# Range Queries Using Huffman Wavelet Trees

Gilad Baruch[1], Shmuel T. Klein[1], and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
gilad.baruch@gmail.com, tomi@cs.biu.ac.il

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@ariel.ac.il

**Abstract.** A Wavelet Tree (WT) is a compact data structure which is used in order to perform various well defined operations directly on the compressed form of a file. Many algorithms that are based on WTs consider balanced binary trees as their shape. However, when non uniform repetitions occur in the underlying data, it may be better to use a Huffman structure, rather than a balanced tree, improving both storage and average processing time. We study distinct range queries and several related problems that may benefit from this change and present theoretical and empirical improvements in time and space complexities.

## 1 Introduction

Given an array $A$ of $n$ elements from an alphabet $\Sigma$, and indices $low$ and $high$, consider the problem named *Distinct Range Queries* that returns the $d$ distinct elements in $A[low, high]$. Here and below, $A[i, j]$ denotes the sub-array of $A$, consisting of the consecutive elements $A[i], A[i+1], \ldots, A[j]$, for $i \leq j$. For example, if $A = \texttt{xxxABRACADABRAyyyyy}$, then $n = 19$, and for range $[4, 14]$, we have $d = 5$ and the sought elements are $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{R}\}$. The goal is to preprocess $A$ and generate a bounded amount of auxiliary information so that given a specific range, the query could be answered efficiently. There are several applications that use such queries. To mention just one, consider the case a list of the most traded stocks for the past $n$ days is given, and one wishes to calculate the set of most traded stocks in some specific period of time, e.g., two months ago.

A trivial solution, without preprocessing, sorts the elements in the given range of size $r$, and computes the set of distinct elements by sequentially rescanning the sorted range in time $r \log r = O(n \log n)$, and without auxiliary storage.

A possible solution with preprocessing and auxiliary storage, would use a sliding window of size $r$, $1 \leq r \leq n$. Given a fixed range of size $r$, it will first compute, in $O(r)$ processing time, the set of distinct elements in the prefix $A[1, r]$ of the array, based on a constant time computation of the corresponding set for $A[1, r-1]$, $r \geq 2$. A table of size $|\Sigma| \lceil \log n \rceil$ bits will store the number of occurrences of each character in $A[1, r]$. The algorithm then slides the window of fixed size $r$, one character at a time, and compares the outgoing character to the incoming one, that is, it compares the first character of the current sliding range to the character just after that range. If these characters are equal, the set of distinct elements does not change. Otherwise, the new set of distinct elements can be determined in constant time by updating the table, for a total of $O(n - r + 1)$ time to process the entire array. The algorithm repeats this process for every $r$, $1 \leq r \leq n$, and stores the answer for every range

Gilad Baruch, Shmuel T. Klein, Dana Shapira: Range Queries Using Huffman Wavelet Trees, pp. 18–29.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0 

$[i, j]$, $1 \le i, j \le n$. Thus, this solution uses $(|\Sigma|n^2 \log n)$ memory space and $O(n^2)$ preprocessing time, but then answers the range query in constant time.

Another line of investigation considers *Wavelet trees*, defined by Grossi et al. [12]. A Wavelet tree (WT) $T$ for an array $A$ of $n$ elements is a full binary tree whose leaves are labeled by the elements of $\Sigma$, and the internal nodes store bitmaps. The bitmap at the root contains $n$ bits, in which the $i^{th}$ bit is set to 0 or 1 depending on whether $A[i]$ is the label of a leaf that is stored in the left or right subtree of $T$. Each internal node $v$ of $T$, is itself the root of a WT $T_v$ for the *subarray* of $A$ consisting only of the labels of the leaves of $T_v$, which are not necessarily consecutive elements of the array $A$. Balanced WTs can be constructed in $O(n \log |\Sigma|)$ time and require $n \log |\Sigma|(1 + o(1))$ bits.

The data structures associated with a WT for general prefix codes require some amount of additional storage (compared to the memory usage of the compressed file itself). Given a text string of length $n$ over an alphabet $\Sigma$, the space required by Grossi et al.'s implementation can be bounded by $nH_h + O(\frac{n \log \log n}{\log_{|\Sigma|} n})$ bits, for all $h \ge 0$, where $H_h$ denotes the $h$th-order empirical entropy of the text, which is at most $\log |\Sigma|$; processing time is just $O(m \log |\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length $m$. Multiary WTs replace the bitmaps by sequences over sublogarithmic sized alphabets in order to reduce the $O(\log |\Sigma|)$ height of binary WTs, and obtain the same space as the binary ones, but their times are reduced by an $O(\log \log n)$ factor. If the alphabet $\Sigma$ is small enough, say $|\Sigma| = O(\text{polylog}(n))$, the tree height is a constant and so are the query times.

