data structure and its basic algorithmics. Section 3 then describes the new solutions to fundamental algorithmic problems. Then we move on to applications of those basic results in IR scenarios. We first review some basic IR concepts in Section 4. Then Section 5 explores generalized IR problems and Section 6 considers inverted list representations for NL applications. We conclude in Section 7.

2. Wavelet Trees

A *wavelet tree* T [3] for a sequence $S[1,n]$ over an ordered alphabet $[1,\sigma]$ is an ordered, strictly binary tree whose leaves are labeled with the distinct symbols in S in order from left to right, and whose internal nodes T_v store binary strings B_v. The binary string at the root contains n bits and each is set to 0 or 1 depending on whether the corresponding character of S is the label of a leaf in T's left or right subtree. For each internal node v of T, the subtree T_v rooted at v is itself a wavelet tree for the *subsequence* S_v of S consisting of the occurrences of the leaf labels in T_v. In this article we only consider *balanced* wavelet trees, where the number of leaves to the left and to the right of any node differ at most by 1. Figure 1 gives an example.

The wavelet tree stores just the tree structure and the bitmaps B_v, together with data structures to carry out binary *rank* and *select* queries on them (these are essential to navigate the tree, as seen soon). The important properties of such a data structure for this article, in terms of space and construction cost, are summarized in the following lemma.

Lemma 1. *The wavelet tree T for a sequence $S[1,n]$ on alphabet $[1,\sigma]$ with u distinct symbols requires at most $n\log\sigma + O(n)$ bits of space, and can be constructed in $O(n\log u)$ time.*

Proof. It is easy to see by induction that the leaves of a (balanced) wavelet tree are at depth $\lfloor\log u\rfloor$ or $\lceil\log u\rceil$. Thus the wavelet tree has height $\lceil\log u\rceil$ and, following the description above, it can be easily built in time $O(n\log u)$ (we need to determine the $u \le \min(n,\sigma)$ distinct values first, but this is straightforward within the same complexity).

As for the space, note that the total length of the B_v bitmaps is at most n at each level of the wavelet tree, which adds up to $n\lceil\log u\rceil$. Those $n\lceil\log u\rceil$ bits can be represented using a data structure [28] that requires $n\log u + O(n)$ bits and gives constant-time access to any bit, as well as constant-time support for (binary) *rank* and *select* operations.

Apart from the bitmaps, one should store the $O(u)$ nodes, which can be an issue if $u = \omega(n/\log n)$. In this case, instead of storing the pointers explicitly, one can concatenate all the bitmaps of the same level and store one level after the other. If all the leaves of depth $\lceil\log u\rceil$ are put to the left of those of depth $\lfloor\log u\rfloor$, the nodes can be easily simulated with no pointers [10].

Algorithm 1 Basic wavelet tree algorithms: On the wavelet tree of sequence S, **access**(v_{root}, i) returns $S[i]$; **rank**(v_{root}, c, i) returns $rank_c(S, i)$; and **select**(v_{root}, c, i) returns $select_c(S, i)$.

```
access(v, i)                                  rank(v, c, i)                                 select(v, c, i)

  if v is a leaf then                           if v is a leaf then                           if v is a leaf then
    return label(v)                               return i                                      return i
  else if B_v[i] = 0 then                       else if c ∈ labels(v_l) then                  else if c ∈ labels(v_l) then

    return access(v_l, rank_0(B_v, i))            return rank(v_l, c, rank_0(B_v, i))           return select_0(B_v, select(v_l, c, i))
  else                                          else                                          else

    return access(v_r, rank_1(B_v, i))            return rank(v_r, c, rank_1(B_v, i))           return select_1(B_v, select(v_r, c, i))
  end if                                        end if                                        end if
```

The u distinct values must be stored as well. Indeed, if $\sigma \leq n$, we can just assume that all the σ values exist, so the wavelet tree will have $\lceil \log \sigma \rceil$ levels and the lemma holds. Otherwise, we can store a mapping between the universe of σ possible values and the $u \leq n$ actual values using an "indexable dictionary" data structure [29]. This requires $u \log \frac{\sigma}{u} + O(u + \log \log \sigma)$ bits and maps in both directions (telling also whether a value from the universe appears in S or not) in constant time. Thus we can act as if S were a sequence over the alphabet $[1, u]$.

Adding up all the spaces we get $n \log u + O(n) + u \log \frac{\sigma}{u} + O(u + \log \log \sigma) \leq n \log \sigma + O(n)$ bits, and the construction time is $O(n \log u)$. □

The most basic operation of T is to replace S, by retrieving any $S[i]$ value in $O(\log u)$ time. The algorithm is as follows. We first examine the ith bit of the root bitmap B_{root}. If $B_{root}[i] = 0$, then symbol $S[i]$ corresponds to a leaf descending from the left child of the root, and from the right otherwise. In the first case we continue recursively on the left child, T_l. However, position i must now be mapped to the subsequence S_l handled at T_l. Precisely, if the 0 at $B_{root}[i]$ is the jth 0 in B_{root}, then $S[i]$ is mapped to $S_l[j]$. In other words, when we go left, we must recompute $i \leftarrow rank_0(B_{root}, i)$. Similarly, when we go right we set $i \leftarrow rank_1(B_{root}, i)$.

When the tree nodes are not explicit, we find out the intervals corresponding to B_v in the levelwise bitmaps B^d, where d is the depth of v, as follows. $B_{root} = B^0$ is a single bitmap. If B_v corresponds to interval $B^d[l, r]$, then its left child corresponds to $B^{d+1}[l, k]$ and its right child to $B^{d+1}[k + 1, r]$, where $k = rank_0(B^d, r) - rank_0(B^d, l - 1)$ [10].

The wavelet tree can also answer $rank_c(S, i)$ queries on S with a mechanism similar to that for retrieving $S[i]$. This time one decides whether to go left or right depending on which subtree of the current node the leaf labeled c appears in, and not on the bit values of B_v. The final i value when one reaches the leaf is the answer. Again, the process requires $O(\log u)$ time.

Finally, $select_c(S, j)$ is also supported in $O(\log u)$ time using the wavelet tree. This time we start from position j at the leaf labeled c; this indeed corresponds to the jth occurrence of symbol c in S. If the leaf is a left child of its parent v, then the position of that c in S_v is $select_0(B_v, j)$, and $select_1(B_v, j)$ if the leaf is a right child of v. We continue recursively from this new j value until reaching the root, where j is the answer. If the tree nodes are not explicitly stored, we first descend to the node labeled c in order to delimit the interval corresponding to the leaf and to all of its ancestors in the levelwise bitmaps.

Algorithm 1 gives pseudocode for the basic *access*, *rank* and *select* algorithms on wavelet trees. For all the pseudocodes in this article we use the following notation: v is a wavelet tree node and v_{root} is the root node. If v is a leaf then its symbol is $label(v) \in [1, \sigma]$. Otherwise v_l and v_r are its left and right children, respectively, B_v is its bitmap. For all nodes v, $labels(v)$ is the range of leaf labels that descend from v; $labels(v) = [label(v), label(v)]$ if v is a leaf. Note that the recursive code for *select* first descends and then ascends, which is compatible with the way to handle the case when nodes are not explicitly represented.

As we make use of *range_count* and a form of *range_report* queries in this article, we give pseudocode for them as well, in Algorithm 2. The code is simplified for point sets of the form $P = \{(i, y_i),\ 1 \leq i \leq n\}$ on grid $[1, n] \times [1, \sigma]$, where the wavelet tree is built on the sequence $P = y_1 y_2 \ldots y_n$. The techniques are easily extended to handle general point grids [12].

In Section 3 we develop new algorithms based on wavelet trees to solve fundamental algorithmic problems. We prove now a few simple lemmas that are useful for analyzing *range_count* and *range_report*, as well as many other algorithms we introduce throughout the article. Most results are folklore but we reprove them here for completeness.

Definition 1. *A node v is said to* cover *all the leaves that descend from it. A set of leaves L is* covered *by a set of nodes V if L is the union of the leaves covered by the nodes in V.*

Algorithm 2 Range algorithms: **count**$(v_{root}, x^s, x^e, [y^s, y^e])$ returns *range_count*(P, x^s, x^e, y^s, y^e) on the wavelet tree of sequence P; and **report**$(v_{root}, x^s, x^e, [y^s, y^e])$ outputs all pairs (y, f), where $y^s \le y \le y^e$ and y appears $f > 0$ times in $P[x^s, y^s]$, this way extending *range_report*(P, x^s, x^e, y^s, y^e).

count(v, x^s, x^e, rng)
  if $x^s > x^e \ \vee \ labels(v) \cap rng = \emptyset$ **then**
    return 0
  else if $labels(v) \subseteq rng$ **then**
    return $x^e - x^s + 1$
  else
    $x^s_l \leftarrow rank_0(B_v, x^s - 1) + 1$
    $x^e_l \leftarrow rank_0(B_v, x^e)$
    $x^s_r \leftarrow x^s - x^s_l$, $x^e_r \leftarrow x^e - x^e_l$
    return count(v_l, x^s_l, x^e_l, rng)+
        count(v_r, x^s_r, x^e_r, rng)
  end if

report(v, x^s, x^e, rng)
  if $x^s > x^e \ \vee \ labels(v) \cap rng = \emptyset$ **then**
    return
  else if v is a leaf **then**
    output $(label(v), x^e - x^s + 1)$
  else
    $x^s_l \leftarrow rank_0(B_v, x^s - 1) + 1$
    $x^e_l \leftarrow rank_0(B_v, x^e)$
    $x^s_r \leftarrow x^s - x^s_l$, $x^e_r \leftarrow x^e - x^e_l$
    report(v_l, x^s_l, x^e_l, rng)
    report(v_r, x^s_r, x^e_r, rng)
  end if

Lemma 2. *Any contiguous range of ℓ leaves in a wavelet tree is covered by $O(\log \ell)$ nodes.*

Proof. First assume u is a power of 2, so all the leaves are at the same depth. We start with a cover V formed by the ℓ leaves. For each consecutive pair of leaves in V that shares the same parent, replace the pair by their parent. At most two leaves are left in V, and at most $\ell/2$ parents are inserted in V. Repeat the operation on the parents just inserted in V (which are also contiguous in their level), and so on. After working on $\lceil \log \ell \rceil$ levels, no more nodes are inserted and V has at most two nodes per level considered, for a total of $O(\log \ell)$ nodes covering the original interval.

