Space Efficient Data Structures for Dynamic Orthogonal Range Counting*

Meng He and J. Ian Munro

Cheriton School of Computer Science, University of Waterloo, Canada
{mhe,imunro}@uwaterloo.ca

Abstract. We present a linear-space data structure that maintains a dynamic set of n points with coordinates of real numbers on the plane to support orthogonal range counting, as well as insertions and deletions, in $O((\frac{\lg n}{\lg \lg n})^2)$ time. This provides faster support for updates than previous results with the same bounds on space cost and query time. We also obtain two other new results by considering the same problem for points on a $U \times U$ grid, and by designing the first succinct data structures for a dynamic integer sequence to support range counting.

1 Introduction

The two-dimensional ***orthogonal range counting*** problem is a fundamental problem in computational geometry. In this problem, we store a set, P, of n points in a data structure so that given a query rectangle R, the number of points in $P \cap R$ can be computed efficiently. This problem has applications in many areas of computer science, including databases and computer graphics, and thus has been studied extensively [4,11,13,12,3]. Among previous previous, Chazelle [4] designed a linear-space data structure for points with real coordinates to support orthogonal range counting in $O(\lg n)$ time[1], while the linear-space data structure of JáJá *et al.* [11] provides $O(\frac{\lg n}{\lg \lg n})$-time support for integer coordinates.

Researchers have also considered the orthogonal range counting problem in dynamic settings. The goal is to maintain a dynamic set, P, of n points to support orthogonal range counting, while allowing points to be inserted into and deleted from P. Chazelle [4] designed a linear-space data structure that supports orthogonal range counting, insertions and deletions in $O(\lg^2 n)$ time. Nekrich [12] designed another data structure of linear space with improved query time. With his data structure, a range counting query can be answered in $O((\frac{\lg n}{\lg \lg n})^2)$ time, matching the lower bound proved by Pătraşcu [13] under the group model, but it takes $O(\lg^{4+\epsilon} n)$ amortized time to insert or delete a point. Thus in this paper, we consider the problem of designing a linear-space data structure that matches Nekrich's query time while providing faster support for updates.

In addition to considering points on the plane, we also define range counting over a dynamic sequence $S[1..n]$ of integers from $[1..\sigma]$: given a range, $[i_1..i_2]$,

* This work was supported by NSERC and the Canada Research Chairs program.

[1] $\lg n$ denotes $\log_2 n$.

of indices and a range, $[v_1..v_2]$, of values, a range counting query returns the number of entries of $S[i_1..i_2]$ that store integers from $[v_1..v_2]$.[2] We are interested in designing ***succinct data structures*** to represent S. Succinct data structures were first proposed by Jacobson [10] to encode bit vectors, trees and planar graphs using space close to the information-theoretic lower bound, while supporting efficient navigation operations in them. As succinct data structures provide solutions to modern applications that process large data sets, they have been studied extensively [14,6,5,7,3,8].

1.1 Our Results

Under the word RAM model with word size $w = \Omega(\lg n)$, we present the following results:

1. A linear-space data structure that maintains a dynamic set, P, of n points on the plane to answers an orthogonal range counting query in $O((\frac{\lg n}{\lg \lg n})^2)$ worst-case time. A point can be inserted into or deleted from P in $O((\frac{\lg n}{\lg \lg n})^2)$ amortized time. This improves the result of Nekrich [12]. The point coordinates are real numbers, and we only require that any two coordinates can be compared in constant time.
2. A linear-space data structure that maintains a dynamic set, P, of n points on a $U \times U$ grid to provide $O(\frac{\lg n \lg U}{(\lg \lg n)^2})$ worst-case time support for orthogonal range counting, insertions and deletions. Note that for large data sets in which $\lg n = \Theta(\lg U)$, the query and update times are both $O((\frac{\lg n}{\lg \lg n})^2)$ in the worst case.
3. A succinct representation of a dynamic sequence $S[1..n]$ of integers from $[1..\sigma]$ in $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits[3] to support range counting, insertion and deletion in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all the operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time. This is the first dynamic succinct data structure that supports range counting.

