Optimal Encodings for Range Majority Queries ☆

Gonzalo Navarro[a,1], Sharma V. Thankachan[b]

[a] *Department of Computer Science, University of Chile, Chile.*
[b] *Georgia Institute of Technology, USA.*

Abstract

We study the problem of designing a data structure that reports the positions of the distinct τ-majorities within any range of an array $A[1, n]$, without storing A. A τ-majority in a range $A[i, j]$, for $0 < \tau < 1$, is an element that occurs more than $\tau(j-i+1)$ times in $A[i, j]$. We show that $\Omega(n\lceil\log(1/\tau)\rceil)$ bits are necessary for any data structure just able to count the number of distinct τ-majorities in any range. Then, we design a structure using $O(n\lceil\log(1/\tau)\rceil)$ bits that returns one position of each τ-majority of $A[i, j]$ in $O((1/\tau)\log\log_w(1/\tau)\log n)$ time, on a RAM machine with word size w (it can output any further position where each τ-majority occurs in $O(1)$ additional time). Finally, we show how to remove a $\log n$ factor from the time by adding $O(n \log\log n)$ bits of space to the structure.

1. Introduction

Given an array $A[1, n]$ of n arbitrary elements, an *array range query* problem asks us to build a data structure over A, such that whenever a range $[i, j]$ with $1 \leq i \leq j \leq n$ arrives as an input, we can efficiently answer queries on the elements in $A[i, j]$ [26]. Many array range queries arise naturally as subproblems of combinatorial problems, and are also of direct interest in data mining applications. Well-known examples are range minimum queries (RMQs, which seek the smallest element in $A[i, j]$) [2], top-k queries (which report the k largest elements in $A[i, j]$) [4], range selection queries (which report the k-th largest element in $A[i, j]$) [7], and colored top-k queries (which report the k largest distinct elements in $A[i, j]$) [17].

An *encoding* for array range queries is a data structure that answers the queries without accessing A. This is useful when the values of A are not of interest themselves, and thus A may be deleted, potentially saving a lot of space. It is also useful when array A does not fit in main memory, so it can be kept in secondary storage while a much smaller encoding can be maintained

☆An early version of this article appeared in *Proc. CPM 2014* [19].

Email addresses: gnavarro@dcc.uchile.cl (Gonzalo Navarro), sharma.thankachan@gmail.com (Sharma V. Thankachan)

[1]Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

in main memory, speeding up queries. In this setting, instead of reporting an element in A, we only report a position in A containing the element. Otherwise, in many cases we would be able to reconstruct A via queries on the encodings, and thus the encodings could not be small (e.g., $A[i]$ would be the only answer to the range query $A[i,i]$ for all the example queries given above). As examples of encodings, RMQs can be solved in constant time using just $2n + o(n)$ bits [12] and, using $O(n \log k)$ bits, top-k queries can be solved in $O(k)$ time [15] and range selection queries in $O(\log k / \log\log n)$ time [18].

Frequency-based array range queries, in particular variants of heavy-hitter-like problems, are very popular in data mining. Queries such as finding the most frequent element in a range (known as the range mode query) are known to be harder than problems like RMQs. For range mode queries, known data structures with constant query time require nearly quadratic space [21]. The best known linear-space solution requires $O(\sqrt{n/\log n})$ query time [5], and conditional lower bounds given in that paper show that a significant improvement is highly unlikely.

Still, efficient solutions exist for some useful variants of the range mode problem. An example are approximate range mode queries, where we are required to output an element whose number of occurrences in $A[i,j]$ is at least $1/(1+\epsilon)$ times the number of occurrences of the mode in $A[i,j]$ [14, 3].

In this paper we focus on a popular variant of range mode queries called *range τ-majority queries*, which ask to report any element that occurs more than $\tau(j-i+1)$ times in $A[i,j]$. A version of the problem useful for encodings can be stated as follows (other variants are possible).

Definition 1. *Given an array $A[1,n]$, a* range τ-majority query *receives a range $[i,j]$ and returns one position in the range where each τ-majority in $A[i,j]$ occurs. A* τ-majority *is any element that occurs more than $\tau(j-i+1)$ times in $A[i,j]$. When $\tau = 1/2$ we simply call it a* majority.

Range majority queries can be answered in constant time by maintaining a linear space (i.e., $O(n)$-word or $O(n \log n)$-bit) data structure [9]. Similarly, range τ-majority queries can be solved in time $O(1/\tau)$ and linear space if τ is fixed at construction time, or $O(n \log\log n)$ space (i.e., $O(n \log n \log\log n)$ bits) if τ is given at query time [1].

In this paper, we focus for the first time on *encodings for range τ-majority queries*. In this scenario, a valid question is how much space is necessary for an encoding that correctly answers such queries (we recall that A itself is not available at query time). We answer that question in Section 3, proving a lower bound for any encoding that solves even a weaker query.

Theorem 1. *Given a real number $0 < \tau < 1$, any encoding able to count the number of range τ-majorities in any range $A[i,j]$ must use $\Omega(n\lceil\log(1/\tau)\rceil)$ bits.*

Since when using $O(n \log n)$ bits we have sufficient space to store $A[1,n]$[2]

[2] Or an equivalent array where each element is replaced by an identifier in $[1,n]$.

Condition	Space (bits)	Query time
$1/\tau = \omega(\text{polylog}\, n)$	$O(n\lceil\log(1/\tau)\rceil)$ *	$O((1/\tau)\log\log_w(1/\tau))$
$1/\tau = \Theta(\text{polylog}\, n)$	$O(n\lceil\log(1/\tau)\rceil)$ *	$O(1/\tau)$ *
$1/\tau = o(\text{polylog}\, n)$	$O(n\lceil\log(1/\tau)\rceil)$ *	$O((1/\tau)\log n)$
$1/\tau = o(\text{polylog}\, n)$	$O(n\log\log n)$	$O(1/\tau)$ *

Table 1: Space-time tradeoffs achieved. We mark the optimal spaces and times with a *.

(and achieve the optimal $O(1/\tau)$ time [1]), encodings for range τ-majorities are asymptotically interesting only for $\log(1/\tau) = o(\log n)$.

In Section 4 we show how range τ-majority queries can be solved using $O((n/\tau)\log\log n)$ bits of space and $O((1/\tau)\log n)$ query time. In Section 5 we reduce the space to the optimal $O(n\lceil\log(1/\tau)\rceil)$ bits and slightly increase the time. After spending this time, the structure can report *any* of the positions of any majority in optimal time (e.g., the leftmost position of each τ-majority in a negligible $O(1/\tau)$ time). In Section 6 we show how to build our structure in $O(n\log n)$ time. All the results hold on the RAM model with word size $w = \Omega(\log n)$ bits.

Theorem 2. *Given a real number $0 < \tau < 1$, there exists an encoding using the optimal $O(n\lceil\log(1/\tau)\rceil)$ bits that answers range τ'-majority queries, for any $\tau \le \tau' < 1$, in time $O((1/\tau)\log\log_w(1/\tau)\log n)$, where $w = \Omega(\log n)$ is the RAM word size in bits. It can report any occ further occurrence positions of the majorities in $O(occ)$ time. The encoding can be built in $O(n\log n)$ time.*

We note that the query time is simply $O((1/\tau)\log n)$ for polylogarithmic values of $1/\tau$. We also note that the time depends on τ, not τ'. In Section 6 we also show how to obtain a query time that is a function of τ', yet using $O(n\lceil\log^2(1/\tau)\rceil)$ bits of space.