If u is not a power of 2, we note that leaves at level $\lceil \log u \rceil$ come in sibling pairs, and thus a preliminary iteration of the algorithm on that level also leaves at most two such leaves in V and creates $\ell/2$ new nodes at level $\lfloor \log u \rfloor$. Then we go on with the algorithm for powers of 2. □

Lemma 3. *Any set of r nodes in a wavelet tree of u leaves has at most $O\left(r \log \frac{u}{r}\right)$ ancestors.*

Proof. Consider the paths from the root to each of the r nodes. They cannot be all disjoint. They share the least if they diverge from depth $\lceil \log r \rceil$. In this case, all the $O(r)$ tree nodes of depth up to $\lceil \log r \rceil$ belong to some path, and from that depth each of the r paths is disjoint, adding at most $\lceil \log u \rceil - \lceil \log r \rceil$ distinct ancestors. The total is $O\left(r + r \log \frac{u}{r}\right)$. □

Lemma 4. *Any set of r nodes covering a contiguous range of leaves in a wavelet tree of u leaves has at most $O(r + \log u)$ ancestors.*

Proof. First assume u is a power of 2, so all the leaves are at the same depth. We first count all the ancestors of the ℓ consecutive leaves covered and then subtract the sizes of the subtrees rooted at the r nodes $v_1, v_2, \ldots, v_r$. Start with marking the ℓ leaves, and then mark all their parents. At most $2 + \lfloor \ell/2 \rfloor$ distinct parents are marked, as most pairs of consecutive leaves will share the same parent. Mark the parents of the parents. At most $2 + \lfloor (2 + \ell/2)/2 \rfloor \le 3 + \ell/4$ parents of parents are marked. In the next level, at most $7/2 + \ell/8$ nodes are marked, and so on. At height h, the number of marked nodes is at most $4 + \ell/2^h$. Adding over all heights, we have that the total number of ancestors is at most $4 \log u + 2\ell$. Now let ℓ_i be the number of leaves covered by node v_i, so that $\sum_{1 \le i \le r} \ell_i = \ell$. The subtree rooted at each v_i has $2\ell_i - 1$ nodes. By subtracting those subtree sizes and adding back the r root nodes we get $4 \log u + 2\ell - (2\ell - r) + r = O(r + \log u)$.

If u is not a power of 2 we can apply the argument ignoring the nodes at depth $\lceil \log u \rceil$. This only makes a difference if some of the nodes v_i are leaves of depth $\lceil \log u \rceil$, in which case the result changes only by $O(r)$. □

From the lemmas we conclude that **count** in Algorithm 2 takes time $O(\log u)$: it finds the $O(\log(y^e - y^s + 1))$ nodes that cover the range $[y^s, y^e]$ (Lemma 2), by working in time proportional to the number of ancestors of those nodes, $O(\log(y^e - y^s + 1) + \log u) = O(\log u)$ (Lemma 4). Interestingly, **report** in Algorithm 2 can be analyzed in two ways. On one hand, it takes time $O(y^e - y^s + \log u)$ as it arrives at most at $y^e - y^s + 1$ consecutive leaves and thus it works on all of their ancestors (Lemma 4). On the other hand, if it outputs r results (which are not necessarily consecutive), it also works proportionally to the number of their ancestors, $O\left(r \log \frac{u}{r}\right)$ (Lemma 3). The latter is an *output-sensitive* analysis. The following lemma shows that the cost is indeed $O\left(\log u + r \log \frac{y^e - y^s + 1}{r}\right)$.

Lemma 5. *Any set of r nodes covering subsets of ℓ contiguous leaves, on a wavelet tree of u leaves, has at most $O\left(\log u + r \log \frac{\ell}{r}\right)$ ancestors.*

Proof. Let $v_1, \ldots, v_r$ be the r nodes. By Lemma 2 the ℓ leaves are covered by $c = O(\log \ell)$ disjoint nodes $u_1, \ldots, u_c$. Let node u_j cover ℓ_j leaves, so $\sum \ell_j = \ell$. Note that each v_i must descend from exactly one u_j. Say that r_j nodes v_i descend from u_j, so $\sum r_j = r$. Then by Lemma 3 the number of ancestors inside u_j of those r_j nodes is $O\left(r_j \log \frac{\ell_j}{r_j}\right)$. By the log-sum inequality[5] the sum of those numbers is $O\left(r \log \frac{\ell}{r}\right)$. Finally, the ancestors that lie above the subtrees u_j are $O(c + \log u) = O(\log u)$ by Lemma 4, for a total of $O\left(\log u + r \log \frac{\ell}{r}\right)$. □

3. New Algorithms

3.1. Range Quantile

Two näive ways of solving query *range_quantile*(i, j, k) are by sequentially scanning the range in time $O(j - i + 1)$ [30], and by storing the answers to the $O\left(n^3\right)$ possible queries in a table and returning answers in $O(1)$ time. Neither of these solutions is really satisfactory.

Until recently there was no work on range quantile queries, but several authors wrote about *range median queries*, the special case in which k is half the length of the interval between i and j. Krizanc et al. [31] introduced the problem of preprocessing for range median queries and gave four solutions, three of which require time superlogarithmic in n. Their fourth solution requires almost quadratic space, storing $O\left(n^2 \log\log n / \log n\right)$ words to answer queries in constant time (a *word* holds $\log \sigma$ bits). Bose et al. [32] considered approximate queries, and Har-Peled and Muthukrishnan [20] and Gfeller and Sanders [33] considered batched queries. Recently, Krizanc et al.'s fourth solution was superseded by one due to Petersen and Grabowski [34, 35], who slightly reduced the space bound to $O\left(n^2(\log\log n)^2 / \log^2 n\right)$ words.

At about the same time we presented the early version of our work [1], Gfeller and Sanders [33] gave a similar $O(n)$-word data structure that supports range median queries in $O(\log n)$ time and observed in a footnote that "a generalization to arbitrary ranks will be straightforward". A few months later, Brodal and Jørgensen [36] gave a more involved data structure that still takes $O(n)$ words but only $O(\log n / \log\log n)$ time for queries. These two papers have now been merged [37]. Very recently, Jørgensen and Larsen [38] proved a matching lower bound for any data structure that takes $n \log^{O(1)} n$ space.

In the sequel we show that, if S is represented using a wavelet tree, we can answer general range quantile queries in $O(\log u)$ time, where $u \leq \min(\sigma, n)$ is the number of distinct symbols in S. As explained in Section 2, within these $n \log \sigma + O(n)$ bits of space we can also retrieve any element $S[i]$ in time $O(\log u)$, so our data structure actually *replaces* S (requiring only $O(n)$ extra bits). The latest alternative structure [38] may achieve slightly better time but it requires $O(n \log n)$ extra bits of space, apart from being significantly more involved.

Theorem 6. *Given a sequence $S[1, n]$ storing u distinct values over alphabet $[1, \sigma]$, we can represent S within $n \log \sigma + O(n)$ bits, so that range quantile queries are solved in time $O(\log u)$. Within that time we can also know the number of times the returned value appears in the range.*

Proof. We represent S using a wavelet tree T, as in Lemma 1. Query *range_quantile*(i, j, k) is then solved as follows. We start at the root of T and consider its bitmap B_{root}. We compute $n_l = rank_0(B_{root}, j) - rank_0(B_{root}, i-1)$, the number of 0s in $B_{root}[i, j]$. If $n_l \geq k$, then there are at least k symbols in $S[i, j]$ that label leaves descending from the left child T_l of T, and thus we must find the kth symbol on T_l. Therefore we continue recursively on T_l with the new values $i \leftarrow rank_0(B_{root}, i-1)+1$, $j \leftarrow rank_0(B_{root}, j)$, and k unchanged. Otherwise, we must descend to the right child, mapping the range to $i \leftarrow rank_1(B_{root}, i-1) + 1$ and $j \leftarrow rank_1(B_{root}, j)$. In this case, since we have discarded n_l numbers that are already to the left of the kth value, we set $k \leftarrow k - n_l$. When we reach a leaf, we just return its label. Furthermore, we have that the value occurs $j - i + 1$ times in the original range. Since T is balanced and we spend constant time at each node as we descend, our search takes $O(\log u)$ time. □

Algorithm 3 (**rqq**) gives pseudocode. Note that, if u is constant, then so is our query time. On the other hand, we are not aware of a way to reduce this $O(\log u)$ time with a multiary wavelet tree (actually, it is not trivial to avoid increasing the complexity as the arity grows).

[5] Given n pairs of numbers $a_j, b_j > 0$, it holds $\sum a_j \log \frac{a_j}{b_j} \geq \left(\sum a_j\right) \log \frac{\sum a_j}{\sum b_j}$.

Algorithm 3 New wavelet tree algorithms: **rqq**(v_{root}, i, j, k) returns $(range\_quantile(S, i, j, k), f)$ on the wavelet tree of sequence S, assuming $k \leq j - i + 1$, and where f is the frequency of the returned element in $S[i, j]$; **rnv**$(v_{root}, i, j, 1, x)$ returns $(range\_next\_value(S, i, j, x), f, p)$, where f is the frequency and p is the smallest rank of the returned element in the multiset $S[i, j]$ (the element is $\perp$ if no answer exists); and **rint**$(v_{root}, i_1, j_1, i_2, j_2, [y^s, y^e])$ solves an extension of query $range\_intersect(S, i_1, j_1, i_2, j_2)$ outputting triples (y, f_1, f_2), where y are the common elements, f_1 is their frequency in $S[i_1, j_1]$, f_2 is their frequency in $S[i_2, j_2]$, and moreover $y^s \leq y \leq y^e$.