Many algorithms that are based on WTs consider balanced binary trees as their shape, that is, during the construction of each of the subtrees of the WT, the corresponding set of elements of $\Sigma$ is split, at each stage, into two subsets of equal size, $\pm 1$. However, when repetitions occur in the underlying data, it may be better using a Huffman structure, rather than a balanced tree, as suggested already in [16], improving both storage and average processing time. The contribution of this paper is to formalize this approach and conduct some empirical studies supporting its efficiency. Let $H$ denote the zeroth order entropy of the given elements in $A$, and $d$ the number of distinct elements in the range of the query, we show how to answer distinct range queries using a Huffman based WT, in $O(d(H + 1))$ processing time on average, and only $O(n(H + 1))$ auxiliary storage.

The rest of the paper is organized as follows. Section 2 reports on previous research. Section 3 presents the algorithm for solving the *distinct range query* problem by means of a Huffman WT. Section 4 considers other problems that can benefit from the use of Huffman WTs rather than balanced ones. Section 5 brings preliminary empirical evidence that using Huffman WTs may enhance processing time as well as storage usage as compared to the corresponding balanced WT.

## 2 Previous Work

Previous work has focused mainly on finding the $k^{th}$ element in a given range, also named *Range Selection Queries*, and specifically on *Range Median Queries* in which $k$ is equal to $\frac{n}{2}$. Krizanc et al. [14] presented the first preprocessing solution for *mode* and *median* queries, the mode of a given set being its most frequent element. In addition to mode and median range queries on lists, they also considered the general settings of *path queries*, in which the input is given as a node labeled tree, and the query consists of two nodes. For the mode query they suggest an $O(n^{\epsilon} \log n)$ time and $O(n^{2-2\epsilon})$ space algorithm, where $0 < \epsilon < \frac{1}{2}$, while the median query could be answered in constant time using an $O(\frac{n^2 \log \log n}{\log n})$ space algorithm. For the median query, Petersen [18] improves the space to $O(\frac{n^2 \log^{(k)} n}{\log n})$, still answering the query in constant time, where $k$ is a constant and $\log^{(k)}$ is the $k$ times iterated logarithm. Unlike the near quadratic space of Petersen, the best known linear space solution is due to Chan et al. [4] and requires $O(\sqrt{\frac{n}{\log n}})$ query time.

*Range Least Frequent Element Queries* on arrays were studied by Chan et al. in [5], and improved by Durocher et al. in [6]. Durocher et al. [7] study the *Range Majority Query* problem, which asks to report the mode in $A[low, high]$ only if the mode occurs more than half of the times in the range. Given a real number $0 < \tau \leq 1$, Navarro et al. [17] consider a generalization where any element occurring a fraction of times larger than $\tau$ in $A[low, high]$ can be reported. Thus a majority corresponds to $\tau = \frac{1}{2}$. They prove a lower bound of $\Omega(n\lceil\log(\frac{1}{\tau})\rceil)$ bits, without storing $A$, for any data structure supporting $\tau$ majorities within any range, and present a data structure that returns a single position of each $\tau$-majority, and obtains this space lower bound, in running time $O(\frac{1}{\tau} \log \log_w(\frac{1}{\tau}) \log n)$, on a RAM machine with word size $w$. As extension, Huffman WTs can also be used when considering Range Least Frequent Element Queries and Range Majority Queries, yielding an improvement as can be found in Table 1.

A problem related to the range selection queries is *Range Rank Queries* (or range dominance queries), where, given indices $i$, $j$ and a value $e$, the goal is to return the number of elements from $A[i, j]$ that are less than or equal to $e$ (dominated by $e$). Brodal et al. [3] designed a static linear space data structure that supports both range selection and range rank queries in $O(\log n/ \log \log n)$ time. In [2] the authors suggest a linear space and $O(n \log n)$ preprocessing time solution to the median range queries problem, with the same time complexity per query. Their data structure sorts the input elements and places them in the leaves of a balanced binary search tree. Consider a search for the $k^{th}$ smallest element in $A[i, j]$. If the left subtree of the root contains $k$ or more elements from $A[i, j]$ then it contains the $k^{th}$ smallest element from $A[i, j]$. If not, the sought element is in the right subtree. Each node of the tree stores the prefix sum such that the number of elements from $A[1, j]$ contained in the left subtree can be determined for any $j$. The space is then reduced to $O(n)$ using $\mathsf{rank}$ and $\mathsf{select}$ data structures defined as:

$\mathsf{rank}_{\sigma}(A, i)$ – returns the **number** of occurrences of $\sigma \in \Sigma$ in $A$ up to and including position $i$;

$\mathsf{select}_{\sigma}(A, i)$ – returns the **position** of the $i$th occurrence of $\sigma \in \Sigma$ in $A$.