2 Data Structures for Range Sum

In this section, we present two data structures that are used in our solutions to dynamic orthogonal range counting. Both data structures represent a two-dimensional array $A[1..r, 1..c]$ of numbers to support ***range sum*** queries that return the sum of the elements in a rectangular subarray of A. More precisely, a range sum query, $\texttt{range\_sum}(A, i_1, j_1, i_2, j_2)$, evaluates $\sum_{u=i_1}^{i_2} \sum_{v=j_1}^{j_2} A[u, v]$, i.e. the sum of the numbers stored in the subarray $A[i_1..i_2, j_1..j_2]$, where $1 \leq i_1 \leq i_2 \leq r$ and $1 \leq j_1 \leq j_2 \leq c$.

We define a special form of range sum queries as ***dominance sum*** queries. These return the sum of the elements stored in subarrays of the form $A[1..i, 1..j]$,

[2] $[i..j]$ denotes a range of integers, while $[i, j]$, $[i, j)$, etc. denote ranges of real values.
[3] H_0 denotes the zeroth empirical entropy of S, which is $\lg \sigma$ in the worst case.

where $1 \leq i \leq r$ and $1 \leq j \leq c$. In other words, we define the operator `dominance_sum`(A, i, j) to be `range_sum`$(A, 1, i, 1, j)$. It is clear that any range sum query can be answered using at most four dominance sum queries.

Our data structures also support update operations to the array A, namely the following three operations:

- `modify`(A, i, j, δ), which sets $A[i, j]$ to $A[i, j]+\delta$ (restrictions on δ will apply);
- `insert`(A, j), which inserts a 0 between $A[i, j-1]$ and $A[i, j]$ for all $1 \leq i \leq r$, thus increasing the number of columns of A by one;
- `delete`(A, j), which deletes $A[i, j]$ for all $1 \leq i \leq r$, decreasing the number of columns of A by one, and to perform this operation, $A[i, j] = 0$ must hold for all $1 \leq i \leq r$.

The two data structures presented in this section solve the dynamic range sum problem under different restrictions on the input data and updates, in order to achieve desired time and space bounds of our range counting structures.

2.1 Range Sum in a Small Two-Dimensional Array

We now design a data structure for a small two-dimensional array $A[1..r, 1..c]$ to support range sum. Let n be a positive integer such that $w = \Omega(\lg n)$, where w is the word size of the RAM model. We require that $rc = O(\lg^{\lambda} n)$ for any constant $\lambda \in (0, 1)$, and that each entry of A stores an nonnegative, b-bit integer where $b = O(\lg n)$. This data structure supports `modify`(A, i, j, δ), where $|\delta| \leq \lg n$, but it does not support `insert` or `delete`.

Our data structure is a generalization of data structure of Raman *et al.* [14] on supporting prefix-sum queries on a small one-dimensional array: the dominance sum query is a two-dimensional version of prefix sum. It is not hard to adapt the approach of Raman *et al.* to 2D and represent A in $O(\lg^{1+\lambda} n)$ bits to support `range_sum` in $O(1)$ worst-case time and `modify` in $O(1)$ amortized time with the help of a universal table, but deamortization is interesting and nontrivial. We first present the following data structure that provides $O(1)$ amortized time support for queries and updates:

Lemma 1. *The array A described above can be represented using $O(\lg^{1+\lambda} n)$ bits to support* `range_sum` *in $O(1)$ worst-case time and* `modify`(A, i, j, δ)*, where $|\delta| \leq \lg n$, in $O(1)$ amortized time. This data structure requires a precomputed universal table of size $O(n^{\lambda'})$ bits for any fixed constant $\lambda' \in (0, 1)$.*

