Data Structures for Range Median Queries

Gerth Stølting Brodal and Allan Grønlund Jørgensen

MADALGO [⋆], Department of Computer Science, Aarhus University, Denmark.
{gerth,jallan}@cs.au.dk

Abstract. In this paper we design data structures supporting *range median* queries, i.e. report the median element in a sub-range of an array. We consider static and dynamic data structures and batched queries. Our data structures support *range selection* queries, which are more general, and dominance queries (*range rank*). In the static case our data structure uses linear space and queries are supported in $O(\log n/\log\log n)$ time. 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.

1 Introduction

The median of a set S of size n is an element in S that is larger than $\lfloor \frac{n-1}{2} \rfloor$ other elements from S and smaller than $\lceil \frac{n-1}{2} \rceil$ other elements from S. In the range median problem one must preprocess an input array A of size n into a data structure that given indices i and j, $1 \leq i \leq j \leq n$, a query must return an index i', $i \leq i' \leq j$, such that $A[i']$ is the median of the elements in the subarray $A[i,j] = [A[i], A[i+1], \ldots, A[j]]$. This problem is considered in [1–5]. In the batched case, the input is an array of size n and a set of k queries, $(i_1, j_1), \ldots, (i_k, j_k)$, and the output is the answer to these k queries [6]. Range median queries are naturally generalized to *range selection*, given indices i, j and s, return the index of the s'th smallest element in $A[i,j]$. A related problem is *range dominance* (or *range rank*) queries, given indices i, j and a value e, return the number of elements from $A[i,j]$ that are less than or equal to e (dominated by e). This corresponds to 3-sided range counting queries for a set of points.
Previous Work. Previously, the best linear space data structure supported range selection queries in $O(\log n)$ time [4, 5]. In the dynamic case the only known data structure uses $O(n\log n)$ space and supports updates and queries in $O(\log^2 n)$ time [4]. For dominance queries, linear space data structures supporting queries in $O(\log n/\log\log n)$ time is known, as well as a matching lower bound [7–9]. In the dynamic case [10] describes an $O(n)$ space data structure that supports dominance queries in $O((\log n/\log\log n)^2)$ time and updates in $O(\log^{9/2} n/(\log\log n)^2)$ time. A query lower bound of $\Omega((\log n/\log\log n)^2)$ for data structures with $O(\log^{O(1)} n)$ update time is proved in [7].
Our Results. In this paper we use the RAM model of computation with word-size $\Theta(\log n)$. Our data structures use the same basic approach as in [4]. We design a static linear space data structure that supports both range selection and

[⋆] Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

range rank queries in $O(\log n/\log\log n)$ time. This is the best known for range median data structures using $O(n\log^{O(1)} n)$ space, and for range dominance queries this is optimal. 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. For dominance queries this query time is optimal. 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 [11], leaving a significant gap to the achieved upper bound. With our static data structure we improve the $O(n\log k + k\log n)$ time bound for the batched range median problem achieved in [4] to $O(n\log k + k\log n/\log\log n)$ time. If $k > \sqrt{n}$ we construct our static data structure in $O(n\log n) = O(n\log k)$ time and perform k queries. This takes $O(n\log n + k\log n/\log\log n) = O(n\log k + k\log n/\log\log n)$ time. If $k < \sqrt{n}$ then $O(n\log k)$ time is already achieved by [6, 4].

2 Simple Range Selection Data Structure

In this section we describe the data structure of Gfeller and Sanders [4], which uses linear space and supports queries in $O(\log n)$ time. First, we describe a data structure that uses $O(n\log n)$ space and supports queries in $O(\log n)$ time. Then the space is reduced to $O(n)$ using standard techniques. The main idea is the following. Sort the input elements and place them in the leaves of a binary search tree. Consider a search for the s'th smallest element in $A[i,j]$. If the left subtree of the root contains s or more elements from $A[i,j]$ then it contains the s'th smallest element from $A[i,j]$. If not, it is in the right subtree. We augment each node of the tree with prefix sums such that the number of elements from $A[1,j]$ contained in the left subtree can be determined for any j, and we use fractional cascading [12] to avoid a search for the needed prefix sums in each node.

