Linear-Space Data Structures for Range Mode Query in Arrays*

Timothy M. Chan† Stephane Durocher‡ Kasper Green Larsen§ Jason Morrison‡
Bryan T. Wilkinson§

February 3, 2013

Abstract

A mode of a multiset S is an element $a \in S$ of maximum multiplicity; that is, a occurs at least as frequently as any other element in S. Given an array $A[1:n]$ of n elements, we consider a basic problem: constructing a static data structure that efficiently answers range mode queries on A. Each query consists of an input pair of indices (i, j) for which a mode of $A[i:j]$ must be returned. The best previous data structure with linear space, by Krizanc, Morin, and Smid (ISAAC 2003), requires $\Theta(\sqrt{n}\log\log n)$ query time in the worst case. We improve their result and present an $O(n)$-space data structure that supports range mode queries in $O(\sqrt{n/\log n})$ worst-case time. In the external memory model, we give a linear-space data structure that requires $O(\sqrt{n/B})$ I/Os per query, where B denotes the block size. Furthermore, we present strong evidence that a query time significantly below $\sqrt{n}$ cannot be achieved by purely *combinatorial* techniques; we show that boolean matrix multiplication of two $\sqrt{n}\times\sqrt{n}$ matrices reduces to n range mode queries in an array of size $O(n)$. Additionally, we give linear-space data structures for the dynamic problem (queries and updates in near $O(n^{3/4})$ time), for orthogonal range mode in d dimensions (queries in near $O(n^{1-1/2d})$ time) and for half-space range mode in d dimensions (queries in $O(n^{1-1/d^2})$ time). Finally, we complement our dynamic data structure with a reduction from the *multiphase problem*, again supporting that we cannot hope for much more efficient data structures.

1 Introduction

The *frequency* of an element x in a multiset S, denoted $\text{freq}_S(x)$, is the number of occurrences (i.e., the multiplicity) of x in S. A *mode* of S is an element $a \in S$ such that for all $x \in S$, $\text{freq}_S(x) \leq \text{freq}_S(a)$. A multiset S may have multiple distinct modes; the frequency of the modes of S, denoted by m, is unique. A unique mode is also known as a *plurality* when $m \leq |S|/2$ and a *majority* when $m > |S|/2$.

Along with the mean and median, the mode is a fundamental statistic in data analysis for which efficient computation is necessary. Given a sequence of n elements ordered in a list A, a range query seeks to compute the corresponding statistic on the multiset determined by a subinterval of the list $A[i:j]$. The objective is to preprocess A to construct a data structure that supports efficient response to one or more subsequent range queries, where the corresponding input parameters (i, j) are provided at query time. Such a data structure is useful as it allows us to report statistics over any window of a given sequence of data.

We assume the standard RAM model of computation with word size $w = \Omega(\log n)$. Although the complete set of possible queries can be precomputed and stored using $\Theta(n^2)$ space, practical data structures require less storage while still enabling efficient response time. For all i, if $i = j$, then a range query must report $A[i]$. Consequently, any range query data structure for a list of n items requires $\Omega(n)$ storage space in the

*A preliminary version of these results appeared at the 29th Symposium on Theoretical Aspects of Computer Science (STACS) [14]. Work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and in part by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

†University of Waterloo, Waterloo, Canada, tmchan@uwaterloo.ca

‡University of Manitoba, Winnipeg, Canada, durocher@cs.umanitoba.ca and jason_morrison@umanitoba.ca

§Aarhus University, Aarhus, Denmark, larsen@cs.au.dk and btw@cs.au.dk

worst case [7]. This leads to a natural question: how quickly can an $O(n)$-space data structure answer range queries?

A range mean query is equivalent to a normalized range sum query (partial sum query), for which a precomputed prefix-sum array provides a linear-space static data structure with constant query time [47]. Range median queries have been analyzed extensively in recent years and are closely related to range counting, where efficient data structures are now known (with linear space and logarithmic or slightly sublogarithmic query time), including both static and dynamic, and both linear- and superlinear-space data structures [7, 11, 12, 16, 37, 38, 41, 42, 44, 45, 47, 54, 55]. In contrast, range mode queries appear more challenging than range mean and median. As expressed recently by Brodal et al. [11, page 2]: "The problem of finding the most frequent element within a given array range is still rather open."

The best previous linear-space data structure for range mode query was by Krizanc et al. [46, 47], who obtained a query time of $O(\sqrt{n}\log\log n)$.[1] No better approaches have been discovered in the intervening years, which leads one to suspect that a $\sqrt{n}$-type bound might be the best one could hope for.

Indeed, we present strong evidence that purely combinatorial approaches cannot avoid the $\sqrt{n}$ effect in the preprocessing or query costs, up to polylogarithmic factors. (Krizanc et al.'s method has near $n^{3/2}$ preprocessing time.) More specifically, we show in Section 9.1 that boolean matrix multiplication (matrix multiplication on $\{0,1\}$-matrices with addition and multiplication replaced by logical OR and AND, respectively) of two $\sqrt{n}\times\sqrt{n}$ matrices reduces to n range mode queries in an array of size $O(n)$. This reduction implies that any data structure for range mode must have either $\Omega(n^{\omega/2})$ preprocessing time or $\Omega(n^{\omega/2-1})$ query time in the worst case, where ω denotes the matrix multiplication exponent. Since the current best matrix multiplication algorithm has exponent 2.3727 [59], we cannot obtain preprocessing time better than $n^{1.18635}$ and query time better than $n^{0.18635}$ simultaneously with current knowledge. Moreover, since the current best *combinatorial* algorithm for boolean matrix multiplication (which avoids algebraic techniques as in Strassen's) has running time only a polylogarithmic factor better than cubic [4], we cannot obtain preprocessing time better than $n^{3/2}$ and query time better than $\sqrt{n}$ simultaneously by purely combinatorial techniques with current knowledge, except for polylogarithmic-factor speedups.

In view of the above hardness result, it is therefore worthwhile to pursue more modest improvements for the range mode problem. Notably, can the extra $\log\log n$ factor in Krizanc et al.'s bound be eliminated?

In Section 3, we give a data structure that accomplishes just that: with $O(n)$ space, we can answer range mode queries in $O(\sqrt{n})$ time. The data structure is based on—and in some ways simplifies—Krizanc et al.'s, since we use only rudimentary structures (mostly arrays), without van Emde Boas trees or repeated binary searches.

In fact, we go beyond eliminating a mere $\log\log n$ factor: in Section 6, we present an $O(n)$-space data structure that answers range mode queries in $o(\sqrt{n})$ time. The precise worst-case time bound is $O(\sqrt{n/w}) \subseteq O(\sqrt{n/\log n})$. As one might guess, bit packing tricks are used to achieve the speedup, but in addition we need a nontrivial combination of ideas, including partitioning elements into two sets (one with small maximum frequency and another with a small number of distinct elements), each handled by a different method, and an interesting application of rank/select data structures (from the world of succinct data structures). In the external memory model we can do even better. In Section 7, we present a linear-space data structure that requires $O(\sqrt{n/B})$ I/Os per query, where B denotes the block size.

We also give efficient data structures for several variations of range mode. In Section 8, we consider a dynamic version of range mode in which an update operation modifies the value of one array element. We obtain the first result with sublinear query and update time bounds: our linear-space data structure supports both queries and updates in near $O(n^{3/4})$ time. Since the current state of the art in unconditional lower bounds remains just polylogarithmic [53], even in the dynamic setting, Section 9.2 presents additional evidence of the hardness of range mode by reducing the *multiphase problem* [48] to dynamic range mode. Based on a widely believed conjecture about the hardness of 3-SUM, this implies that any dynamic range mode data structure must have either update time or query time polynomial in the size of the array.

In Section 10, we consider a natural higher-dimensional generalization of the problem: given a set of

[1] The original data structure described by Krizanc et al. [47] supports queries in $O(\sqrt{n}\log n)$ time. As they remarked, this time can be reduced to $O(\sqrt{n}\log\log n)$ by using van Emde Boas trees for predecessor search [30].

coloured points in $\mathbb{R}^d$, support queries for the most frequently occurring colour in some query range. We obtain the first nontrivial results for this geometric problem. For example, for orthogonal ranges, we give a near-linear space data structure that supports queries in near $O(n^{1-1/2d})$ time. For half-space ranges, we give a linear-space data structure that supports queries in $O(n^{1-1/d^2})$ time. This latter result is obtained using an interesting application of geometric *cuttings* [18], in addition to standard range searching data structures.