Proof. In addition to storing A, we construct and maintain a two-dimensional array $B[1..r, 1..c]$, in which $B[i, j]$ stores $\sum_{u=1}^{i} \sum_{v=1}^{j} A[u, v]$, i.e. the result of `dominance_sum`$(A, i, j)$. We however cannot always keep B up-to-date under updates, so we allow B to get slightly "out of date". More precisely, B is not changed each time `modify` is performed; instead, after every rc `modify` operations, we reconstruct B from A to make B up-to-date. Since it takes $O(rc)$ time to reconstruct B, the amortized cost is $O(1)$ per `modify` operation.

As mentioned before, to support the `range_sum` operation, it suffices to provide support for `dominance_sum`. In order to answer dominance sum queries correctly using B, we maintain another two-dimensional array $C[1..r, 1..c]$, whose content is set to all zeros each time we construct B. Otherwise, after an operation `modify`(A, i, j, δ) is performed, we set $C[i, j] \leftarrow C[i, j] + \delta$. Thus we have:

$$\texttt{dominance\_sum}(A, i, j) = B[i, j] + \sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v] \qquad (1)$$

To use the above identity to compute `dominance_sum`(A, i, j) in constant time, it suffices to compute $\sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v]$ in constant time. Since we set $C[i, j]$ to 0 after every rc `modify` operations and we require $|\delta| \leq \lg n$, we have $|C[i, j]| \leq rc \lg n = O(\lg^{1+\lambda} n)$. Hence each entry of C can be encoded in $O(\lg \lg n)$ bits. Thus array C can be encoded in $O(\lg^{\lambda} n \lg \lg n) = o(\lg n)$ bits, which allows us to build a $O(n^{\lambda'})$-bit precomputed table to perform the above computation in constant time. It is clear that `modify` can be supported in $O(1)$ amortized time, and the arrays A, B and C can be encoded in $O(\lg^{1+\lambda} n)$ bits in total. □

To eliminate amortization, we design the following approach:

1. We construct a new table C' after rc `modify` operations have been performed since the table C was created, i.e. after the values of C have been changed rc times. Initially, all entries of C' are zeroes.
2. After we create C', for the next rc `modify` operations, if the operation is `modify`(A, i, j, δ), we set $C'[i, j] \leftarrow C'[i, j] + \delta$ without changing the content of C. We use the following identity instead of Identity 1 to answer queries:

$$\texttt{dominance\_sum}(A, i, j) = B[i, j] + \sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v] + \sum_{u=1}^{i} \sum_{v=1}^{j} C'[u, v] \qquad (2)$$

3. We also maintain a pointer called ***refresh pointer*** that moves from $B[1, 1]$ to $B[r, c]$ in row-major order. When we create the table C', the refresh pointer points to $B[1, 1]$. After each `modify`, we move the pointer by one position. Right before we move the pointer that points to $B[i, j]$, we perform the following process of ***refreshing*** $B[i, j]$:
   (a) Set $B[i, j] \leftarrow B[i, j] + C[i, j]$;
   (b) If $i < r$, set $C[i + 1, j] \leftarrow C[i + 1, j] + C[i, j]$;
   (c) If $j < c$, set $C[i, j + 1] \leftarrow C[i, j + 1] + C[i, j]$;
   (d) If $i < r$ and $j < c$, set $C[i + 1, j + 1] \leftarrow C[i + 1, j + 1] - C[i, j]$;
   (e) Set $C[i, j] \leftarrow 0$.
4. After we refresh $B[r, c]$, rc `modify` operations have been performed since we created C'. At this time, all the entries of C are zeroes. We then deallocate C, rename C' by C. Note that at this time, rc `modify` operations have already been performed on the new array C (when it was named C'), so it is time to go back to step 1, create a new table C', and repeat this process.

