Towards Optimal Range Medians*

Gerth Stølting Brodal[1], Beat Gfeller**[2], Allan Grønlund Jørgensen[1], and Peter Sanders***[3]

[1] MADALGO‡, Department of Computer Science, Aarhus University, Denmark, {gerth,jallan}@cs.au.dk
[2] IBM Research, Zurich, Switzerland, bgf@zurich.ibm.com
[3] Universität Karlsruhe, Germany, sanders@ira.uka.de

Abstract. We consider the following problem: Given an unsorted array of n elements, and a sequence of intervals in the array, compute the median in each of the subarrays defined by the intervals. We describe a simple algorithm which needs $O(n \log k + k \log n)$ time to answer k such median queries. This improves previous algorithms by a logarithmic factor and matches a comparison lower bound for $k = O(n)$. The space complexity of our simple algorithm is $O(n \log n)$ in the pointer-machine model, and $O(n)$ in the RAM model. In the latter model, a more involved $O(n)$ space data structure can be constructed in $O(n \log n)$ time where the time per query is reduced to $O(\log n / \log \log n)$. We also give efficient dynamic variants of both data structures, achieving $O(\log^2 n)$ query time using $O(n \log n)$ space in the comparison model and $O((\log n / \log \log n)^2)$ query time using $O(n \log n / \log \log n)$ space in the RAM model, and show that in the cell-probe model, any data structure which supports updates in $O(\log^{O(1)} n)$ time must have $\Omega(\log n / \log \log n)$ query time.
Our approach naturally generalizes to higher-dimensional range median problems, where element positions and query ranges are multidimensional — it reduces a range median query to a logarithmic number of range counting queries.

1 Introduction and Related Work

The classic problem of finding the *median* is to find the element of rank $\lceil n/2 \rceil$ in an unsorted array of n elements.[4] Clearly, the median can be found in $O(n \log n)$ time by sorting the elements. However, a classic algorithm finds the median in $O(n)$ time [BFP+72], which is asymptotically optimal.

More recently, the following generalization, called the *Range Median Problem (RMP)*, has been considered [KMS05,HPM08]:

‡ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

* This paper contains results previously announced in [GS09,BJ09].

** The author gratefully acknowledges the support of the Swiss SBF under contract no. C05.0047 within COST-295 (DYNAMO) of the European Union.

*** Partially supported by DFG grant SA 933/3-1.

[4] An element has rank i if it is the i-th element in some sorted order.

Input: An unsorted array A with n elements, each having a *value*. Furthermore, a sequence of k *queries* $Q_1, \ldots, Q_k$, each defined as an interval $Q_i = [L_i, R_i]$. In general the sequence of queries is given in an online fashion and we want the answer to a query before the next query, and k is not known in advance.
Output: A sequence $y_1, \ldots, y_k$ of values, where y_i is the median of the elements in $A[L_i, R_i]$, i.e. the set of all elements whose index in A is at least L and at most R.

Note that instead of the median, any specified rank might be of interest. We restrict ourselves to the median to simplify notation, but a generalization to arbitrary ranks is straightforward for all our results except the ones in Section 8. RMP naturally fits into a larger group of problems, in which an unsorted array is given, and for a query one wants to compute a certain function of all the elements in a given interval. Instead of the median, natural candidates for such a function are:

- Sum: This problem can be trivially solved with $O(n)$ preprocessing time and $O(1)$ query time by computing prefix sums, since each query can be answered by subtracting two prefix sums.
- Semigroup operator: This problem is significantly more difficult than the sum since subtraction is not available. However, there exists a very efficient solution: For any constant c, preprocessing in $O(nc)$ time and space allows to answer queries in $O(\alpha_c(n))$ time, where α_c is the inverse of a certain function at the $(c/2)$th level of the primitive recursion hierarchy. In particular, using $O(n)$ processing time and space, each query can be answered in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function [Yao82]. A matching lower bound is known [Yao85].
- Maximum, Minimum: For these particular semigroup operators, the problem can be solved slightly more efficiently: $O(n)$ preprocessing time and space is sufficient to allow $O(1)$ time queries (see e.g. [GBT84]).
- Mode: The problem of finding the most frequent element within a given array range is still rather open. Using $O(n^2 \log\log n / \log^2 n)$ space (in words), constant query time is possible [PG09], and with $O(n^{2-2\varepsilon})$ space, $0 < \varepsilon \leq 1/2$, $O(n^{\varepsilon})$ query time can be achieved [Pet08]. Some earlier space-time tradeoffs were given in [KMS05].
- Rank: The problem of finding the number of elements smaller than a query element within a query range. This problem has been studied extensively. A linear space data structure supporting queries in $O(\log n / \log\log n)$ time is presented in [JMS04], and a matching lower bound in the cell probe model for any data structure using $O(n \log^{O(1)} n)$ space is shown in [Pătr07,Pătr08].

In addition to being a natural extension of the median problem, RMP has applications in practice, namely obtaining a "typical" element in a given time series out of a given time interval [HPM08].

Natural special cases of RMP are an *offline* variant, where all queries are given in a batch, and a variant where we want to do all *preprocessing up front* and are then interested in good worst case bounds for answering a single query.

The authors of [HPM08] give a solution of the online RMP which requires $O(n \log k + k \log n \log k)$ time and $O(n \log k)$ space. In addition, they give a lower bound of $\Omega(n \log k)$ time for comparison-based algorithms. They basically use a one-dimensional range tree over the input array, where each inner node corresponds to a subarray defined by an interval. Each such subarray is sorted, and stored with the node. A range median query then corresponds to selecting the median from $O(\log k)$ sorted subarrays (whose union is the queried subarray) of total length $O(n)$, which requires $O(\log n \log k)$ time [VSIR91]. The main difficulty of their approach is to show that the subarrays need not be fully sorted, but only presorted in a particular way, which reduces the construction time of the tree from $O(n \log n)$ to $O(n \log k)$.

Concerning the preprocessing variant of RMP, [KMS05] give a data structure to answer queries in $O(\log n)$ time, which uses $O(n \log^2 n / \log \log n)$ space. They do not analyze the required preprocessing time, but it is clearly at least as large as the required space in machine words. Moreover, they give a structure which uses only $O(n)$ space, but query time $O(n^\varepsilon)$ for arbitrary $\varepsilon > 0$. To obtain $O(1)$ query time, the best-known data structure [PG09] requires $O(n^2(\log \log n / \log n)^2)$ space (in words), which improves upon [KMS05] and [Pet08].[5]

Our results. First, in Section 2 we give an algorithm for the pointer-machine model which solves the RMP for an online sequence of k queries in $O(n \log k + k \log n)$ time and $O(n \log k)$ space. This improves the running time of $O(n \log k + k \log n \log k)$ reported in [HPM08] for $k \in \omega(n / \log n)$. Our algorithm is also considerably simpler. The idea is to reduce a range median query to a logarithmic number of related range counting queries. Similar to quicksort, we descend a tree that stems from recursively partitioning the values in array A. The final time bound is achieved using the technique of fractional cascading. In Section 2.1, we explain why our algorithm is time-optimal in the comparison model for $k \in O(n)$ and at most $\Omega(\log n)$ from optimal for $k \in \omega(n)$.

In Section 3 we achieve linear space in the RAM model using techniques from succinct data structures — the range counting problems are reduced to rank computations in bit arrays. To achieve the desired bound, we compress the recursive subproblems in such a way that the bit arrays remain dense at all times. The latter algorithm can be easily modified to obtain a linear space data structure using $O(n \log n)$ preprocessing time that allows arbitrary range median queries to be answered in time $O(\log n)$. In Section 4 we increase the degree of the partitioning of A to $\Theta(\log^\varepsilon n)$ and by adopting bit-tricks and table lookups we reduce the query time to $O(\log n / \log \log n)$ while keeping linear space and $O(n \log n)$ preprocessing time. Note that the previously best linear-space data structure required $O(n^\varepsilon)$ query time [KMS05].

This means that in the RAM model we get an algorithm for the batched RMP that uses $O(n \log k + k \log n / \log \log n)$ time and $O(n)$ space as follows. If

[5] Note that the data structures in [KMS05,Pet08,PG09] work only for a specific quantile (e.g. the median), which must be the same for all queries.

$k \leq \sqrt{n}$ we use the simple data structure and use $O(n \log k)$ time. If $k > \sqrt{n}$ we build the more complicated RAM data structure in $O(n \log n) = O(n \log k)$ time and do the k queries in $O(k \log n / \log \log n)$ time. Our data structures also give solutions with the same query bounds for the range rank problem, which means that our structure also solves the range rank problem optimally [Pät07].