Throughout the paper, let m denote the maximum frequency (i.e., the mode of the overall array), and let Δ denote the number of distinct elements ($\max\{m, \Delta\} \leq n$).

2 Related Work

2.1 Computing a Mode

The mode of a multiset S of n items can be found in $O(n \log n)$ time by sorting S and scanning the sorted list to identify the longest sequence of identical elements. By reduction from element uniqueness, a matching $\Omega(n \log n)$ lower bound in the comparison model follows [58]. Better bounds on the worst-case time can be obtained by parameterizing in terms of m or Δ. A worst-case time of $O(n \log \Delta)$ is easily achieved by inserting the n elements into a balanced search tree in which each node stores a key and its frequency. Munro and Spira [51] and Dobkin and Munro [25] described an $O(n \log(n/m))$-time algorithm and a matching comparison-based lower bound. On the word RAM model, a mode can be computed in linear expected time by hashing.

2.2 Range Mode Query

As mentioned, a range mode data structure of Krizanc et al. [47] requires linear space and provides $O(\sqrt{n} \log \log n)$ query time. Krizanc et al. also considered larger-space structures. They described data structures that provide constant-time queries using $O(n^2 \log \log n / \log n)$ space and $O(n^\epsilon \log n)$-time queries using $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in (0, 1/2]$. Petersen and Grabowski [55] improved the first bound to constant time and $O(n^2 \log \log n / \log^2 n)$ space and Petersen [54] improved the second bound to $O(n^\epsilon)$-time queries using $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in [0, 1/2)$. Although its space requirement is almost linear in n as ϵ approaches $1/2$, the data structure of Petersen [54] requires $\omega(n)$ space (the number of levels in a hierarchical set of tables and hash functions approaches ∞ as $\epsilon \to 1/2$). Our new approach can also lead to improved space-time trade-offs (see the statement of Theorem 7 with the parameter $s = n^{1-\epsilon}$): we can obtain $O(n^\epsilon)$ query time with $O(n^{2-2\epsilon} / \log n)$ space for any fixed $\epsilon \in [0, 1/2]$. This improves Petersen's result (though for $\epsilon = 0$, Petersen and Grabowski's result remains slightly better). Finally, Greve et al. [40] prove a lower bound of $\Omega(\log n / \log(s \cdot w/n))$ query time for any data structure that uses s memory cells of w bits in the cell probe model.

2.3 Approximate Range Mode Query

Bose et al. [7] considered approximate range mode queries, in which the objective is to return an element whose frequency is at least αm. They gave a data structure that requires $O(n/(1-\alpha))$ space and answers approximate range mode queries in $O(\log \log_{1/\alpha} n)$ time for any fixed $\alpha \in (0, 1)$, as well as data structures that provide constant-time queries for $\alpha \in \{1/2, 1/3, 1/4\}$, using space $O(n \log n)$, $O(n \log \log n)$, and $O(n)$, respectively. Greve et al. [40] gave data structures that support approximate range mode queries in $O(1)$ time using $O(n)$ space for $\alpha = 1/3$, and $O(\log(\alpha/(1-\alpha)))$ time using $O(n\alpha/(1-\alpha))$ space for any fixed $\alpha \in [1/2, 1)$.

2.4 Continuous Space versus Array Input

A vast literature studies the problems of geometric range searching in continuous Euclidean space; that is, data points are positioned arbitrarily in $\mathbb{R}^d$. See the survey by Agarwal [1] for an overview of results. The

range query problems considered in this paper, however, restrict attention to array input. Although a range query on an array can be viewed as a restricted case of a more general range searching problem (e.g., a point set with regular grid spacing), the algorithmic techniques differ greatly between the two settings when $d \geq 2$. When $d = 1$, however, a geometric range mode query problem reduces to array range mode query. In particular, the rank of each data point in Euclidean space corresponds to its array index. It suffices to compute the ranks of the respective successor and predecessor of the endpoints of the query interval to identify the indices i and j, and to return the corresponding array range mode query on $A[i : j]$.

2.5 Other Related Range Query Problems

In addition to results on the median, mode, and sum range query problems discussed in Sections 1 and 2.2, other range query problems examined on arrays include semigroups [2, 60, 61], extrema (e.g., range minimum or maximum) [5, 6, 21, 23, 26, 31, 32, 33, 34, 35], selection or quantiles (for which the median is a special case) [37, 38, 44, 45], dominance or rank (counting the number of elements in the query range that exceed a given input threshold) [43, 44], coloured range (counting/enumerating the distinct elements in the query range) [37], and k-frequency (determining whether any element has frequency k) [40].

Related to range mode query, Durocher et al. [27] described an $O(n)$-space data structure that supports constant-time range majority queries; this data structure is then extended to range α-majority queries, a generalization of range majority. Recently, Chan et al. [15] examined the least-frequent element range query problem, which is analogous to range mode query, except that an element of minimum frequency is returned. They give an $O(n)$-space data structure that supports queries in $O(\sqrt{n})$ time.

Finally, range query problems have been examined on multidimensional arrays, including partial sums [19], range minimum [3, 9, 8, 23, 39, 56, 57, 62], median [38], selection [37], and α-majority [36]. Similarly, range query problems have been examined in the dynamic setting, including median and selection [11, 12, 38, 42], dominance [52], majority [29], and minimum [10, 20]. To the authors' knowledge, this paper contains the first examinations of range mode query in the multidimensional or dynamic settings.

3 First Method: $O(\sqrt{n})$ Query Time and $O(n)$ Space

We begin by presenting a linear-space data structure with $O(\sqrt{n})$ query time, improving Krizanc et al.'s result [47] by a factor of $\log \log n$. We build on the data structure of Krizanc et al. and introduce a different technique that avoids the need for predecessor search. We will actually establish the following time-space trade-off—the linear-space result follows by setting the parameter $s = \lceil \sqrt{n} \rceil$.

Theorem 1 *Given an array $A[1 : n]$ and any fixed value $s \in [1, n]$, there exists a data structure requiring $O(n + s^2)$ space that supports range mode queries on A in $O(n/s)$ time.*

The following observation will be useful:

Lemma 2 (Krizanc et al. [47]) *Let A_1 and A_2 be any multisets. If c is a mode of $A_1 \cup A_2$ and $c \notin A_1$, then c is a mode of A_2.*

3.1 Data Structure Precomputation

Given input array $A[1 : n]$, let D denote the set of distinct elements stored in A and assume some arbitrary ordering on the elements. We first apply rank space reduction: construct an array $\bar{A}[1 : n]$ such that for each i, $\bar{A}[i]$ stores the rank of $A[i]$ in D. Here, $\bar{A}[i] \in \{1, \ldots, \Delta\}$. For any a, i, and j, $\bar{A}[a]$ is a mode of $\bar{A}[i : j]$ if and only if $A[a]$ is a mode of $A[i : j]$. For simplicity, we describe our data structures in terms of array $\bar{A}$; a table look-up provides a direct bijective mapping from $\{1, \ldots, \Delta\}$ to D. Set D, array $\bar{A}$, and the value Δ are independent of any query range and can be computed in $O(n \log \Delta)$ time during preprocessing.