In the above approach, `modify` clearly takes $O(1)$ worst-case time, and A, B, C and C' can be encoded in $O(\lg^{1+\lambda} n)$ bits. To show the correctness of the above process, it is not hard to see that Identity 2 always holds. Finally, we need argue that the right-hand side of Identity 2 can be evaluated in constant time. The term $\sum_{u=1}^{i}\sum_{v=1}^{j} C'[u,v]$ can be evaluated in constant time using the precomputed universal table. However, it is not clear whether $\sum_{u=1}^{i}\sum_{v=1}^{j} C[u,v]$ can still be evaluated in constant time using this table: Because of the refresh process, it is not trivial to show that each entry of C can still be encoded in $O(\lg\lg n)$ bits. For this we first present these two easy-to-prove lemmas (we omit their proofs):

Lemma 2. *For any integers $i \in [1,m]$ and $j \in [1,d]$, a refresh process does not change the value of $\sum_{u=1}^{i}\sum_{v=1}^{j} C[u,v]$ unless this process refreshes $B[i,j]$.*

Lemma 3. *Let $C^*[u,v]$ be the value of $C[u,v]$ when the table C' is created. Then immediately before we refresh $B[i,j]$, the value of $C[i,j]$ is $\sum_{u=1}^{i}\sum_{v=1}^{j} C^*[u,v]$.*

We can now show that each entry of C can be encoded in $O(\lg\lg n)$ bits:

Lemma 4. *The absolute value of any entry, $C[i,j]$, of C never exceeds $4rc\lg n$.*

Proof. We prove this lemma for $i > 1$ and $j > 1$; the other cases can be handled similarly. When we create the table C' and start to refresh the entries of B, rc `modify` operations have been performed since C was created (recall that initially C was named C'). Hence when we start to refresh the entries of B, the absolute value of $C[i,j]$ is at most $rc\lg n$. When we refresh $B[i,j]$, we set $C[i,j]$ to 0 and never changes its value till we deallocate C. Before $B[i,j]$ is refreshed, the value of $C[i,j]$ changes at most three times: (i) when we refresh $B[i-1,j-1]$; (ii) when we refresh $B[i-1,j]$; and (iii) when we refresh $B[i,j-1]$. In (i), we set $C[i,j] \leftarrow C[i,j]-C[i-1,j-1]$ before we set $C[i-1,j-1]$ to 0. By Lemma 3, the absolute value of $C[i-1,j-1]$ before we set it to 0 is at most $rc\lg n$. Hence the absolute value of $C[i,j]$ does not exceed $2rc\lg n$ after (i). By similar reasoning, we can show that the absolute values of $C[i,j]$ do not exceed $3rc\lg n$ and $4rc\lg n$ after (ii) and (iii), respectively.[4] □

Our result in this section immediately follows from Lemma 4:

Lemma 5. *Let n be a positive integer such that $w = \Omega(\lg n)$, where w is the word size of the RAM model. A two-dimensional array $A[1..r,1..c]$ of non-negative, b-bit integers, where $b = O(\lg n)$ and $rc = \lg^{\lambda} n$ for any constant $\lambda \in (0,1)$, can be represented using $O(\lg^{1+\lambda} n)$ bits to support* `range_sum` *and* `modify`(A,i,j,δ), *where $|\delta| \leq \lg n$, in $O(1)$ worst-case time. This data structure can be constructed in $O(rc)$ time, and it requires a precomputed universal table of size $O(n^{\lambda'})$ bits for any fixed constant $\lambda' \in (0,1)$.*

[4] With greater care, we can show that the absolute value of any element of C never exceeds $rc\lg n$ using the identity in Lemma 3, although this would not affect the time/space bounds in Lemma 5.