Finally, in Section 7 we derive a new variant that may use more space but removes the $\log n$ term from the time complexity.

Theorem 3. *Given a real number $0 < \tau < 1$, there exists an encoding using $O(n\lceil\log(1/\tau)\rceil + n\log\log n)$ bits that answers range τ'-majority queries, for any $\tau \le \tau' < 1$, in time $O((1/\tau)\log\log_w(1/\tau))$, where $w = \Omega(\log n)$ is the RAM word size in bits. It can report any occ further occurrence positions of the majorities in $O(occ)$ time. The encoding can be built in $O(n\log n)$ time.*

By combining the results of Theorems 2 and 3, we obtain the combinations given in Table 1.

2. Related Work

In this section we first cover the state of the art for answering range τ-majority queries. Then, we survey a few results on bitmap representation, and give a new result that will be useful for this paper. Again, all these results hold on the RAM model with word size $w = \Omega(\log n)$ bits.

2.1. Range Majorities

Range τ-majority queries were introduced by Karpinski and Nekrich [16], who presented an $O(n/\tau)$-words structure with $O((1/\tau)(\log\log n)^2)$ query time. Durocher et al. [9] improved their word-space and query time to $O(n\lceil\log(1/\tau)\rceil)$ and $O(1/\tau)$, respectively. Gagie et al. [13] presented another trade-off, where the space is $O(n(H+1))$ *bits* and the query time is $O((1/\tau)\log\log n)$. Here $H \le \lg n$ denotes the empirical entropy of the distribution of elements in A (we use lg to denote the logarithm in base 2). The best current result in general is by Belazzougui et al. [1], where the space is $O(n)$ words and the query time is $O(1/\tau)$. All these results assume that τ is fixed at construction time.

For the case where τ is also a part of the query input, data structures of space (in words) $O(n(H+1))$ and $O(n\log n)$ were proposed by Gagie et al. [13] and Chan et al. [6], respectively. Very recently, Belazzougui et al. [1] brought down the space occupancy to $O(n\log\log\sigma)$ words, where σ is the number of distinct elements in A. The query time is $O(1/\tau)$ in all cases. Belazzougui et al. [1] also presented a compressed solution using $nH + o(n\log\sigma)$ bits, with slightly higher query time. All these solutions include a (sometimes compressed) representation of A, thus they are not encodings. As far as we know, ours is the first encoding for this problem.

For further reading, we recommend the recent survey by Skala [26].

2.2. Bitmap Representations

Let $B[1,m]$ be a bitmap with n 1s. Operation $rank(B,i)$ returns, for any given parameter i, the number of 1s in $B[1,i]$. Operation $select(B,j)$ gives, for any parameter j, the position of the j-th 1 in B. Both operations can be solved in constant time with data structures that use $o(m)$ bits in addition to a plain representation of B [8]. Instead, it is possible to compress B to $n\lg\frac{m}{n} + O(n) + o(m)$ bits while retaining constant time for both operations [24]. This is most useful when $n = o(m)$.

When $n = o(m/\operatorname{polylog} m)$, even the $o(m)$ extra bits of that compressed representation [24] are troublesome, and an Elias-Fano-based [11, 10] compressed representation [20] is useful. It requires $n\lg\frac{m}{n} + O(n)$ bits, solves *select* in $O(1)$ time and *rank* in $O(\log\frac{m}{n})$ time. The representation considers the positions of all the 1s in B, $x_i = select(B,i)$, and encodes the lowest $b = \lceil\lg\frac{m}{n}\rceil$ bits of each x_i in an array $L[1,n]$, $L[i] = x_i \bmod 2^b$. Then it defines a bitmap $H[1,2n]$ that encodes the highest bits of the x_i values: all the bits at positions $i + (x_i \text{ div } 2^b)$ are set in H. Bitmap H is indexed for constant-time *rank* and *select* queries [8]. The space for $L[1,n]$ is $n\lceil\lg\frac{m}{n}\rceil$ and H uses $2n + o(n)$ bits.

Now, $select(B,j) = 2^b(select(H,j) - j) + L[i]$ can be computed in constant time. For $rank(B,i)$, we observe that the h-th 0 in H represents the point where the position $B[2^b h]$ is reached in the process of setting the 1s at positions $i + (x_i \text{ div } 2^b)$, that is, $x_{i-1} < 2^b h \le x_i$. The number of 1s in H up to that position is $rank(B, 2^b h)$. Therefore, if we write $i = 2^b h + l$, then $rank(B, i)$ is between $j_1 = rank(H, select_0(H, h)) + 1$ and $j_2 = rank(H, select_0(H, h + 1))$. Here, operation $select_0(H, h)$ gives the position of the h-th 0 in H, and it is also computed in constant time with a structure using $o(n)$ bits [8]. Now we binary search for l in $L[j_1, j_2]$, which is increasing in that range. The range is of length at most 2^b, so the search takes $O(b) = O(\log \frac{m}{n})$ time. The final position j returned by the search is $rank(B, i)$.

The time can be improved to $O(\log \log_w \frac{m}{n} + \log s)$ on a RAM machine having w-bit words by sampling, for each increasing interval of L of length more than s, one value out of s. Predecessor data structures are built on the samples of each interval, taking at most $O((n/s) \log \frac{m}{n})$ bits. Then we first run a predecessor query on $L[j_1, j_2]$, which takes time $O(\log \log_w \frac{m}{n})$ [22], and finish with an $O(\log s)$-time binary search between the resulting samples.

Lemma 1. *A bitmap $B[1, m]$ with n 1s can be stored in $n \log \frac{m}{n} + O((n/s) \log \frac{m}{n} + n)$ bits, so that select queries take $O(1)$ time and rank queries take $O(\log \log_w \frac{m}{n} + \log s)$, for any s, on a RAM machine of w bits.*

3. Lower Bounds

We derive a lower bound on the minimum size range τ-majority encodings may have, even if we just ask them to count the number of distinct τ-majorities present in any range. The idea is to show that we can encode a certain combinatorial object in the array A, so that the object can be recovered via range τ-majority queries. Therefore, in the worst case, the number of bits needed to solve such queries must be at least the logarithm of the number of distinct combinatorial objects that can be encoded.

Consider a sequence of m permutations on $[k]$. There are $k!^m$ such sequences, thus any encoding for them must use at least $m \lg(k!)$ bits in the worst case. Now consider the following encoding. Array A will have length $n = 4 \cdot k \cdot m$. To encode the i-th permutation, $\pi_i = (x_1\ x_2\ \ldots\ x_k)$, we will write the following chunk to array locations $A[4k(i-1)+1, 4ki]$:

$$1, 2, 3, \ldots, k, \quad -1, -2, -3, \ldots, -2k, \quad x_1, x_2, x_3, \ldots, x_k.$$

We will set $\tau = 1/(2k+2)$ and perform τ-majority queries on parts of A to recover any permutation.

Let us show how to obtain π_i. Let $C[1, 4k] = A[4k(i-1)+1, 4ki]$. Consider an interval of the form

$$C[\ell, 3k+g] = \ell, \ell+1, \ldots, k, -1, -2, \ldots, -2k, x_1, x_2, \ldots, x_g,$$

for $1 \leq \ell, g \leq k$. Note that $x_1, \ldots, x_g$ are the only values that may appear twice in $C[\ell, 3k+g]$, precisely, if they belong to $\{\ell, \ldots, k\}$. Note that elements appearing once in $C[\ell, 3k+g]$ are not τ-majorities, since $1 \leq \tau(3k+g-\ell+1)$ for any values k, ℓ, g. On the other hand, if an element appears twice in $C[\ell, 3k+g]$, then it is a τ-majority, since $2 > \tau(3k+g-\ell+1)$ for any values k, ℓ, g.