Given a range $[low, high]$ and an element $x$, the *Range Counting Query* problem is counting the number of occurrences of $x$ in $A[low, high]$. Krizanc et al. [14] use a series of sorted arrays, one for each element in $\Sigma$. The array for element $x$, denoted by $A_x$, contains the indices $1 \leq i \leq n$ such that $a_i = x$ in sorted order. Given a range $[low, high]$ and an element $x$, binary search is applied on $A_x$ in order to find the indices $\ell$ and $h$ of $low$ and $high$, respectively. The number of occurrences of $x$ is then $h - \ell + 1$. This solution uses $O(n)$ words of storage and $O(\log n)$ processing time. It should be noted that a space of $O(n)$ words is equal to $O(n \log n)$ bits in the word-RAM model, in which a word size is $\Theta(\log n)$. By applying the predecessor data structure of van Emde Boas [8] instead of binary search, Range Counting Queries over the integer alphabet $[1..u]$ can be answered in $O(\log \log u)$ time using $O(u \log u)$ bits. If the length of the string is much smaller than the alphabet size, i.e., if $n \ll u$, then *Y-fast tries* can be used, with $O(\log \log n)$ time using $O(n \log n)$ bits [20].

Muthukrishnan [15] solved the *Distinct range query* problem, also called the *colored range listing* problem, as part of a solution to the *document listing problem* for listing all distinct documents containing a given pattern. His solution is based on defining an additional array $C$, so that $C[k]$ is the largest value $i < k$ such that $A[i] = A[k]$, or 0 if there is no such $i$. $A[k]$ is then the first occurrence of this element in the range $A[i, j]$ if and only if $C[k] < i$. Thus, if the minimum value in $C[i, j]$ is $C[k]$, the element $A[k]$ is reported as a new element in the range if and only if $C[k] < i$. All other distinct elements in the (original) range are reported by recursively applying the same method on the sub-arrays $C[i, k-1]$ and $C[k+1, j]$. The constant time *Range Minimum Queries (RMQ)* data structure, due to Gabow et al. [9] is used for a total of $O(d)$ time and $O(n \log n)$ space, where $d$ is the number of distinct elements.

Välimäki and Mäkinen [19] reduce the space of Muthukrishnan's data structure by means of a multiary wavelet tree, using $O(n \log |\Sigma|)$ bits and $O(d \frac{\log(|\Sigma|}{\log \log n})$ time. Their idea is based on the $\mathsf{rank}$ and $\mathsf{select}$ data structures used in the internal nodes of the mulitary wavelet tree. They give an alternative way for computing the value $C[k]$ used in Muthukrishnan's solution as $C[k] = \mathsf{select}_{A[k]}(A, \mathsf{rank}_{A[k]}(A, k) - 1)$.

Gagie et al. [10] eliminate the use of RMQ's and suggest a binary WT for solving range quantile queries and distinct range queries, using the same size of auxiliary space and $O(d \log |\Sigma|)$ processing time. In particular, range counting queries are solved by them in $O(\log |\Sigma|)$ time. Unlike this solution which is based on a binary balanced Wavelet tree, we examine the use of the Huffman tree that corresponds to the number of occurrences of the items in $A$ as the structure of the WT.

Concentrating on the shape of the WT was recently done by Klein and Shapira [13] and Baruch et al. [1], where a pruning strategy was applied to the WTs in order to reduce the overhead of the additional storage used by the data structures for processing the stored bitmaps. Moreover, the average path lengths corresponding to the codewords was also decreased, thus implying a reduction of the average random access time.

Table 1 summarizes the results. The variable $w = \Omega(\log n)$ stands for the word size.

## 3 Distinct Range Queries

Recall that the binary tree $T_C$ corresponding to a prefix code $C$ is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node $v$ is associated with the bit string obtained by concatenating the labels on the edges on the path from the root to $v$; finally, $T_C$ is defined as the binary tree for which the set of bit strings associated with its leaves is the code $C$.