2.2 Range Sum in a Narrow Two-Dimensional Array

Our second data structure for dynamic range sum requires the array $A[1..r, 1..c]$ to be "narrow", i.e. $r = O(\lg^{\gamma} c)$ for a fixed constant $\gamma \in (0, 1)$. Dominance sum queries on this data structure can be viewed as a 2-dimensional versions of prefix sum queries in the Collections of Searchable Partial Sums (CSPSI) problem defined by González and Navarro [7]. Our data structure in based on the solution to the CSPSI problem given by He and Munro [8,9], and the main change is to use Lemma 5 to encode information encoded as small 2D arrays. We have the following result:

Lemma 6. *Let $A[1..r][1..c]$ be a two-dimensional array of nonnegative integers, where $r = O(\lg^{\gamma} c)$ for any constant $\gamma \in (0, 1)$, and each integer of A is encoded in $b = O(w)$ bits, where $w = \Omega(\lg c)$ is the word size of the RAM model. A can be represented using $O(rcb+w)$ bits to support* `range_sum`*,* `search`*,* `modify`(C, i, j, δ) *where $|\delta| \leq \lg c$,* `insert` *and* `delete` *in $O(\frac{\lg c}{\lg \lg c})$ time with a $O(c \lg c)$ bit buffer.*

3 Range Counting in Integer Sequences

A basic building block for many succinct data structures [6,5,7,3,8] is a highly space-efficient representation of a sequence $S[1..n]$ of integers from $[1..\sigma]$ to support the fast evaluation of `rank` and `select`[5]. Under dynamic settings, the following operations are considered:

- `access`(S, i), which returns $S[i]$;
- `rank`$_\alpha(S, i)$, which returns the number of occurrences of integer α in $S[1..i]$;
- `select`$_\alpha(S, i)$, which returns the position of the i^{th} occurrence of integer α in the string S;
- `insert`$_\alpha(S, i)$, which inserts integer α between $S[i-1]$ and $S[i]$;
- `delete`(S, i), which deletes $S[i]$ from S.

He and Munro [8] designed a succinct representation of S to support the above operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. In this section, we extend their results to support range counting on integer sequences. We are interested in the operation `range_count`(S, p_1, p_2, v_1, v_2), which returns the number of entries in $S[p_1..p_2]$ whose values are in the range $[v_1..v_2]$.

3.1 Sequences of Small Integers

We first consider the case in which $\sigma = O(\lg^{\rho} n)$ for any constant $\rho \in (0, 1)$. In our approach, we encode S using a B-tree as in [8]. Each leaf of this B-tree contains a ***superblock*** that has at most $2L$ bits, where $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Entries of S are stored in superblocks. A two-dimensional array $F[1..\sigma, 1..t]$ is constructed,

[5] Many papers define S as a string of characters over alphabet $[1..\sigma]$, which is equivalent to our definition. We choose to define S as a sequence of integers as it seems more natural to introduce range counting on integers.

where t denotes the number of superblocks. An entry $F[\alpha, i]$ stores the number of occurrences of integer α in superblock i. The array F is encoded using Lemma 6. We defer the details of our algorithms and data structures to the full version of our paper, and only present our results here. We first present our result on representing dynamic sequences of small integers to support range counting:

Lemma 7. *Under the word RAM model with word size $w = \Omega(\lg n)$, a sequence $S[1..n]$ of integers from universe $[1..\sigma]$, where $\sigma = O(\lg^\rho n)$ for any constant $\rho \in (0, 1)$, can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* `access`*,* `rank`*,* `select`*,* `range_count`*,* `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time.*

We also have the following lemma to show that a batch of update operations performed on a contiguous subsequence S can be supported efficiently:

Lemma 8. *Let S be a sequence represented by Lemma 7. Consider a batch of m update operations performed on subsequence $S[a..a + m - 1]$, in which the i^{th} operation changes the value of $S[a+i-1]$. If $m > 5L/\lg \sigma$, then the above batch of operations can be performed in $O(m)$ time.*