We discuss dynamic data structures for RMP in Section 5. Our dynamic data structure uses $O(n \log n / \log \log n)$ space and supports queries and updates in $O((\log n / \log \log n)^2)$ time. In Section 6 we prove an $\Omega(\log n / \log \log n)$ time lower bound on range median queries for data structures that can be updated in $O(\log^{O(1)} n)$ time using a reduction from the marked ancestor problem [AHR98], leaving a significant gap to the achieved upper bound. We note that for dynamic range rank queries, an $O(n)$ space data structure that supports queries in $O((\log n / \log \log n)^2)$ time and updates in $O(\log^{4+\epsilon} n / (\log \log n)^2)$ time is described in [Nek09], and a matching lower bound for data structures with $O(\log^{O(1)} n)$ update time is proved in [Pät07]. After a few remarks on generalizations for higher dimensional inputs in Section 7, we consider random input arrays and give a construction with constant expected query time using $O(n^{3/2})$ space in expectation in Section 8. Section 9 concludes with a summary and some open problems.

2 A Pointer Machine Algorithm

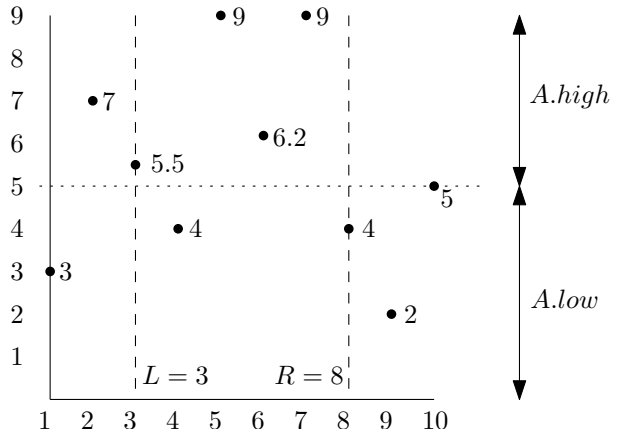


Fig. 1. An example with the array $A = [3, 7, 5.5, 4, 9, 6.2, 9, 4, 2, 5]$. The median value used to split the set of points is 5. For the query $L = 3, R = 8$, there are two elements inside $A.low[L, R]$ and four elements in $A.high[L, R]$. Hence, the median of $A[L, R]$ is the element of rank $p = \lfloor (2 + 4)/2 \rfloor - 2 = 1$ in $A.high[L, R]$.

Our algorithm is based on the following key observation (see also Figure 1): Suppose we partition the elements in array A of length n into two smaller arrays: $A.low$ which contains all elements with the $n/2$ smallest[6] values in A, and $A.high$ which contains all elements with the $n/2$ largest values. The elements in $A.low$ and $A.high$ are sorted by their index in A, and each element e in $A.low$ and $A.high$ is associated with its index $e.i$ in the original input array, and its value $e.v$. Now, if we want to find the element of rank p in the subarray $A[L, R]$, we can do the following: We count the number m of elements in $A.low$ which are contained in $A[L, R]$. To obtain m, we do a binary search for both L and R in $A.low$ (using the $e.i$ fields). If $p \leq m$, then the element of rank p in $A[L, R]$ is the element of rank p in $A.low[L, R]$. Otherwise, the element of rank p is the element of rank $p - m$ in $A.high[L, R]$.

Hence, using the partition of A into $A.low$ and $A.high$, we can reduce the problem of finding an element of a given rank in array $A[L, R]$ to the same problem, but on a smaller array (either $A.low[L, R]$ or $A.high[L, R]$). Our algorithm applies this reduction recursively.

Algorithm overview. The basic idea is therefore to subdivide the n elements in the array into two parts of (almost) equal size by computing the median of their values and using it to split the list into a list of the $n/2$ elements with smaller values and a list of the $n/2$ elements with larger values. The two parts are recursively subdivided further, but only when required by a query (this technique is sometimes called "deferred data structuring", see [KMR88]). To answer a range median query, we determine in which of the two parts the element of the desired rank lies (initially, this rank corresponds to the median, but this may change during the search). Once this is known, the search continues recursively in the appropriate part until a trivial problem of constant size is encountered.

We will show that the total work involved in splitting the subarrays is $O(n \log k)$ and that the search required for any query can be completed in $O(\log n)$ time using fractional cascading [CG86]. Hence, the total running time is $O(n \log k + k \log n)$.

Detailed description and analysis. Algorithm 1 gives pseudocode for the query, which performs preprocessing (i.e., splitting the array into two smaller arrays) only where needed. Note that we have to keep three things separate here: values that are relevant for median computation and partitioning the input, positions in the input sequence that are relevant for finding the elements within the range $[L, R]$, and positions in the subdivided arrays that are important for counting elements.

Let us first analyze the time required for processing a query not counting the 'preprocessing' time within lines 4–6: The query descends $\log n$ levels of

[6] To simplify notation we ignore some trivial rounding issues and also sometimes assume that all elements have unique values. This is without loss of generality because we could artificially expand the size of A to the next power of two and because we can use the index of an element in A to break ties in element comparisons.

Algorithm 1: Query(A, L, R, p)

```
 1 Input: range select data structure A, query range [L,R], desired rank p
 2 if |A| = 1 then return A[1]
 3 if A.low is undefined then
 4 |   Compute the median value y of the elements in A
 5 |   A.low  := ⟨e ∈ A : e.v ≤ y⟩
 6 |   A.high := ⟨e ∈ A : e.v > y⟩
 7 |   { ⟨e ∈ A : Q⟩ is an array containing all elements e of A satisfying the given
 8 |     condition Q, ordered as in A }
 9 { Find(A,q) returns max{j : A[j].i ≤ q} (with Find(A,0) = 0) }
10 l := Find(A.low, L − 1) ;          // # of low elements left of L
11 r := Find(A.low, R) ;               // # of low elements up to R
12 m := r − l ;                  // # of low elements between L and R
13 if p ≤ m then return Query(A.low, L, R, p)
14 else          return Query(A.high, L, R, p − m)
```

recursion.[7] On each level, Find-operations for L and R are performed on the lower half of the current subproblem. If we used binary search, we would get a total execution time of up to $\sum_{i=1}^{\log_2 n} O(\log \frac{n}{2^i}) = \Theta(\log^2 n)$. However, the fact that in all these searches, we search for the same key (L or R) allows us to use a standard technique called fractional cascading [CG86] that reduces the search time to a constant, once the resulting position of the first search is known. Indeed, we only need a rather basic variant of fractional cascading, which applies when each successor list is a sublist of the previous one [dBvKOS00]. Here, it suffices to augment an element e of a list with a pointer to the position of some element e' in each subsequent list (we have two successors: $A.low$ and $A.high$). In our case, we need to point to the largest element in the successor that is no larger than e. We get a total search time of $O(\log n)$.

Now we turn to the preprocessing code in lines 4–6 of Algorithm 1. Let $s(i)$ denote the level of recursion at which query i encountered an undefined array $A.low$ for the first time. Then the preprocessing time invested during query i is $O(n/2^{s(i)})$ if a linear time algorithm is used for median selection [BFP+72] (note that we have a linear recursion with geometrically decreasing execution times). This preprocessing time also includes the cost of finding the pointers for fractional cascading while splitting the list in lines 4–6. Since the preprocessing time during query i decreases with $s(i)$, the total preprocessing time is maximized if small levels $s(i)$ appear as often as possible. However, level j can appear no more than 2^j times in the sequence $s(1), s(2), \ldots, s(k)$.[8] Hence, we get an upper bound for the preprocessing time when the smallest $\lfloor \log k \rfloor$ levels are used as often as possible ('filled') and the remaining levels are $\lceil \log k \rceil$. The preprocessing

[7] Throughout the paper, $\log n$ denotes the binary logarithm of n.

[8] Indeed, for $j > 0$ the maximal number is 2^{j-1} since the other half of the available subintervals have already been covered by the preprocessing happening in the layer above.

time at every used level is $O(n)$ giving a total time of $O(n \log k)$. The same bound applies to the space consumption since we never allocate memory that is not used later. We summarize the main result of this section in a theorem:

Theorem 1. *The online range median problem (RMP) on an array with n elements and k range queries can be solved in time $O(n \log k + k \log n)$ and space $O(n \log k)$.*

Another variant of the above algorithm invests $O(n \log n)$ time and space into complete preprocessing up front. Subsequently, any range median query can be answered in $O(\log n)$ time. This improves the preprocessing space of the corresponding result in [KMS05] by a factor $\log n / \log \log n$ and the preprocessing time by at least this factor.

2.1 Lower Bounds

We briefly discuss how far our algorithm is from optimality. In [HPM08], a comparison-based lower bound of $\Omega(n \log k)$ is shown for the range median problem.[9] As our algorithm shows, this bound is (asymptotically) tight if $k \in O(n)$. For larger k, the above lower bound is no longer valid, as the construction requires $k < n$. Yet, a lower bound of $\Omega(n \log n)$ is immediate for $k \geq n$, by reduction to the sorting problem. Furthermore, $\Omega(k)$ is a trivial lower bound. Note that in our algorithm, the number of levels of the recursion is actually bounded by $O(\min\{\log k, \log n\})$, and thus for any $k \geq n$ our algorithm has running time $O(n \log n + k \log n)$, which is up to $\Omega(\log n)$ from the trivial linear bound.