For each $a \in \{1, \ldots, \Delta\}$, let $Q_a = \{b \mid \bar{A}[b] = a\}$. That is, Q_a is the set of indices b such that $\bar{A}[b] = a$. For any a, a range counting query for element a in $\bar{A}[i : j]$ can be answered by searching for the predecessors

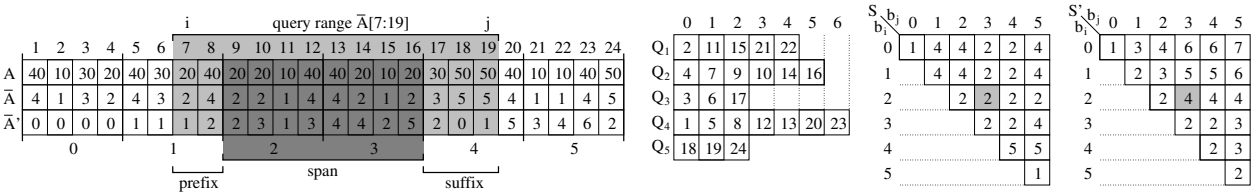

Figure 1: The array has size $n = 24$ (of which $\Delta = 5$ are distinct), partitioned into $s = 6$ blocks of size $t = 4$. The query range is $A[i : j] = A[7 : 19]$, for which the unique mode is 20, occurring with frequency 5. The corresponding mode of $\bar{A}[i : j]$ is 2. The query range is partitioned into the prefix $\bar{A}[7 : 8]$, the span $\bar{A}[9 : 16]$, and the suffix $\bar{A}[17 : 19]$. The span covers blocks $b_i = 2$ to $b_j = 3$, for which the corresponding mode is $S[2, 3] = 2$, occurring with frequency $S'[2, 3] = 4$.

of i and j, respectively, in the set Q_a; the difference of their indices is the frequency of a in $\bar{A}[i : j]$ [47]. Such a range counting query can be implemented using an efficient predecessor data structure in $\Theta(\log \log n)$ time in the worst case (e.g., [30]).

The following related decision problem, however, can be answered in *constant* time by a linear-space data structure: does $\bar{A}[i : j]$ contain at least q instances of element $\bar{A}[i]$? This question can be answered by a "select" query that returns the index of the qth instance of $\bar{A}[i]$ in $\bar{A}[i : n]$. For each $a \in \{1, \ldots, \Delta\}$, store the set Q_a as an ordered array (also denoted Q_a for simplicity). Define a rank array $\bar{A}'[1 : n]$ such that for all b, $\bar{A}'[b]$ denotes the rank (i.e., the index) of b in $Q_{\bar{A}[b]}$. Given any q, i, and j, to determine whether $\bar{A}[i : j]$ contains at least q instances of $\bar{A}[i]$ it suffices to check whether $Q_{\bar{A}[i]}[\bar{A}'[i] + q - 1] \leq j$. Since array $Q_{\bar{A}[i]}$ stores the sequence of indices of instances of element $\bar{A}[i]$ in $\bar{A}$, looking ahead $q-1$ positions in $Q_{\bar{A}[i]}$ returns the index of the qth occurrence of element $\bar{A}[i]$ in $\bar{A}[i : n]$; if this index is at most j, then the frequency of $\bar{A}[i]$ in $\bar{A}[i : j]$ is at least q. If the index $\bar{A}'[i] + q - 1$ exceeds the size of the array $Q_{\bar{A}[i]}$, then the query returns a negative answer. This gives the following lemma:

Lemma 3 *Given an array $A[1 : n]$, there exists a data structure requiring $O(n)$ space that can determine in constant time for any $0 \leq i \leq j \leq n$ and any q whether $A[i : j]$ contains at least q instances of element $A[i]$.*

Following Krizanc et al. [47], given any $s \in [1, n]$ we partition array $\bar{A}$ into s blocks of size $t = \lceil n/s \rceil$. That is, for each $i \in \{0, \ldots, s - 2\}$, the ith block spans $\bar{A}[i \cdot t + 1 : (i + 1)t]$ and the last block spans $\bar{A}[(s - 1)t + 1 : n]$. We precompute tables $S[0 : s - 1, 0 : s - 1]$ and $S'[0 : s - 1, 0 : s - 1]$, each of size $\Theta(s^2)$, such that for any $0 \leq b_i \leq b_j < s$, $S[b_i, b_j]$ stores a mode of $\bar{A}[b_i t + 1 : (b_j + 1)t]$ and $S'[b_i, b_j]$ stores the corresponding frequency.

The arrays $Q_1, \ldots, Q_\Delta$ can be constructed in $O(n)$ total time in a single scan of array $\bar{A}$. The arrays S and S' (which we call the *mode table*) can be constructed in $O(n \cdot s)$ time by scanning array $\bar{A}$ s times, computing one row of each array S and S' per scan. Thus, the total precomputation time required to initialize the data structure is $O(n \cdot s)$.

3.2 Query Algorithm

Given a query range $\bar{A}[i : j]$, let $b_i = \lceil (i - 1)/t \rceil$ and $b_j = \lfloor j/t \rfloor - 1$ denote the respective indices of the first and last blocks completely contained within $\bar{A}[i : j]$. We refer to $\bar{A}[b_i t + 1 : (b_j + 1)t]$ as the *span* of the query range, to $\bar{A}[i : \min\{b_i t, j\}]$ as its *prefix*, and to $\bar{A}[\max\{(b_j + 1)t + 1, i\} : j]$ as its *suffix*. One or more of the prefix, span, and suffix may be empty; in particular, if $b_i > b_j$, then the span is empty. See Figure 1.

The value $c = S[b_i, b_j]$ is a mode of the span with frequency $f_c = S'[b_i, b_j]$. If the span is empty, then let $f_c = 0$. By Lemma 2, either c is a mode of $\bar{A}[i : j]$ or some element of the prefix or suffix is a mode of $\bar{A}[i : j]$. Thus, to find a mode of $\bar{A}[i : j]$, we verify for every element in the prefix and suffix whether its frequency in $\bar{A}[i : j]$ exceeds f_c and, if so, we identify this element as a *candidate* mode and count its additional occurrences in $\bar{A}[i : j]$.

We now describe how to compute the frequency of all candidate elements in the prefix, storing the value and frequency of the current best candidate in c and f_c; an analogous procedure is applied to the suffix. Sequentially scan the items in the prefix starting at the leftmost index, i, and let x denote the index of the current item. If $Q_{\bar{A}[x]}[\bar{A}'[x] - 1] \geq i$, then an instance of element $\bar{A}[x]$ appears in $\bar{A}[i : x - 1]$, and its frequency has been counted already; in this case, simply skip $\bar{A}[x]$ and increment x. Otherwise, check whether the frequency of $\bar{A}[x]$ in $\bar{A}[i : j]$ (which is equivalent to the frequency of $\bar{A}[x]$ in $\bar{A}[x : j]$) is at least f_c by Lemma 3 (i.e., by testing whether $Q_{\bar{A}[x]}[\bar{A}'[x] + f_c - 1] \leq j$). If not, we again skip $\bar{A}[x]$. Otherwise, $\bar{A}[x]$ is a candidate, and the exact frequency of $\bar{A}[x]$ in $\bar{A}[i : j]$ can be counted by a linear scan[2] of $Q_{\bar{A}[x]}$, starting at index $\bar{A}'[x] + f_c - 1$ and terminating upon reaching either an index y such that $Q_{\bar{A}[x]}[y] > j$ or the end of array $Q_{\bar{A}[x]}$ (i.e., $y = |Q_{\bar{A}[x]}| + 1$). That is, $Q_{\bar{A}[x]}[y]$ denotes the index of the first instance of element $\bar{A}[x]$ that lies beyond the query range $\bar{A}[i : j]$ (or no such element exists). Consequently, the frequency of $\bar{A}[x]$ in $\bar{A}[i : j]$ is $f_x = y - \bar{A}'[x]$. Update the current best candidate and its frequency: $c \leftarrow \bar{A}[x]$ and $f_c \leftarrow f_x$.

After all elements in the prefix and suffix have been processed, a mode of $\bar{A}[i : j]$ and its frequency are stored in c and f_c, respectively.

3.3 Analysis

Excluding the linear scans of $Q_{\bar{A}[x]}$, the query cost is clearly bounded by $O(t)$. For each candidate $\bar{A}[x]$ encountered during the processing of the prefix, the cost of the linear scan of $Q_{\bar{A}[x]}$ is $O(f_x - f_c)$. Since f_c is at least the frequency of the mode of the span, at least $f_x - f_c$ instances of $\bar{A}[x]$ must occur in the prefix or suffix. We can thus charge the cost of the scan to these instances. Since each element $\bar{A}[x]$ is considered a candidate at most once (during its first appearance) in the prefix, we conclude that the total cost of all the linear scans is proportional to the total number of elements in the prefix, i.e., $O(t)$. An analogous argument holds for the cost of processing the suffix. Therefore, a range mode query requires $O(t) = O(n/s)$ total time. The data structure requires $O(n)$ space to store the arrays A, $\bar{A}$, and $\bar{A}'$, $O(n)$ total space to store the arrays $Q_1, \ldots, Q_\Delta$, and $O(s^2)$ space to store the tables S and S'. This proves Theorem 1.