3.2 General Integer Sequences

To generalized our result on sequences of small integers to general integer sequences, we combine the techniques of Lemma 7 with generalized wavelet trees proposed by Ferragina *et al.* [6]. Similar ideas were used by He and Munro [8] to support `rank` and `select` operations on dynamic sequences, and by Bose *et al.* [3] for static orthogonal range search structures on a grid. Here we apply these techniques on range counting in dynamic settings:

Theorem 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a sequence $S[1..n]$ of integers from $[1..\sigma]$ can be represented using $nH_0+O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}})$ $+O(w)$ bits to support* `access`*,* `rank`*,* `select`*,* `range_count`*,* `insert` *and* `delete` *operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all these operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time.*

4 Range Counting in Planar Point Sets

We now consider orthogonal range counting over a dynamic set of n points on the plane. In Section 4.1, we consider a special case in which each point is on a fixed $U \times U$ grid, i.e. each x or y-coordinate is an integer from universe $[1..U]$, while in Section 4.2, points on the plane have arbitrary (real) coordinates as long as any two coordinates can be compared in constant time.

4.1 Range Counting on a $U \times U$ Grid

Our orthogonal range counting structure for a dynamic set of n points on a $U \times U$ grid is based on our data structure supporting range counting over an integer sequence. The key idea is to convert coordinates in one dimension, say, the x-coordinates, to rank space.

Theorem 2. *Under the word RAM model with word size $w = \Omega(\lg n)$, there is an $O(n)$ word data structure that can maintain a dynamic set, P, of n points on a $U \times U$ grid to answer orthogonal range counting queries in $O(\frac{\lg n \lg U}{(\lg\lg n)^2})$ time. A point can be inserted to or deleted from P in $O(\frac{\lg n \lg U}{(\lg\lg n)^2})$ time.*

Proof. Let P_x be the set of x-coordinates. Without loss of generality, we assume that x-coordinates are distinct. We construct an augmented red-black tree T_x to represent P_x: For each node v in T_x, we store additional information that encodes the number of nodes in the subtree rooted at v. With such information, given a value x, we can easily find out, in $O(\lg n)$ time, the number of elements in P_x that are less than or equal to x. Maintaining such information does not slow down the $O(\lg n)$-time support for insertions and deletions of values in P_x. This is because each insertion or deletion requires at most 3 tree rotations, so we need only update the information of subtree size for the constant number of nodes directly involved in the rotations and their $O(\lg n)$ ancestors.

We construct a sequence $S[1..n]$ of integers from $[1..U]$, in which $S[i] = u$ if and only if the point in P with the i^{th} smallest x-coordinate has y-coordinate u. We represent S using Theorem 1. Since T_x maps x-coordinates to rank space, it is easy to use T_x and S to support query and update. □

4.2 Range Counting for General Point Sets

For general point sets, we can still use the augmented red-black tree designed in the proof of Theorem 2 to map the set of x-coordinates to the rank space, since this tree structure does not require any assumptions on the values stored. Handling the other dimension, however, is challenging: We cannot simply use a generalized wavelet tree, which is the main building block of the representation of the sequence S used in the proof of Theorem 2. This is because a (generalized) wavelet tree has, up to now, only been used to handle data of a two-dimensional nature for which the range of values in at least one dimension is fixed [6,5,3,7,8], such as sequences of integers from a fixed range in Theorem 1. To overcome this difficulty, our main strategy is to combine the notion of range trees [2] with generalized wavelet trees. Our work is the first that combines these two powerful data structures.

Let P_x and P_y denote the set of x and y-coordinates of the points in P, respectively. Without loss of generality, we assume that the values in P_x are distinct, and so are the values in P_y. We construct the following data structures:

1. An augmented red-black tree, T_x, that represents the set P_x, as described in the proof of Theorem 1. Recall that this structure supports the computation of the number of values in P_x that are less than or equal to a given value.
2. As amortizing a rebuilding cost to insertions or deletions will be crucial, we use a weight-balanced B-tree [1], T_y. This is constructed over P_y, with branching factor $d = \Theta(\lg^\epsilon n)$ for a fixed constant $\epsilon \in (0, 1)$ and leaf parameter 1. Hence each internal node has at least $d/4$ and at most $4d$ children, except the root for which the lower bound on degree does not apply. Each leaf represents a range $[y, y')$, where y and y' are in P_y, and y' is the immediate successor of y. The

(contiguous) range represented by an internal node is the union of the ranges represented by its children. The levels of T_y are numbered $0, 1, 2, \cdots$, starting from the root level. We store the tree structure of T_y together with the start and end values of the range represented by each node.