WTs can be defined for a text array over any prefix code and the tree structure is inherited from the tree usually associated with the code. Considering the WT as

| | *Processing Time* | *Space (bits)* |
|---|---|---|
| **Distinct Range Queries** | | |
| Välimäki et al. [19] | $O(d\frac{\log(\|\Sigma\|)}{\log\log n})$ | $O(n\log\|\Sigma\|)$ |
| Gagie et al. [10] | $O(d\log\|\Sigma\|)$ | $O(n\log\|\Sigma\|)$ |
| Section 3 | $O(d(H+1))$ average time | $O(n(H+1))$ |
| **Range Counting Queries** | | |
| Krizanc et al. [14] | $O(\log\log n)$ | $O(n\log n)$ |
| Gagie et al. [10] | $O(\log\|\Sigma\|)$ | $O(n\log\|\Sigma\|)$ |
| Section 4 | $O(H+1)$ average time | $O(n(H+1))$ |
| **Range Mode Queries** | | |
| Petersen [18] | $O(1)$ | $O(\frac{n^2\log^{(k)} n}{\log n})$ |
| Chan et al. [4] | $O(\sqrt{n/\log n})$ | $O(n\log n)$ |
| Section 4 | $O(d(H+1))$ average time | $O(n(H+1))$ |
| **Range Least Frequent Element Queries** | | |
| Chan et al. [5] | $O(\sqrt{n})$ | $O(n\log n)$ |
| Durocher et al. [6] | $O(\sqrt{n/w})$ | $O(n\log n)$ |
| Section 4 | $O(d(H+1))$ average time | $O(n(H+1))$ |
| **Range Majority Queries** | | |
| Chan et al. [7] | $O(1)$ | $O(n\log n)$ |
| Section 4 | $O(H+1)$ average time | $O(n(H+1))$ |

**Table 1.** Time and space complexities for range queries.

associated with the prefix code, rather than with the text array itself, yields the following equivalent definition, as alternative to the one given in the introduction. The root holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in the encoded text. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively with 1. This process is repeated similarly on the next levels: the grand-children of the root hold the bitmaps obtained by concatenating the *third* bit of the sequence of codewords starting, respectively, with 00, 01, 10 or 11, if they exist at all, etc.

The bitmaps in the nodes of the WT can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node once the size $n$ of the text is given in the header of the file.

Let the weights $\{w_1, w_2, \ldots, w_k\}$ be the number of occurrences of the individual characters in $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, respectively. It is well known that Huffman's encoding is optimal, and assigns codeword lengths $\{\ell_1, \ell_2, \ldots, \ell_k\}$ so that $W = \sum_{i=1}^{k} w_i\ell_i$ is minimal. Let us assume that $\sigma_1, \ldots, \sigma_k \in \Sigma$ occur $\{w'_1, w'_2, \ldots, w'_k\}$ times in $A[low, high]$ ($w'_i = 0$ for characters that do not occur in the given range). A Huffman based WT requires only $O(W)$ space and $O(\sum_{i=1}^{k} w'_i\ell_i)$ processing time. Notice the following:

1. There are $d$ non zero terms in $\sum_{i=1}^{k} w'_i\ell_i$;
2. $W \leq n\log|\Sigma|$;
3. $\sum_{i=1}^{k} w'_i\ell_i \leq d\log|\Sigma|$;

The last two points indicate that Huffman based WTs may improve both space and processing time of the WTs of Gagie et al. [10].

The algorithm for extracting the distinct elements in the range $[low, high]$ of an array $A$ by means of a Huffman WT rooted by $v_{root}$ is given in Algorithm 1, using the function call distinct($v_{root}$, $low$, $high$). $B_v$ denotes the bitmap belonging to vertex $v$ of the Wavelet tree. The variables $num_0$ and $num_1$ are assigned the number of 0s and 1s in the given range in lines 3.1 and 3.2, respectively, by subtracting the number of 0s/1s up to the beginning of the range from the number of 0s/1s up to the end of the range. Branching left or right depends on whether there are 0s or 1s in the current range. If $num_0$ is greater than 0, the process continues on the left subtree, and if $num_1$ is greater than 0, it continues (also) on the right subtree. Computing the new range in the following bitmap is done by applying the rank operation on both ends of the current range. As a side effect, when processing a leaf $v$, the number of occurrences of the corresponding element is also computed, based on the number of 0s or 1s in the parent node of $v$.

---

Distinct($v, low, high$)
1 $num \leftarrow high - low + 1$
2 if $v$ is a leaf
2.1 output element corresponding to $v$ and its frequency $num$
2.2 return
3 else
3.1 $num_0 \leftarrow rank_0(B_v, high) - rank_0(B_v, low - 1)$
3.2 $num_1 \leftarrow num - num_0$
3.3 if $num_0 > 0$
3.3.1 Distinct $(left(v), rank_0(B_v, low - 1) + 1, rank_0(B_v, high))$
3.4 if $num_1 > 0$
3.4.1 Distinct $(right(v), rank_1(B_v, low - 1) + 1, rank_1(B_v, high))$