4 Second Method: $O(\sqrt{n/w})$ Query Time and $O(n)$ Space When $m \leq \sqrt{nw}$

Our second method is a refinement of the first method (from Section 3), in which we store the mode tables (S and S') more compactly by an encoding scheme that enables efficient retrieval of the relevant information, using techniques from succinct data structures, specifically, for *rank/select* operations. We show how to reduce a query to four rank/select operations. These new ideas allow us to improve the space bound in Theorem 1 by a factor of w, which enables us to use a slightly larger number of blocks, s, which in turn leads to an improved query time. However, there is one important caveat: our space-saving technique only works when the maximum frequency is small, namely, when $m \leq s$. Specifically, we will prove the following theorem in this section: choosing $s = \lceil \sqrt{nw} \rceil$ gives $O(n)$ space and $O(\sqrt{n/w})$ query time for $m \leq \sqrt{nw}$.

Theorem 4 *Given an array $A[1 : n]$ and any fixed $s \in [1, n]$ such that $m \leq s$ (where m is the frequency of the overall mode), there exists a data structure requiring $O(n + s^2/w)$ space that supports range mode queries on A in $O(n/s)$ time.*

4.1 Modified Data Structure

Recall that for a span from block b_i to block b_j, the mode tables store a mode of the span and its frequency in $S[b_i, b_j]$ and $S'[b_i, b_j]$, respectively. As we will show, a mode of the span can be computed efficiently if its frequency is known; consequently, we omit table S. Also, instead of storing the frequency of the mode

[2]Although the time required to complete a linear scan could be reduced by instead using a binary search or a more efficient predecessor data structure, the asymptotic worst-case time remains unchanged; for simplicity, a linear scan suffices.

explicitly, we store column-to-column frequency deltas (i.e., differences of adjacent frequency values); observe that frequency values are monotonically increasing across each row. We encode the frequency deltas for a single row as a bit string, where a zero bit represents an increment in the frequency of the mode (i.e., each frequency delta is encoded in unary) and a one bit represents a former cell boundary. In any row, the number of ones is at most the number of blocks, s, and the number of zeroes is at most $m \leq s$. Precompute a data structure that uses a linear number of bits to support $O(1)$-time binary *rank* and *select* operations on each row (e.g., see [50]):[3] given a binary string, for each $a \in \{0, 1\}$, $\text{rank}_a(i)$ returns the number of times a occurs in the first i positions of the string, and $\text{select}_a(i)$ returns the position of the ith occurrence of a in the string. Thus, each row of the table uses $O(s)$ bits of space. The table has s rows and requires $O(s^2)$ bits of space in total. We pack these bits into words, resulting in an $O(s^2/w)$-space data structure.

4.2 Modified Query Algorithm

Assuming we know a mode of the span and its frequency, we can process the prefix and suffix ranges in $O(t)$ time as before. Our attention turns now to determining a mode of the span and its frequency. We first obtain the frequency of the mode of the span in $O(1)$ time using rank and select queries on the bit string of the b_ith row:

$$pos_{b_j} \leftarrow \text{select}_1(b_j - b_i + 1), \qquad \text{and} \qquad freq \leftarrow \text{rank}_0(pos_{b_j}).$$

Having found the frequency of the mode, identifying a mode itself is still a tricky problem. We proceed in two steps. We first determine the block in which the last occurrence of a mode lies, in $O(1)$ time, as follows:

$$pos_{\text{last}} \leftarrow \text{select}_0(freq), \qquad \text{and} \qquad b_{\text{last}} \leftarrow \text{rank}_1(pos_{\text{last}}) + b_i.$$

Next we find a mode of the span by iteratively examining each element in block b_{last}, using a technique analogous to that for processing a suffix from Section 3. By Lemma 3 (reversed with $j \leq i$), we can check whether each element $\bar{A}[x]$ in b_{last} has frequency *freq* in $\bar{A}[b_i t + 1 : x]$, in $O(1)$ time per element. If the mode occurs multiple times in block b_{last}, its last occurrence will be successfully identified. Processing block b_{last} requires $O(t)$ total time. We conclude that the total query time is $O(t) = O(n/s)$ time. This proves Theorem 4.

5 Third Method: $O(\Delta)$ Query Time and $O(n)$ Space

In this section, we take a quick detour and consider a third method that has query time sensitive to Δ, the number of distinct elements; this "detour" turns out to be essential in assembling our final solution. We show the following:

Theorem 5 *Given an array $A[1 : n]$, there exists a data structure requiring $O(n)$ space that supports range mode queries on A in $O(\Delta)$ time, where Δ denotes the number of distinct elements in A.*

The proof is simple: to answer a range mode query, the approach is to compute the frequency (in the query range) for each of the Δ possible elements explicitly, and then just compute the maximum in $O(\Delta)$ time.

5.1 Data Structure Precomputation

As before, we work with the array $\bar{A}$ by rank space reduction. This time, we divide $\bar{A}$ into blocks of size $t = \Delta$. For each $i \in \{1, \ldots, \lfloor n/\Delta \rfloor\}$, and for every $x \in \{1, \ldots, \Delta\}$, store the frequency $C_i[x]$ of x in the range $\bar{A}[1 : i\Delta]$. The total size of all these *frequency tables* is $O((n/\Delta) \cdot \Delta) = O(n)$. The preprocessing time required is $O(n)$ (or $O(n \log \Delta)$ time if Δ or $\bar{A}$ must be computed).

[3] Succinct data structures can ensure that space usage is very close to the length of the bit string up to lower-order terms, but this fact is not needed in our application.

5.2 Query Algorithm

Given a query range $\bar{A}[i:j]$, as mentioned, it suffices to compute the frequency of x in $\bar{A}[i:j]$ for every $x \in \{1, \ldots, \Delta\}$.

Let $b_j = \lfloor j/\Delta \rfloor - 1$. We can compute the frequency $C(x)$ of x in the suffix $\bar{A}[b_j\Delta + 1 : j]$ for every $x \in \{1, \ldots, \Delta\}$ by a linear scan, in $O(\Delta)$ time since the suffix has size at most Δ. Then the frequency of x in $\bar{A}[1:j]$ is given by $C_{b_j}[x] + C(x)$. The frequency of x in $\bar{A}[1 : i-1]$ can be computed similarly. The frequency of x in $\bar{A}[i:j]$ is just the difference of these two numbers. The total query time is clearly $O(\Delta)$. This proves Theorem 5.

6 Final Method: $O(\sqrt{n/w})$ Query Time and $O(n)$ Space

We are finally ready to present our improved linear-space data structure with $O(\sqrt{n/w})$ query time. Our final idea is simple: if the elements all have small frequencies, the second method (Section 4) already works well; otherwise, the number of distinct elements with large frequencies is small, and so the third method (Section 5) can be applied instead.