rqq(v, i, j, k)

if v is a leaf **then**
  return $(label(v), j - i + 1)$
else
  $i_l \leftarrow rank_0(B_v, i - 1) + 1$
  $j_l \leftarrow rank_0(B_v, j)$
  $i_r \leftarrow i - i_l$, $j_r \leftarrow j - j_r$
  $n_l \leftarrow j_l - i_l + 1$
  if $k \leq n_l$ **then**
    return rqq(v_l, i_l, j_l, k)
  else
    return rqq$(v_r, i_r, j_r, k - n_l)$
  end if
end if

rnv(v, i, j, p, x)

if $i > j$ **then**
  return $(\perp, 0, 0)$
else if v is a leaf **then**
  return $(x, j - i + 1, p)$
else
  $i_l \leftarrow rank_0(B_v, i - 1) + 1$
  $j_l \leftarrow rank_0(B_v, j)$
  $i_r \leftarrow i - i_l$, $j_r \leftarrow j - j_r$
  $n_l \leftarrow j_l - i_l + 1$
  if $x \in labels(v_r)$ **then**
    return rnv$(v_r, i_r, j_r, p + n_l, x)$
  else
    $(y, f, p') \leftarrow$ **rnv**(v_l, i_l, j_l, p, x)
    if $y \neq \perp$ **then**
      return (y, f, p')
    else
      return rnv$(v_r, i_r, j_r, p + n_l, \min labels(v_r))$
    end if
  end if
end if

rint$(v, i_1, j_1, i_2, j_2, rng)$

if $i_1 > j_1 \vee i_2 > j_2$ **then**
  return
else if $labels(v) \cap rng = \emptyset$ **then**
  return
else if v is a leaf **then**
  output $(label(v), j_1 - i_1 + 1, j_2 - i_2 + 1)$
else
  $i_{1l} \leftarrow rank_0(B_v, i_1 - 1) + 1$
  $j_{1l} \leftarrow rank_0(B_v, j_1)$
  $i_{1r} \leftarrow i_1 - i_{1l}$, $j_{1r} \leftarrow j_1 - j_{1r}$
  $i_{2l} \leftarrow rank_0(B_v, i_2 - 1) + 1$
  $j_{2l} \leftarrow rank_0(B_v, j_2)$
  $i_{2r} \leftarrow i_2 - i_{2l}$, $j_{2r} \leftarrow j_2 - j_{2r}$
  rint$(v_l, i_{1l}, j_{1l}, i_{2l}, j_{2l}, rng)$
  rint$(v_r, i_{1r}, j_{1r}, i_{2r}, j_{2r}, rng)$
end if

3.2. Range Next Value

Again, two naive ways of solving query $range\_next\_value(i, j, x)$ on sequence $S[1, n]$ are scanning in $O(j - i + 1)$ worst-case time, and precomputing all the possible answers in $O(n^3)$ space to achieve constant time queries. Crochemore et al. [22] reduced the space to $O(n^2)$ words while preserving constant query time. Later, Crochemore et al. [24] further improved the space to $O(n^{1+\epsilon})$ words. Mäkinen et al. [39, Lemma 4] give a simple $O(n)$-words space solution based on an augmented binary search tree, with query time $O(\log u)$, where once again $u \leq \min(n, \sigma)$ is the number of distinct symbols in S and $[1, \sigma]$ is the domain of values. Yu et al. [40] improved the time to $O(\log n / \log\log n)$, within linear space. For the special case of semi-infinite queries (i.e., $i = 1$ or $j = n$) one can use an $O(n)$-words and $O(\log\log n)$ time solution by Gabow et al. [41].

By using wavelet trees, we also solve the general problem in time $O(\log u)$. Our space is better than the simple linear-space solution, $n + O(n/\log\sigma)$ words (n of which actually replace the sequence).

Theorem 7. *Given a sequence $S[1, n]$ storing u distinct values over alphabet $[1, \sigma]$, we can represent S within $n \log\sigma + O(n)$ bits, so that range next value queries are solved in time $O(\log u)$. Within the same time we can return the position of the first occurrence of the value in the range.*

Proof. We represent S using a wavelet tree T, as in Lemma 1. Query $range\_next\_value(i, j, x)$ is then solved as follows. We start at the root of T and consider its bitmap B_{root}. If x labels a leaf descending by the right child T_r, then the left subtree is irrelevant and we continue recursively on T_r, with the new values $i \leftarrow rank_1(B_{root}, i - 1) + 1$ and $j \leftarrow rank_1(B_{root}, j)$. Otherwise, we must descend to the left child T_l, mapping the range to $i \leftarrow rank_0(B_{root}, i - 1) + 1$ and $j \leftarrow rank_0(B_{root}, j)$. If our interval $[i, j]$ becomes empty at any point, we return with no value.

When the recursion returns from T_r with no value, we return no value as well. When it returns from T_l with no value, however, there is still a chance that a number $\geq x$ appears on the right in the interval $[i, j]$. Indeed, if we descend to T_r and map i and j accordingly, and the interval is not empty, then we want the minimum value of that interval. Thus from this node we change x by $\min labels(T_r)$ and we are sure to find the value on the leftmost leaf of T_r, without further backtracking. The overall time is $O(\log u)$. □

Algorithm 3 (**rnv**) gives pseudocode. While our space gain may not appear very impressive, we point out that our solution requires only $O(n)$ extra bits on top of the sequence (if we accept the logarithmic slowdown in accessing S via the wavelet tree). Moreover, we can use the same wavelet tree to carry out the other algorithms, instead of requiring a different data structure for each. This is relevant for applications that need support for several operations simultaneously, as we see later in this article. Again, it is not obvious whether multiary wavelet trees could help reduce the time complexity.

3.3. Range Intersection

The query *range_intersect*(i_1, j_1, i_2, j_2), which finds the common symbols in two ranges of a sequence $S[1, n]$ over alphabet $[1, \sigma]$, appears naturally in many cases. In particular, a simplified variant where the two ranges to intersect are sorted in increasing order arises when intersecting full-text inverted lists, when solving intersection, phrase, or proximity queries (see Section 4).

Worst-case complexity measures depending only on the range sizes are of little interest for this problem, as an adversary can always force us to completely traverse both ranges, and time complexity $O(j_1 - i_1 + j_2 - i_2 + 1)$ is easily achieved through merging[6]. More interesting are *adaptive* complexity measures, which define a finer *difficulty measure* for problem instances. For example, in the case of sorted ranges, an instance where the first element of the second range is larger than the last element of the first range is easier (one can establish the emptiness of the result with just one well-chosen comparison) than another where elements are mixed.

A popular measure for this case is called *alternation*, denoted α [19]. For two sorted sequences without repetitions, α can be defined as the number of switches from one sequence to the other in the sorted union of the two ranges. Equivalently, α is the time complexity of a nondeterministic program that guesses which comparisons to carry out. This definition can be extended to intersecting k ranges $[i_r, j_r]$. Formally, the measure α is defined through a function $G : [1, \sigma] \to [0, k]$, where $G[c]$ can be the number of any range (1 to k) in which symbol c does not appear, and $G[c] = 0$ if c appears in all ranges. Then α is the number of zeros in G plus the minimum possible number of switches (i.e., $G[c] \neq G[c + 1]$) in such a function. A lower bound in terms of alternation (holding even for randomized algorithms) [19] is $\Omega\left(\alpha \cdot \sum_{1 \le r \le k} \log \frac{n_r}{\alpha}\right)$, where $n_r = j_r - i_r + 1$. There exist adaptive algorithms matching this lower bound [42, 19, 43].

We show now that the wavelet tree representation of $S[1, n]$ allows a rather simple intersection algorithm that approaches the lower bound, even if one starts from ranges of *disordered* values, possibly with repetitions. For $k = 2$, we start from both ranges $[i_1, j_1]$ and $[i_2, j_2]$ at the root of the wavelet tree. If either range is empty, we stop. Otherwise we map both ranges to the left child of the root using $rank_0$, and to the right child using $rank_1$. We continue recursively on the branches where both intervals are nonempty. If we reach a leaf, then its corresponding symbol is in the intersection, and we know that there are $j_1 - i_1 + 1$ copies of the symbol in the first range, and $j_2 - i_2 + 1$ in the second. For k ranges $[i_r, j_r]$, we maintain them all at each step, and abandon a path as soon as any of the k ranges becomes empty. Algorithm 3 (**rint**) gives pseudocode for the case $k = 2$.

Theorem 8. *Given a sequence $S[1, n]$ storing u distinct values over alphabet $[1, \sigma]$, we can represent S within $n \log \sigma + O(n)$ bits, so that range intersection queries are solved in time $O\left(\alpha k \log \frac{u}{\alpha}\right)$, where k is the number of ranges intersected and α is the alternation complexity of the problem.*

Proof. Consider the function $p : [1, u] \to \{0, 1\}^*$, so that $p(c)$ is a bit stream of length equal to the depth of the leaf representing symbol c in the wavelet tree. More precisely, $p[i]$ is 0 if the leaf descends from the left child of its ancestor at depth i, and 1 otherwise. That is, $p(c)$ describes the path from the root to the wavelet tree leaf labeled c.

Now let T_r be the *trie* (or digital tree) formed by the strings $p(c)$ for all those c appearing in $S[i_r, j_r]$, and let $T_\cap$ be the trie formed by the branches present in all T_r, $1 \le r \le k$. It is easy to see that $T_\cap$ contains precisely the wavelet tree nodes where our intersection algorithm goes beyond line 5. As at most two further children can be visited from those nodes, Algorithm **rint** visits $O(|T_\cap|)$ wavelet tree nodes. Thus its complexity is $O(k \cdot |T_\cap|)$ because we maintain up to k intervals as we traverse $O(|T_\cap|)$ nodes.

We first show that $T_\cap$ has at most α leaves of T. The leaves of $T_\cap$ that are also leaves of T correspond to the symbols that belong to the intersection, and thus to the number of 0s in any function G. This is accounted for in measure α. Let us now focus on the other leaves of $T_\cap$. Assume that G is decomposed into $O(\alpha)$ ranges $G[s_i, e_i] = r_i$. Then each range of leaves $[s_i, e_i]$ is covered, by Lemma 2, by $O(\log(e_i - s_i + 1))$ nodes of T. The union of all those covers gives a cover V of $[1, u]$ of size $O(\sum \log(e_i - s_i + 1))$, which by convexity is $O\left(\alpha \log \frac{u}{\alpha}\right)$. The leaves in $T_\cap$ also cover the leaf interval $[1, u]$ in T, and no node in this second cover can descend from a node

[6] If the ranges are already ordered; otherwise a previous sorting is necessary.

$v \in V$ (as the recursion would have stopped at node v, since it contains no elements appearing in the r_ith interval). The exceptions are the areas where $G[s_i, e_i] = 0$, where $T_\cap$ reaches the leaves of T, but those leaves have already been counted. Therefore $T_\cap$ also has $O\left(\alpha \log \frac{u}{\alpha}\right)$ leaves. By Lemma 4, $T_\cap$ has $O\left(\log u + \alpha \log \frac{u}{\alpha}\right) = O\left(\alpha \log \frac{u}{\alpha}\right)$ nodes overall. □

Our algorithm complexity is pretty close to the lower bound, matched when all $n_r = u$. Note also that our algorithm is easily extended to handle the so-called (t, k)-*thresholded problem* [19], where we return any symbol appearing in at least t of the k ranges. It is simply a matter of abandoning a range only when more than $k - t$ ranges have become empty.