With this tool, we can discover x_1 as follows. First, x_1 is for sure a τ-majority in $C[1, 3k+1]$, since it appears twice. Now we query the range $C[2, 3k+1]$, which lacks number 1 compared to $C[1, 3k+1]$. If there is no τ-majority, then $x_1 \notin \{2, \ldots, k\}$, and we conclude that $x_1 = 1$. If there is, then $x_1 \in \{2, \ldots, k\}$ and we query the range $C[3, 3k+1]$. If there is no τ-majority, then $x_1 \notin \{3, \ldots, k\}$ and we conclude that $x_1 = 2$, and so on. The process is continued, if necessary, until reaching the range $C[k, 3k+1]$, where we know that $x_1 = k$.

To look for x_2, we consider ranges of the form $C[\ell, 3k+2]$, with identical reasoning. This time, it is possible that element x_1 is also counted as an answer, but since we already know the value of x_1, we simply subtract 1 from the count in any range $C[\ell, 3k+2]$ with $\ell \leq x_1$. This process continues analogously until we identify x_k.

Example. Consider encoding one permutation $\pi = (3\ 1\ 2)$, of size $k = 3$ (i.e., $m = 1$). Then we set $\tau = 1/8$ and the array $A[1, 12]$ is as follows:

$$1, 2, 3, \quad -1, -2, -3, -4, -5, -6, \quad 3, 1, 2$$

Now we will find x_1 (which is 3, but we do not know it yet). We know that $A[1, 10]$ has one τ-majority, since x_1 must appear twice. Since $A[2, 10]$ still has one τ-majority, we know that $x_1 \in \{2, 3\}$. And since $A[3, 10]$ still has one τ-majority, we know that $x_1 \in \{3\}$, thus we learn $x_1 = 3$.

Now let us find x_2. We know that $A[1, 11]$ has two τ-majorities, since x_1 and x_2 must appear twice. Now, $A[2, 11]$ has only one τ-majority, thus only one of $\{x_1, x_2\}$ is in $\{2, 3\}$. But we know $x_1 = 3$, thus $x_2 \notin \{2, 3\}$, and we learn $x_2 = 1$.

Finally, it can only be that $x_3 = 2$. $\square$

Now, since $n = 4km$ and $\tau = 1/(2k+2)$, we have that any encoding able to answer the above queries requires at least

$$m \lg(k!) \geq m\,(k \lg k - k \lg e + 1) > \frac{n}{4}\left(\lg\left(\frac{1}{2\tau} - 1\right) - \lg e\right)$$

bits[3]. This is $\Omega(n\lceil\log(1/\tau)\rceil)$ as long as $1/\tau$ is bounded from below by a constant larger than $2 + 2e$. Thus, to complete the proof, it is sufficient to show that $\Omega(n)$ is a lower bound for any constant $1/\tau \leq 8$, since $8 > 2 + 2e$.

To show that $\Omega(n)$ bits are necessary for any $\tau \geq 1/8$, consider encoding a bitmap $B[1, m]$ in an array $A[1, 8m]$ so that, if $B[i] = 0$, then $A[8(i-1)+1] = 1$, $A[8(i-1)+2] = 2$, and so on until $A[8i] = 8$. Instead, if $B[i] = 1$, then $A[8(i-1)+1, 8i] = 1$. Then, for any $\tau \geq 1/8$, there is a τ-majority in $A[8(i-1)+1, 8i]$ iff $B[i] = 1$. As there are 2^m possible bitmaps B and our array is of length $n = 8m$, we need at least $m = n/8 = \Omega(n)$ bits for any encoding. Then the proof of Theorem 1 is complete.

[3]Bounding $\lg(k!)$ with integrals one obtains $k \lg(k/e)+1 \leq \lg(k!) \leq (k+1)\lg((k+1)/e)+1$.

4. An $O((n/\tau)\log\log n)$ Bits Encoding for Range τ-Majorities

In this section we obtain an encoding using $O((n/\tau)\log\log n)$ bits and solving τ-majority queries in $O((1/\tau)\log n)$ time. In the next section we improve the space usage. We assume that τ is fixed at construction time. At query time, we will be able to solve any τ'-majority query for any $\tau \le \tau' < 1$.

4.1. The Basic Idea

Consider each distinct symbol x appearing in $A[1,n]$. Now consider the set S_x of all the segments within $[1,n]$ where x is a τ-majority (this includes, in particular, all the segments $[k,k]$ where $A[k]=x$). Segments in S_x may overlap each other. Now let $A_x[1,n]$ be a bitmap such that $A_x[k]=1$ iff position k belongs to some segment in S_x. We define a second bitmap related to x, M_x, so that if $A_x[k]=1$, then $M_x[rank(A_x,k)]=1$ iff $A[k]=x$, where operation $rank$ was defined in Section 2.2.

Example. Let our running example array be $A[1,7] = \langle 1\ 3\ 2\ 3\ 3\ 1\ 1\rangle$, and $\tau = 1/2$. Then we have the segments S_x:

$$\begin{aligned} S_1 &= \{[1,1],[6,6],[7,7],[6,7],[5,7]\},\\ S_2 &= \{[3,3]\},\\ S_3 &= \{[2,2],[4,4],[5,5],[4,5],[2,4],[3,5],[4,6],[2,5],[1,5],[2,6]\}, \end{aligned}$$

and the corresponding bitmaps A_x:

$$A_1 = \langle 1\ 0\ 0\ 0\ 1\ 1\ 1\rangle, \quad A_2 = \langle 0\ 0\ 1\ 0\ 0\ 0\ 0\rangle, \quad A_3 = \langle 1\ 1\ 1\ 1\ 1\ 1\ 0\rangle.$$

Finally, the corresponding bitmaps M_x are:

$$M_1 = \langle 1\ 0\ 1\ 1\rangle, \quad M_2 = \langle 1\rangle, \quad M_3 = \langle 0\ 1\ 0\ 1\ 1\ 0\rangle.$$

$\square$

Then, the following result is not difficult to prove.

Lemma 2. *An element x is a τ'-majority in $A[i,j]$ iff $A_x[k]=1$ for all $i \le k \le j$, and 1 is a τ'-majority in $M_x[rank(A_x,i), rank(A_x,j)]$.*

Proof. If x is a τ'-majority in $A[i,j]$, then it is also a τ-majority. Thus, by definition, $[i,j] \in S_x$, and therefore all the positions $k \in [i,j]$ are set to 1 in A_x. Therefore, the whole segment $A_x[i,j]$ is mapped bijectively to $M_x[rank(A_x,i), rank(A_x,j)]$, which is of the same length. Finally, the number of occurrences of x in $A[i,j]$ is the number of occurrences of 1 in $M_x[rank(A_x,i), rank(A_x,j)]$, which establishes the result.

Conversely, if $A_x[k]=1$ for all $i \le k \le j$, then $A[i,j]$ is bijectively mapped to $M_x[rank(A_x,i), rank(A_x,j)]$, and the 1s in this range correspond one to one with occurrences of x in $A[i,j]$. Therefore, if 1 is a τ'-majority in $M_x[rank(A_x,i), rank(A_x,j)]$, then x is a τ'-majority in $A[i,j]$. $\square$

Example. Value 1 is a majority in $A[5,7]$, and it holds that $A_1[5,7] = \langle 1\ 1\ 1\rangle$ and $M_1[rank(A_1,5), rank(A_1,7)] = M_1[2,4] = \langle 0\ 1\ 1\rangle$, where 1 is a majority. □

Thus, with A_x and M_x we can determine whether x is a majority in a range.