More precisely, let s be any fixed value in $[1, n]$. Partition the elements of A into those with *low* frequencies, i.e., at most s, and those with *high* frequencies, i.e., greater than s. A mode of the low-frequency elements has frequency at most s. Thus we can apply Theorem 4 to build an $O(n + s^2/w)$-space range mode query data structure on the low-frequency elements to support $O(n/s)$ query time. On the other hand, there are at most n/s distinct high-frequency elements. Thus we can apply Theorem 5 to build an $O(n)$-space range mode query data structure on the high-frequency elements to support $O(n/s)$ query time. The following simple decomposition lemma allows us to combine the two structures:

Lemma 6 *Given an array $A[1:n]$ and any ordered partition of A into two arrays $B_1[1:n']$ and $B_2[1:n-n']$ such that no element in B_1 occurs in B_2 nor vice versa, if there exist respective $s_1(n)$- and $s_2(n)$-space data structures that support range mode queries on B_1 and B_2 in $t_1(n)$ and $t_2(n)$ time, then there exists an $O(n + s_1(n) + s_2(n))$-space data structure that supports range mode query on A in $O(t_1(n) + t_2(n))$ time.*

Proof. For each $a \in \{1, 2\}$ and $i \in \{1, \ldots, n\}$, precompute $I_a[i]$, the index in the B_a array of the first element in A to the right of $A[i]$ that lies in B_a; and precompute $J_a[i]$, the index in the B_a array of the first element in A to the left of $A[i]$ that lies in B_a. Given a range query $A[i:j]$, we can compute the mode in the range $B_1[I_1[i], J_1[j]]$ and the mode in the range $B_2[I_2[i], J_2[j]]$ and determine which has larger frequency; this is a mode of $A[i:j]$. □

We have thus completed the proof of our main theorem:

Theorem 7 *Given an array $A[1:n]$ and any fixed $s \in [1, n]$, there exists a data structure requiring $O(n + s^2/w)$ space that supports range mode queries on A in $O(n/s)$ time. In particular, by setting $s = \lceil \sqrt{nw} \rceil$, there exists a data structure requiring $O(n)$ space that supports range mode queries on A in $O(\sqrt{n/w})$ time.*

7 External Memory Data Structure

We turn our attention to the external memory model. In this model, we have a two-level memory hierarchy: the disk and the cache. The disk is divided into blocks of size B and our data structure can only operate on blocks that are in the cache. Our goal is to minimize the number of I/Os (i.e., disk accesses) during a query. Any internal memory data structure with query time $q(n)$ requires $O(q(n))$ I/Os in the external memory model, so we already have linear-space data structures that require $O(\sqrt{n})$, $O(\Delta)$, or $O(\sqrt{n/w})$ I/Os per query. We present the following theorem:

Theorem 8 *Given an array $A[1:n]$, there exists a data structure requiring $O(n)$ space that supports range mode queries on A in $O(\sqrt{n/B})$ I/Os.*

First, note that using the same decomposition technique as in Section 6, we can solve the general case by separately considering the cases of having a low number of distinct elements Δ and having a low maximum frequency m. Decomposing a query takes constant time and thus constant I/Os. Next, we observe that the data structure of Theorem 5 needs only $O(\Delta/B)$ I/Os during a query. A query consists of two linear scans of frequency tables of size Δ as well as linear scans of the prefix and suffix, which are also of size Δ. Each scan requires $O(\Delta/B)$ I/Os since the elements of each array are stored contiguously in memory. Although queries to our data structures that give $O(\sqrt{n})$ and $O(\sqrt{n/w})$ query time also reduce to linear scans of the prefix and suffix, at each step of the scan we query the data structure of Lemma 3 which requires a potentially non-local memory access. Thus, a similar observation does not hold for these data structures. The final piece is a linear-space data structure that requires $O(m)$ I/Os per query. We give a stronger result: an internal memory data structure that requires $O(m)$ query time.

Theorem 9 *Given an array $A[1:n]$, there exists a data structure requiring $O(n)$ space that supports range mode queries on A in $O(m)$ time, where m denotes the frequency of the mode of A.*

Proof. We use a combination of ideas from our first method and from an approximate range mode query data structure of Greve et al. [40].

We proceed as in Section 3, except that now we divide into blocks of size $t = m$ and we replace the mode tables S and S' with the following: for each $i \in \{1, \ldots, \lfloor n/s \rfloor\}$, construct an array $F_i[1:m]$ such that for each x, $F_i[x]$ stores the largest $j \leq n$ such that the mode of $A[i:j]$ has frequency at most x; a corresponding mode is also stored. The space for these arrays is $O(m \cdot n/t) = O(n)$.

A query range is divided into prefix, span, and suffix subarrays as before. As observed by Greve et al. [40], a mode of the span and its frequency can be computed by finding the successor of j in F_i; this can be achieved in $O(\log m)$ time by binary search in F_i. Once the mode of the span and its frequency have been found, we can compute the mode of the query range by processing the prefix and suffix as in Section 3, in $O(t) = O(m)$ time. The resulting worst-case query time is $O(m)$. □

We consider distinct elements of A with frequency at most $\sqrt{n/B}$ separately from those with frequency greater than $\sqrt{n/B}$. Using the data structure of Theorem 9 for the low-frequency elements, we support queries in $O(\sqrt{n/B})$ I/Os. Since there are $O(\sqrt{nB})$ elements with frequency greater than $O(\sqrt{n/B})$, the data structure of Theorem 5 built for the high-frequency elements supports queries in $O(\sqrt{nB}/B) = O(\sqrt{n/B})$ I/Os. This proves Theorem 8.

8 Dynamic Data Structure

We now consider a dynamic version of the problem in which an update operation can modify an element in the array A. As in Section 3, we partition A into s blocks of size $t = \lceil n/s \rceil$. In addition to the parameter s, our data structure also takes as input a user specifiable parameter k. For each of the $\Theta(s^2)$ spans, we store the top k most frequent elements as well as their frequencies in the data structure described in Lemma 10. We call such a data structure the top list of a span. The size of this component of our data structure is $O(k \cdot s^2)$.

Lemma 10 *There is a data structure that maintains a dynamic multiset S of at most N elements requiring linear space with respect to the number of unique elements in S and supporting the following operations:*

- *Update(S, x, μ): Add μ instances of element x to multiset S in $O(\log \log N)$ expected time. If μ is negative, instances of x are removed from S.*
- *Top(S, k): Report the top k most frequent elements in multiset S in $O(1 + k)$ worst-case time.*

Proof. We use a linear-space dynamic van Emde Boas tree [30] to store key/value pairs, where a key is a frequency in the universe $\{1, \ldots, N\}$ and its associated record is a doubly-linked list of all elements whose

frequency in S matches the key. We use linear-space dynamic hashing to map an element to its doubly-linked list node in constant expected time.

Assume we are given an update for the frequency of an element x. If there is already an element $x \in S$, we start by removing its doubly-linked list node from the list associated with its original frequency f. If the list for frequency f becomes empty, we delete it from our van Emde Boas tree in $O(\log \log N)$ expected time. If there is no other element $x \in S$, we create a new doubly-linked list node for the new unique element and the original frequency f is 0. Unless $f + \mu = 0$, we add the doubly-linked list node of x to the list associated with frequency $f + \mu$. If there is no other element with frequency $f + \mu$, this involves creating a new list and inserting it into our van Emde Boas tree in $O(\log \log N)$ expected time.

To find the top k elements in S we iterate through the entries in our van Emde Boas tree from most frequent to least frequent. The list of most frequent elements is kept at the root of the van Emde Boas tree and can be accessed in constant time. Assuming the van Emde Boas tree is augmented to include pointers between adjacent entries we can find the next list of elements to output in constant time. Once we have iterated through the k most frequent elements we stop, resulting in a running time of $O(1 + k)$. □

Each of our top lists represents elements from a subarray of A containing at most n elements and thus supports update operations in $O(\log \log n)$ time. We first note that we can build all $\Theta(s^2)$ top lists in $O((k \cdot s^2 + sn) \log \log n)$ expected time. For each block b_i, we create all of the top lists for the spans that start at b_i in one pass through A. During the pass, we add each element in A starting at the beginning of b_i to a global top list. At the end of a block b_j, we get the top k elements seen so far in the global top list and insert them into the top list for the span from b_i to b_j in $O(k \log \log n)$ time. The cost of a single pass through A thus includes the $O(n \log \log n)$ cost of adding elements to the global top list and the $O(k \cdot s \log \log n)$ cost of creating top lists for spans. All s passes thus take $O((k \cdot s^2 + sn) \log \log n)$ time.

For each unique element we also store a linear-space dynamic ranking data structure (e.g., Dietz's data structure [24]) for the indices at which the element resides in A. With these data structures we can perform range counting queries for a specific element in $O(\log n / \log \log n)$ worst-case time. The cumulative space cost of all of these dynamic ranking data structures is $O(n)$.