In a very restricted model (sometimes called "Pointer Machine"), where a memory location can be reached only by following pointers, and not by direct addressing, our algorithm is indeed optimal also for $k \geq n$: it takes $\Omega(\log n)$ time to even access an arbitrary element of the input (which is initially given as a linked list). Since every element of the input is the answer to at least one range query (e.g. the query whose range contains only this element), the lower bound follows. For a matching upper bound, the array based algorithm described in this section can be transformed into an algorithm for the strict pointer machine model by replacing the arrays $A.low$ and $A.high$ by balanced binary search trees.

An interesting question is whether a lower bound $\Omega(k \log n)$ could be shown in more realistic models. However, note that any comparison-based lower bound (as the one in [HPM08]) cannot be higher than $\Omega(n \log n)$: With $O(n \log n)$ comparisons, an algorithm can determine the permutation of the array elements, which suffices to answer any query without further element comparisons. Therefore, one would need to consider more realistic models (e.g. the "cell-probe" model), in which proving lower bounds is significantly more difficult.

[9] The authors derive a lower bound of $\log l$, where $l := \frac{n!}{k!((n/k-1)!)^k}$, and n is a multiple of $k < n$. Unfortunately, the analysis of the asymptotics of l given in [HPM08] is erroneous; however, a corrected analysis shows that the claimed $\Omega(n \log k)$ bound holds.

3 A Linear Space RAM Implementation

Our starting point for a more space efficient implementation of Algorithm 1 is the observation that we do not actually need all the information available in the arrays stored at the interior nodes of our data structure. All we need is support for the operation $Find(x)$ that counts the number of elements e in $A.low$ that have index $e.i \leq x$. This information can already be obtained from a bit-vector where a 1-bit indicates whether an element of the original array is in $A.low$. For this bit-vector, the operation corresponding to *Find* is called *rank*. In the RAM model, there are data structures that need space $n + o(n)$ bits, can be constructed in linear time and support *rank* in constant time (e.g., [Cla88,OS06][10]). Unfortunately, this idea alone is not enough since we would need to store 2^j bit arrays consisting of n positions each on every level j. Summed over all levels, this would still need $\Omega(n \log^2 n)$ bits of space even if optimally compressed data structures were used. This problem is solved using an additional idea: for a node of our data structure with value array A, we do not store a bit array with n possible positions but only with $|A|$ possible positions, i.e., bits represent positions in A rather than in the original input array. This way, we have n positions on every *level* leading to a total space consumption of $O(n \log n)$ bits. For this idea to work, we need to transform the query range in the recursive call in such a way that *rank* operations in the contracted bit arrays are meaningful. Fortunately, this is easy because the rank information we compute also defines the query range in the contracted arrays. Algorithm 2 gives pseudocode specifying the details. Note that the algorithm is largely analogous to Algorithm 1. In some sense, the algorithm becomes simpler because the distinction between query positions and array positions for counting disappears (If we still want to report the positions of the median values in the input, we can store this information at the leaves of the data structure using linear space). Using an analysis analogous to the analysis of Algorithm 1, we obtain the following theorem:

Theorem 2. *The online range median problem (RMP) on an array with n elements and k range queries can be solved in time $O(n \log k + k \log n)$ and space $O(n)$ words in the RAM model.*

By doing all the preprocessing up front, we obtain an algorithm with preprocessing time $O(n \log n)$ using $O(n)$ space and query time $O(\log n)$. This improves the space consumption compared to [KMS05] by a factor $\log^2 n / \log \log n$.

[10] Indeed, since we only need the rank operation, there are very simple and efficient implementations: store a table with ranks for indices that are a multiple of $w = \Theta(\log n)$. General ranks are then the sum of the next smaller table entry and the number of 1-bits in the bit array between this rounded position and the query position. Some processors have a POPCNT instruction for this purpose. Otherwise we can use lookup tables.

Algorithm 2: Query(A, L, R, p)

```
1  Input: range select data structure A, query range [L, R] and desired rank p
2  if |A| = 1 then return A[1]
3  if A.low is undefined then
4  |  Compute the median y of the values in A
5  |  A.lowbits := BitVector(|A|, {i ∈ [1, |A|] : A[i] ≤ y})
6  |  A.low  := ⟨A[i] : i ∈ [1, |A|], A[i] ≤ y⟩
7  |  A.high := ⟨A[i] : i ∈ [1, |A|], A[i] > y⟩
8  |  deallocate the value array of A itself
9  l := A.lowbits.rank(L − 1)
10 r := A.lowbits.rank(R)
11 m := r − l
12 if  p ≤ m then  return Query(A.low, l + 1, r, p)
13 else            return Query(A.high, L − l, R − r, p − m)
```

4 Improving Query Time

In this section, we describe how for the offline variant of the problem, the query time for selecting the element of rank s in the array $A[L, R]$ can be reduced to $O(\log n / \log\log n)$ time while still using $O(n)$ space (in machine words) only.

The initial idea is to use a tree with branching factor $f = \lceil \log^{\epsilon} n \rceil$ for some $0 < \epsilon < 1$, instead of a binary tree. This reduces the depth of the tree to $O(\log n / \log\log n)$, but this is only useful if the branching decision can still be computed in constant amortized time per level. It turns out that using word-level parallelism, this can indeed be achieved. We first give a brief overview before describing the algorithm in detail. As before, a query is answered by descending the tree, starting from the root, until the element of the given rank s in $A[L, R]$ is found. However, there will now be two different cases for computing the child to which to descend: First, an attempt is made to identify the correct child in $O(1)$ time, by computing, for every $l \in \{1, \ldots, f\}$, an approximation of how many elements of $A[L, R]$ lie in the first l childrens' subtrees of the current node (called *prefix count* in the following). Note that the element of rank s is contained in the leftmost subtree of T_v whose prefix count is at least s. To compute all approximate prefix counts in constant time, the approximation is only accurate up to $g = O(\log n / f)$ bits. This computation reduces the branching decision to an interval $[\ell_1, \ell_2]$ among which the child containing the element of rank s is contained. If $[\ell_1, \ell_2]$ contains only one or two indices, the branching decision in this node can be made in constant time, by exactly computing the prefix count for these indices. Yet, since only an approximation is used, this attempt may fail, i.e., more than two children remain as candidates for containing the desired element. In this case, the correct child is computed using a binary search (where in each step, one computes the exact prefix count for a given subtree), which takes $O(\log f) = O(\log\log n)$ time. As we will show, after each such binary search, the number of bits in all prefix counts that are relevant for the search is reduced by g. Therefore, this second case can only occur $O(\log n / g) = O(f)$

times, and the total time for the search remains $O(f \log\log n + \log n/\log\log n) = O(\log n/\log\log n)$.

For clarity, we initially describe a data structure that uses slightly more than $O(n)$ space. Then we reduce the space to linear using standard space compression techniques.

4.1 Structure

The data structure is a balanced search tree T storing the n elements from $A = [y_1, \ldots, y_n]$ in the leaves in sorted order. The fan-out of T is $f = \lceil \log^{\varepsilon} n \rceil$ for some constant $0 < \varepsilon < 1$. For a node v in T, let T_v denote the subtree rooted at v, and $|T_v|$ the number of leaves in T_v. We also use T_v to refer to the set of elements stored in the leaves of T_v. With each node $v \in T$, we associate $f \cdot |T_v|$ prefix sums: For each element $y_i \in T_v$, and for each child index, $1 \leq \ell \leq f$, we denote by t_ℓ^i the number of elements from $A[1, i]$ that reside in the first ℓ subtrees of T_v. These prefix sums are stored in $|T_v|$ bit-matrices, one matrix, M_i, for each $y_i \in T_v$. The ℓ'th row of bits in M_i is the number t_ℓ^i. The rows form a non-decreasing sequence of numbers by construction. The matrices are stored consecutively in an array A_v, i.e. M_i is stored before M_j if $i < j$, and the number of elements from $A[1, i]$ in T_v, is the position of M_i in A_v. Each matrix is stored in two different ways. In the first copy each row is stored in one word. In the second copy each matrix is divided into sections of $g = \lfloor \log n/f \rfloor = \Theta(\log^{1-\varepsilon} n)$ columns. The first section contains the first g bits of each of the f rows, and these are stored in one word. This is the g most significant bits of each prefix sum stored in the matrix. The second section contains the last three bits of the first section and then the following $g - 3$ bits [11], and so on. The reason for this overlap of three bits will become clear later. We think of each section as an $f \times g$ bit matrix.