Lemma 3. *It is sufficient to have rank-enabled bitmaps A_x and M_x to determine, in constant time, whether x is a τ'-majority in any $A[i,j]$.*

Proof. We use Lemma 2. We compute $i' = rank(A_x, i)$ and $j' = rank(A_x, j)$. If $j' - i' \neq j - i$, then $A_x[k] = 0$ for some $i \leq k \leq j$ and thus x is not a τ-majority in $A[i,j]$, hence it is also not a τ'-majority. Otherwise, we find out whether 1 is a τ'-majority in $M_x[i', j']$, by checking whether $rank(M_x, j') - rank(M_x, i'-1) > \tau'(j' - i' + 1)$. □

To find any position $i \leq k \leq j$ where $A[k] = x$, we need the operation $select(B, j)$, defined in Section 2.2. Then, for example, if x is a τ'-majority in $A[i,j]$, its leftmost occurrence in $A[i,j]$ is $i - i' + select(M_x, rank(M_x, i'-1)+1)$. In general, for any $1 \leq t \leq rank(M_x, j') - rank(M_x, i'-1)$, we can retrieve the t-th occurrence with $i - i' + select(M_x, rank(M_x, i'-1) + t)$.

4.2. Coalescing the Bitmaps

We cannot afford to store (and probe!) all the bitmaps A_x and M_x for all x, however. The next lemma is the first step to reduce the total space to slightly superlinear.

Lemma 4. *For any position $A[k] = x$ there are at most $2\lceil 1/\tau \rceil$ 1s in A_x.*

Proof. Consider a process where we start with $A[k] = \perp$ for all k, and set the values $A[k] = x$ progressively. We will distinguish three kinds of changes.

(1) New segments around $A[k]$ are created in S_x. Setting $A[k] = x$ creates in S_x all the segments of the form $[k - k_l, k + k_r]$ for $1 > \tau(k_r + k_l + 1)$, or $k_l + k_r < 1/\tau - 1$. Their union is the area $A_x[k - \lceil 1/\tau \rceil + 2, \ldots, k + \lceil 1/\tau \rceil - 2] = 1$, which may increase the number of 1s in A_x by up to $2\lceil 1/\tau \rceil - 3$.

(2) Segments already covering $A[k]$ are extended. Any maximal segment $[l, r] \in S_x$ covering $A_x[k]$ contains $c > \tau(r - l + 1)$ occurrences of x, but it holds that $c \leq \tau(r - l + 2)$, otherwise there would also exist segments $[l-1, r]$ and $[l, r+1]$ in S_x, and $[l, r]$ would not be maximal. Therefore, adding one more occurrence, $A[k] = 1$, we get $c + 1 \leq \tau(r - l + 2 + 1/\tau)$ occurrences in $[l, r]$. Now it holds that x may be a τ-majority in segments $[l - k_l, r + k_r]$ for all $0 \leq k_l + k_r < 1 + 1/\tau$ (i.e., where $c + 1 > \tau(r - l + 1 + k_l + k_r)$, using only that $c + 1 \leq \tau(r - l + 2 + 1/\tau)$), and therefore we can extend $[l, r]$ to the left by up to $\lceil 1/\tau \rceil$, or to the right by up to $\lceil 1/\tau \rceil$.

(3) Segments reaching close to $A[k]$ are extended. The same reasoning as for the previous case applies, even if $[l, r]$ does not originally contain position k. There are more restrictions, since now $[l - k_l, r + k_r]$ must be so that it contains k, and the same limit $0 \leq k_l + k_r < 1 + 1/\tau$ applies. Thus, in addition to being possible to extend them by at most $\lceil 1/\tau \rceil$ cells in either direction, position k must lie within the extended area.

Total extension. The three cases above are superimposed. Let ℓ_l and ℓ_r the closest positions $\ell_l \leq k \leq \ell_r$ where $A_x[\ell_l] = A_x[\ell_r] = 1$. Then, if $\ell_l = k$, we can set at most $\lceil 1/\tau \rceil$ new 1s in A_x to the left of k by extending segments using case (2). Otherwise, if $k - \ell_l \leq \lceil 1/\tau \rceil$, we can cover the area $A_x[\ell_l + 1, \ldots, k]$ and add up to $\lceil 1/\tau \rceil - (k - \ell_l)$ further cells to the left, using case (3). Otherwise, if $k - \ell_l > \lceil 1/\tau \rceil$, we set $\lceil 1/\tau \rceil - 2$ cells to the left, apart from k, using case (1). The same reasoning applies to the right, and therefore $2\lceil 1/\tau \rceil$ is an upper bound to the number of 1s in A_x produced by each new occurrence of x in A. $\square$

The lemma shows that all the A_x bitmaps add up to $O(n/\tau)$ 1s, and thus the lengths of all the M_x bitmaps add up to $O(n/\tau)$ as well (recall that M_x has one position per 1 in A_x). Therefore, we can store all the M_x bitmaps within $O(n/\tau)$ bits of space. We cannot, however, store all the A_x bitmaps, as they may add up to $O(n^2)$ 0s (note there can be $O(n)$ distinct symbols x), and we still cannot probe all the A_x bitmaps for all x in $o(n)$ time.

Instead, we will *coalesce* all the bitmaps A_x into a smaller number of bitmaps A'_r (which will be called coalesced bitmaps). Coalescing works as follows. Let us write $A[i, j] = b$ to mean $A[\ell] = b$ for all $i \leq \ell \leq j$. We start with all $A'_r[1, n] = 0$ for all r. Then we take each maximal area of all 1s of each bitmap, $A_x[i, j] = 1$, choose some r such that $A'_r[i - 1, j + 1] = 0$, and set $A'_r[i, j] = 1$. That is, we copy the run of 1s from A_x to some coalesced bitmap A'_r such that the run does not overlap nor touch other previous runs already copied (i.e., there must be at least one 0 between any two copied runs of 1s). We associate to each such A'_r a bitmap M'_r where the areas of each M_x corresponding to each coalesced area of A_x are concatenated, in the same order of the coalesced areas. That is, if $A'_r[i_t, j_t] = 1$, the t-th left-to-right run of 1s in A'_r, was copied from A_x, then $M_x[rank(A_x, i_t), rank(A_x, j_t)]$ will be the t-th segment appended to M'_r.

Example. We can coalesce the whole bitmaps A_1 and A_2 into $A' = \langle 1\,0\,1\,0\,1\,1\,1 \rangle$, with the corresponding bitmap $M' = \langle 1\ 1\ 0\ 1\ 1 \rangle$. $\square$

The coalesced bitmaps A'_r and M'_r will replace the original bitmaps A_x and M_x. At query time, we check for the area $[i, j]$ of each coalesced bitmap using Lemma 3. We cannot confuse the areas of different symbols x because we force that there is at least one 0 between any two areas. We cannot report the same τ'-majority x in more than one coalesced bitmap, as both areas should overlap on $[i, j]$ and then they would have been merged as a single area in A_x. If we find one τ'-majority in one coalesced bitmap, we know that there is a τ'-majority x and can spot all of its occurrences (or the leftmost, if desired) in optimal time, even if we cannot know the identity of x. Moreover, we will find all the distinct τ'-majorities in this way.