Given a query, we follow the approach of Krizanc et al. [47] using range counting queries that run in $O(\log n / \log \log n)$ worst-case time, instead of the $O(\log \log n)$-time range counting queries that are possible in the static case. The result is a worst-case query running time of $O((n/s) \cdot (\log n / \log \log n))$.

Assume we are given an update to change the element at index i from x to y. We note that the element at index i is contained in $O(s^2)$ spans and we must propagate the change to each of their top lists. For each top list, we reduce the frequency of x by 1 if x is in the top list. Also, for each top list, we increase the frequency of y by 1 if y is in the top list. If y is not in the top list, we perform a range counting query for element y in the span and increase the frequency of y by this count. Propagating these changes thus takes $O(s^2 \log n / \log \log n)$ time due to the range counting queries.

After k updates the most frequent element in a span may not be in the span's top list. Consider when all k elements in the original top list have the same frequency f and there is one more element x with frequency f. If each of the k elements in the top list is replaced by some infrequent element, then x becomes the most frequent element in the span. For this reason, as well as to ensure that the space required by a top list remains $O(k)$, we rebuild all of our top lists after every k updates. The amortized cost of this step for each update is $O((s^2 + s \cdot n/k) \log \log n)$. Including the cost of propagating changes to all of the top lists, the overall cost of an update is $O(s^2 \log n / \log \log n + (s \cdot n/k) \log \log n)$.

Theorem 11 *Given an array $A[1 : n]$ and any fixed $s, k \in \{1, \ldots, n\}$, there exists a data structure requiring $O(n + k \cdot s^2)$ space that supports range mode queries on A in $O((n/s) \cdot (\log n / \log \log n))$ worst-case time and changes to the elements of A in $O(s^2 \log n / \log \log n + (s \cdot n/k) \log \log n)$ amortized expected time. In particular, by setting $s = \lceil n^{1/4} \rceil$ and $k = \lceil \sqrt{n} \rceil$, there exists a data structure requiring $O(n)$ space that supports range mode queries on A in $O(n^{3/4} \log n / \log \log n)$ worst-case time and changes to the elements of A in $O(n^{3/4} \log \log n)$ amortized expected time.*

Note that if space is not a concern, we can set $s = \lceil n^{1/3} \rceil$ and $k = \lceil n^{2/3} \rceil$ and obtain an $O(n^{4/3})$-space structure with $O(n^{2/3} \operatorname{polylog} n)$ query and amortized update time.

9 Conditional Lower Bounds

In this section, we present strong evidence that range mode cannot be solved much more efficiently than what we have achieved in this paper. First we present a reduction from boolean matrix multiplication to range mode queries and then a reduction from the multiphase problem to dynamic range mode.

9.1 Boolean Matrix Multiplication and Range Mode

In the following, we show that boolean matrix multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices reduces to n range mode queries in an array of size $O(n)$. Greve et al. [40] observe the following:

Observation 12 (Greve et al. [40]) *Let S be a multiset whose elements belong to a universe U. Adding one of each element in U to S increases the frequency of the mode of S by one.*

Observation 13 (Greve et al. [40]) *Let S_1 and S_2 be two sets (not multisets) and let S be the multiset union of S_1 and S_2. The frequency of the mode of S is one if $S_1 \cap S_2 = \emptyset$ and it is two if $S_1 \cap S_2 \neq \emptyset$.*

Now let A and B be two $\sqrt{n} \times \sqrt{n}$ boolean matrices for which we are to compute the product $C = A \cdot B$. The entry $c_{i,j}$ in C must be 1 precisely if there exists at least one index k, where $1 \leq k \leq \sqrt{n}$, such that $a_{i,k} = b_{k,j} = 1$. Our goal is to determine whether this is the case using one range mode query for each entry $c_{i,j}$. Our first step in achieving this is to transform each row of A and each column of B into respective sets. For the ith row of A, we construct the set A_i containing all those indices k for which $a_{i,k} = 1$, i.e., $A_i = \{k \mid a_{i,k} = 1\}$. Similarly we let $B_j = \{k \mid b_{k,j} = 1\}$. Clearly $c_{i,j} = 1$ if and only if $A_i \cap B_j \neq \emptyset$. By Observation 13, this can be tested if we can determine the frequency of the mode in the multiset union of A_i and B_j. Our last step is thus to embed all the sets A_i and B_j into an array, such that we can use range mode queries to perform these intersection tests for every pair i, j. Our constructed array M has two parts, a left part L and a right part R. The array M is then simply the concatenation of L and R. The array L represents all the sets A_i. It consists of $\sqrt{n}$ blocks of $\sqrt{n}$ entries. The ith block (entries $L[(i-1)\sqrt{n}+1 : i\sqrt{n}]$) represents the set A_i, and it consists of the elements $\{1, \ldots, \sqrt{n}\} \setminus A_i$ in some arbitrary order, followed by the elements of A_i in some arbitrary order. The array R similarly represents the sets B_j and it also consists of $\sqrt{n}$ blocks of $\sqrt{n}$ entries. The jth block represents the set B_j and it consists of the elements in B_j in some arbitrary order, followed by the elements $\{1, \ldots, \sqrt{n}\} \setminus B_j$ in some arbitrary order.

Now assume that $|A_i|$ and $|B_j|$ are known for each set A_i and B_j. We can now determine whether $A_i \cap B_j \neq \emptyset$ (i.e., whether $c_{i,j} = 1$) from the result of the range mode query on $M[\text{start}(i) : \text{end}(j)]$, where

$$\text{start}(i) = (i-1)\sqrt{n} + 1 + \sqrt{n} - |A_i| \qquad \text{and} \qquad \text{end}(j) = n + (j-1)\sqrt{n} + |B_j|.$$

To see this, first observe that $\text{start}(i)$ is the first index in M of the elements in A_i, and that $\text{end}(j)$ is the last index in M of the elements in B_j. In addition to a suffix of the block representing A_i and a prefix of the block representing B_j, the subarray $M[\text{start}(i) : \text{end}(j)]$ contains $\sqrt{n} - i$ complete blocks from L and $j-1$ complete blocks from R. Since a complete block contains all the elements $\{1, \ldots, \sqrt{n}\}$, it follows from Observations 12 and 13 that $A_i \cap B_j \neq \emptyset$ (i.e., $c_{i,j} = 1$) if and only if the frequency of the mode in $M[\text{start}(i), \text{end}(j)]$ is $2 + \sqrt{n} - i + j - 1$. The answer to the query $(\text{start}(i), \text{end}(j))$ thus allows us to determine whether $c_{i,j} = 1$ or 0. The array M and the values $|A_i|$ and $|B_j|$ can clearly be computed in linear time when given matrices A and B, thus we have the following result:

Theorem 14 *Let $p(n)$ be the preprocessing time of a range mode data structure and $q(n)$ its query time. Then boolean matrix multiplication on two $\sqrt{n} \times \sqrt{n}$ matrices can be solved in time $O(p(n) + n \cdot q(n) + n)$.*

9.2 The Multiphase Problem and Dynamic Range Mode

In the following, we show that the multiphase problem, introduced by Pătraşcu [53], reduces to dynamic range mode. The multiphase problem is the following dynamic version of set disjointness:

Phase I. We are given k sets, $S_1, \ldots, S_k \subset \{1, \ldots, n\}$. We may preprocess the sets in time $O(nk \cdot \tau)$.

Phase II. We are given another set $T \subseteq \{1, \ldots, n\}$, and have time $O(n \cdot \tau)$ to read and update memory locations from the data structure constructed in Phase I.

Phase III. Finally, we are given an index $i \in \{1, \ldots, k\}$ and must, in time $O(\tau)$, answer whether S_i is disjoint from T.

Pătraşcu made the following conjecture about the hardness of the multiphase problem:

Conjecture 1 (Pătraşcu [53]) *There exists constants $\gamma > 1$ and $\delta > 0$ such that the following holds. If $k = \Theta(n^\gamma)$, any solution to the multiphase problem in the word RAM model requires $\tau = \Omega(n^\delta)$.*

This conjecture is supported by a reduction from the problem known as 3-SUM (given n integers, find three that sum to zero). It is widely believed that the 3-SUM problem cannot be solved in truely subquadratic time, and if this is true, then so is Conjecture 1. In the following, we show that a data structure for dynamic range mode solves the multiphase problem.