2.1 Basic Structure

Let $A = [y_1,\ldots,y_n]$ be the input array. We sort A and build a complete binary search tree T that stores the n elements in the leaves in sorted order. We introduce the following notation. 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. The *x-predecessor* of an index (x-coordinate) i in T_v is the largest index i' such that $i' \le i$ and $y_{i'} \in T_v$. If no such index exists the x-predecessor of i is zero. The *x-rank* of an index i in T_v is the number of elements from $A[1,i]$ contained in T_v. An x-rank is essentially a prefix sum. If we know the x-rank of $i-1$ and j in T_v, we know the number of elements from $A[i,j]$ in T_v since this is the difference between the two. Notice that in T_v, the x-rank of j and the x-rank of the x-predecessor of j are equal.

Each node v of T stores two indices for each element $y_i \in T_v$ in an array A_v of size $|T_v|$. Let $y_i \in T_v$ and r_i be the x-rank of i in T_v. The r_i'th pair of indices stored in A_v is the x-rank of i in the left subtree, and the x-rank of i in the right subtree of v. These are fractional cascading indices, meaning that the x-rank of i in the left (right) subtree is the position of the indices stored for the x-predecessor of i in the left (right) subtree. The arrays are constructed by scanning A, starting with y_1, and *inserting* the elements, y_i, in increasing i order into T. For each element y_i, the search path to y_i is traversed, and in each visited node v the pair of indices for y_i are appended to A_v. The data structure can be built in $O(n \log n)$ time and uses $O(n \log n)$ words of space.

Range Selection Query. A range selection query is given two indices i and j, and an integer s, where $1 \leq i \leq j \leq n$, and must return the s'th smallest element in $A[i, j]$. In a node v of T the search is guided using the x-ranks stored for the x-predecessor of $i-1$ and j in T_v. In the root this is the $i-1$'th and j'th pair stored in the root's array. By subtracting the x-ranks for the left subtree we learn how many elements, s', from $A[i, j]$ the left subtree of v contains. If $s \leq s'$ the search continues in the left subtree. Otherwise, we set $s = s - s'$ and continue the search in the right subtree. Notice that each step learns the x-ranks of $i-1$ and j in the children nodes, which are needed to lookup the indices stored for the x-predecessors of $i-1$ and j in the subsequent step. A query takes $O(\log n)$ time since each step takes constant time. Given indices i and j in a range median query we return the $s = \lfloor \frac{j-i}{2} \rfloor + 1$'th smallest element in $A[i, j]$. Given indices i, j and a value e in a range rank query, we do a predecessor search for e in T: In each step where the search continues to the right child, the number of elements from $A[i, j]$ in the left subtree is computed as above, and these are added up. When a leaf is reached this sum is the rank of e in $A[i, j]$.

2.2 Getting Linear Space

We reduce the space usage of the data structure to $O(n)$ by replacing the arrays stored in each node by simple rank and select data structures [13] as follows. In each node v, the array A_v is partitioned into chunks of size $\log n$. The last entry of each chunk, i.e. every $\log n$'th entry of A_v, is stored as before. For the remaining entries of a chunk, one bit is stored, indicating whether the corresponding element resides in the left or right subtree. These bits are packed in order into one word, which we denote a direction word. This reduces the space to $O(n)$ bits per level of T. Even though v no longer stores an x-rank for each element in T_v, a needed x-rank is easily computed from the stored chunks in constant time. Let r_j be the x-rank of j in T_v, and let $j' = \lfloor j / \log n \rfloor$. The indices stored in the $j'-1$'th chunk yields the x-rank, r_λ, in the left subtree of $y_\lambda \in T_v$. The first $r_j - j' \log n$ bits in the direction word from the j''th chunk determines how many elements from $A[\lambda + 1, j]$ that reside in the left subtree. The sum of these is the x-rank of j in the left subtree. The latter is computed using complete tabulation. The extra table needed for this uses $O(n)$ additional space.

3 Improving Query Time

In this section we generalize the data structure from Section 2 and obtain a linear space data structure that supports range selection queries in $O(\log n / \log \log n)$ time. First, we describe a data structure that supports queries in $O(\log n/\log\log n)$ time but uses slightly more than $O(n)$ space. Then we reduce the space to $O(n)$ by generalizing the ideas from Section 2.2.