---

**Algorithm 1**. Extracting the distinct elements of $A[low, high]$ from a Wavelet tree.

Consider for example the tree in Figure 1, which represents a Wavelet tree for some array $A$. Assume that the substring of $A$ from position 4 to position 14 contains `abracadabra` and consider the query with $low = 4$ and $high = 14$. Note that the leaves are sorted from left to right according to the number of their occurrences in the entire array $A$. At the beginning we are looking for the leftmost leaf corresponding to an element that occurs in the given range. There are 0s in the given range in the bitmap stored in the root, meaning that the range contains elements corresponding to the left subtree, thus $v$ is assigned the left child of the root. The new range is computed to be from 3 to 7, according to the number of 0s $num_0 = 5$ in the range $[4, 14]$ in the bitmap of the root, and the number of 0s preceding the range, which is 2 in this example. As all bits in the range $[3, 7]$ in the bitmap of the left child of the root are 1s, the element `e` does not occur in the range, and the left subtree can be skipped, going directly to the right child of the left child of the root. The new range is computed to be [2,6], and as the corresponding bitmap is all 0s, the algorithm continues with the left child, and character `a` with frequency 5 is reported. This process continues until all elements of the range are reported, skipping subtrees that do not contain leaves with labels in the range.

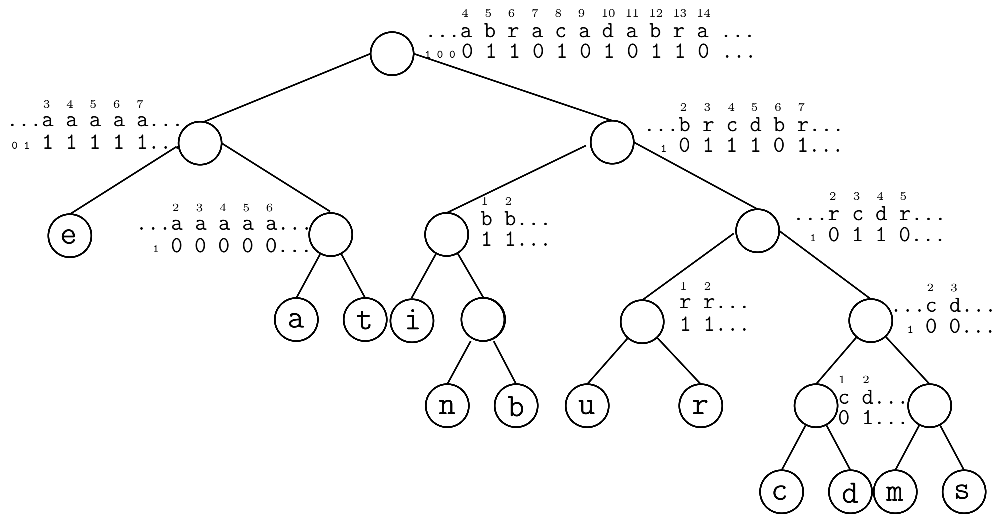

**Figure 1.** A Range Query on the Wavelet tree induced by the canonical Huffman tree corresponding to the frequencies {20, 9, 9, 9, 5, 5, 5, 5, 2, 2, 2, 2} of {e, a, t, i, n, b, u, r, c, d, m, s}, respectively.

As mentioned in Section 2, the algorithm of Gagie et al. [10] for Distinct Range Queries, runs in $O(d \log |\Sigma|)$ time, and uses $O(n \log |\Sigma|)$ space. It is important to note that given a specific range, the running time $O(d(H + 1))$ of Algorithm 1, could be longer than the $O(d \log |\Sigma|)$, suggested by Gagie. This happens when the distribution of the characters within the given range significantly deviates from this distribution in the entire text. However, the improvement of the average running time is based on the assumption that there is no such discrepancy between the partial range and one spanning the entire text, resulting in a reduction in running time. Nevertheless, the storage of the entire WT requires generally less space than a balanced WT, and only if the distribution of the character frequencies is close to uniform, both will produce an $O(n \log |\Sigma|)$ space data structure.

Another interesting bound can be derived on the worst case running time of Algorithm 1. The Range Distinct Elements algorithm runs on the Huffman tree, possibly skipping several subtrees in case the relevant bitmap contains only 0s or only 1s. In the worst case, when all characters of $\Sigma$ appear in the given range, the entire Huffman tree is processed. Thus, the running time is bounded by the total number of nodes in the Huffman tree, which is $O(|\Sigma|)$, and may be independent of $n$.