4.3. Bounding the Number of Coalesced Bitmaps

This scheme will work well if we obtain just a few coalesced bitmaps overall. Next we show how to obtain only $O((1/\tau)\log n)$ coalesced bitmaps.

Lemma 5. *At most $2\log_{1+\tau} n$ distinct values of x can have $A_x[k] = 1$ for a given k.*

Proof. First, $A[k] = x$ is a τ-majority in $A[k, k]$, thus $A_x[k] = 1$. Now consider any other element $x' \neq x$ such that $A_{x'}[k] = 1$. This means that x' is a τ-majority in some $[i, j]$ that contains k. Since $A[k] \neq x'$, it must be that x' is a τ-majority in $[i, k-1]$ or in $[k+1, j]$ (or in both). We say x' is a left-majority in the first case and a right-majority in the second. Let us call $y_1, y_2, \ldots$ the x' values that are left-majorities, and $i_1, i_2, \ldots$ the starting points of their segments (if they are τ-majorities in several segments covering k, we choose one arbitrarily). Similarly, let $z_1, z_2, \ldots$ be the x' values that are right-majorities, and $j_1, j_2, \ldots$ the ending points of their segments. Assume the left-majorities are sorted by decreasing values of i_r and the right-majorities are sorted by increasing values of j_r. If a same value x' appears in both lists, we arbitrarily remove one of them. As an exception, we will start both lists with $y_0 = z_0 = x$, with $i_0 = j_0 = k$.

It is easy to see by induction that y_r must appear at least $(1+\tau)^r$ times in the interval $[i_r, k]$ (or in $[i_r, k-1]$, which is the same). This clearly holds for $y_0 = x$. Now, by the inductive hypothesis, values $y_0, y_1, \ldots, y_{r-1}$ appear at least $(1+\tau)^0, (1+\tau)^1, \ldots, (1+\tau)^{r-1}$ times within $[i_{r-1}, k-1]$ (which contains all the intervals), adding up to $\frac{(1+\tau)^r-1}{\tau}$ occurrences. Thus $k - 1 - i_{r-1} + 1 \geq \frac{(1+\tau)^r-1}{\tau}$. In order to be a left-majority, element y_r must appear strictly more than $\tau(k - i_{r-1}) \geq (1+\tau)^r - 1$ times in $[i_r, k-1]$, to outweight all the occurrences of the previous symbols. The case of right-majorities is analogous. This shows that there cannot be more than $\log_{1+\tau} n$ left-majorities and $\log_{1+\tau} n$ right-majorities. $\square$

In the following it will be useful to define C_x as the set of maximal contiguous areas of 1s in A_x. That is, C_x is obtained by merging all the segments of S_x that touch or overlap. Note that segments of C_x do not overlap, unlike those of S_x. Since a segment of C_x covers a position k iff some segment of S_x covers position k (and iff $A_x[k] = 1$), it follows by Lemma 5 that any position is covered by at most $2\log_{1+\tau} n$ segments of C_x of distinct symbols x.

Note that a pair of consecutive positions $A[k] = x$ and $A[k+1] = y$ is also covered by at most $2\log_{1+\tau} n$ such segments: the right-majorities for $A[k]$ either are y or are also right-majorities for $A[k+1]$, and those are already among the $\log_{1+\tau} n$ right-majorities of $A[k+1]$. And vice versa.

We obtain $O(\log_{1+\tau} n)$ coalesced bitmaps as follows. We take the union of all the sets C_x of all the symbols x and sort the segments by their starting points. Then we start filling coalesced bitmaps. We check if the current segment can be added to an existing bitmap without producing overlaps (and leaving a 0 in between). If we can, we choose any appropriate bitmap, otherwise we start a new bitmap. If at some point we need more than $2\log_{1+\tau} n$ bitmaps, it is because all the last segments of the current $2\log_{1+\tau} n$ bitmaps overlap either the starting point of the current segment or the previous position, a contradiction.

Example. We have $C_1 = \{[1,1],[5,7]\}$, $C_2 = \{[3,3]\}$, and $C_3 = \{[1,6]\}$. Now, we take $C_1 \cup C_2 \cup C_3 = \{[1,1],[1,6],[3,3],[5,7]\}$, and the process produces precisely the coalesced bitmaps A', corresponding to the set $\{[1,1],[3,3],[5,7]\}$, and A_3, corresponding to $\{[1,6]\}$. □

Note that in general the coalesced bitmaps may not correspond to the union of complete original bitmaps A_x, but areas of a bitmap A_x may end up in different coalesced bitmaps.

Therefore, the coalescing process produces $O(\log_{1+\tau} n) = O((1/\tau)\log n)$ bitmaps. Consequently, we obtain $O((1/\tau)\log n)$ query time by simply checking the coalesced bitmaps one by one using Lemma 3.

Finally, representing the $O((1/\tau)\log n)$ coalesced bitmaps A', which have total length $O((n/\tau)\log n)$ and contain $O(n/\tau)$ 1s, requires $O((n/\tau)\log\log n)$ bits if we use a compressed bitmap representation [24] that still offers constant-time *rank* and *select* queries (recall Section 2.2). The coalesced bitmaps M' still have total length $O(n/\tau)$.

This completes the first part of our result. Next, we will reduce the space usage of our encoding.

5. Reducing the Space to $O(n\lceil\log(1/\tau)\rceil)$ Bits

We introduce a different representation of the coalesced bitmaps that allows us to store them in $O(n\lceil\log(1/\tau)\rceil)$ bits, while retaining the same mechanism described above. We note that, although there can be $O(n/\tau)$ bits set in the bitmaps A_x, each new element x produces at most one new *run* of contiguous 1s (case (1) in the proof of Lemma 4). Therefore there are at most n runs in total. We will use a representation of coalesced bitmaps that takes advantage of these runs.

We will distinguish segments of C_x by their lengths, separating lengths by ranges between $\lceil 2^\ell/\tau\rceil$ and $\lceil 2^{\ell+1}/\tau\rceil - 1$, for any *level* $0 \le \ell \le \lg(\tau n)$ (level 0 is special in that it contains lengths starting from 1). In the process of creating the coalesced bitmaps described in the previous section, we will have separate coalesced bitmaps for inserting segments within each range of lengths; these will be called bitmaps of level ℓ. There may be several bitmaps of the same level. It is important that, even with this restriction, our coalescing process will still generate $O((1/\tau)\log n)$ bitmaps, because only $O(1/\tau)$ coalesced bitmaps of each level ℓ will be generated.