For technical reasons, we ensure that the first column of each matrix only contains zero-entries by prepending a column of zeroes to all matrices before the division into sections.

An overview of the data structure is shown in Figure 2.

4.2 Range Selection Query

Given L, R and s, a query locates the s'th smallest element in $A[L, R]$. Consider the matrix M', whose ℓ'th row is defined as $t_\ell^R - t_\ell^{L-1}$ (in other words, $M' = M_R - M_{L-1}$ with row-wise subtraction). We compute the smallest ℓ such that the ℓ'th row in M' stores a number greater than or equal to s, which defines the subtree containing the s'th smallest element in T_v. In the following pages we describe how to compute ℓ without explicitly constructing the entire matrix M'.

The intuitive idea to guide a query in a given node, v, is as follows. Let K be the number of elements from $A[L, R]$ contained in T_v. We consider the

[11] Here and in the following sections we assume that ε and n are such that the overlap is strictly smaller than g.

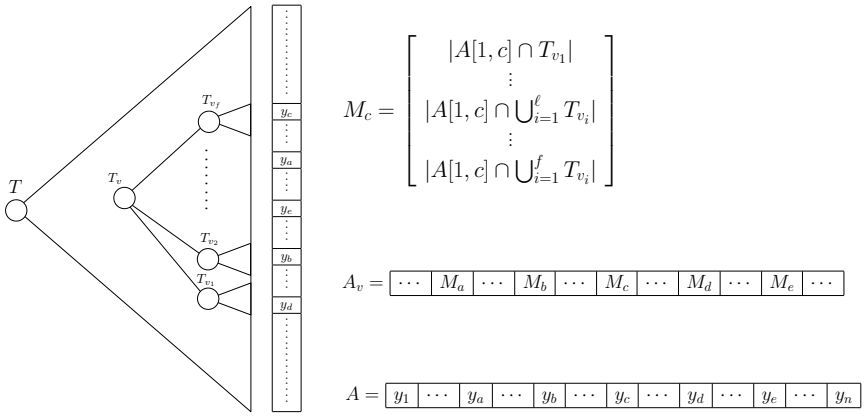


Fig. 2. A graphic overview of the data structure. As shown in the tree, $y_d < y_b < y_e < y_a < y_c$ and they are all contained in T_v. A concrete example of a matrix is shown in Figure 3.

section from M' containing the $\lceil \log K \rceil$'th least significant bit of each row. All the bits stored in M' before this section are zero and thus not important. Using word-level parallelism we find an interval $[\ell_1, \ell_2] \subseteq [1, f]$, which contains the indices to all rows of M' where the g bits in M' match the corresponding g bits of s, plus the following row. By definition, this interval contains the index of the subtree of T_v that contains the s'th smallest element in T_v. We then try to determine which of these subtrees contains the s'th smallest element. First, we consider the children of v defined by the endpoints of the interval, ℓ_1 and ℓ_2. Suppose neither of these contains the s'th smallest element in $A[L, R]$, and consider the subtree of T_v containing the s'th smallest element. This subtree then contains approximately a factor of 2^g fewer elements from $A[L, R]$ than T_v does, since the g most significant bits of the prefix sum of the row corresponding to this subtree are the same as the bits in the preceding row. In this case we determine ℓ in $O(\log \log n)$ time using a standard binary search. The point is that this can only occur $O(\log n/g)$ times, and the total cost of these searches is $O(f \log \log n) = O(\log^{\varepsilon} n \log \log n) = o(\log n / \log \log n)$. In the remaining nodes we use constant time.

There are several technical issues that must be worked out. The most important is that we cannot actually produce the needed section of M' in constant time. Instead, we compute an approximation where the number stored in the g bits of each row of the section is at most one too large when compared to the g bits of that row in M'. The details are as follows.

In a node $v \in T$ the search is guided using $M_{p(L)}$ and $M_{p(R)}$, where $p(L)$ and $p(R)$ are the maximal indices less than $L - 1$ and R respectively and $y_{p(L)}$ and $y_{p(R)}$ are contained in T_v. For clarity we use M_{L-1} and M_R for the description. A query maintains an index c, *initially one*, defining which section of the bit-

matrices is currently in use. We maintain the following invariant regarding the c'th section of M' in the remaining subtree: in M', all bits before the c'th section are zero, i.e., the important bits of M' are stored in the c'th section or to the right of it (in sections with higher index). For technical reasons, we ensure that the most important bit of the c'th section of M' is zero. This is true before the query starts since the first bit in each row of each stored matrix is zero.

We compute the approximation of the c'th section of M' from the c'th section of M_R and M_{L-1}. This approximation we denote $w^{L,R}$ and think of it as a $f \times g$ bit-matrix. Basically, the word containing the c'th section of bits from M_{L-1} is subtracted from the corresponding word in M_R. However, subtracting the c'th section of g bits of t_ℓ^{L-1} from the corresponding g bits of t_ℓ^R does not encompass a potential cascading carry from the lower order bits when comparing the result with the matching g bits of $t_\ell^R - t_\ell^{L-1}$, the ℓ'th row of M'. This means that in the c'th section, the ℓ'th row of M_{L-1} could be larger than the ℓ'th row of M_R. To ensure that each pair of rows is subtracted independently in the computation of $w^{L,R}$, we prepend an extra one bit to each row of M_R and an extra zero bit to each row of M_L to deal with cascading carries. Then we subtract the c section of M_{L-1} from the c'th section of M_R, and obtain $w^{L,R}$. After the subtraction we ignore the value of the most significant bit of each row in $w^{L,R}$ (it is masked out). After this computation, each row in $w^{L,R}$ contain a number that either matches the corresponding g bits of M', or a number that is one larger. Since the most important bit of the c'th section of M' is zero, we know that the computation does not *overflow*. An example of this computation is shown in Figure 3.

Searching $w^{L,R}$. Let $s_b = s_1, \ldots, s_g$ be the g bits of s defined by the c'th section, initially the g most important bits of s. If we had actually computed the c'th section of M', then only rows matching s_b and possibly the first row containing a larger number can define the subtree containing the s'th smallest element. However, since the rows can contain numbers that are one *too large*, we also consider all rows matching $s_b + 1$, and the first row storing a larger number. Therefore, the algorithm locates the first row of $w^{L,R}$ storing a number greater than or equal to s_b and the first row greater than $s_b + 1$. The indices of these rows we denote ℓ_1 and ℓ_2, and the subtree containing the s'th smallest element corresponds to a row between ℓ_1 and ℓ_2. Subsequently, it is checked whether the ℓ_1'th or ℓ_2'th subtree contains the s'th smallest element in T_v using the first copy of the matrices (where the rows are stored separately). If this is not the case, then the index of the correct subtree is between $\ell_1 + 1$ and $\ell_2 - 1$, and it is determined by a binary search. The binary search uses the first copy of the matrices. In the c'th section of M', the g bits from the $\ell_1 + 1$'th row represent a number that is at least $s_b - 1$, and the $(\ell_2 - 1)$'th row a number that is at most $s_b + 1$. Therefore, the difference between the numbers stored in row $\ell_1 - 1$ and $\ell_2 - 1$ in M' is at most two. This means that in the remaining subtree, the c'th section of bits from M' ($t_\ell^R - t_\ell^{L-1}$ for $1 \leq \ell \leq f$) is a number between zero and two. Since the following section stores the last three bits of the current section, the algorithm safely skips the current section in the remaining subtree, by increasing c by one, without violating the invariant: we need two bits to express a number between

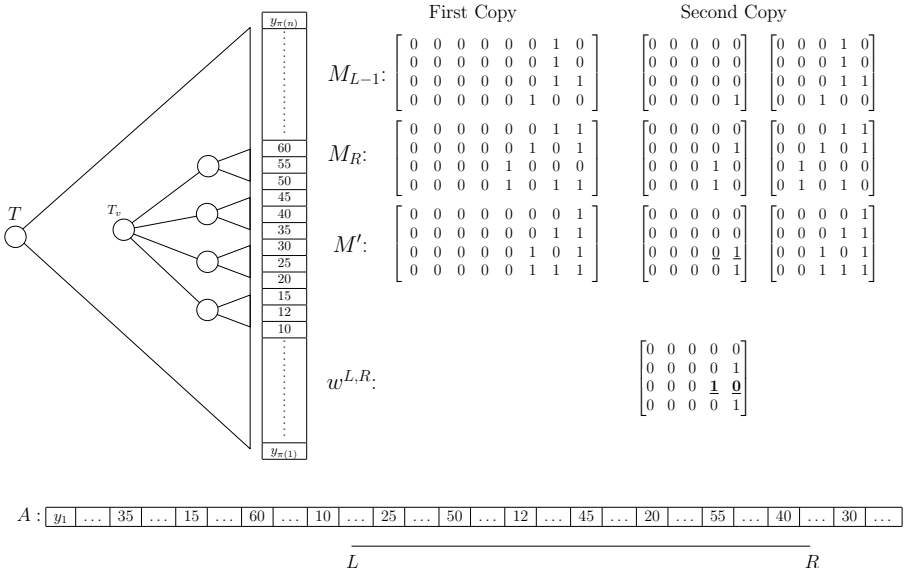


Fig. 3. A concrete example of the matrices used. In this example, $f = 4$ and $g = 5$. The figure shows the matrices M_L, M_R and M', how they appear when they are divided into sections. The figure also shows the $f \times g$ matrix $w^{L,R}$ a query produces. Notice that the third row of $w^{L,R}$ stores a number one larger than the corresponding bits in matching section of M' (underlined).