The results can be summarized in the following theorem:

Theorem 1: *There exists a data structure of size $O(W)$ bits which can be built in $O(W)$ time, that answers distinct range queries on $A[i, j]$ for $1 \leq i \leq j \leq n$ in $O(d(H + 1))$ average time.*

## 4 Range Mode, Range Least, Range Counting, and Range Majority Queries

The operation $rank_\sigma(A, i)$ is defined as computing the number of occurrences of $\sigma$ in $A$ up to position $i$. This can be adapted quite easily in order to compute the number of occurrences of $\sigma$ in a given range $[low, high]$ by simply calculating $rank_\sigma(A, high) - rank_\sigma(A, low - 1)$. A WT can be used to compute $rank_\sigma(A, i)$ in time proportional to the length of the path starting at the root and ending at the leaf corresponding to $\sigma$. Using a Huffman based WT, this time is $O(H+1)$ on average, where the WT occupies $O(W)$ bits. Though the $O(\log\log n)$ time algorithm of Krizanc et al. [14] for Range Counting Queries is usually faster than the $O(H+1)$ average time of our suggested algorithm, their $O(n\log n)$ memory space is larger than the $O(W)$ space we use.

Given a range $[low, high]$, the *Range Mode Query* reports the most frequent element in $A[low, high]$, or one of them if there are several. As mentioned above, Chan et al. [4] present an $O(\sqrt{\frac{n}{\log n}})$ query time algorithm for this problem, using $O(n\log n)$ bits for storage. We note that the problem of finding the mode of a given range can also be solved by using a balanced Wavelet tree, by computing Range Counting Queries for each distinct element in the range. This solution suggests a method requiring $O(d\log|\Sigma|)$ processing time and $O(n\log|\Sigma|)$ space. By applying Huffman shaped WTs, the time is reduced to $O(d(H+1))$ and to only $O(W)$ space. In more details, the algorithm presented for Distinct Range Queries can also be used to solve Range Mode Queries, no matter whether the underlying shape of the Wavelet tree is balanced or Huffman. As described above, as a side effect of this algorithm, the number of occurrences of each element is also computed each time a leaf is processed. We can therefore answer Range Mode Queries in time $O(d(H+1))$, using a Huffman shaped WT, and in both cases the times are bounded by $O(|\Sigma|)$.

Note that if an unbounded alphabet $\Sigma$ is assumed, the traditional WT and the Huffman shaped WT algorithms are worse than the $O(\sqrt{n/\log n})$ of Chan et al., but reduce the processing time in the case of a finite alphabet. However, the WTs algorithms may still be useful when the number of distinct elements $d$ in the given range is small, e.g., when $d = \log n$, which can happen even in the case of an unbounded alphabet. Moreover, in the bounded and unbounded cases, using WTs needs only $O(n\log|\Sigma|)$ and $O(W)$ space for traditional and Huffman shaped Wavelet trees, respectively, as compared to $O(n\log n)$ of Chan et al.. The same discussion applies also to a symmetric problem named *Range Least Frequent Element*.

The algorithm for solving Range Majority Queries in a given range $[low, high]$ of an array $A$, by means of a Huffman WT rooted at $v_{root}$, is given in Algorithm 2, using the function call $\mathsf{majority}(v_{root}, low, high, (high-low+1)/2)$. As the majority depends on the number of elements in the original range, the last argument of the function giving the majority bound is passed through all recursive calls. The variables $B_v$, $num_0$ and $num_1$ are the same as in Algorithm 1. Branching left or right depends on whether the number of 0s or 1s is greater than the required target value $m = (high-low+1)/2$ in the current range. This time, at most one of the subtrees will be processed. If $num_0$ is greater than $m$, the process continues on the left subtree, otherwise, if $num_1$ is greater than $m$, it continues on the right subtree. If neither of $num_0$ and $num_1$ are greater than $m$, there is no majority element in $A$, and the process terminates after reporting so. This algorithm runs in $H+1$ time on average, unlike the constant time of Durocher et al. [7]. However, it only uses $O(W) \le O(n(H+1))$ space rather than $O(n\log n)$.

Gagie et al. [10] use a balanced WT for finding the $k^{th}$ element in time $O(\log|\Sigma|)$ and $O(n\log n)$ space. In our paradigm the elements are sorted by frequencies in the *entire* array, thus the problem is now finding the $k^{th}$ frequent element in a given range. In fact, the same algorithm can be used on a Huffman shaped WT, and produces an

```
majority(v, low, high, m)
num ← high − low + 1
if v is a leaf
    output element corresponding to v
    return
else
    num_0 ← rank_0(B_v, high) − rank_0(B_v, low − 1)
    num_1 ← num − num_0
    if num_0 ≥ m
        Majority (left(v), rank_0(B_v, low − 1) + 1, rank_0(B_v, high), m)
    else if num_1 ≥ m
        Majority (right(v), rank_1(B_v, low − 1) + 1, rank_1(B_v, high), m)
    else
        output "no Majority in Range"
        return
```

**Algorithm 2**. Majority Query on $A[low, high]$.

average running time of $O(H+1)$ and only $O(W) \leq O(n(H+1))$ space. The algorithm is similar to Algorithm 2.