Lemma 6. *There can be at most $4/\tau$ segments of any C_x, of length between $\lceil 2^\ell/\tau\rceil$ and $\lceil 2^{\ell+1}/\tau\rceil - 1$, covering a given position k, for any ℓ.*

Proof. Any such segment must be contained in the area $A[k - \lceil 2^{\ell+1}/\tau \rceil + 1, k + \lceil 2^{\ell+1}/\tau \rceil - 1]$, and if x is a τ-majority in it, it must appear more than $\tau \lceil 2^{\ell}/\tau \rceil \geq 2^{\ell}$ times. There can be at most $4/\tau$ different values of x appearing more than 2^{ℓ} times in an area of length less than $2^{\ell+2}/\tau$. □

Consider a coalesced bitmap $A'[1,n]$ of level ℓ. All of its 1s come in runs of lengths at least $b = \lceil 2^{\ell}/\tau \rceil$. We cut A' into *chunks* of length b and define two bitmaps: $A'_1[1,n/b]$ will have $A'_1[i] = 1$ iff the i-th chunk of A' is all 1s, and $A'_2[1,n/b]$ will have $A'_2[i] = 1$ iff the i-th chunk of A' has 0s and 1s. Note that, since the runs of 1s are of length at least b, inside a chunk with 0s and 1s there can be at most one 01 and at most one 10, and the 10 can only come before the 01. Let $p_{10}[j]$ be the position, in the j-th chunk with 0s and 1s, of the 1 preceding a 0, where $p_{10}[j] = 0$ if the chunk starts with a 0. Similarly, let $p_{01}[j]$ be the position of the 0 preceding a 1, with $p_{01}[j] = b$ if the chunk ends with a 0. It always holds that $p_{10}[j] < p_{01}[j]$, and the number of 1s in the chunk is $r(j) = p_{10}[j] + (b - p_{01}[j])$. Also, the rank up to position k in the chunk, $r(j,k)$, is k if $k \leq p_{10}[j]$, $p_{10}[j]$ if $p_{10}[j] < k \leq p_{01}[j]$, and $p_{10}[j] + (k - p_{01}[j])$ if $k > p_{01}[j]$. Then it holds that

$$rank(A', i) \; = \; b \cdot r_1 + \left(\sum_{j=1}^{r_2} r(j) \right) + \begin{cases} r(r_2+1, k) & \text{if } A'_2[1 + \lfloor i/b \rfloor] = 1, \\ A'_1[1 + \lfloor i/b \rfloor] \cdot k & \text{otherwise,} \end{cases}$$

where $r_1 = rank(A'_1, \lfloor i/b \rfloor)$, $r_2 = rank(A'_2, \lfloor i/b \rfloor)$, and $k = i \bmod b$. Note this can be computed in constant time as long as we have constant-time $rank$ data structures on A'_1 and A'_2, and constant-time access and sums on p_{10} and p_{01}.

Example. Using $b = 2^{\ell}$ to make it more interesting, we would have three coalesced bitmaps: $A' = \langle 1\ 0\ 1\ 0\ 0\ 0\ 0 \rangle$, of level $\ell = 0$, for the segments $[1,1]$ and $[3,3]$; $A'' = \langle 0\ 0\ 0\ 0\ 1\ 1\ 1 \rangle$, of level $\ell = 1$, for the segment $[5,7]$; and $A''' = \langle 1\ 1\ 1\ 1\ 1\ 1\ 0 \rangle$, of level $\ell = 2$, for the segment $[1,6]$. Consider level $\ell = 0$ and $b = 2$, and let us focus on A'. Then, we would have $A'_1 = \langle 0\ 0\ 0\ 0 \rangle$, $A'_2 = \langle 1\ 1\ 0\ 0 \rangle$, $p_{10} = \langle 1\ 1 \rangle$, and $p_{01} = \langle 2\ 2 \rangle$. □

To have constant-time sums on p_{10} (p_{01} is analogous), we store its values in a bitmap A'_{10}, where we set all the bits at positions $r + \sum_{j=1}^{r} p_{10}[j]$ to 1, for all r. Then we can recover $\sum_{j=1}^{r} p_{10}[j] = select(A'_{10}, r) - r$. We use a bitmap representation [20] that solves *select* in constant time (recall Section 2.2). Let n' be the number of segments C_x represented in bitmap A'. Then there are at most $2n'$ chunks with 0s and 1s, and A'_{10} contains at most $2n'$ 1s and $2n'b$ 0s (as $0 \leq p_{10}[j] \leq b$). The size of the bitmap representation [20] is in this case $O(n' \log b) = O(n'(\ell + \log(1/\tau)))$ bits. On the other hand, bitmaps A'_1 and A'_2 are represented in plain form [8], requiring $O(n/b) = O(n\tau/2^{\ell})$ bits.

Considering that there are $O(n/\tau)$ 1s overall, and that the runs of level ℓ are of length at least $2^{\ell}/\tau$, we have that there can be at most $n/2^{\ell}$ runs across the $O(1/\tau)$ bitmaps of level ℓ. Therefore, adding up the space over the bitmaps of level ℓ, we have $O(n(\ell + \log(1/\tau))/2^{\ell})$ bits. Added over all the levels ℓ, this gives $O(n\lceil\log(1/\tau)\rceil)$ bits.

Let us now consider the representation of the coalesced bitmaps M'. They have total length $O(n/\tau)$ and contain n 1s overall, therefore using the representation of Lemma 1 with $s = 1$, we have $O(n\lceil\log(1/\tau)\rceil)$ bits of space. They solve *rank* queries in time $O(\log\log_w(1/\tau))$, and *select* in constant time.

As we have to probe $O((1/\tau)\log n)$ coalesced bitmaps M' in the worst case, this raises our query time to $O((1/\tau)\log\log_w(1/\tau)\log n)$. This concludes the proof of Theorem 2, except for the construction time (see the next section).

In our previous work [19], we had obtained $O((1/\tau)\log n)$ time, but using $O((n/\tau)\log^* n)$ bits of space. It is not hard to obtain that time, using $O(n/\tau)$ bits, by simply representing the coalesced bitmaps M' using plain *rank*/*select* structures [8], or even using $O(n\lceil\log(1/\tau)\rceil + (n/\tau)/\operatorname{polylog} n)$ bits, for any $\operatorname{polylog} n$, using compressed representations [23]. The extra $O(\log\log_w(1/\tau))$ time factor arises when we insist in obtaining the optimal $O(n\lceil\log(1/\tau)\rceil)$ bit space. We note that this time penalty factor vanishes when $1/\tau = w^{O(1)}$, which includes the case where $1/\tau$ grows polylogarithmically with n.

6. Construction

The most complex part of the construction of our encoding is to build the sets C_x. Once these are built, the structures described in Section 5 can be easily constructed in $o(n \log n)$ time:

1. The $O(n)$ segments C_x belong to $[1, n]$, so they are sorted by starting point in $O(n)$ time.
2. We maintain a priority queue for each level ℓ, containing the last segment of each coalesced bitmap. We use the queue to find the segment that finishes earliest in order to try to add the new segment of C_x after it. We carry out, in total, $O(n)$ operations on those queues, and each contains $O(1/\tau)$ elements, thus they take total time $O(n\lceil\log(1/\tau)\rceil) = o(n\log n)$.
3. The bitmaps A' of each level ℓ, represented with A'_1, A'_2, A'_{01} and A'_{10}, are easily built in $O(n/b) = O(n\tau/2^{\ell})$ time. Added over the $O(1/\tau)$ coalesced bitmaps of level ℓ this is $O(n/2^{\ell})$, and added over all the levels ℓ this gives $O(n)$ total time.
4. The coalesced bitmaps M' have $O(n)$ 1s overall, so their representation (Lemma 1) is also built in $O(n)$ time, except for the predecessor structures, which need construction of deterministic dictionaries. This can be done in $o(n \log n)$ total time [25].

Now we show that the sets C_x can be built in $O(n \log n)$ time, thus finishing the proof of Theorem 2.

We build the set of increasing positions P_x where x appears in A, for each x, in $O(n \log n)$ total time (the elements of A can be of any atomic type, so we only rely on a comparison-based dictionary to maintain the set of different x values and their P_x lists). Now we build C_x from each P_x using a divide-and-conquer approach, in $O(|P_x| \log |P_x|)$ time, for a total construction time of $O(n \log n)$.