3.1 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 $0 < \varepsilon < 1$. The height of T is $O(\log n/\log f) = O(\log n/\log\log n)$. Each node $v \in T$ contains $f \cdot |T_v|$ prefix sums: For each element, $y_i \in T_v$, and for each child index, $1 \le \ell \le f$, v stores the number of elements from $A[1,i]$ that reside in the first ℓ subtrees of T_v. We denote by t_ℓ^i such a prefix sum. 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 as above, i.e. M_i is stored before M_j if $i < j$, and the x-rank of i in T_v defines the position of M_i in A_v. If $y_j \notin T_v$ then v does not store a matrix M_j, but it is still well defined and equal to the matrix $M_{j'}$, that is stored in v, where j' is the x-predecessor of j in T_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 $g = \lfloor \log n/f \rfloor$ 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, 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 contain zero entries by prepending a column of zeroes to all matrices before the division into sections.

3.2 Range Selection Query

A query is given indices i, j and s and locates the s'th smallest element in $A[i,j]$. In each node we consider the matrix $M' = M_j - M_{i-1}$ (row-wise subtraction). The ℓ'th row of M' is $t_\ell^j - t_\ell^{i-1}$, i.e. the number of elements from $A[i,j]$ contained in the first ℓ subtrees. We compute the smallest ℓ such that the ℓ'th row in M' stores a number greater than or equal to s, and this 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 $K = |T_v \cap A[i,j]|$ be the number of elements from $A[i,j]$ contained in T_v. We consider the 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 $[\ell_1, \ell_2] \subseteq [1, f]$, where the g bits of M' match the corresponding g bits of s and the following row. These indices define the subtrees of T_v that can contain the s'th smallest element in T_v. We then try to determine which of these subtrees contain the

s'th smallest element. First, we consider the children of v defined by the endpoints of the interval, ℓ_1 and ℓ_2. If neither of these contain the s'th smallest element in $A[i,j]$, we know that the subtree of T_v containing the s'th smallest element stores approximately a factor of 2^g elements from $A[i,j]$ fewer than T_v, 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. Stated differently, the number of elements in this subtree does not influence the g most important bits of the prefix sum, and thus it must be small. 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 $O(\log n \log\log n/f) = O(\log^{1-\varepsilon} n \log\log n) = o(\log n/\log\log n)$. 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_i} and M_{p_j} where p_i is the x-predecessor of $i-1$ in T_v and p_j is the x-predecessor of j in T_v. For clarity we use M_{i-1} and M_j for the description. A query maintains an index c, initially one, defining which section of the bit-matrices that is currently in use i.e. c defines the section of M' containing the $\lceil \log K \rceil$'th least significant bit. 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. 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_j and M_i. This approximation we denote $w^{i,j}$ and think of it as a $f \times g$ bit-matrix. Basically, the word containing the c'th section of bits from M_{i-1} is subtracted from the corresponding word in M_j. However, subtracting the c'th section of g bits of t_ℓ^{i-1} from the corresponding g bits of t_ℓ^j does not encompass a potential cascading carry from the lower order bits when comparing the result with the matching g bits of $t_\ell^j - t_\ell^{i-1}$, the ℓ'th row of M'. This means that in the c'th section, the ℓ'th row of M_{i-1} could be larger than ℓ'th row of M_j. To ensure that each pair of rows is subtracted independently in the computation of $w^{i,j}$, we prepend an extra one bit to each row of M_j and an extra zero bit to each row of M_i to deal with cascading carries. Then we subtract the c section of M_{i-1} from the c'th section of M_j, and obtain $w^{i,j}$. After the subtraction we ignore the value of the most significant bit of each row in $w^{i,j}$ (it is masked out). After this computation, each row in $w^{i,j}$ 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*. If all bits in $w^{i,j}$ are zero the algorithm never needs to consider the current section again, and it is skipped in the remaining subtree by increasing c by one, without breaking the invariant, and $w^{i,j}$ is recomputed.