## 5 Experimental Results

For our preliminary experiments we considered two different files of different languages and alphabet sizes. The Bible (King James version) in English, *ebib*, in which the text was stripped of all punctuation signs, and the French version of the European Union's JOC corpus, *ftxt*, which is a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project. Our implementation used the *Succinct Data Structure Library* [11], which is an open-source library implementing succinct data structures efficiently in C++. All experiments were conducted on a machine running 64 bit Linux Ubuntu with an Intel Core i7-4720 at 2.60 GHz processor, 6144K L3 cache size of the CPU, and 4 GB of main memory.

The files were encoded as a sequence of characters as well as a sequence of *words* (a maximal sequence of non whitespace characters), producing two different alphabets, a small and a large one. Table 2 presents some information on the data files involved. The second column presents the original file sizes in MB. The third and fourth columns give the number of elements in the character alphabet (chars) and the word alphabet (words), respectively. The size of the word alphabet is given in thousands of (different) words. The number of words in the file, including repetitions, is given in the fifth column, in millions.

Our first experiment compares the processing times for the distinct elements range query problem, using balanced and Huffman WTs. The range sizes were chosen as a series of increasing powers of 2, starting with 1 and up to the size of 256. For each of the test files and range sizes, the range query was run 1000 times, with randomly chosen starting points. The displayed plots are the averages over these runs. Figures 2 and 3 present the processing times for our dataset for the alphabet of characters and

| File | size (MB) | chars | words (in thousands) | Words in text (in millions) |
|---|---|---|---|---|
| *ebib* | 3.5 | 53 | 11 | 0.6 |
| *ftxt* | 7.6 | 132 | 75 | 1.2 |

**Table 2.** Information about the used datasets

words, respectively. The plots are given on a log scale, showing the processing time, in microseconds, as function of the range size, measured in number of characters.

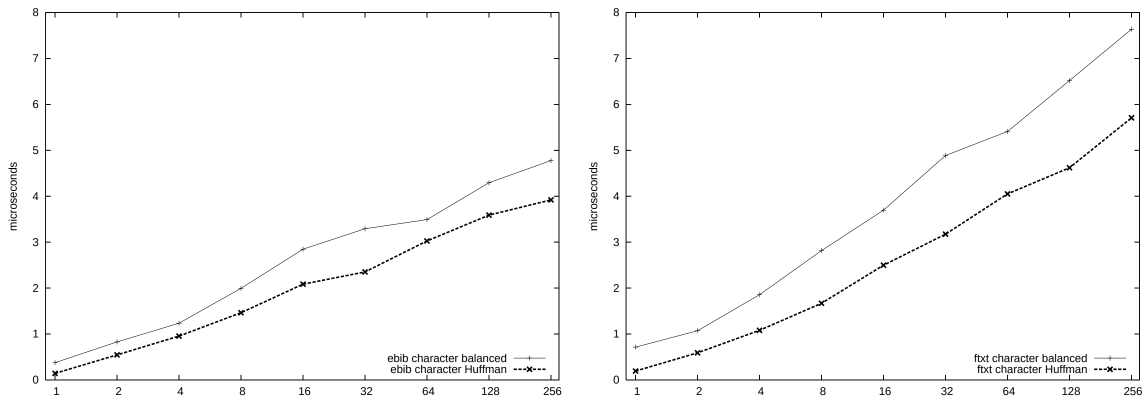


**Figure 2.** Processing time as function of the range size with character alphabet.

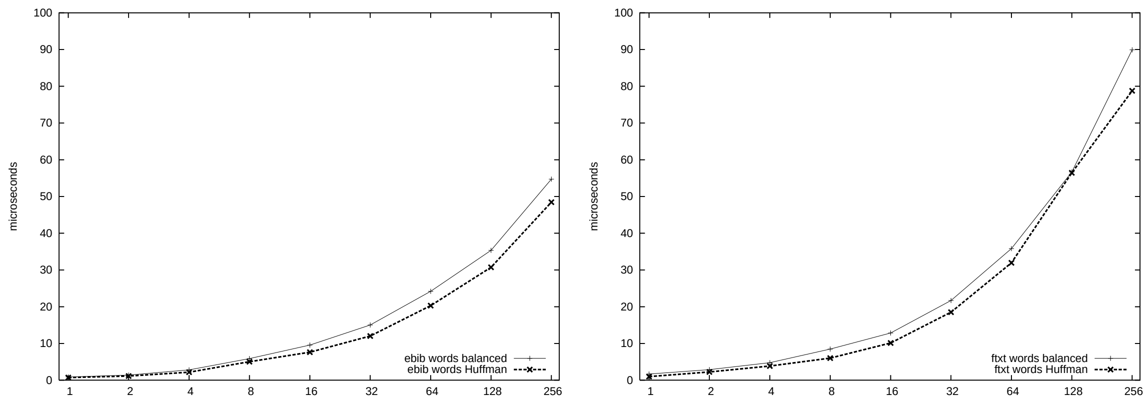


**Figure 3.** Processing time as function of the range size with word alphabet.

As can be seen, processing the Huffman WT is consistently faster than processing the balanced one, for ranges up to 256. The ratio of the improvement of Huffman over balanced WTs reduces as the ranges become longer. This can be explained by the fact that the probability that longer ranges include also less frequent characters becomes higher, requiring longer processing times for the deeper leaves. Thus, there are cases in which for a given range the running time of the balanced WT can be faster than the Huffman one, and the advantage of the Huffman structure vanishes.