zero and two, and the third bit ensures that the most significant bit of the c'th section of M' is zero. After the subtree T_ℓ, containing the s'th smallest element, is located, s is updated as before: $s := s - (t^R_{\ell-1} - t^{L-1}_{\ell-1})$. Let $r_{L-1} = t^{L-1}_\ell - t^{L-1}_{\ell-1}$ be the number of elements from $A[1, L-1]$ in T_ℓ, and let $r_R = t^R_\ell - t^R_{\ell-1}$ be the number of elements from $A[1, R]$ contained in T_ℓ. In the subsequent node the algorithm uses the r_{L-1}'th and the r_R'th stored matrix to guide the search (fractional cascading).

In the next paragraph we explain how to determine ℓ_1 and ℓ_2 in constant time. Thus, if the search continues in the ℓ_1'th or ℓ_2'th subtree, the algorithm used $O(1)$ time in the node. Otherwise, a binary search is performed, which takes $O(\log f)$ time, but in the remaining subtree an additional section is skipped. An additional section may be skipped at most $\lceil 1 + \log n/(g-3) \rceil = O(f)$ times. When the search is guided using the last section there will not be any problems with cascading carries. This means that the search continues in the subtree corresponding to the first row of $w^{L,R}$ where the number stored is at least as large as s_b, and a binary search is never performed in this case. We conclude that a range selection query takes $O(\log n/\log\log n + f \log f) = O(\log n/\log\log n)$ time.

The data structure stores a matrix for each element on each level of the tree, and every matrix uses $O(f)$ of space. There are $O(\log n/\log\log n)$ levels giving a total space of $O(nf\log n/\log\log n) = O(n\log^{1+\varepsilon} n/\log\log n)$.

We briefly note that range rank queries can be answered quite simply in $O(\log n/\log\log n)$ time using a similar data structure: Given L, R and an element e in a rank query we use a linear space predecessor data structure (van Emde Boas tree [vEBKZ77]) that in $O(\log\log n)$ time yields the predecessor e_p of e in the sorted order of A. Then, the path from e_p to the root in T is traversed, and during this walk the number of elements from $A[L, R]$ in subtrees hanging off the path to the left are added up using the first copy of the bit matrices.

Lemma 1. *The data structure described uses $O(n\log^{1+\varepsilon} n/\log\log n)$ words of space and supports range selection and range rank queries in $O(\log n/\log\log n)$ time.*

Determining ℓ_1 and ℓ_2. The remaining issue is to compute ℓ_1 and ℓ_2. A query maintains a search word, s_w, that contains f independent blocks of the g bits from s that corresponds to the c'th section. Initially, this is the g most important bits of s. To compute s_w we store a table that maps each g-bit number to a word that contains f copies of these g bits. After updating s we update s_w using a bit-mask and a table look-up. A query knows $w^{L,R} = v_1^1, \ldots, v_g^1, \ldots, v_1^d, \ldots, v_g^d$ and s_w which is $s_b = s_1, \ldots, s_g$ concatenated f times. The g-bit block $v_1^\ell, \ldots, v_g^\ell$ from $w^{L,R}$ we denote $w_\ell^{L,R}$ and the ℓ'th block of $s_1, \ldots, s_g$ from s_w we denote s_w^ℓ. We only describe how to find ℓ_1, since ℓ_2 can be found similarly. Remember that ℓ_1 is the index of the first row in $w^{L,R}$ that stores a number greater than or equal to s_b. We make room for an extra bit in each block and make it the most significant. We set the extra bit of each $w_\ell^{L,R}$ to one and the extra bit of each s_w^ℓ to zero. This ensures that $w_\ell^{L,R}$ is larger than s_w^ℓ, for all ℓ, when both are considered $g+1$ bit numbers. s_w is subtracted from $w^{L,R}$ and because of the extra bit, this operation subtracts s_w^ℓ from $w_\ell^{L,R}$, for $1 \leq \ell \leq f$, independently of the other blocks. Then, all but the most significant (fake) bit of each block are masked out. The first one-bit in this word reveals the index ℓ of the first block where $w_\ell^{L,R}$ is at least as large as s_w^ℓ. This bit is found using complete tabulation.

4.3 Getting Linear Space

In this section we reduce the space usage of our data structure to $O(n)$ words. Let $t = \lceil f\log n\rceil$. In each node, the sequence of matrices/elements is divided into chunks of size t and only the last matrix of each chunk is explicitly stored. For each of the remaining elements in a chunk, $\lceil\log f\rceil$ bits are used to describe in which subtree it resides. The description for $d = \lfloor\log n/\lceil\log f\rceil\rfloor$ elements are stored in one word, which we denote a direction word. Prefix sums are stored after each direction word summing up all previous direction words in the chunk. After each direction word we store f prefix sums summing up all the previous direction words in the chunk. Since each chunk stores the directions of t elements,

at most $\lceil f\lceil \log t\rceil / \log n\rceil = O(1)$ words are needed to store these f prefix sums. We denote these prefix words. The data structure uses $O(n)$ words of space.

Range Selection Query. The query works similarly to above. The main difference is that we do not use the matrices M_{L-1} and M_R to compute $w^{L,R}$ since they are not necessarily stored. Instead, we use two matrices that are stored which are *close* to M_{L-1} and M_R. The direction and update words enable us to exactly compute any row of M_R and M_{L-1} in constant time (explained below). Therefore, the main difference compared to the previous data structure is that the potential difference between the computed approximation of the c'th section of M' and the c'th section of M' is marginally larger, and for this reason the overlap between blocks is increased to four.

In a node $v \in T$ a query is guided as follows. Let r_L and r_R be the number of elements from $A[1, L-1]$ and $A[1, R]$ in T_v respectively, and let $L' = \lfloor r_L/t \rfloor$ and $R' = \lfloor r_R/t \rfloor$. We use the matrices contained in the L''th and R''th chunk respectively to guide the search, and we denote these M_a and M_b. Since v stores a matrix for every t'th element in T_v, we have $t^R_\ell - t^b_\ell \leq t$ for any $1 \leq \ell \leq f$. Now consider the matrix $\bar{M} = M_b - M_a$, the analog to M' for a, b. Then the ℓ'th row of $\bar{M}$ is at most t smaller than or larger than the ℓ'th row of M'. If we add the difference between the ℓ'th row of $\bar{M}$ and M' to the ℓ'th of $\bar{M}$ and ignore cascading carries then only the least $\lceil \log t\rceil = \lceil (1+\varepsilon)\log\log n\rceil$ significant bits change. Stated differently, unless we use the last section, the number stored in the ℓ'th row of the c'th section of $\bar{M}$ is at most one from the corresponding number stored in M'.

We can obtain the value of any row in M_R as follows. For each ℓ between 1 and f, we compute how many of the first $r_R - R't$ elements represented in the $R'+1$'th chunk that are contained in the first ℓ children from the direction and prefix words. These are the elements considered in M^R but not in M^b. Formally, the $p = \lfloor (r_R - R't)/d \rfloor$'th prefix word stores how many of the first pd elements from the chunk reside in the first ℓ children for $1 \leq \ell \leq f$. Using complete tabulation on the following direction word, we obtain a word storing how many of the following $r_R - R't - pd$ elements from the chunk reside in the first ℓ children for all ℓ, $1 \leq \ell \leq f$. Adding this to the p'th prefix word yields the difference between the ℓ'th row of M_R and M_b, for all $1 \leq \ell \leq f$. The difference between M_a and M_{L-1} can be computed similarly. Thus, any row of M_R and M_{L-1}, and the last section of M_R and M_{L-1} can be computed in constant time.

If the last section is used, it is computed exactly in constant time and the search is guided as above. Otherwise, we compute the difference between each row in the c'th section of M_a and M_b, yielding $w^{a,b}$. As above, the computation of $w^{a,b}$ does not consider cascading carries from lower order bits and for this reason the ℓ'th row of $w^{a,b}$ may be one too large when compared to the same bits in $\bar{M}$. Furthermore, the number stored in the ℓ'th row of $\bar{M}$ could be one larger or one smaller than the corresponding bits of M'.