Searching $w^{i,j}$. 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 the first row containing an even larger number can define the subtree containing the s'th smallest element. However, since the rows can contain numbers that are one *to large*, we also consider all rows matching $s_b + 1$, and the first row storing a number even larger. Therefore, the algorithm locates the first row of $w^{i,j}$ 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 at 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 represents at 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' ($t^j_\ell - ^{i-1}_\ell$ 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 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, $s = s - (t^j_{\ell-1} - t^{i-1}_{\ell-1})$. Let $r_{i-1} = t^{i-1}_\ell - t^{i-1}_{\ell-1}$, be the x-rank of $i-1$ in T_ℓ, and $r_j = t^j_\ell - t^j_{\ell-1}$, the x-rank of j in T_ℓ. In the subsequent node the algorithm uses the r_{i-1}'th and the r_j'th stored matrix to guide the search. This corresponds to the matrix stored for the x-predecessor of $i-1$ and the x-predecessor of j in T_ℓ (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 $w^{i,j}$ 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 query takes $O(\log n/\log\log n + f \log f) = O(\log n/\log\log n)$ time.

Given i, j and e in a rank query we use a linear space predecessor data structure (van Emde Boas tree [14]) 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 $A[i, j]$ in subtrees

hanging of to the left are added up using the first copy of the bit matrices. The data structures uses $O(nf \log n / \log\log n) = O(n \log^{1+\varepsilon} n / \log\log n)$ space.
Determining ℓ_1 and ℓ_2. The remaining issue is 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 $w^{i,j} = v_1^1, \ldots, v_g^1, \ldots, v_1^d, \ldots, v_g^d$ 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 $w^{i,j}$ we denote $w_\ell^{i,j}$ 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, ℓ_2 can be found similarly. Remember that ℓ_1 is the index of the first row in $w^{i,j}$ 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 $w_\ell^{i,j}$ to one and the extra bit of each s_w^ℓ to zero. This ensures that $w_\ell^{i,j}$ is larger than s_w^ℓ, for all ℓ, when both are considered $g+1$ bit numbers. s_w is subtracted from $w^{i,j}$ and because of the extra bit, this operation subtracts s_w^ℓ from $w_\ell^{i,j}$, for $1 \le \ell \le f$, 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 $w_\ell^{i,j}$ is at least as large as s_w^ℓ. This bit is found using complete tabulation.

3.3 Getting Linear Space

In this section we reduce the space usage of our data structure to $O(n)$ words. The previous data structure stores a matrix for each element on each level of the tree, and every matrix uses $O(f \log n)$ bits of space. Instead we only store a matrix for every $t = \lceil f \log n \rceil$'th element. In each node, the sequence of matrices 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 directions words in the chunk, i.e. storing how many elements that was inserted in the first ℓ subtrees for $\ell = 1, \ldots, f$. Since each chunk stores the direction of t elements, at most $\lceil f \log t / \log n \rceil = O(1)$ words are needed to store these f prefix sums. We denote it a prefix word. 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_{i-1} and M_j to compute $w^{i,j}$ since they are not necessarily stored. Instead, we use two matrices that are stored which are *close* to M_{i-1} and M_j. The direction and update words enables us to exactly compute any row of M_j and M_{i-1} in constant time. Therefore, the main difference compared to the previous data structure, is that the potential difference between $w^{i,j}$, that we compute, 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_i be the x-rank of $i-1$ and r_j the x-rank of j in T_v. Let $i' = \lfloor r_i/t \rfloor$ and $j' = \lfloor r_j/t \rfloor$. The matrices stored in the i''th and j''th chunk respectively are used to guide the search. These matrices we denote M_a and M_b. Since v stores every t'th matrix from above, $t_\ell^j - t_\ell^b \leq t$ for any $1 \leq \ell \leq f$. If we ignore a potential cascading carry, then adding $t_\ell^j - t_\ell^b$ to t_ℓ^b only affects the last $\log t = (1+\varepsilon)\log\log n$ bits of t_ℓ^b. This means that, unless the search is using the last section, each row in the currently considered section of M_b represents a number that is at most one smaller than if we had used the corresponding section from M_j. The same is true for M_a.