We pick the middle element $k \in P_x$ and compute in linear time the segment $[l, r] \in C_x$ that contains k. To compute l, we find the leftmost element $p_l \in P_x$ such that x is a τ-majority in $[p_l, k_r]$, for some $k_r \in P_x$ with $k_r \geq k$.

To find p_l, we note that it must hold that $(w(p_l, k-1) + w(k, k_r))/(k_r - p_l + 1) > \tau$, where $w(i, j)$ is the number of occurrences of x in $A[i, j]$. The condition is equivalent to $w(p_l, k-1)/\tau + p_l - 1 > k_r - w(k, k_r)/\tau$. Thus we compute in linear time the minimum value v of $k_r - w(k, k_r)/\tau$ over all those $k_r \in P_x$ to the right of k, and then traverse all those $p_l \in P_x$ to the left of k, left to right, to find the first one that satisfies $w(p_l, k-1)/\tau + p_l + 1 > v$, also in linear time. Once we find the proper p_l and its corresponding k_r, the starting position of the segment is slightly adjusted to the left of p_l, to be the smallest value that satisfies $w(p_l, k_r)/(k_r - l + 1) > \tau$, that is, l satisfies $l > -w(p_l, k_r)/\tau + k_r + 1$, or $l = k_r - \lceil w(p_l, k_r)/\tau \rceil + 2$.

Once p_r and then r are computed analogously, we insert $[l, r]$ into C_x and continue recursively with the elements of P_x to the left of p_l and to the right of p_r. Upon return, it might be necessary to join $[l, r]$ with the rightmost segment of the left part and/or with the leftmost segment of the right part, in constant time. The total construction time is $T(n) = O(n) + 2T(n/2) = O(n \log n)$.

Building multiple structures. In order to answer τ'-majority queries for any $\tau \leq \tau' < 1$ in time related to $1/\tau'$ and not to $1/\tau$, we build the encoding of Theorem 2 for values $\tau'' = 1/2, 1/4, 1/8, \ldots, 1/2^{\lceil \lg 1/\tau \rceil}$. Then, a τ'-majority query is run on the structure built for $\tau'' = 1/2^{\lceil \lg 1/\tau' \rceil}$. Since $\tau'/2 < \tau'' \leq \tau'$, the query time is $O((1/\tau'') \log \log_w (1/\tau'') \log n) = O((1/\tau') \log \log_w (1/\tau') \log n)$.

As for the space, we build $O(\lceil \log(1/\tau) \rceil)$ structures, so we use $O(n \lceil \log^2(1/\tau) \rceil)$ bits, and the construction time is $O(n \lceil \log(1/\tau) \rceil \log n)$.

Corollary 1. *Given a real number $0 < \tau < 1$, there exists an encoding using $O(n \lceil \log^2(1/\tau) \rceil)$ bits that answers range τ'-majority queries, for any $\tau \leq \tau' < 1$, in time $O((1/\tau') \log \log_w (1/\tau') \log n)$, where $w = \Omega(\log n)$ is the RAM word size in bits. The structure can be built in time $O(n \lceil \log(1/\tau) \rceil \log n)$.*

7. A Faster Data Structure

In this section we show how, by adding $O(n \log \log n)$ bits to our data structure, we can slash a $\log n$ factor from the query time, that is, we prove Theorem 3. The result, as discussed in the Introduction, yields the optimal query time $O(1/\tau)$ when $1/\tau = O(\text{polylog}\, n)$, although the resulting space may not be optimal anymore.

The idea is inspired by a previous non-encoding data structure for majority queries [9]. Consider a value ℓ. Then we will cut A into consecutive pieces of length 2^ℓ (said to be of *level* ℓ) in two overlapped ways: $A[2^\ell k + 1, 2^\ell (k+1)]$ and $A[2^\ell k + 2^{\ell-1} + 1, 2^\ell (k+1) + 2^{\ell-1}]$, for all $k \geq 0$. We carry out this partitioning for every $\lceil \lg(1/\tau) \rceil \leq \ell \leq \lceil \lg n \rceil$.

Note that there are $O(n/2^\ell)$ pieces of level ℓ, and any interval $A[i,j]$ of length up to $2^\ell/2$ is contained in some piece P of level ℓ. Now, given a query interval $A[i,j]$, let $\ell = \lceil \lg(j-i+1) \rceil + 1$. Then, not only $A[i,j]$ is contained in a piece P of level ℓ, but also any τ-majority x in $A[i,j]$ must be a $\tau/4$-majority in P: Since $j-i+1 > 2^\ell/4$, x occurs more than $\tau(j-i+1) > (\tau/4)2^\ell$ times in $A[i,j]$, and thus in P.

Consider a $\tau/4$-majority x in a given piece P of level ℓ that is also a τ-majority for some range $A[i,j]$ within P, where $2^\ell/4 < j-i+1 \leq 2^\ell/2$. By construction of our previous structures, there exists a maximal segment C_x that contains the range $[i,j]$. If there is another range $A[i',j']$ within P where x is a τ-majority, then there exists another maximal segment C'_x for the same x within P. By our construction, if $C'_x \neq C_x$, then C'_x is disjoint with C_x, and thus each of them contains at least $(\tau/4)2^\ell$ distinct occurrences of x. Obviously, segments C_y for τ-majorities $y \neq x$ contain other $(\tau/4)2^\ell$ occurrences disjoint from those of x. Therefore, the number of distinct maximal segments C that contain τ-majorities at any range $A[i,j]$ (with $j-i+1 > 2^\ell/4$) within P is upper bounded by $4/\tau$. We will say those segments C are *relevant* to P.

Therefore, for each piece P of level ℓ, we will store the index r of the coalesced bitmap A'_r (and its companion M'_r) to which each maximal segment C that is relevant to P belongs. Since there are at most $4/\tau$ such coalesced bitmaps to record, out of a total of $O((1/\tau)\log n)$ coalesced bitmaps, γ-codes on a differential encoding of the subset values requires $O((1/\tau)\log\log n)$ bits.[4] Added up over the $O(n/2^\ell)$ pieces of level $\ell \geq \lceil \lg(1/\tau) \rceil$, this yields $\sum_{\ell \geq \lceil \lg(1/\tau) \rceil} O((n/2^\ell)(1/\tau)\log\log n) = O(n \log\log n)$ bits.

This information reduces the search effort to that of verifying $O(1/\tau)$ coalesced bitmaps A'_r and M'_r for the range $[i,j]$, and thus to $O((1/\tau)\log\log_w(1/\tau))$ query time. However, for ranges shorter than $1/\tau$, where no piece structure has been built, we still have the original query time. To speed up this case, we build a second structure where, for each element $A[k]$, we identify the coalesced bitmap where the maximal segment $C_{A[k]}$ containing the segment $A[k,k]$ belongs, and store the identifier r of the corresponding coalesced bitmap A'_r (and M'_r) associated with k. This requires $O(n\lceil \log((1/\tau)\log n) \rceil) = O(n\lceil \log(1/\tau) \rceil + n\log\log n)$ further bits, and allows checking only one coalesced bitmap A'_r (and M'_r) for each of the $O(1/\tau)$ positions that need to be checked.