We locate the first row of $w^{a,b}$ that is at least $s_b - 1$ and the first row greater than $s_b + 2$ as above, and the subtree we are searching for is defined by a row between these two, and if it is none of these, a binary search is used to determine

it. In this case, by the same arguments as earlier, each row in the c'th section of M' in the remaining subtree, represents a number between zero and six. Since we have an overlap of four bits between sections, we safely move to the next section after every binary search.

Range rank queries are supported similarly to above.

Lemma 2. *The data structure described uses linear space and supports range selection and range rank queries in $O(\log n / \log\log n)$ time.*

4.4 Construction in $O(n \log n)$ time

In this section we describe how to construct the linear space data structure from the previous section in $O(n \log n)$ time. We sort the input elements, and build a tree with fan-out f on top of them. Then we construct the nodes of the tree level by level starting with the leaves.

In each node v, we scan the elements in T_v in chunks of size t ordered by their positions in A, and write the direction and prefix words. After processing each chunk, we construct its corresponding matrix. We use an array C of length f, there is one entry in C for each child of v. We scan the elements by their position in A, and if y_i is contained in the ℓ'th subtree then we add one to $C[\ell]$, and append ℓ to the description word using $\lceil \log f \rceil$ bits. After $\lfloor \log n / \lceil \log f \rceil \rfloor$ steps we build a prefix word by making an array D such that $D[i] = \sum_{\ell=1}^{i} C[i]$ and store it in $O(1)$ words. After t steps we store D as a matrix, i.e., the ℓ'th row of the matrix stores the number $D[\ell]$. We make the second copy of the matrix by repeatedly extracting the needed bits from the first copy. For the next chunk we reset C and repeat. When we build the next matrix we add the numbers stored in D to the previously built matrix, and get the first copy. The second copy is computed as before.

Merging the f lists of elements from the children takes $O(|T_v| \log f)$ time. Adding a number to a direction word takes constant time. Constructing a prefix word takes $O(f)$ time and we do that for every $O(\log(n)/\log f)$'th element. Constructing the matrices takes $O(f)$ time for the first copy and $O(f^2)$ time for the second, and we do this for every $O(f \log n)$'th element. We conclude that we use $O(n \log f) = O(n \log\log n)$ time per level of the tree, and there are $O(\log n / \log\log n)$ levels.

Theorem 3. *The data structure described uses linear space, supports range selection and range rank queries in $O(\log n / \log\log n)$ time, and it can be constructed in $O(n \log n)$ time.*

5 Dynamic Range Medians

In this section, we consider a dynamic variant of the RMP, where a set of points, $S = \{(x_i, y_i)\}$, is maintained under insertions and deletions. A query is given values L, R and an integer s and returns the point with the s'th smallest y value among the points in S with x-value between L and R.

Using standard dynamization techniques (a weight-balanced tree, where associated data structures are rebuilt from scratch when a rotation occurs), the simple data structure of Section 2 yields a dynamic solution with $O(\log n)$ query time, $O(\log^2 n)$ amortized update time and $O(n \log n)$ space (for details, see [GS09]). In the rest of this section, we describe a dynamic variant of the data structure in Section 4, which uses $O(n \log n / \log\log n)$ space and supports queries and updates in $O((\log n / \log\log n)^2)$ time, worst case and amortized respectively. It is based on the techniques just mentioned, but some work is needed to combine these techniques with the static data structure.

We store the points from S in a weight-balanced search tree [NR72,AV03], ordered by y-coordinate. In each node of the tree we maintain the bit-matrices, defined in the static structure, dynamically using a weight-balanced search tree over the points in the subtree, ordered by x-coordinate. The main issue is efficient generation of the needed sections of the bit-matrices used by queries. The quality of the approximation is worse than in the static data structure, and we increase the overlap between sections to $O(\log\log n)$. Otherwise, a search works as in the static data structure.

We note that using this dynamic data structure for the one-dimensional RMP, we can implement a two-dimensional median filter, by scanning over the image, maintaining all the pixels in a strip of width r. In this way, we obtain a running time of $O(\log^2 r / \log\log r)$ per pixel, which is a factor $\log\log r$ better than the state-of-the-art solution for this problem [GW93].

5.1 Structure

The data structure is a weight-balanced B-tree T with $B = \lceil \log^\varepsilon n \rceil$, where $0 < \varepsilon < \frac{1}{2}$, containing the n points in S in the leaves, ordered by their y-coordinates. Each internal node $v \in T$ stores a *ranking tree* R_v. This is also a weight-balanced B-tree with $B = \lceil \log^\varepsilon n \rceil$, containing the points stored in T_v, ordered by their x-coordinates. Each leaf in a ranking tree stores $\Theta(B^2)$ elements. Since the data structure depends on n, it is rebuilt every $\Theta(n)$ updates. Let $h = O(\log_B n)$ be the maximal height the trees can get and $f = O(B)$ the maximal fan-out of a node (until the next rebuild).

Let v be a node in T and denote the $a \leq f$ subtrees of v by $T_1, \ldots, T_a$. The ranking tree R_v stored in v is structured as follows. Let u be a node in R_v and denote its $b \leq f$ subtrees by $R_1, \ldots, R_b$. The node u stores a bit-matrices $M_1^u, \ldots, M_a^u$. In the matrix M_q^u the p'th row stores the number of elements from $\bigcup_{1 \leq i \leq q} R_i$ that are contained in $\bigcup_{1 \leq i \leq p} T_i$. Additionally, u also stores up to B^2 updates, each describing from which subtree of v the update came, from which subtree of $u \in R_v$ the update came, and whether it was an insert or a delete. These updates are stored in $O(B^2 \log B / \log n) = O(1)$ words.

As in the static case, each matrix is stored in two ways. In the first copy each row is stored in one word. For the second copy, each matrix is divided into sections of $g = \lfloor \log n / f \rfloor$ bits, and each section is stored in one word. These sections have $\lceil \log(2h+2) \rceil + 1 = O(\log\log n)$ bits of overlap.

Finally, a linear space dynamic predecessor data structure [BF02] containing the $|T_v|$ elements in R_v, ordered by x, is also stored.

If we ignore the updates stored in update blocks, each matrix M_j, as defined in the static data structure, corresponds to the row-wise sum of at most h matrices from R_v. Consider the path from the root of R_v to the leaf storing the point in T_v with maximal x coordinate smaller than or equal to j, i.e. the predecessor of j in T_v when the points are ordered by x their coordinates. Starting with the zero matrix we add up matrices as follows. If the path continues in the ℓ'th subtree at the node u, then we add the $\ell - 1$'th matrix stored at u ($M^u_{\ell-1}$) to the sum. Summing up, the p'th row of the computed matrix is the number of points, $(x, y) \in T_v$ where $x \leq j$, contained in the first p subtrees of T_v, and this is exactly the definition of M_j in the static data structure.

5.2 Range Selection Query

Given values L, R and an index s in a query, we perform a topdown search in T. As in the static data structure, we maintain an index c that defines which section of the matrices is currently in use and approximate the c'th section of the matrix M'.

In a node $v \in T$, we compute an approximation, $w^{L,R}$, of the c'th section of M' from the associated ranking tree as follows. First, we locate the leaves of the ranking tree containing the points with maximal x-coordinates less than $L - 1$ and R using the dynamic predecessor data structure. Then we traverse the paths from these two leaves in parallel until the paths merge, which must happen at some node because both paths lead to the root. We call them the left and right path.

Initially, we set $w^{L,R}$ to the zero matrix. Assume that we reach node u_L from its p_L'th child on the left path and the node u_R from its p_R'th child on the right path. Then we subtract the c'th section from the $p_L - 1$'th matrix stored in u_L from the c'th section of the $p_R - 1$'th matrix stored in u_R. The subtraction of sections is performed as in the static data structure. We add the result to $w^{L,R}$. If the paths are at the same node we stop, otherwise, we continue up the tree.

Since we do not consider cascading carries, each subtraction might produce a section containing rows that are one to large. Similarly, we add a section to $w^{L,R}$ in each step, which ignores cascading carries from the lower order bits, giving numbers that could be one to small. Furthermore, we ignore up to B^2 updates in each step.

Now consider the difference between $w^{L,R}$ and the c'th section of M'. First of all we have ignored up to B^2 updates in two nodes on each level of T, which means that in the matrices we consider the combined number that is stored in the ℓ'th row may differ by up to $2hB^2$ compared to M'. As in the static data structure, unless we are using the last section, this only affects the number stored in each row by one.

Furthermore, in the computation of $w^{L,R}$ we do h subtractions and h additions of sections and each of these operations ignores the lower order bits.

Combining this with the ignored updates, we get that each row of $w^{L,R}$ is at most h smaller or larger than the corresponding value in M'.