In the following table we present the storage usage in MBs of balanced versus Huffman WTs on both our datasets, and for the two kinds of alphabets. As expected, the storage of the entire Huffman WT, including the rank and select data structures, requires less space than the corresponding balanced WT, because of the skewed probabilities of the underlying alphabets. Although we expected that the word based WTs will generally save space as compared to that corresponding to characters, this is not the case for the Huffman WTs on *ftxt*. This can be explained by the overhead requirements of the rank and select data structures that are needed for a larger set of nodes.

| File | **Character alphabet** | | **Word alphabet** | |
|---|---|---|---|---|
| | Balanced | Huffman | Balanced | Huffman |
| *ebib* | 3.92 | 2.77 | 2.54 | 2.01 |
| *ftxt* | 11.38 | 7.16 | 9.69 | 8.34 |

**Table 3.** Comparison of storage usage.

## References


1. Baruch, G. and Klein, S. T. and Shapira, D., A Space Efficient Direct Access Data Structure, *The Journal of Discrete Algorithms*, (2017) 26–37.
2. G.S. Brodal, B. Gfeller, A.G. Jørgensen, P. Sanders, Towards Optimal Range Median, *Theoretical Computer Science, Special issue of ICALP'09*, **412**(24) (2011) 2588–2601.
3. G.S. Brodal, A.G. Jørgensen, Data Structures for Range Median Queries, *Algorithms and Computation*, (2009) 822-831.
4. T.M. Chan, S. Durocher, K.G. Larsen, J. Morrison, B.T. Wilkinson, Linear-Space Data Structures for Range Mode Query in Arrays, *Theory Comput. Syst.*, (2014) 719–741.
5. T.M. Chan, S. Durocher, M. Skala, B.T. Wilkinson, Linear-Space Data Structures for Range Minority Query in Arrays. *In Proc. SWAT*, **7357** (2012) 295–306.
6. S. Durocher, R. Shah, M. Skala, S.V. Thankachan, Linear-Space Data Structures for Range Frequency Queries on Arrays and Trees, *MFCS*, (2013) 325–336.
7. S. Durocher, M. He, J.I. Munro, P.K. Nicholson, M. Skala, Range majority in constant time and linear space, *Inf. Comput.*, (2013) 169–179.
8. P. van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space, *Information Processing Letters*, **6**(3) (1977) 80–82.
9. H. N. Gabow, J. L. Bentely, R. E. Tarjan Scaling and related techniques for geometry problems, *Proc. STOC*, (1984) 135–143.
10. T. Gagie, S.J. Puglisi, A. Turpin, Range Quantile Queries: Another Virtue of Wavelet Trees, *SPIRE '09 Proceedings of the 16th International Symposium on String Processing and Information Retrieval* (2009) 1–6.
11. S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, *13th International Symposium on Experimental Algorithms, (SEA 2014)*, Copenhagen (2014) 326–337.

12. R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA)* (2003) 841–850.
13. Klein, S. T. and Shapira, D., Random access to Fibonacci codes, *Discrete Applied Mathematics*, **212**, (2016) 115–128.
14. D. Krizanc, P. Morin, M.H.M. Smid, Range mode and range median queries on list and trees, *Nordic Journal of Computing*, **12** (2005) 1–17.
15. S. Muthukrishnan, Efficient algorithms for document retrieval problems. *Proc. SODA'02*, (2002) 657–666.
16. G. Navarro, Wavelet Trees for All, *Journal of Discrete Algorithms*, **25** (2014) 2–20.
17. G. Navarro, S.V. Thankachan, Optimal Encodings for Range Majority Queries, *Algorithmica*, **74** (2016) 1082–1098.
18. H. Petersen, Improved bounds for range mode and range median queries, *Proc. of the 34th Conference on Current Trends in Theory and Practice of Computer Science*, (2008) 418–423.
19. N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, *Proc. CPM*, (2007) 205–215.
20. D. E. Willard Log-logarithmic worst-case range queries are possible in space $\Theta(N)$ *proc. Information Processing Letters*, (1983) **17**(2) 81–84.