To finish the proof we must consider the construction time. The second structure (for short ranges) is easily built with the general structure, taking asymptotically the same amount of time, by keeping track of which maximal segment $C_{A[k]}$ contains each segment $A[k,k]$ and which coalesced bitmap it is assigned. With this, the structure for long ranges can be built as follows: for each position $A[k]$ contained in a piece P of level ℓ, consider the maximal segment $C_{A[k]}$ that contains it and determine whether it is relevant to P. A weak test for this is to consider the coalesced bitmap M' where $C_{A[k]}$ is represented (which is precisely what the first structure stores associated with k) and ask

[4]We could also afford to store them in plain form, in $O((1/\tau)(\log(1/\tau) + \log\log n))$ bits.

whether M' contains more than $(\tau/4)2^\ell$ 1s in the range of P. This must be the case if $C_{A[k]}$ is relevant to P. Although including the identifier of each M' that passes the test may add some nonrelevant ones, we still cannot include more than $4/\tau$ coalesced bitmaps in the set, as the 1s in the M' bitmaps are disjoint.

The *rank* operations on bitmaps M' take $O(\log\log_w(1/\tau))$ time, so we avoid using them to count how many 1s M' contains in the range of P. Instead, we perform a preprocessing pass over P as follows: We initialize to zero a set of $O((1/\tau)\log n)$ counters, one per coalesced bitmap M', and process P left to right. We increase the counter associated with the bitmap M' of each element $A[k]$ in P. At the end, we know all the desired values. This takes $O(2^\ell)$ time, and a similar postprocessing pass clears the counter for the next piece.

Therefore, we process all the pieces P of level ℓ in time $O(2^\ell)$, which amounts to $O(n)$ time per level. Added over all the levels, this gives $O(n\log n)$ total time. This concludes the proof of Theorem 3.

8. Conclusions

A τ-majority query on array $A[1,n]$ receives a range $[i,j]$ and returns all the elements appearing more than $\tau(j-i+1)$ times in $A[i,j]$. We have obtained the first results about *encodings* for answering range τ-majority queries. Encodings are data structures that use less space than what is required to store A and answer queries without accessing A at all. In the encoding scenario we do not report the τ-majorities themselves, but one of their positions in $A[i,j]$.

We have proved that $\Omega(n\lceil\log(1/\tau)\rceil)$ bits are necessary for any such encoding, even if it can only count the number of τ-majorities in any range. Then we presented an encoding that uses the optimal $O(n\lceil\log(1/\tau)\rceil)$ bits, and answers queries in $O((1/\tau)\log\log_w(1/\tau)\log n)$ time in the RAM model with word size $w=\Omega(\log n)$ bits. We also showed that this time can be divided by $\log n$ if we add $O(n\log\log n)$ bits to the space. This yields various space/time tradeoffs, shown in Table 1. Our encoding can actually report any occurrence of each τ-majority, in optimal extra time. The structure is built in $O(n\log n)$ time.

An open question is whether it is possible to achieve optimal query time within optimal space for all values of $1/\tau$. As seen in Table 1, we reach this only for $\log(1/\tau)=\Theta(\log\log n)$. This is also possible when $\log(1/\tau)=\Omega(\log n)$, where we leave the non-encoding scenario [1]. Instead, our results for $\log(1/\tau)$ between $\log\log n$ and $\log n$ have a small factor $O(\log\log_w(1/\tau))$ over the optimal time, and those for $\log(1/\tau)$ below $\log\log n$ either require nonoptimal $O(n\log\log n)$ bits of space, or an $O(\log n)$ factor over the optimal time. It is not clear whether combined optimality can be reached.

Another open question is whether we can do better for weaker versions of the problem we have not studied. For example, if we are only required to report *any* occurrence of *any* τ-majority (or, even less, telling whether or not there exists a τ-majority), our lower bound based on representing a bitmap B shows that $\Omega(n)$ bits are necessary, but we do not know if this bound is tight.

Acknowledgements. We thank the reviewers for their valuable comments.

References

[1] Belazzougui, D., Gagie, T., Navarro, G., 2013. Better space bounds for parameterized range majority and minority. In: Proc. 11th Annual Workshop on Algorithms and Data Structures (WADS). pp. 121–132.

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

[3] Bose, P., Kranakis, E., Morin, P., Tang, Y., 2005. Approximate range mode and range median queries. In: Proc. 22nd International Symposium on Theoretical Aspects of Computer Science (STACS). pp. 377–388.

[4] Brodal, G., Fagerberg, R., Greve, M., López-Ortiz, A., 2009. Online sorted range reporting. In: Proc. 20th Annual International Symposium on Algorithms and Computation (ISAAC). pp. 173–182.

[5] Chan, T., Durocher, S., Larsen, K., Morrison, J., Wilkinson, B., 2012. Linear-space data structures for range mode query in arrays. In: Proc. 29th International Symposium on Theoretical Aspects of Computer Science (STACS). pp. 290–301.

[6] Chan, T., Durocher, S., Skala, M., Wilkinson, B., 2012. Linear-space data structures for range minority query in arrays. In: Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT). pp. 295–306.

[7] Chan, T., Wilkinson, B., 2013. Adaptive and approximate orthogonal range counting. In: Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 241–251.

[8] Clark, D., 1996. Compact PAT trees. Ph.D. thesis, University of Waterloo, Canada.

[9] Durocher, S., He, M., Munro, I., Nicholson, P., Skala, M., 2013. Range majority in constant time and linear space. Information and Computation 222, 169–179.

[10] Elias, P., 1974. Efficient storage and retrieval by content and address of static files. Journal of the ACM 21, 246–260.

[11] Fano, R., 1971. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts.

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

[13] Gagie, T., He, M., Munro, I., Nicholson, P., 2011. Finding frequent elements in compressed 2d arrays and strings. In: Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 295–300.

[14] Greve, M., Jørgensen, A., Larsen, K. D., Truelsen, J., 2010. Cell probe lower bounds and approximations for range mode. In: Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP). pp. 605–616.

[15] Grossi, R., Iacono, J., Navarro, G., Raman, R., Satti, S. R., 2013. Encodings for range selection and top-k queries. In: Proc. 21st Annual European Symposium on Algorithms (ESA). pp. 553–564.

[16] Karpinski, M., Nekrich, Y., 2008. Searching for frequent colors in rectangles. In: Proc. 20th Canadian Conference on Computational Geometry (CCCG). pp. 11–14.

[17] Karpinski, M., Nekrich, Y., 2011. Top-k color queries for document retrieval. In: Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 401–411.

[18] Navarro, G., Raman, R., Rao, S. S., 2014. Asymptotically optimal encodings for range selection. In: Proc. 34th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). pp. 291–302.

[19] Navarro, G., Thankachan, S., 2014. Encodings for range majority queries. In: Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS 8486. pp. 262–272.

[20] Okanohara, D., Sadakane, K., 2007. Practical entropy-compressed rank/select dictionary. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 60–70.

[21] Petersen, H., Grabowski, S., 2009. Range mode and range median queries in constant time and sub-quadratic space. Information Processing Letters 109 (4), 225–228.

[22] Pătraşcu, M., Thorup, M., 2008. Time-space trade-offs for predecessor search. CoRR cs/0603043v1, `http://arxiv.org/pdf/cs/0603043v1`.

[23] Pătraşcu, M., 2008. Succincter. In: Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS). pp. 305–313.

[24] Raman, R., Raman, V., Rao, S. S., 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3 (4), article 43.

[25] Ružić, M., 2008. Constructing efficient dictionaries in close to sorting time. In: Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP). LNCS 5125. pp. 84–95 (part I).

[26] Skala, M., 2013. Array range queries. In: Space-Efficient Data Structures, Streams, and Algorithms. LNCS. Springer, pp. 333–350.