Let s_b be the number defined by the c'th section of g bits from s. The query locates the first row that is at least $s_b - h$ and the first row greater than $s_b + h$. Then the algorithm checks whether either of these corresponds to the subtree containing the answer. If not, the c'th section of the remaining matrices store a number between zero and $2h$, and for this reason the overlap between sections is set to $\lceil \log(2h+1) \rceil + 1 = O(\log \log n)$ bits. In this case we use a binary search that in each step traverses R_v as above (path up from leaves containing the maximal x-coordinate less than $L-1$ and R) to compute the needed rows of M' exactly using the first copy of the stored matrices and the information stored in the update words. Extracting information from update words is done by complete tabulation on each of the $O(1)$ update words.

If we are using the last section, there will not be problems with the lower order bits in the additions and subtractions of sections. We add the changes stored in the update words on the paths and obtain the last section of M' exactly, and a binary search is never performed in this case. As before, the current section is skipped in the remaining subtree in this case.

5.3 Updates

An element e is inserted into the data structure as follows. First, e is added to T. If a node in T splits, two new structures are built from the existing one, and the parent node is rebuilt. Then, we traverse path from e to the root. In each node $v \in T$ on this path, e is inserted into the ranking tree R_v, and the dynamic predecessor data structure. If a node splits in R_v, two new nodes are constructed from the old one, and the parent rebuilt. After R_v has been updated, e is inserted in each node in R_v on the path from e to the root, by appending a description of the update to the update words. When B^2 updates have been appended to a node in a ranking tree, all bit-matrices in this node are recomputed and the update words are cleared. Deletes are handled similarly, except that updates to the base tree and ranking trees are handled using global rebuilding, e.g. deleted nodes are marked and after $n/2$ updates the tree is completely rebuild.

Theorem 4. *The data structure uses $O(n \log n / \log \log n)$ words of space and queries and updates are supported in $O((\log n / \log \log n)^2)$ time worst case and amortized respectively.*

Proof. The height of T is $O(\log n / \log B) = O(\log n / \log \log n)$ and in each node on a path we traverse a ranking tree of height $O(\log n / \log B) = O(\log n / \log \log n)$. We do a binary search $O(\log n / B)$ times and each binary search takes $O(\log B \log n / \log \log n) = O(\log n)$ time. The combined time for all binary searches is $O(\log^2 n / B) = o((\log n / \log \log n)^2)$ time.

Updating the base tree T (splitting nodes) takes $O(B)$ time amortized per update. Updating a ranking also takes $O(B)$ time amortized per update and $O(\log n / \log B)$ ranking trees are changed in each update. Furthermore, an update is appended to a node on each level of a ranking tree, on each level of T.

This means that $O((\log n/\log\log n)^2)$ update words are changed. The matrices stored in a node in a ranking tree can be recomputed in $O(f^2) = O(B^2)$ time. This amounts to $O(1)$ time amortized per update added to the node. Each element defines a node in a ranking tree on each level of T. A node in a ranking tree stores at most f bit-matrices, each storing f numbers, and up to B^2 updates, each using $O(\log B)$ bits of space. Thus, a node in the ranking tree uses $O(f^2 + B^2 \log B/\log n) = O(B^2)$ words of space, and each ranking tree, R_v, contains $O(|T_v|/B^2)$ nodes.

6 Lower Bound for Dynamic Data Structures

In this section we describe a reduction from the marked ancestor problem to a dynamic range median data structure. In the marked ancestor problem the input is a complete tree of degree b and height h. An update marks or unmarks a node of the tree, initially all nodes are unmarked. A query is provided a leaf v of the tree and must return whether there exists a marked ancestor of v. Let t_q and t_u be the query and update time for a marked ancestor data structure. Alstrup et al. proved the following lower bound trade-off for the problem: $t_q = \Omega(\frac{\log n}{\log(t_u w \log n)})$, where w is the word size [AHR98].

Reduction. Let T denote a marked ancestor tree of height h and degree b. We associate two pairs of elements with each node v in T, which we denote *start-mark* and *end-mark*. We translate T into an array of size $4|T|$ by a recursive traversal of T, where for each node v, we output its start-mark, then recursively visit each of v's children, and then output v's end-mark. Start-marks are used to mark a node, and end-marks ensure that markings only influence the answer for queries in the marked subtree. When a node v is unmarked, start-mark=end-mark=(0,1) and when v is marked, start-mark is set to (1,1) and end-mark to (0,0).

A marked ancestor query for a leaf v is answered by returning yes if and only if the range median from the subarray ranging from the beginning of the array to the start-mark element associated with v is one. If zero nodes are marked, the array is of the form $[0, 1, 0, 1, \ldots, 0, 1]$. Since the median in any range that can be considered by a query is zero, any marked ancestor query returns no. If v or one of its ancestors is marked there will be more ones than zeros in the range for v, and the query answers yes. A node u that is not an ancestor of v has both its start-mark and end-mark placed either before v's marks or after v's marks, and independently of whether u is marked or not, it contributes an equal number of zeroes and ones to v's query range. Since the reduction requires an overhead of $O(1)$ for both queries and updates we get the following lower bound.

Theorem 5. *Any data structure that supports updates in $O(\log^{O(1)} n)$ time uses $\Omega(\log n/\log\log n)$ time to support a range median query.*

7 Higher Dimensions

Since our algorithm from Section 2 decomposes the values rather than the positions of elements, it can be naturally generalized to higher dimensional point sets. We obtain an algorithm that needs $O(n \log k)$ preprocessing time plus the time for supporting range counting queries on each level. The amortized query time is the time for $O(\log n)$ range counting queries. Note that query ranges can be specified in any way we wish: (hyper)-rectangles, circles, etc., without affecting the way we handle values. For example, using the data structure for 2D range counting from [JMS04] we obtain a data structure for the 2D rectangular range median problem that needs $O(n \log n \log k)$ preprocessing time, $O(n \log k / \log\log n)$ space, and $O(\log^2 n / \log\log n)$ query time. This not only applies to 2D arrays consisting of n input points but to arbitrary two-dimensional point sets with n elements.

Unfortunately, further improvements, e.g. to logarithmic query time, seem difficult. Although the query range is the same at all levels of recursion, fractional cascading becomes less effective when the result of a rectangular range counting query is defined by more than a constant number of positions within the data structure because we would have to follow many forwarding pointers. Also, the array contraction trick that allowed us to use dense bit arrays in Section 3 does not work anymore because an array with half the number of bits need not contain any empty rows or columns.

Another indication that logarithmic query time in two dimensions might be difficult to achieve is that there has been intensive work on the more specialized median-filtering problem in image processing where we ask for *all* range medians with query ranges that are squares of size $(2r+1) \times (2r+1)$ in an image with n pixels. The best previous algorithms known here need time $\Theta\big(n \log^2 r\big)$ [GW93] unless the range of values is very small [PH07,CWE07]. Our result above improves this by a factor $\log\log r$ (by applying the general algorithm to input pieces of size $3r \times 3r$) but this seems to be of theoretical interest only.

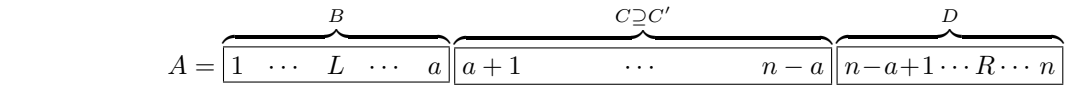
8 Towards Constant Query Time

Using $O(n^2)$ space, we can trivially precompute all medians so that the query time becomes constant. This space requirement is reduced by somewhat less than a logarithmic factor in [KMS05,Pet08]. An interesting question is whether we can save more than a polylogarithmic factor. We now outline an algorithm that needs space $O(n^{3/2})$ (machine words), preprocessing time $O(n^{3/2} \log n)$ and achieves constant query time on the average, i.e., the expected query time is constant for random inputs[12]. Note that the results of this section only apply to computing range medians rather than general range selection.

We first consider median queries for a range $[L, R]$ where $L \leq a+1$ and $R \geq n-a$ with $a \in \Theta(\sqrt{n})$, i.e., the range contains a large middle part $C = A[a+$

[12] The analysis is for inputs with distinct elements where every permutation of ranks is equally likely. The queries can be arbitrary.

$1, n-a]$ of the input array. Furthermore let $B = A[1, a]$ and $D = A[n-a+1, n]$.