Assume the availability of a dynamic range mode data structure with update time t_u and query time t_q on an array M of size $n(k+1)$ entries. We assume all values in the array are initialized to the same fixed value. We think of the array as being partitioned into $k+1$ consecutive chunks of n entries, where the jth chunk from the left corresponds to set S_j and the last chunk corresponds to the set T. In Phase I, we update the entries in chunks $j = 1, \ldots, k$ such that the chunks represents the sets $S_1, \ldots, S_k$. We follow the ideas in Section 9.1 uneventfully: the jth chunk consists of the elements $\{1, \ldots, n\} \setminus S_j$ in some arbitrary order, followed by the elements of S_j in some arbitrary order. In addition to the range mode data structure, we store an array with $k+1$ entries, such that the jth entry stores the value $|S_j|$ and the last entry will store $|T|$. In Phase II, we update the last chunk by inserting the elements of T in the beginning of the chunk. We also update the last entry of our array of set sizes, such that it stores $|T|$. Finally, in Phase III, we are given a query index i and we must determine whether $S_i \cap T = \emptyset$. For this, we first retrieve the values $|S_i|$ and $|T|$ and then ask a range mode query on $M[\text{start}(i) : \text{end}(T)]$, where

$$\text{start}(i) = (i-1)n + 1 + n - |S_i| \qquad \text{and} \qquad \text{end}(T) = nk + |T|.$$

As in Section 9.1, this query range contains precisely the elements of S_i, the elements of T and $k-i$ complete permutations of $\{1, \ldots, n\}$. It follows from Observation 12 and 13 that $S_i \cap T \neq \emptyset$ precisely if the frequency of the mode in $M[\text{start}(i) : \text{end}(T)]$ is $2 + k - i$. We have thus obtained a solution for the multiphase problem using $O(knt_u)$ time in Phase I, $O(nt_u)$ time in Phase II and $O(t_q)$ time in Phase III. It follows that Conjecture 1 implies $\max\{t_u, t_q\} = \Omega(n^\delta)$ for some constant $\delta > 0$ when $k = \Theta(n^\gamma)$ for some constant $\gamma > 1$, i.e., either the query time or the update time of a dynamic range mode data structure must be polynomial in the size of the array.

10 Higher Dimensions

We now consider generalizations of the range mode problem to Euclidean spaces of constant dimension d. Given a set P of n points in $\mathbb{R}^d$, each of which is assigned a colour, we consider the problem of constructing an efficient data structure to support queries that return a most frequently occurring colour in $P \cap Q$ for a query range $Q \subseteq \mathbb{R}^d$. We consider orthogonal range queries in Section 10.1 and half-space range queries in Section 10.2.

10.1 Orthogonal Ranges

We generalize the technique of Krizanc et al. [47] by dividing space into s^d grid cells such that there are $O(n/s)$ points between any two consecutive parallel grid hyperplanes. The generalization of a span of a query range Q is the largest rectangle inside Q whose sides lie along grid hyperplanes. There are s^{2d} distinct spans

and for each we precompute and store the mode of the span. This component of our data structure thus requires $O(s^{2d})$ space. For each set of points of a given colour, we also build an orthogonal range counting data structure [22] with polylogarithmic space overhead that answers queries in polylogarithmic time (see [43] for the best known solution, using $O(n(\log n/\log\log n)^{d-2})$ space and with $O((\log n/\log\log n)^{d-1})$ query time). Across all colours, these data structures require $O(n \operatorname{polylog} n)$ space.

Given a query hyperrectangle Q we use binary search amongst the grid hyperplanes in order to determine the slabs in which Q's sides lie. We then determine the mode of Q's span in $O(1)$ time from our precomputed table. For each of the $2d$ sides of Q we must additionally consider each of the $O(n/s)$ points in the slab in which the side lies. For each such point, we count the number of points of its colour in Q using the range counting data structure of its colour in polylogarithmic time to find the actual mode. So, the running time of a query is $O\left(2d \cdot (n/s) \cdot \operatorname{polylog} n\right) = O\left((n/s) \cdot \operatorname{polylog} n\right)$ time.

Theorem 15 *Given a set P of n points in $\mathbb{R}^d$, each of which is assigned a colour, and any fixed $s \in \{1, \ldots, n\}$, there exists a data structure requiring $O(n \operatorname{polylog} n + s^{2d})$ space that supports orthogonal range mode queries in $O\left((n/s) \cdot \operatorname{polylog} n\right)$ time. In particular, by setting $s = \lceil n^{1/2d} \rceil$, there exists a data structure requiring $O(n \operatorname{polylog} n)$ space that supports range mode queries in $O(n^{1-1/2d} \operatorname{polylog} n)$ time.*

Note that in the special case of *dominance* (i.e., d-sided) ranges in $\mathbb{R}^d$, the query bound in the $O(n \operatorname{polylog} n)$-space structure can be improved to $O(n^{1-1/d} \operatorname{polylog} n)$.

Alternatively, we can guarantee $O(n)$ space if we increase the query time by an n^ϵ factor, by switching to a linear-space data structure for orthogonal range counting with $O(n^\epsilon)$ query time (by using a range tree [22] with n^ϵ fan-out).

10.2 Halfspace Ranges

We now consider half-space range queries. We work in *dual* space [22], where the input is transformed into n hyperplanes, each assigned a colour, and a query half-space is transformed into a point. A query for a dual point q returns the most frequently occurring colour amongst the hyperplanes that lie below q. Let $s \in \{1, \ldots, n\}$ be a fixed parameter specified by the user. We use the key concept of cuttings [18] from computational geometry. Given a set of n hyperplanes in $\mathbb{R}^d$, a $(1/r)$-*cutting* is a partition of $\mathbb{R}^d$ into simplicial cells such that each cell intersects at most n/r hyperplanes. The following is known [17, 18]:

Lemma 16 *For any set of n hyperplanes in $\mathbb{R}^d$, there exists a $(1/r)$-cutting with $O(r^d)$ cells. Furthermore, there is a data structure for point location in the $(1/r)$-cutting, also requiring $O(r^d)$ space and answering queries in $O(\log r)$ time.*

We set $r = (n \cdot s^{d-1})^{1/d}$. For each cell γ in the cutting, we store the mode of the hyperplanes that lie strictly below γ. This component of our data structure requires $O(r^d) = O(n \cdot s^{d-1})$ space. In primal space, we build a simplex range reporting data structure [13, 49] for all of the points with $S = O(n \cdot s^{d-1})$ space. This data structure reports the k points in a query simplex in $O((n/S^{1/d}) \operatorname{polylog} n + k) = O((n/s)^{1-1/d} \operatorname{polylog} n + k)$ time. Also, for each colour i, we build a separate half-space range counting data structure [13, 49] for the n_i points of colour i, with $S_i = O(n_i \cdot s^{d-1})$ space and $O(n_i/S_i^{1/d} + \log n_i) = O((n_i/s)^{1-1/d} + \log n)$ query time. The total space is $O(n \cdot s^{d-1})$.

Given a dual query point q, we first identify the cell γ of the $(1/r)$-cutting that contains q in $O(\log r)$ time. The mode of the hyperplanes below q is either the colour stored at cell γ or one of the colours of the hyperplanes intersecting γ. We can find the $O(n/r)$ hyperplanes intersecting γ by simplex range reporting in primal space in $O((n/s)^{1-1/d} \operatorname{polylog} n + n/r)$ time, since the set of all hyperplanes intersecting a simplex dualizes to a polyhedron of $O(1)$ size. For each hyperplane that intersects γ and lies below q, we perform a half-space range counting query for the points of the colour of the hyperplane in primal space to determine the actual mode. The running time of this step is $O\left(\sum_{i=1}^{O(n/r)} (n_i/s)^{1-1/d} + (n/r) \log n\right)$. By Hölder's inequality, the sum in the first term is bounded by $O((n/r)^{1/d} \cdot (n/s)^{1-1/d}) = O((n/s)^{1-1/d^2})$ for $r = (n \cdot s^{d-1})^{1/d}$. The second term $(n/r) \log n = (n/s)^{1-1/d} \log n$ does not dominate except when $n/s = O(\operatorname{polylog} n)$.