An alternative way to carry out the intersection is by means of the query *range_next_value*(S, i, j, x): Start with $x_1 \leftarrow$ *range_next_value*$(S, i_1, j_1, 1)$ and $x_2 \leftarrow$ *range_next_value*(S, i_2, j_2, x_1). If $x_2 > x_1$ then continue with $x_1 \leftarrow$ *range_next_value*(S, i_1, j_1, x_2); if now $x_1 > x_2$ then continue with $x_2 \leftarrow$ *range_next_value*(S, i_2, j_2, x_1); and so on. If at any moment $x_1 = x_2$ then output it as part of the intersection and continue with $x_1 \leftarrow$ *range_next_value*$(S, i_1, j_1, x_2 + 1)$. It is not hard to see that there must be a switch in G for each step we carry out, and therefore the cost is $O(\alpha \log u)$.

To reduce the cost to $O\left(\alpha \log \frac{u}{\alpha}\right)$, we carry out a *fingered search* in *range_next_value* queries, that is, we remember the path traversed from the last time we called *range_next_value*(S, i, j, x) and only retraverse the necessary part upon calling *range_next_value*(S, i, j, x') for $x' > x$. For this purpose we move upwards from the leaf where the query for x was solved until reaching the first node v such that $x' \in labels(v)$, and complete the **rnv** procedure from that node. Since the total work done by this variant is proportional to the number of distinct ancestors of the α leaves arrived at, the complexity is $O\left(\alpha \log \frac{u}{\alpha}\right)$ by Lemma 3.

This second procedure is the basis of most algorithms for intersecting two or more lists [44]. The **rint** method we have presented has the same complexity, yet it is simpler, potentially faster, and more flexible (e.g., it is easily adapted to t-thresholded queries). Moreover, it is specific to the wavelet tree.

4. Information Retrieval Concepts

4.1. Suffix and Document Arrays

Let C be a collection of *documents* (which are actually strings over an alphabet $[1, \sigma]$) $D_1, D_2, \ldots, D_m$. Assume strings are terminated by a special character "$", which does not occur elsewhere in the collection. Now we define C as the concatenation of all the documents, $C[1, n] = D_1 D_2 \ldots D_m$. Each position i defines a *suffix* $C[i, n]$. A *suffix array* [26] of C is an array $A[1, n]$ where the integers $[1, n]$ are ordered in such a way that the suffix starting at $A[i]$ is lexicographically smaller than that starting at $A[i + 1]$, for all $1 \leq i < n$.

Put another way, the suffix array lists all the suffixes of the collection in lexicographic order. Since any substring of C is the prefix of a suffix, finding the occurrences of a query string q in C is equivalent to finding the suffixes that start with q. These form a lexicographic range of suffixes, and thus can be found via two binary searches in A (accessing C for the string comparisons). As each step in the binary search may require comparing up to $|q|$ symbols, the total search time is $O(|q| \log n)$. Once the interval $A[sp, ep]$ is determined, all the occurrences of q start at $A[i]$ for $sp \leq i \leq ep$. Compressed full-text self-indexes permit representing both C and A within the space required to represent C in compressed form, and for example determine the range $[sp, ep]$ within time $O(|q| \log \sigma)$ and list each $A[i]$ in time $O\left(\log^{1+\epsilon} n\right)$ for any constant $\epsilon > 0$ [4, 45].

For listing the distinct documents where q appears, one option is to find out the document to which each $A[i]$ belongs and remove duplicates. This, however, requires $\Omega(ep - sp + 1)$ time; that is, it is proportional to the *total* number of occurrences of q, $occ = ep - sp + 1$. This may be much larger than the number of distinct documents where q appears, $docc$.

Muthukrishnan [27] solved this problem optimally by defining a so-called *document array* $D[1, n]$, so that $D[i]$ is the document suffix $A[i]$ belongs to. Other required data structures in his solution are an array $E[1, n]$, so that $E[i] = \max\{j < i, D[j] = D[i]\}$, and a data structure to compute range minimum queries on E, $RMQ_E(i, j) = \operatorname{argmin}_{i \leq k \leq j} E[k]$. Muthukrishnan was able to list all the distinct documents where q appears in time $O(docc)$ once the interval $A[sp, ep]$ was found. However, the data structures occupied $O(n \log n)$ bits of space, which is too much if we consider the compressed self-indexes that solve the basic string search problem. Another problem is that the resulting documents are not retrieved in ascending order, which is inconvenient for several purposes.

Välimäki and Mäkinen [18] were the first to illustrate the power of wavelet trees for this problem. By representing D with a wavelet tree, they simulated $E[i] = select_{D[i]}(D, rank_{D[i]}(D, i - 1))$ without storing it. By using a $2n$-bit data structure for RMQ [46], the total space was reduced to $n \log m + O(n)$ bits, and still Muthukrishnan's algorithm was simulated within reasonable time, $O(docc \log m)$.

Ranked document retrieval is usually built around two measures: *term frequency*, $tf_{d,q} = doc\_frequency(C, q, d)$, is the number of times the query q appears in document d, and the *document frequency*, df_q, is the number of different documents where q appears. For example a typical weighting formula is $w_{d,q} = tf_{d,q} \times idf_q$, where $idf_q = \log \frac{m}{df_q}$ is called the *inverse document frequency*. Term frequencies are computed with wavelet trees as $doc\_frequency(C, q, d) = rank_d(D, ep) - rank_d(D, sp - 1)$. Document frequencies can be computed with just $2n + o(n)$ more bits for the case of the D array [47], and on top of a wavelet tree for the E array for more general scenarios [48].

In Section 5 we show how our new algorithms solve the document listing problem within the same time complexity $O(docc \log m)$, without using any RMQ data structure, while reporting the documents in increasing order. This is the basis for a novel algorithm to list the documents where two (or more) queries appear simultaneously. We extend these solutions to temporal and hierarchical document collections.

4.2. Inverted Indexes

The *inverted index* is a classical IR structure [49, 50], lying at the heart of most modern Web search engines and applications handling natural-language text collections. By "natural language" texts one refers to those that can be easily split into a sequence of *words*, and where queries are also limited to words or sequences thereof (*phrases*). An inverted index is an array of *lists*. Each array entry corresponds to a different word of the collection, and its list points to the documents where that word appears. The set of different words is called the *vocabulary*. Compared to the document retrieval problem for general strings described above, the restriction of word queries allows inverted indexes to precompute the answer to each possible word query.

Two main variants of inverted indexes exist [51, 52]. *Ranked retrieval* is aimed at retrieving documents that are most "relevant" to a query, under some criterion. As explained, a popular formula for relevance is $w_{d,q} = tf_{d,q} \times idf_q$, but others built on tf and df, as well as even more complex ones, have been used (see, e.g., Zobel and Moffat [53]). In inverted indexes for ranked retrieval, the lists point to the documents where each word appears, storing also the weight of the word in that document (in the case of $tf \times idf$, only tf values are stored, since idf depends only on the word and is stored with the vocabulary). IR queries are usually formed by various words, so the relevance of the documents is obtained by some form of combination of the various individual weights. Algorithms for this type of query have been intensively studied, as well as different data organizations for this particular task [54, 50, 52, 55, 56]. List entries are usually sorted by *descending weights* of the term in the documents.

Ranked retrieval algorithms try to avoid scanning all the involved inverted lists. A typical scheme is Persin's [54]. It first retrieves the shortest list (i.e., with highest idf), which becomes the candidate set, and then considers progressively longer lists. Only a prefix of the subsequent lists is considered, where the weights are above a threshold. Those documents are merged with the candidate set, accumulating relevance values for the documents that contain both terms. The longer the list, the least relevant is the term (as the tfs are multiplied by a lower idf), and thus the shorter the considered prefix of its list. The threshold provides a time/quality tradeoff.

The second variant is the inverted indexes for so-called *full-text retrieval* (also known as *boolean retrieval*). These simply find all the documents where the query appears. In this case the lists point to the documents where each term appears, usually in *increasing document* order. Queries can be single words, in which case the retrieval consists simply of fetching the list of the word; or disjunctive queries, where one has to fetch the sorted lists of all the query words and merge them; or conjunctive queries, where one has to intersect the lists. Intersection queries are nowadays more popular, as this is Google's default policy to treat queries of several words. Another important query where intersection is essential is the phrase query, where intersecting the documents where the words appear is the first step.

While intersection can be achieved by scanning all the lists in synchronization, faster approaches aim to exploit the phenomenon that some lists are much shorter than others [57]. This general idea is particularly important when the lists for many terms need to be intersected. The amount of recent research on intersection of inverted lists witnesses the importance of the problem [42, 19, 58, 59, 60, 61, 62, 43] (see Barbay et al. [44] for a comprehensive survey). In particular, in-memory algorithms have received much attention lately, as large main memories and distributed systems make it feasible to hold the inverted index entirely in RAM.

Needless to say, space is an issue in inverted indexes, especially when combined with the goal of operating in main memory. Much research has been carried out on compressing inverted lists [50, 63, 52, 62], and on the interaction of compression with query algorithms, including list intersections. Most of the list compression algorithms for full-text indexes rely on the fact that the document identifiers are increasing, and that the differences between consecutive entries are smaller on the longer lists. The differences are thus represented with encodings that favor small numbers [50]. Random access is supported by storing sampled absolute values. For lists sorted

by decreasing weights, these techniques can still be adapted: most documents in a list have small weight values, and within the same weight one can still sort the documents by increasing identifier.

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

In Section 6 we show that wavelet trees allow one to build a data structure that permits, within the same space required for a single compressed inverted index, retrieving the list of documents of any term in either decreasing-weight or increasing-identifier order, thus supporting both types of retrieval. Moreover, we can efficiently support the operations needed to implement any of the intersection algorithms, namely: retrieve the ith element of a list, retrieve the first element larger than x, retrieve the next element, and several more complex ones.

In addition, our structure offers novel ways of carrying out several operations of interest. These include, among others, the support for stemming and for structured document retrieval without any extra space cost. *Stemming* is a useful tool to enhance recall [64, 65] in which terms having the same root word are treated as the same term. For example, an IR system using stemming would treat `ironing`, `ironed` and `irons` all as the same term: `iron`. One common way to support stemming is by coalescing terms having the same root at index construction time. However, the index is then unable to provide non-stemmed searching. One can of course index the stemmed and non-stemmed occurrence of each term, but this costs space. Once again, our method can provide both types of search without using any extra space, provided all the variants of the same stemmed word be contiguous in the vocabulary (this is in many cases automatic as stemmed terms share the same root, or prefix).