We can obtain the value of any row in M_j as follows. From the direction and prefix words from the j''th chunk we compute for each ℓ, $1 \leq \ell \leq f$, how many of the first $r_j - j't$ elements represented in the chunk that reside in the first ℓ children. These are the elements considered in M^j but not in M^b. Formally, the $p = \lfloor (r_j - j't)/d \rfloor$'th prefix word stores how many of the first pd elements from the chunk that 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 $r_j - j't - pd$ elements from the chunk that reside in the first ℓ children for all $1 \leq \ell \leq f$. Adding this to the p'th prefix word, gives for each $1 \leq \ell \leq f$, the difference between the ℓ'th row of M_j and M_b. The difference between M_a and M_{i-1} can be computed similarly. Thus, any row of M_j and M_{i-1}, and the last section of M_j and M_{i-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 $w^{a,b}$. Since the ℓ'th row, for $1 \leq \ell \leq f$, in the current section of M_b might be one to small compared to ℓ'th row in the current section of M_j, the ℓ'th row in $w^{a,b}$ may be one to small compared to the corresponding g bits of M'. Similarly, each row in $w^{a,b}$ might also one to the large since the current section of bits from M_a may be one smaller than in the current section of M_{i-1}. 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 additionally be one to large when compared to the same bits in M'. Therefore, the first row of $w^{a,b}$ that is at least $s_b - 1$ and the first row greater than $s_b + 2$ are located as above. As above, the subtree we are searching for is defined by a row between these two, and if it is not one 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 at 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.

Dominance queries are supported similarly to above.

4 Dynamic Range Selection

In this section we briefly sketch how our data structure can be made dynamic. 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, worst case and amortized respectively. Details will appear in the full paper.

Our data structure maintains a set of points, $S = \{(x_i, y_i)\}$, under insertions and deletions. A query is given values x_l, x_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 x_l and x_r. We store the points from S in a weight-balanced search tree [15, 16], 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.

5 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 exist 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, $t_q = \Omega(\frac{\log n}{\log(t_u w \log n)})$ [11], where w is the word size.
Reduction. Let T denote a marked ancestor tree of height h and degree b. For each node v in T we associate two pairs of elements, 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 we for each node v outputs 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 influences 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 on the form $[0, 1, \ldots, 0, 1]$. 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 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 and 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 1. *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.*

6 Main Open Problems

There are two main open problems. First, what is the lower bound on the query time for range selection queries in static $O(n \log^{O(1)} n)$ space data structures? We can prove that any $O(n \log^{O(1)} n)$ space data structure needs $\Theta(\log n / \log\log n)$ time for three-sided range median queries by a reduction from two dimensional rectangle-stabbing [8]. Furthermore, there is a gap between the upper and lower bounds for batched range median problem for $k = \Omega(n^{1+\varepsilon})$, and the lower bound [6] is only valid in the comparison model.

References

1. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nord. J. Comput. **12**(1) (2005) 1–17
2. Petersen, H.: Improved bounds for range mode and range median queries. In: Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science. (2008) 418–423
3. Petersen, H., Grabowski, S.: Range mode and range median queries in constant time and sub-quadratic space. Inf. Process. Lett. **109**(4) (2009) 225–228
4. Gfeller, B., Sanders, P.: Towards optimal range medians. In: Proc. 36th International Colloquium on Automata, Languages and Programming. (2009) 475–486
5. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. 16th String Processing and Information Retrieval Symposium. (2009) 1–6
6. Har-Peled, S., Muthukrishnan, S.: Range medians. In: Proc. 16th Annual European Symposium on Algorithms. (2008) 503–514
7. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proc. 39th ACM Symposium on Theory of Computing. (2007) 40–46
8. Pătraşcu, M.: (Data) STRUCTURES. In: Proc. 49th Annual IEEE Symposium on Foundations of Computer Science. (2008) 434–443
9. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Proc. 15th International Symposium on Algorithms and Computation. (2004) 558–568
10. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. In: Proc. 10th International Workshop on Algorithms and Data Structures. (2007) 15–26
11. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (1998) 534–543
12. Chazelle, B., Guibas, L.J.: Fractional cascading: I. A data structuring technique. Algorithmica **1**(2) (1986) 133–162
13. Jacobson, G.J.: Succinct static data structures. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1988)
14. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory **10** (1977) 99–127
15. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Proc. 4th Annual ACM symposium on Theory of computing. (1972) 137–142
16. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM Journal on Computing **32**(6) (2003) 1488–1508