If the median value v of $A[L, R]$ comes from C, then its rank within C must be in $[\lceil \frac{n}{2} \rceil - 2a, \lceil \frac{n}{2} \rceil]$ because the median of the elements in C has rank $\lceil \frac{n}{2} \rceil - a$, and adding at most $2a$ elements outside C can increase or decrease the rank of the median by at most a. The basic idea is to precompute a sorted array $C'[1, 2a+1]$ of these *central elements*. The result of a query then only depends on $A[L, a]$, C', and $A[n-a+1, R]$. For a start, let us assume that all elements in B and D are either smaller than $C'[1]$ or larger than $C'[2a+1]$. Suppose $A[L, a]$ and $A[n-a+1, R]$ contain s_l and s_r values smaller than $C'[1]$, respectively. Then the median of $A[L, R]$ is $C'[a+1+\lfloor \frac{b-s}{2} \rfloor]$, where $s := s_l + s_r$ and $b := R-L+1+2a-n-s$ (b is the number of values larger than $C'[2a+1]$ in $A[L, a]$ and $A[n-a+1, R]$). Note that s_l can be precomputed for all possible values of L using time and space $O(\sqrt{n})$ (and the same is true for s_r and R). For general contents of B and D, elements in B and D with value between $C'[1]$ and $C'[2a+1]$ are stored explicitly. Since, for a random input, each element from B and D has probability $\Theta(1/a)$ to lie within this range, only $O(1)$ elements have to be stored on the average. During a query, these extra elements are scanned[13] and those with position within $[L, R]$ are moved to a sorted extra array X. The median of $A[L, R]$ is then the element with rank $a+1+\lfloor \frac{b-s}{2} \rfloor + \frac{|X|}{2}$ in $C' \cup X$. Equivalently, we can take the element with rank $|X|/2$ in $C'[a+1+\lfloor \frac{b-s}{2} \rfloor, a+1+\lfloor \frac{b-s}{2} \rfloor + |X|] \cup X$. We are facing a selection problem from two sorted arrays of size $|X|$ which is possible in time $O(\log |X|)$ (see e.g. [VSIR91]), i.e., $O(1)$ on the average.

To generalize for arbitrary ranges, we can cover the input array A with subarrays obeying the above rules in such a way that every possible query can be performed in one subarray. Here is one possible covering scheme: In category $i \in [1, \lfloor \sqrt{n} \rfloor]$ we want to cover all queries with ranges $R-L+1$ in $[i^2, i^2+2i]$. Note that these ranges cover all of $[1, n]$ since $i^2+2i+1 = (i+1)^2$, i.e. subsequent range intervals $[i^2, i^2+2i]$ and $[(i+1)^2, (i+1)^2+2(i+1)]$ are contiguous. In category i, we use subarrays of size $2i + (i^2 - i) + 2i$ such that the central part C of the j-th subarray starts at position $ij+1$ for $j \in [0, n/i - i]$.[14] A query $[L, R]$ is now handled by category $i = \lfloor \sqrt{R-L+1} \rfloor$.[15] It remains to determine the number j of the subarray within category i. If $R-L+1 \in [i^2, i^2+1]$ we use $j = \lfloor L/i \rfloor$, otherwise $j = \lfloor L/i \rfloor + 1$. In both cases, L is within part B of the j-th subarray and R is within part D of the j-th subarray.

[13] An algorithm that is more robust for nonrandom inputs could avoid scanning by using a selection algorithm working on one sorted array and a data structure that supports fast range-rank queries (see also Section 3). This way, we would get an algorithm running in time logarithmic in the number of extra elements.

[14] We pad the input array A on both sides with random values in order to avoid special cases.

[15] Note that we can precompute all required square roots if desired.

A subarray of category i needs space $O(i)$ and there are $O(n/i)$ arrays from category i. Hence in total, the arrays of category i need space $O(n)$. Since $O(\sqrt{n})$ categories suffice to cover the entire array, the overall space consumption is $O(n^{3/2})$. Precomputing the arrays of category i can be done in time $O(n \log i)$ by keeping the elements of the current subarray in a search tree sorted by element values. Finding the central elements for the next subarray then amounts to i deletions, i insertions and one range reporting query in this search tree. This takes time $O(i \log i)$. The remaining precomputations can be performed in time $O(i)$. Summing over all categories yields preprocessing time $O(n^{3/2} \log n)$.

The above bounds for space and preprocessing time are deterministic worst case bounds. The average space consumption can be reduced by a factor $\log n$: First, instead of precomputing the counts for s_l and s_r they can be computed using a bit array with fast rank operation (see also Section 3). Second, instead of blindly storing the worst case number of $2a+1$ central elements, we may only store the elements actually needed by any query. This number is upper bounded by the number of elements needed for queries of the form $[a+1, R]$ plus the number of elements needed for queries of the form $[L, n-a]$. Let us consider the position of the median in queries of the form $[a+1, R]$ as a function of R. This position (ignoring rounding) performs a random walk on the line with step-width 1/2. In $a = O(\sqrt{n})$ steps, the expected maximum distance from its starting point reached by such a random walk is $O(\sqrt{\sqrt{n}}) = O(n^{1/4})$. Queries of the form $[L, n-a]$ behave analogously.

9 Conclusion

We have presented improved upper bounds for the range median problem. Except for the results in Section 8, they generalize to finding the element of any given rank inside the query range. In the comparison model, the query time of our solution is asymptotically optimal for $k \in O(n)$. For larger values of k, our solution is at most a factor $\log n$ from optimal. In a very restricted pointer model where no arrays are allowed, our solution is optimal for all k. Moreover, in the RAM model, our data structure requires only $O(n)$ space, which is clearly optimal. It is open whether the term $O(k \log n / \log \log n)$ in the query time could be reduced to $O(k)$ in the RAM model when k is sufficiently large. The interesting range here is when k lies between $\Theta(n)$ and $\Theta(n^2)$. Making the data structure dynamic adds an amortized factor $\log n$ to the query time in the pointer machine model, or $\log n / \log \log n$ in the RAM model. We obtain all these bounds also for the range rank problem.

Given the simplicity of some of our data structures from Section 3, a practical implementation would be easily possible. To avoid the large constants involved when computing medians for recursively splitting the array, one could consider using a randomized scheme for selecting pivots, which works well in practice.

It would be interesting to find faster solutions for the dynamic RMP or the two-dimensional (static) RMP: Either would lead to a faster median filter for images, which is a basic tool in image processing [Wei06].

References

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 534–543, Washington, DC, USA, 1998. IEEE Computer Society.

[AV03] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.

[BFP+72] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Linear Time Bounds for Median Computations. In *4th Annual Symp. on Theory of Computing (STOC)*, pages 119–124, 1972.

[BJ09] Gerth Stølting Brodal and Allan Grønlund Jørgensen. Data structures for range median queries. In *Proc. 20th Annual International Symposium on Algorithms and Computation*, volume 5878 of *LNCS*. Springer Verlag, Berlin, 2009.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[Cla88] David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1988.

[CWE07] David Cline, Kenric B. White, and Parris K. Egbert. Fast 8-bit median filtering based on separability. *IEEE International Conference on Image Processing (ICIP)*, 5:281–284, 2007.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[GBT84] Harold N. Gabow, Jon L. Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, New York, NY, USA, 1984. ACM.

[GS09] Beat Gfeller and Peter Sanders. Towards optimal range medians. In *36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *LNCS*, pages 475–486. Springer, 2009.

[GW93] Joseph Gil and Michael Werman. Computing 2-d min, median, and max filters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(5):504–507, May 1993.

[HPM08] Sariel Har-Peled and S. Muthukrishnan. Range Medians. In *16th Annual European Symp. on Algorithms (ESA)*, volume 5193 of *LNCS*, pages 503–514. Springer, 2008.

[JMS04] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *19th International Symp. on Algorithms and Computation (ISAAC)*, volume 3341 of *LNCS*, pages 558–568. Springer, 2004.

[KMR88] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17(5):883–902, 1988.

[KMS05] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range Mode and Range Median Queries on Lists and Trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[Nek09] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

[NR72] Jürg Nievergelt and Edward M. Reingold. Binary Search Trees of Bounded Balance. In *4th Annual Symposium on Theory of Computing (STOC)*, pages 137–142. ACM, 1972.

[OS06] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *The Computing Research Repository (CoRR)*, abs/cs/0610001, 2006.

[Păt07] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proc. 39th ACM Symposium on Theory of Computing*, pages 40–46, 2007.

[Păt08] Mihai Pătraşcu. (Data) STRUCTURES. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 434–443, 2008.

[Pet08] Holger Petersen. Improved Bounds for Range Mode and Range Median Queries. In *34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 418–423, 2008.

[PG09] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

[PH07] Simon Perreault and Patrick Hébert. Median filtering in constant time. *IEEE Transactions on Image Processing*, 16(9):2389–2394, Sept. 2007.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[VSIR91] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):171–177, 1991.

[Wei06] Ben Weiss. Fast median and bilateral filtering. *ACM Transactions on Graphics*, 25(3):519–526, 2006.

[Yao82] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *14th Annual Symposium on Theory of Computing (STOC)*, pages 128–136, 1982.

[Yao85] Andrew Chi-Chih Yao. On the Complexity of Maintaining Partial Sums. *SIAM J. Comput.*, 14(2):277–288, 1985.