5. Document Listing and Intersections

The algorithm for *range_report*(P, x^s, x^e, y^s, y^e) queries described in Section 2 can be used to solve *doc_listing*(C, q), as follows. As explained in Section 4.1, use a (compressed) suffix array A to find the range $A[sp, ep]$ corresponding to query q, and use a wavelet tree on the document array $D[1, n]$ on alphabet $[1, m]$, so that the answer is the set of distinct document numbers $d_1 < d_2 < \ldots < d_{docc}$ in $D[sp, ep]$. Then *range_report*$(D, sp, ep, 1, m)$ returns the $docc$ document numbers, in order, in total time $O\left(\log m + docc \log \frac{m}{docc}\right)$, due to Lemma 5. Moreover, procedure **report** in Algorithm 2 also retrieves the frequencies of each d_i in $D[sp, ep]$, outputting the pairs $(d_i, tf_{d_i,q})$ within the same cost. (As explained, arbitrary frequencies $tf_{d,q} = doc\_frequency(C, q, d)$ can also be obtained in time $O(\log m)$ by two $rank_d$ queries on D.)

Corollary 9. *Let C be a text collection of m documents and $C[1, n]$ their concatenation. Then, given the suffix array interval of a query q in C, we can solve query doc_listing(C, q) in time $O\left(\log m + docc \log \frac{m}{docc}\right)$, where $docc$ is the size of the output, using a data structure that occupies $n \log m + O(n)$ bits. Within the same time we also give the term frequencies $tf_{d_i,q}$ of the retrieved documents d_i.*

As explained in Section 4.1, this solution is simpler and requires less space than various previous ones[7], and has the additional benefit of delivering the documents in increasing document identifier order. This enables us to extend the algorithm to more complex scenarios, as shown in Section 6. In those scenarios, alternative solutions using *range_quantile* or *range_next_value* queries, instead of *range_report*, will be of interest.

Now consider k queries $q_1, q_2, \ldots, q_k$, and the problem of listing the documents where all those queries appear (i.e., problem *doc_intersect*$(C, q_1, \ldots, q_k)$). With the suffix array we can map the queries to ranges $[sp_r, ep_r]$, and then the problem is that of finding the distinct document numbers that appear in all those ranges. This corresponds exactly to query *range_intersect*$(D, sp_1, ep_1, \ldots, sp_k, ep_k)$, which we have solved in Section 3.3 (**rint** in Algorithm 3). We also delivered the tf_{d,q_r} values.

Corollary 10. *Let C be a text collection of m documents and $C[1, n]$ their concatenation. Then, given the suffix array intervals of queries $q_1, \ldots, q_k$ in C, we can solve query doc_intersect$(C, q_1, \ldots, q_k)$ in time $O\left(\alpha k \log \frac{m}{\alpha}\right)$, where α is the alternation complexity of the intersection problem, using a data structure that occupies $n \log m + O(n)$ bits. Within the same time we also give the term frequencies tf_{d_i,q_r} of the delivered documents d_i.*

We have indeed solved a more general variant where we list the documents where at least t of the k terms appear. This corresponds to the disjunctive query for the case $t = 1$ and to a conjunctive query for $t = k$. Note that the data structure referred to in both corollaries is the same.

[7] It is even better than our previous solution based on *range_quantile* queries [1], which takes time $O(docc \log m)$.

5.1. Temporal and Hierarchical Documents

The simplest extension when we have versioned or hierarchical documents is to restrict queries *doc_listing*(C, q) and *doc_intersect*$(C, q_1, \ldots, q_k)$ to a range of documents $[d_{min}, d_{max}]$, which represents a temporal interval or a subtree of the hierarchy in which we are interested. Such a restricted document listing and intersection is easily supported by setting $rng = [d_{min}, d_{max}]$ in procedures **report** (Algorithm 2) and **rint** (Algorithm 3), respectively. The complexities are $O\left(\log m + docc \log \frac{d_{max}-d_{min}+1}{docc}\right)$ for listing (due to Lemma 5), and $O\left(k\left(\log m + \alpha \log \frac{d_{max}-d_{min}+1}{\alpha}\right)\right)$ for intersections (due to a simple adaptation of Theorem 8).

When the hierarchical documents represent nodes in an XML collection, other queries become of obvious interest. Indeed, how to carry out ranking on XML collections is an unresolved issue, with very complex ranking proposals counterweighted by others advocating simple measures. Rather than trying to cover such a broad topic, we refer the reader to comprehensive surveys and discussions in the article by Hiemstra and Mihajlović [66], the PhD thesis of Pehcevski [67, Ch. 2], and the recent book by Lalmas [68, Ch. 6].

In most models, the frequency of a term within a subtree, and the size of such subtree, are central to the definition of ranking strategies. The latter is usually easy to compute from the sequence representation. The former, a generalization of *doc_frequency* to ranges, can actually be computed with query *range_count*(D, sp, ep, dl, dr) (see Algorithm 2), where $[sp, ep]$ is the suffix array range corresponding to query q, and $[dl, dr]$ is the range of documents corresponding to our structural element. This query also takes time $O(\log m)$.

5.2. Restricting to Retrievable Units

We focus now on a more complex issue that is also essential for XML ranked retrieval. Query languages such as XPath and XQuery define structural constraints together with terms of interest. For example, one might wish to retrieve books about the term `cryptography`, or rather book sections about that term, in each case ranked by the relevance of the term. Thus the definition of the *retrievable unit* (books, sections) comes in the query together with the terms (`cryptography`) whose relevance is to be computed with respect to the retrievable units that contain it. We show now how to support a simple model where the retrievable units are defined by an XML *tag* name, and consider other models at the end. We assume that retrievable units of the same type do not nest.

Following common models of XML data, we consider that text data can appear only at the leaves of the XML structure (the general case is easily managed with this model [69]). Thus, each leaf of the XML tree will be associated with a document number, 1 to m, so that d will be the document associated to the dth leaf. The XML tree, containing n nodes, will be represented using a sequence $P[1, 2n]$ of parentheses [70]. These are obtained through a preorder traversal, by appending an opening parenthesis when we reach a node and a closing one when we leave it. A tree node will be identified with the position of its opening parenthesis in P. Several succinct data structures can represent the parentheses within $2n + o(n)$ bits and simulate a wealth of tree operations in constant time (e.g., [71]).

In addition we represent a sequence $Tag[1, 2n]$ giving the tag name associated to each parenthesis in P. Sequence Tag is represented using a data structure that requires $2n \log \tau + O(n)$ bits of space, where τ is the number of distinct tags in the collection, and answers *rank*/*select* queries on Tag in time $O(\log \log \tau)$ [72].

A first task we can carry out is, given an occurrence in document number (i.e., leaf) d, compute *expand*(t, d), the range of documents (i.e., leaves) corresponding to its ancestor tagged t, or determine there is no such ancestor. We use operation $j = selectLeaf(P, d)$ to find the dth leaf of P. Then $p = select_t(Tag, rank_t(Tag, j))$ finds the rightmost parenthesis preceding j corresponding to a node tagged t. If $P[p] = '('$, then p is the ancestor of d tagged t. Otherwise, there is no ancestor of d tagged t and the next node tagged t is $p = select_t(Tag, rank_t(Tag, j) + 1)$. In either case, we return the range of leaves corresponding to p, $expand(t, d) = leaf\_range(p) = [rankLeaf(P, p) + 1, rankLeaf(P, p + 2 \cdot subtreeSize(P, p) - 1)]$, where $rankLeaf(P, s)$ counts the number of leaves contained in $P[1, s]$ and $subtreeSize(P, p)$ counts the number of nodes of the subtree rooted at p. The process takes $O(\log \log \tau)$ time, dominated by the costs to operate on Tag. Algorithm 4 (**exp** and **leafRange**) gives pseudocode.

If we now want to count the number of occurrences of our query q in a retrievable node p, we need to count the number of occurrences of leaves (i.e., document numbers) below p within the interval $D[sp, ep]$ corresponding to query q. Such a range is easily obtained in constant time as $[dl, dr] = leaf\_range(p)$. Then the result is *range_count*(D, sp, ep, dl, dr), as explained (see **hdfreq** in Algorithm 4).

To carry out document listing restricted to structural elements tagged t, we build on range next value queries. We start with $d_1 = range\_next\_value(D, sp, ep, 1)$, which gives us the smallest (leaf) document number in $D[sp, ep]$. Now we compute $[dl_1, dr_1] = expand(t, d_1)$, the range of the node tagged t that contains d_1 (or the leftmost following it). If $d_1 \in [dl_1, dr_1]$ we report the range and find the next document using $d_2 = range\_next\_value(D, sp, ep, dr_1 + 1)$, otherwise we do not report the range and compute $d_2 = range\_next\_value(D, sp, ep, dl_1)$. We continue until no more occurrences are found. Algorithm 4 (**hdlist**) gives pseudocode.

Algorithm 4 Algorithms for hierarchical document listing and intersections: **exp**(Tag, P, t, d) computes the node in P for $expand(t, d)$ and **leafRange**(P, p) computes $leaf\_range(p)$; **hdfreq**(P, D, sp, ep, p) computes the frequency of p in $D[sp, ep]$; and **hdlist**$(A, D, Tag, P, t, q, rng)$ lists the retrievable units where q appears (*pattern_search(A,q)* returns the interval of the suffix array A where the occurrences of the pattern q lie). **hdlist** only reports documents in range rng (which is assumed not to split any retrievable unit).

exp(Tag, P, t, d)
  $j \leftarrow selectLeaf(P, d)$
  $r \leftarrow$ **rank**(Tag, t, j)
  $p \leftarrow$ **select**(Tag, t, r)
  if $P[p] = ')'$ **then**
    $p \leftarrow$ **select**$(Tag, t, r + 1)$
  end if
  return p

leafRange(P, p)
  return $[rankLeaf(P, p) + 1,$
  $rankLeaf(P, p+2 \cdot subtreeSize(P, p)-1)]$

hdfreq(P, D, sp, ep, p)

  $[dl, dr] \leftarrow$ **leafRange**(P, p)
  return count$(D, sp, ep, [dl, dr])$

hdlist$(A, D, Tag, P, t, q, [d_{min}, d_{max}])$
  $[sp, ep] \leftarrow pattern\_search(A, q)$
  $v \leftarrow root(D)$
  $(d, f, r) \leftarrow$ **rnv**$(v, sp, ep, 1, d_{min})$
  while $d \neq \perp \ \wedge \ d \leq d_{max}$ **do**
    $p \leftarrow$ **exp**(Tag, P, t, d)
    $[dl, dr] \leftarrow$ **leafRange**(P, p)
    if $d \in [dl, dr]$ **then**
      output $[dl, dr]$
      $(d, f, r) \leftarrow$ **rnv**$(v, sp, ep, 1, dr+1)$
    else
      $(d, f, r) \leftarrow$ **rnv**$(v, sp, ep, 1, dl)$
    end if
  end while

The cost per step is $O(\log\log\tau + \log m)$, and it is easy to see that the total number of steps is $O(\alpha)$, where α is the alternation complexity of the problem of intersecting the list of endpoints of nodes tagged t and the leaf documents where q occurs. Using the fingered search on **rnv** outlined in Section 3.3, the overall cost is $O\left(\log m + \alpha\left(\log\log\tau + \log\frac{m}{\alpha}\right)\right)$. If we wish to additionally restrict the retrieval to documents in the range $[d_{min}, d_{max}]$, we simply start with $d_1 = range\_next\_value(D, sp, ep, d_{min})$ and stop when we retrieve a document larger than d_{max}. The cost improves to $O\left(\log m + \alpha\left(\log\log\tau + \log\frac{d_{max}-d_{min}+1}{\alpha}\right)\right)$ due to Lemma 5. The complexity returns to $O(\alpha(\log\log\tau + \log m))$ if we compute also the frequency in each retrievable unit using **hdfreq**.