Theorem 17 *Given a set P of n points in $\mathbb{R}^d$, each of which is assigned a colour, and any fixed $s \in \{1, \ldots, n\}$, there exists a data structure requiring $O(n \cdot s^{d-1})$ space that supports half-space range mode queries in $O((n/s)^{1-1/d^2} + \operatorname{polylog} n)$ time. In particular, by setting $s = 1$, there exists a data structure requiring $O(n)$ space that supports half-space range mode queries in $O(n^{1-1/d^2})$ time.*

A similar approach works for other ranges (e.g., simplices, balls, and other constant-degree semialgebraic sets) by transforming query ranges to query points in a higher dimension, and using cuttings in this higher-dimensional space.

11 Discussion and Directions for Future Research

We close by mentioning a few interesting open problems. A useful generalization of the problem is to return the kth most frequently occurring element (or the k most frequent elements) in a query range. Due to its dependence on precomputed modes stored in array S, an analogous generalization of our methods (except for the third method) seems unlikely without a significant increase in space, if k is large.

Open Problem 1 *Construct an $O(n)$-space data structure for identifying the kth most frequently occurring element (or the k most frequent elements) in the range $A[i:j]$ in time $O(n^{1-\epsilon})$ (or $O(n^{1-\epsilon} + k)$) for some constant $\epsilon > 0$, where i, j, and k are given at query time.*

We have given (near-)linear-space data structures for multiple variants of range mode, including dynamic range mode in an array, orthogonal range mode for a d-dimensional point set and half-space range mode for a d-dimensional point set. Our results, in various ways, build on and generalize the techniques of Krizanc et al. [47]. It is unknown whether there are entirely different approaches that can achieve smaller exponents on n in the query and update times of these data structures.

Open Problem 2 *Is there a linear-space dynamic data structure for range mode in an array that supports queries and updates in $O(\sqrt{n} \operatorname{polylog} n)$ time?*

Open Problem 3 *Is there a (near-)linear-space data structure for orthogonal range mode in $\mathbb{R}^d$ that supports queries in $o(n^{1-1/2d})$ time?*

Open Problem 4 *Is there a linear-space data structure for half-space range mode in $\mathbb{R}^d$ that supports queries in $o(n^{1-1/d^2})$ time?*

Lastly, the following open problem is likely difficult since currently no techniques seem capable of proving unconditional super-polylogarithmic cell probe lower bounds:

Open Problem 5 *Prove a tight, unconditional lower bound on the worst-case query time required by any $O(n)$-space data structure that supports range mode queries on an array of n items.*

Acknowledgements

The authors thank Peyman Afshani, Francisco Claude, Meng He, Ian Munro, Patrick Nicholson, Matthew Skala, and Norbert Zeh for discussing various topics related to range searching.

References

[1] P. K. Agarwal. Range searching. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 809–837. CRC Press, New York, 2nd edition, 2004.

[2] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report 71/87, Tel-Aviv University, 1987.

[3] A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, volume 4580 of *Lecture Notes in Computer Science*, pages 286–294. Springer, 2007.

[4] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 745–754, 2009.

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the Latin American Theoretical Informatics Symposium (LATIN)*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[6] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 309–319, 1989.

[7] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3404 of *Lecture Notes in Computer Science*, pages 377–388. Springer, 2005.

[8] G. S. Brodal, P. Davoodi, M. Lewenstein, R. Raman, and S. S. Rao. Two dimensional range minimum queries and Fibonacci lattices. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume 7501 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2012.

[9] G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume 6346/6347 of *Lecture Notes in Computer Science*. Springer, 2010.

[10] G. S. Brodal, P. Davoodi, and S. S. Rao. Path minima queries in dynamic weighted trees. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*, volume 6844 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2011.

[11] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011.

[12] G. S. Brodal and A. G. Jørgensen. Data structures for range median queries. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 5878 of *Lecture Notes in Computer Science*, pages 822–831. Springer, 2009.

[13] T. M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47(4):661–690, 2012.

[14] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. Linear-space data structures for range mode query in arrays. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 14 of *Leibniz International Proceedings in Informatics*, pages 291–301, 2012.

[15] T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson. Linear-space data structures for range minority query in arrays. In *Proceedings of the Scandanavian Symposium and Workshops on Algorithm Theory (SWAT)*, volume 7357 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012.

[16] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 161–173, 2010.

[17] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9(2):145–158, 1993.

[18] B. Chazelle. Cuttings. In *Handbook of Data Structures and Applications*, pages 25.1–25.10. CRC Press, 2005.

[19] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the ACM Symposium on Computational Geometry (SoCG)*, pages 131–139, 1989.

[20] P. Davoodi. *Data Structures: Range Queries and Space Efficiency*. PhD thesis, Aarhus University, 2011.

[21] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *Proceedings of the International Computing and Combinatorics Conference (COCOON)*, volume 7434 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2012.

[22] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 3rd edition, 2008.

[23] E. D. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 2009.

[24] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer, 1989.

[25] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12(3):255–263, 1980.

[26] S. Durocher. A simple linear-space data structure for constant-time range minimum query. *CoRR*, abs/1109.4460, 2011.

[27] S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6755 of *Lecture Notes in Computer Science*, pages 244–255. Springer, 2011.

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

[29] A. Elmasry, M. He, J. I. Munro, and P. Nicholson. Dynamic range majority data structures. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2011.

[30] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[31] J. Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the Latin American Theoretical Informatics Symposium (LATIN)*, volume 6034 of *Lecture Notes in Computer Science*, pages 158–169. Springer, 2010.

[32] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[33] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.

[34] J. Fischer and V. Heun. Finding range minima in the middle: Approximations and applications. *Mathematics in Computer Science*, 3(1):17–30, 2010.

[35] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 135–143, 1984.

[36] T. Gagie, M. He, J. I. Munro, and P. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proceedings of the Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7024 of *Lecture Notes in Computer Science*, pages 295–300. Springer, 2011.

[37] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the String Processing and Information Retrieval Symposium (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2009.

[38] B. Gfeller and P. Sanders. Towards optimal range medians. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 2009.

[39] M. J. Golin, J. Iacono, D. Krizanc, R. Raman, and S. S. Rao. Encoding 2D range maximum queries. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *Lecture Notes in Computer Science*, pages 180–189. Springer, 2011.

[40] M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6198 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2010.

[41] S. Har-Peled and S. Muthukrishnan. Range medians. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume 5193 of *Lecture Notes in Computer Science*, pages 503–514. Springer, 2008.

[42] M. He, J. I. Munro, and P. Nicholson. Dynamic range selection in linear space. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 7074 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2011.

[43] J. JáJá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004.

[44] A. G. Jørgensen. *Data Structures: Sequence Problems, Range Queries, and Fault Tolerance*. PhD thesis, Aarhus University, 2010.

[45] A. G. Jørgensen and K. D. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–813, 2011.

[46] D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 2906 of *Lecture Notes in Computer Science*, pages 517–526. Springer, 2003.

[47] D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12:1–17, 2005.

[48] K. G. Larsen. The cell probe complexity of dynamic range counting. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 85–94, 2012.

[49] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10(2):157–182, 1993.

[50] J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

[51] J. I. Munro and M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, 1976.

[52] Y. Nekrich. Orthogonal range searching in linear and almost-linear space. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*, volume 4619 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2007.

[53] M. Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 603–610, 2010.

[54] H. Petersen. Improved bounds for range mode and range median queries. In *Proceedings of the Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 4910 of *Lecture Notes in Computer Science*, pages 418–423. Springer, 2008.

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

[56] C. K. Poon. Optimal range max datacub for fixed dimensions. In *Proceedings of the International Conference on Database Theory (ICDT)*, volume 2572 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2003.

[57] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5:12–22, 2007.

[58] S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

[59] V. Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 887–898, 2012.

[60] A. C. Yao. Space-time tradeoff for answering range queries. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 128–136, 1982.

[61] A. C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14:277–288, 1985.

[62] H. Yuan and M. J. Atallah. Data structures for range minimum queries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 150–160, 2010.