Corollary 11. *Let C be a hierarchical text collection of n tree nodes with tags in $[1, \tau]$ and m text nodes at the leaves, being $C[1, N]$ the concatenation of all the texts, and where tags do not nest. Then, given a tag t and the suffix array interval of a query q in C, we can list the distinct tree nodes tagged t that contain an occurrence of q, restricted to any desired subtree containing m' text nodes, in time $O\left(\log m + \alpha\left(\log\log\tau + \log\frac{m'}{\alpha}\right)\right)$, using a data structure that occupies $N\log m + 2n\log\tau + O(N)$ bits. Here α is the alternation complexity of the problem of intersecting the tag endpoints and the document leaves where q appears.*

Finally, to carry out intersections restricted to retrievable units, we proceed in principle as **rint** in Algorithm 3 (Section 3.3). The only difference is that, instead of outputting each result, we **expand** it and report the retrievable unit, if any. Then we advance as in algorithm **hdlist**, to $dr + 1$ or to dl. It is not hard to see that the complexity is $O\left(\alpha\left(\log\log\tau + \log\frac{m}{\alpha}\right)\right)$, where now α refers to the alternation complexity of the k sequences to intersect plus the sequence of starting at ending points of tag t.

Other possibilities for marking the retrievable documents can be supported, as long as one is able to expand any leaf. For example we could mark retrievable nodes in a bitmap $B[1, 2n]$ aligned with P, where we set to 1 the opening and closing parentheses of retrievable nodes. Then we can compute $expand(B, i)$ via *rank* and *select* operations on B in constant time as follows. We start with $j = selectLeaf(P, d)$, then $p = select_1(rank_1(B, j))$, then if $P[p] = ')'$ position d has no covering retrievable unit, else $expand(B, d) = leaf\_range(p)$.

6. Inverted Lists

Recall m is the total number of documents in the collection and let v be the number of different terms. Let $L_t[1, df_t]$ be the list of document identifiers where term t appears, in decreasing weight order (for concreteness we will assume we store tf values in the lists as weights, but any weight will do). Let $n = \sum_t df_t$ be the total number of occurrences of *distinct* terms in the documents, and $N = \sum_{t,d} tf_{d,t}$ the total length, in words, of the text collection (thus $m \leq n \leq \min(mv, N)$). Finally, let $|q|$ be the number of terms in query q.

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

According to Lemma 1, the wavelet tree of L occupies is $n \log m + O(n)$ bits. The classical encoding of inverted files, when documents are sorted by increasing document identifier, records the consecutive differences using Rice codes [50]. This needs at most $\sum_t df_t \log \frac{m}{df_t} + O(df_t) \leq n \log \frac{mv}{n} + O(n)$ bits, which is asymptotically less than our space. If, however, the lists are sorted by decreasing *tf* values, then differential encoding can only be used on some parts of the lists. Yet, $n \log m + O(n)$ is still an upper bound to the space required to list the documents. As can be seen, no inverted index representation takes more space than our wavelet tree. However, our wavelet tree will offer the combined functionality of *both* inverted indexes, and more.

Sequence s_t is represented using a bitmap $S[1,n]$ providing *rank*/*select* operations. Thus we can recover $s_t = select_1(S,t)$, and also $rank_1(S,i)$ tells us which list $L[i]$ belongs to. A "fully indexable dictionary" [29] provides these operations in constant time using $v \log \frac{n}{v} + O(v) + o(n)$ bits. These spaces are similar to those used to represent this data in traditional *tf*-sorted indices.

We will now consider the classical and extended operations that can be carried out with our data structure. In particular we will show how to give some support for hierarchical document retrieval (as already seen for general documents) and for stemmed searches without using any extra space.

6.1. Full-Text Retrieval

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

6.1.1. Direct retrieval

First, with our wavelet tree representation of L we can compute any specific value $F_t[k]$ in time $O(\log m)$. This is equivalent to finding the kth smallest value in $L[s_t, s_{t+1} - 1]$, that is, query $range\_quantile(L, s_t, s_{t+1} - 1, k)$ described in Section 3.1.

We can also extract any segment $F_t[k,k']$, in order, in time $O(\log m + k' - k)$. The algorithm is the same as for *range_quantile* on quantiles k to k' simultaneously, going just by one branch when both k and k' choose the same branch, and splitting the interval into two separate searches when they do not. We arrive at $k' - k + 1$ consecutive leaves of the wavelet tree, thus the cost follows from Lemma 4. The same complexity is achieved using the *fingered search* on *range_next_value* queries outlined at the end of Section 3.3.

A more general fingered search operation is to find $F_t[k']$ after having visited $F_t[k]$, for some $k' > k$. We need to store $\log m$ values m_δ, e_δ and v^δ, where $m_0 = \infty$ and $e_1 = 0$, and the others are computed as follows when we obtain $F_t[k]$: at wavelet tree node v of depth δ (the root being depth 1) we set $v^\delta \leftarrow v$ and, if we must go to the left child, then we set $m_\delta \leftarrow e_\delta + n_l$ and $e_{\delta+1} \leftarrow e_\delta$; else we set $m_\delta \leftarrow m_{\delta-1}$ and $e_{\delta+1} \leftarrow e_\delta + n_l$. Here n_l is the value local to the node (recall **rqq** in Algorithm 3). Therefore e_δ counts the values skipped to the left, and m_δ is the maximum k' value such that the downward paths to compute $F_t[k]$ and $F_t[k']$ coincide up to depth δ. Now, to compute $F_t[k']$, we consider all the δ values, from largest to smallest, until finding the first one such that $k' \leq m_\delta$. From there on we recompute the downward path, resetting m_δ, e_δ, and v^δ accordingly.

If we carry out this operation r times, across a range $[k,k']$, the cost is $O\left(\log m + r \log \frac{k'-k+1}{r}\right)$ by Lemma 5. Algorithm 5 depicts the new extended variants of **rqq**.

6.1.2. Intersection algorithms

The most important operation in the various list intersection algorithms described in the literature is to find the first k such that $F_t[k] \geq d$, given d. This is usually solved with a combination of sampling and linear, exponential, or binary search. In our case, this operation takes time $O(\log m)$ with query $range\_next\_value(L, s_t, s_{t+1} - 1, d)$ described in Section 3.2. Our time complexity is not far from the $O(\log(s_{t+1} - s_t))$ of traditional approaches. Moreover, as explained in Section 3.3, we can use fingered searches on **rnv** to achieve time $O\left(\log m + r \log \frac{m}{r}\right)$ for r accesses. Furthermore, if all the accesses are for documents in a range $[d,d']$ then, by Lemma 5, the cost will be $O\left(\log m + r \log \frac{d'-d+1}{r}\right)$ time. This is indeed the time required by r successive searches using exponential search.

Finally, we can intersect the lists F_t and $F_{t'}$ using $range\_intersect(L, s_t, s_{t+1} - 1, s_{t'}, s_{t'+1} - 1)$, in adaptive time $O\left(\alpha \log \frac{m}{\alpha}\right)$ — recall Section 3.3. As explained, this can be extended to intersecting $|q|$ terms simultaneously, and to report documents where a minimum number of the terms appear.

The following corollary summarizes the most fundamental results.

Algorithm 5 Extended variants of range quantile algorithms: **mrqq**(v_{root}, i, j, k, k') outputs all the (distinct) values *range_quantile*(S, i, j, k) to *range_quantile*(S, i, j, k'), with their frequencies, on the wavelet tree of sequence S, assuming $k' \leq j - i + 1$; **frqq1**(v_{root}, i, j, k) returns the same as **rqq**(v_{root}, i, j, k) but prepares the iterator for subsequent fingered searches; those are carried out by calling **frqq**(v_{root}, k), where it is assumed that the k values increase at each call; **frqq′** is the recursive procedure that reprocesses the needed part of the path.

mrqq(v, i, j, k, k')

if v is a leaf **then**
  output $(label(v), j - i + 1)$
else
  $i_l \leftarrow rank_0(B_v, i - 1) + 1$
  $j_l \leftarrow rank_0(B_v, j)$
  $i_r \leftarrow i - i_l$, $j_r \leftarrow j - j_r$
  $n_l \leftarrow j_l - i_l + 1$
  if $k \leq n_l$ **then**
    mrqq$(v_l, i_l, j_l, k, \min(n_l, k'))$
  end if
  if $k' > n_l$ **then**
    mrqq$(v_r, i_r, j_r, \max(k - n_l, 1), k')$
  end if
end if

frqq1(v, i, j, k)

$m_0 \leftarrow \infty$
$e_1 \leftarrow v$
$i^* \leftarrow i$
$j^* \leftarrow j$
return frrq′$(v, i, j, k, 1)$

frqq(v, k)

$\delta \leftarrow$ height of v
while $k > m_{\delta-1}$ **do**
  $\delta \leftarrow \delta - 1$
end while

return frqq′$(v^\delta, i^*, j^*, k, \delta)$

frqq′(v, i, j, k, δ)

if v is a leaf **then**
  output $(label(v), j - i + 1)$
else
  $v^\delta \leftarrow v$
  $i_l \leftarrow rank_0(B_v, i - 1) + 1$
  $j_l \leftarrow rank_0(B_v, j)$
  $i_r \leftarrow i - i_l$, $j_r \leftarrow j - j_r$
  $n_l \leftarrow j_l - i_l + 1$
  if $k \leq n_l$ **then**
    $m_\delta \leftarrow e_\delta + n_l$
    $e_{\delta+1} \leftarrow e_\delta$
    return frqq′$(v_l, i_l, j_l, k, \delta + 1)$
  else
    $m_\delta \leftarrow m_{\delta-1}$
    $e_{\delta+1} \leftarrow e_\delta + n_l$
    return frrq′$(v_r, i_r, j_r, k, \delta + 1)$
  end if
end if

Corollary 12. *Let C be a collection over ν distinct words, formed by m documents adding up to N words. Let df_t be the number of distinct documents where term t appears and $n = \sum df_t$. Call F_t the virtual inverted list where the document identifiers are sorted increasingly. Then there exists a data structure using $n \log m + O(n)$ bits carrying out the following operations: (a) extract r values in $F_t[k, k']$, at increasing positions, in time $O\left(\log m + r \log \frac{k'-k+1}{r}\right)$; (b) extract r values from $[d, d']$ in F_t, with increasing lower bounds, in time $O\left(\log m + r \log \frac{d'-d+1}{r}\right)$; (c) intersect $|q|$ lists F_t in time $O\left(\alpha|q| \log \frac{m}{\alpha}\right)$, where α is the alternation complexity of the intersection problem.*

6.1.3. Other operations of interest

If the range of terms $[t, t']$ represents the derivatives of a single stemmed root, we might wish to act as if we had a single list $F_{t,t'}$ containing all the documents where they occur. Indeed, if we apply our previous algorithm to obtain $F_t[k]$ from $L[s_t, s_{t+1} - 1]$, on the range $L[s_t, s_{t'+1} - 1]$, we obtain precisely $F_{t,t'}[k]$, if we understand that a document d may repeat several times in the list if different terms in $[t, t']$ appear in d. Still we can obtain the list of *docc* distinct documents for a range of terms $[t, t']$ with exactly the same method as for the D array, described at the beginning of Section 5, in time $O\left(docc \log \frac{m}{docc}\right)$.

Furthermore, the algorithms to find the first k such that $F_t[k] \geq d$, can be applied verbatim to obtain the same result for $F_{t,t'}[k] \geq d$. All the variants of these queries are directly supported as well. Our intersection algorithm can also be applied verbatim in order to intersect stemmed terms.

Additionally, note that we can compute some *summarization* information. More precisely, we can obtain the *local vocabulary* of a document d, that is, the set of different terms that appear in d. By executing $rank_1(S, select_d(L, i))$ for successive i values, we obtain all the local vocabulary, in order, and in time $O(\log m)$ per term. This allows, for example, merging the vocabularies of different documents. We can also search for a particular term in a particular document via two *rank* operations on L: $rank_d(L, s_{t+1} - 1) - rank_d(L, s_t - 1)$; then the corresponding position can be obtained by $select_d(L, 1 + rank_d(L, s_t - 1))$.

Finally, the data structure provides some basic support for temporal and hierarchical documents, by restricting the inverted lists F_t to a range of document values $[d_{min}, d_{max}]$ (recall Section 5.1). A simple way to proceed is to first carry out a query *range_next_value*$(L, s_t, s_{t+1} - 1, d_{min})$ with **rnv** (see Algorithm 3), which will also give us the rank p of the first document $\geq d$. Then any subsequent range quantile query on F_t must increase its argument by $p - 1$, and discard answers larger than d_{max}. Finally, the restriction to retrievable units works exactly the same as in Section 5.2.

6.2. Ranked Retrieval

We focus now on the operations of interest for ranked retrieval, which are also simulated in $O(\log m)$ time or less. In this case we also need to maintain the *tf* values. We store them in differential and run-length compressed form, in a separate sequence, so as to permit powerful operations.

More precisely, we mark the $v_t \leq df_t$ different $tf_{d,t}$ values of each list in a bitmap $T_t[1, N_t]$, where $N_t = \max_d tf_{d,t}$, and the v_t points in $L_t[1, df_t]$ where value $tf_{d,t}$ changes, in a bitmap $R_t[1, df_t]$. Thus one can obtain $tf_{L_t[i],t} = select_1(T_t, v_t - rank_1(R_t, i) + 1)$. We use Okanohara and Sadakane's representation [73][8] for T_t and the "fully indexable dictionary" [29] for R_t. This gives total space $v_t \log \frac{N_t}{v_t} + O(v_t) + v_t \log \frac{df_t}{v_t} + o(df_t)$ bits and retain constant time access to *tf* values. This space is similar to that needed to represent, in a traditional *tf*-sorted index, each new $tf_{d,t}$ value and the number of entries that share it. Overall our extra structures take at most $n \log \frac{Nv}{n} + O(n)$ bits.

6.2.1. Direct access and Persin's algorithm

The L_t lists used for ranked retrieval are directly concatenated in L, so $L_t[i]$ is obtained by accessing symbol $L[s_t + i - 1]$ using the wavelet tree. Recall that the term frequencies *tf* are available in constant time. A range $L_t[i, i']$ is obtained in time $O\left(\log m + (i' - i + 1) \log \frac{m}{i'-i+1}\right)$ by using query $range\_report(L, s_t + i, s_t + i', [1, m])$ (Algorithm 2), due to Lemma 5.

This algorithm has the problem of retrieving the documents in document order, not in *tf* order as they are in L_t. Note, however, that retrieving the highest-*tf* documents in document order is indeed beneficial for Persin's algorithm [54] (recall Section 4.2), where a problem is how to accumulate results across unordered document sets. More precisely, assume we have the current candidate set as an array ordered by increasing document identifier. Persin's algorithm computes a threshold term frequency f, so that the next list to consider, L_t, should be processed only for *tf* values that are at least p. Instead of traversing L_t by decreasing *tf* values and stopping when these fall below f, we can compute $p = select_1(R_t, v_t - rank_1(T_t, f) + 1) - 1$, so that $L_t[1, p]$ is precisely the prefix where the term frequencies are at least f. Now we extract all the values as explained. As they are obtained in increasing document identifier order, they are easily merged with the current candidate set, in order to accumulate frequencies in common documents.

Corollary 13. *The data structure considered in Corollary 12, joined with a data structure using $n \log \frac{Nv}{n} + O(n)$ bits, can carry out the following operations, where L_t is the virtual list of term t with documents sorted by decreasing $tf_{d,t}$ values: (a) extract the values in $L_t[i, i']$, in increasing document order and with their $tf_{d,t}$ values, in time $O\left(\log m + (i' - i + 1) \log \frac{m}{i'-i+1}\right)$; (b) execute Persin's algorithm in time $O\left(\sum_{t \in q} p_t \log \frac{m}{p_t}\right) = O\left(p \log \frac{m|q|}{p}\right)$, where p_t is the length of the prefix of the list of term t considered by the algorithm, and $\sum_{t \in q} p_t = p$.*

6.2.2. Other operations of interest

Any candidate document d in Persin's algorithm can be directly evaluated, obtaining its $tf_{d,t}$ values, by finding d within L_t for each $t \in q$ (with $rank_d$ and $select_d$ on L, as explained), and its *tf* obtained from R_t and T_t, all in $O(|q| \log m)$ time.

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

Maintaining the *tf* values aligned to the leaf order yields some support for hierarchical queries. Assume a retrievable unit (recall Section 5.2) spans the document range $[dl, dr]$, and thus we wish to compute the total *tf* of t in range $[dl, df]$. Any such range is exactly covered by $O(\log m)$ wavelet tree nodes (Lemma 2). We can descend, projecting the range of L_t in L, until those nodes, and then add up the accumulated *tf* values of those $O(\log m)$ nodes, in overall time $O(\log m)$.

We can also support temporal and hierarchical documents by restricting our accesses in L_t only to documents within a range $[d_{min}, d_{max}]$ (recall Section 5.1). It is sufficient to use $[d_{min}, d_{max}]$ as the last argument when we

[8] We use a constant-time *select* structure [74] for their internal array $H[1, 2v_t]$, which needs $O(v_t)$ bits, and thus the overall structure supports *select* in constant time.

use the *range_report* query that underlies our support for accessing L_t. This automatically yields, for example, Persin's algorithm restricted to a range of documents.

7. Conclusions

The wavelet tree data structure [3] has had an enormous impact on the implementation of space-efficient text databases. In this article we have shown that it has several other under-explored capabilities. We have proposed three new algorithms on wavelet trees that solve fundamental problems, improving upon the state of the art in some aspects. For range intersections we achieve an adaptive complexity that matches the one achieved for sorted ranges. For range quantile and range next value problems, we match or approach the best known time complexities while using less space: basically that needed to represent the sequence $S[1,n]$ plus $O(n)$ extra bits, versus the $O(n \log n)$ extra bits required by previous solutions. The wavelet tree methods also adapt gracefully with the alphabet size. Furthermore, if we use compressed bitmap representations [29] in our wavelet trees, we retain the time complexities and achieve zero-order compression in the representation of S [3], that is, our overall space including the sequence becomes $nH_0(S) + O(n + \sigma)$, where $[1,\sigma]$ is the alphabet of S and $H_0(S)$ is its empirical zero-order entropy.

We have also explored a number of applications of those novel algorithms to two areas of Information Retrieval (IR): document retrieval on general string databases, and inverted indexes. In both cases we obtained support for a number of powerful operations without further increasing the space required to support basic ones.

The algorithms are elegant and simple to implement, so they have the potential to be useful in practice. Future work involves implementing them within an IR framework and evaluating their practical performance. Although we have used some theoretical data structures for handling bitmaps within convenient space bounds, practical variants of *rank*/*select*-capable plain and compressed bitmaps, as well as various wavelet tree implementations, are publicly available[9]. Some preliminary experiments [75] show that an early version of our results [1] do improve significantly in practice upon the previous state of the art on document retrieval for general strings. Our improved versions presented in this article should widen the gap. In the case of inverted indexes we do not expect our representation to be faster for the basic operations, yet it is likely that it requires less space than that of a full-text plus a ranked-retrieval inverted index, and that it is more efficient on sophisticated operations.

Aknowledgements

We thank Jérémy Barbay for his help in understanding the adaptive complexity measures for intersections, and Meg Gagie for righting our grammar.

References

[1] T. Gagie, S. Puglisi, A. Turpin, Range quantile queries: another virtue of wavelet trees, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5721, 2009, pp. 1–6.

[2] G. Navarro, S. J. Puglisi, Dual-sorted inverted lists, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, 2010, pp. 310–322.

[3] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.

[4] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2) (2007) article 20.

[5] P. Ferragina, G. Manzini, Indexing compressed texts, Journal of the ACM 52 (4) (2005) 552–581.

[6] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, An alphabet-friendly FM-index, in: Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 3246, 2004, pp. 150–160.

[7] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nordic Journal of Computing 12 (1) (2005) 40–66.

[8] V. Mäkinen, G. Navarro, Implicit compression boosting with applications to self-indexing, in: Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 4726, 2007, pp. 214–226.

[9] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, SIAM Journal on Computing 17 (3) (1988) 427–462.

[10] V. Mäkinen, G. Navarro, Position-restricted substring searching, in: Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN), LNCS 3887, 2006, pp. 703–714.

[11] P. Bose, M. He, A. Maheshwari, P. Morin, Succinct orthogonal range search structures on a grid with applications to text indexing, in: Proc. 11th International Symposium on Algorithms and Data Structures (WADS), 2009, pp. 98–109.

[12] N. Brisaboa, M. Luaces, G. Navarro, D. Seco, A fun application of compact data structures to indexing geographic data, in: Proc. 5th International Conference on Fun with Algorithms (FUN), LNCS 6099, 2010, pp. 77–88.

[9] See for example `http://libcds.recoded.cl`.

[13] J. Barbay, G. Navarro, Compressed representations of permutations, and applications, in: Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS), 2009, pp. 111–122.
[14] J. Barbay, F. Claude, G. Navarro, Compact rich-functional binary relation representations, in: Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN), LNCS 6034, 2010, pp. 172–185.
[15] G. Navarro, Indexing text using the Ziv-Lempel trie, Journal of Discrete Algorithms 2 (1) (2004) 87–114.
[16] Y.-F. Chien, W.-K. Hon, R. Shah, J. S. Vitter, Geometric Burrows-Wheeler transform: Linking range searching and text indexing, in: Proc. Data Compression Conference (DCC), 2008, pp. 252–261.
[17] F. Claude, G. Navarro, Self-indexed text compression using straight-line programs, in: Proc. 34th International Symposium on Mathematical Foundations of Computer Science (MFCS), LNCS 5734, 2009, pp. 235–246.
[18] N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, in: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4580, 2007, pp. 205–215.
[19] J. Barbay, C. Kenyon, Adaptive intersection and t-threshold problems, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 390–399.
[20] S. Har-Peled, S. Muthukrishnan, Range medians, in: Proc. 16th European Symposium on Algorithms (ESA), LNCS 5193, 2008, pp. 503–514.
[21] S. Stolinski, S. Grabowski, W. Bieniecki, On efficient implementations of median filters in theory and practice, unpublished manuscript (2010).
[22] M. Crochemore, C. S. Iliopoulos, M. Rahman, Finding patterns in given intervals, in: Proc. 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS), LNCS 4708, 2007, pp. 645–656.
[23] O. Keller, T. Kopelowitz, M. Lewenstein, Range non-overlapping indexing and successive list indexing, in: Proc. 10th International Workshop on Algorithms and Data Structures (WADS), LNCS 4619, 2007, pp. 625–636.
[24] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. Rahman, T. Walen, Improved algorithms for the range next value problem and applications, in: Proc. 25th Symposium on Theoretical Aspects of Computer Science (STACS), 2008, pp. 205–216.
[25] W.-K. Hon, R. Shah, S. Thankachan, J. S. Vitter, String retrieval for multi-pattern queries, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, 2010, pp. 55–66.
[26] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.
[27] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proc 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 657–666.
[28] M. Pătraşcu, Succincter, in: Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS), 2008, pp. 305–313.
[29] R. Raman, V. Raman, S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 233–242.
[30] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan, Time bounds for selection, Journal of Computer and System Sciences 7 (4) (1973) 448–461.
[31] D. Krizanc, P. Morin, M. H. M. Smid, Range mode and range median queries on lists and trees, Nordic Journal of Computing 12 (1) (2005) 1–17.
[32] P. Bose, E. Kranakis, P. Morin, Y. Tang, Approximate range mode and range median queries, in: Proc. 22nd Symposium on Theoretical Aspects of Computer Science (STACS), 2005, pp. 377–388.
[33] B. Gfeller, P. Sanders, Towards optimal range medians, in: Proc. 36th International Colloquium on Automata, Languages and Programming, (ICALP), LNCS 5555, 2009, pp. 475–486.
[34] H. Petersen, Improved bounds for range mode and range median queries, in: Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), LNCS 4910, 2008, pp. 418–423.
[35] H. Petersen, S. Grabowski, Range mode and range median queries in constant time and sub-quadratic space, Information Processing Letters 109 (4) (2009) 225–228.
[36] G. S. Brodal, A. G. Jørgensen, Data structures for range median queries, in: Proc. 20th International Symposium on Algorithms and Computation (ISAAC), LNCS 5878, 2009, pp. 822–831.
[37] G. S. Brodal, B. Gfeller, A. G. Jørgensen, P. Sanders, Towards optimal range medians, Theoretical Computer Science 412 (24) (2011) 2588–2601.
[38] A. G. Jørgensen, K. D. Larsen, Range selection and median: Tight cell probe lower bounds and adaptive data structures, in: Proc. 22nd Symposium on Discrete Algorithms (SODA), 2011, pp. 805–813.
[39] V. Mäkinen, G. Navarro, E. Ukkonen, Transposition invariant string matching, Journal of Algorithms 56 (2) (2005) 124–153.
[40] C.-C. Yu, W.-K. Hon, B.-F. Wang, Efficient data structures for the orthogonal range successor problem, in: Proc. 15th International Computing and Combinatorics Conference (COCOON), 2009, pp. 96–105.
[41] H. Gabow, J. Bentley, R. Tarjan, Scaling and related techniques for geometry problems, in: Proc. 16 ACM Symposium on Theory of Computing (STOC), 1984, pp. 135–143.
[42] E. Demaine, I. Munro, Adaptive set intersections, unions, and differences, in: Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 743–752.
[43] J. Barbay, C. Kenyon, Alternation and redundancy analysis of the intersection problem, ACM Transactions on Algorithms 4 (1).
[44] J. Barbay, A. López-Ortiz, T. Lu, A. Salinger, An experimental investigation of set intersection algorithms for text searching, ACM Journal of Experimental Algorithmics 14 (3) (2009) article 7.
[45] G. Navarro, V. Mäkinen, Compressed full text indexes, ACM Computing Surveys 39 (1) (2007) article 2.
[46] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Proc. 1st ESCAPE, LNCS 4614, 2007, pp. 459–470.
[47] K. Sadakane, Succinct data structures for flexible text retrieval systems, Journal of Discrete Algorithms 5 (1) (2007) 12–22.
[48] T. Gagie, G. Navarro, S. J. Puglisi, Colored range queries and document retrieval, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, 2010, pp. 67–81.
[49] R. Baeza-Yates, B. Ribeiro, Modern Information Retrieval, Addison-Wesley, 1999.
[50] I. Witten, A. Moffat, T. Bell, Managing Gigabytes, 2nd Edition, Morgan Kaufmann Publishers, 1999.
[51] R. Baeza-Yates, A. Moffat, G. Navarro, Searching large text collections, in: Handbook of Massive Data Sets, Kluwer Academic Publishers, 2002, pp. 195–244.
[52] J. Zobel, A. Moffat, Inverted files for text search engines, ACM Computing Surveys 38 (2) (2006) art. 6.
[53] J. Zobel, A. Moffat, Exploring the similarity space, ACM SIGIR Forum 32 (1) (1998) 18–34.
[54] M. Persin, J. Zobel, R. Sacks-Davis, Filtered document retrieval with frequency-sorted indexes, Journal of the American Society for

Information Sicence 47 (10) (1996) 749–764.
[55] V. Anh, A. Moffat, Pruned query evaluation using pre-computed impacts, in: Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), 2006, pp. 372–379.
[56] T. Strohman, B. Croft, Efficient document retrieval in main memory, in: Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), 2007, pp. 175–182.
[57] G. Zipf, Human Behaviour and the Principle of Least Effort, Addison-Wesley, 1949.
[58] R. Baeza-Yates, A fast set intersection algorithm for sorted sequences, in: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 3109, 2004, pp. 400–408.
[59] R. Baeza-Yates, A. Salinger, Experimental analysis of a fast intersection algorithm for sorted sequences, in: Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 3772, 2005, pp. 13–24.
[60] J. Barbay, A. López-Ortiz, T. Lu, Faster adaptive set intersections for text searching, in: Proc. 5th International Workshop on Experimental Algorithms (WEA), LNCS 4007, 2006, pp. 146–157.
[61] P. Sanders, F. Transier, Intersection in integer inverted indices, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007.
[62] J. S. Culpepper, A. Moffat, Compact set representation for information retrieval, in: Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 4726, 2007, pp. 137–148.
[63] G. Navarro, E. Moura, M. Neubert, N. Ziviani, R. Baeza-Yates, Adding compression to block addressing inverted indexes, Information Retrieval 3 (1) (2000) 49–77.
[64] D. A. Hull, Stemming algorithms: A case study for detailed evaluation, Journal of the American Society for Information Science 47 (1) (1996) 70–84.
[65] J. Xu, W. B. Croft, Corpus-based stemming using cooccurrence of word variants, ACM Transactions on Information Systems 16 (1) (1998) 61–81.
[66] D. Hiemstra, V. Mihajlović, The simplest evaluation measures for XML information retrieval that could possibly work, in: Proc. INEX Workshop on Element Retrieval Methodology, 2005.
[67] J. Pehcevski, Evaluation of effective XML information retrieval, Ph.D. thesis, RMIT University, Australia (2006).
[68] M. Lalmas, XML Retrieval, Vol. 1, Morgan & Claypool Publishers, 2009.
[69] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, N. Välimäki, Fast in-memory XPath search over compressed text and tree indexes, in: Proc. 26th IEEE International Conference on Data Engineering (ICDE), 2010, pp. 417–428.
[70] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.
[71] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 134–149.
[72] A. Golynski, I. Munro, S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 368–373.
[73] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007.
[74] I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.
[75] J. S. Culpepper, G. Navarro, S. J. Puglisi, A. Turpin, Top-k ranked document search in general text databases, in: Proc. 18th Annual European Symposium on Algorithms (ESA), LNCS 6347, 2010, pp. 194–205 (part II).