3. Next we use ideas analogous to those of generalized wavelet trees [6]. A sequence $L_\ell[1..n]$ of integers from $[1..4d]$ is constructed for each level ℓ except the leaf level, which is encoded using Lemma 7: For each internal node v at level ℓ of T_y, we construct a sequence, S_v of integers from $[1..4d]$. Each entry of S_v corresponds to a point in P whose y-coordinate is in the range represented by v, and $S_v[i]$ corresponds to, among all the points with y-coordinates within the range represented by v, the one with the i^{th} smallest x-coordinate. $S_v[i]$ does not store this point directly. Instead, $S_v[i]$ stores j if the y-coordinate of the corresponding point in P is within the range represented by the j^{th} child of v. We further concatenate all the sequences constructed for the nodes, from left to right, at level ℓ to get the sequence L_ℓ. It is important to understand that, for the top level of T_y, the entries of L_0 correspond to points in P ordered by x-coordinates, but as we move down the tree T_y, the ordering gradually changes: The entries of $L_1, L_2, \cdots$ do not correspond to points ordered by x-coordinates, and at the bottom level, the leaves correspond to points ordered by y-coordinates.

To analyze the space cost of our data structures, it is clear that T_x and T_y use linear space. Our third set of data structures consist of $O(\frac{\lg n}{\lg\lg n})$ subsequences, each storing n integers from $[1..4d]$. By Lemma 7, they occupy $O(n\lg d + w) \times O(\frac{\lg n}{\lg\lg n}) = O(n\lg n + w \times \frac{\lg n}{\lg\lg n})$ bits in total, where w is the size of a word. This space cost is $O(n)$ words. We now use these data structures to support queries:

Lemma 9. *Under the word RAM model with word size $w = \Omega(\lg n)$, the above data structures support orthogonal range counting in $O((\frac{\lg n}{\lg\lg n})^2)$ time.*

Proof. We first give an overview of our algorithm for orthogonal range counting. Let $R = [x_1, x_2] \times [y_1, y_2]$ be the query rectangle. We use T_x to find two x-coordinates x'_1 and x'_2 in P_x that are the immediate successor of x_1 and the immediate predecessor of x_2, respectively (if a value is present in P_x, we define its immediate predecessor/successor to be itself). We then perform a top-down traversal in T_y to locate the (up to two) leaves that represent ranges containing y_1 and y_2. During this traversal, at each level ℓ of T_y, at most two nodes are visited. For a node v visited at level ℓ, we answer a range counting query $\texttt{range\_count}(S_v, i_v, j_v, c_v, d_v)$, where $S_v[i_v..j_v]$ is the longest contiguous subsequence of S_v whose corresponding points in P have x-coordinates in the range $[x'_1, x'_2]$, and the children of v representing ranges that are entirely within $[y_1..y_2]$ are children $c_v, c_v + 1, \cdots, d_v$ (child i refers to the i^{th} child). The sum of the results of the above range queries at all levels is the number of points in $N \cap R$.

To show how to perform the above process, we first observe that for the root r of T_y, i_r and j_r are the numbers of values in P_x that are less than or equal to x'_1 and x'_2, respectively, which can be computed using T_x in $O(\lg n)$ time.

To compute c_r and d_r, we can perform binary searches on the up to $4d$ ranges represented by the children of r, which takes $O(\lg\lg n)$ time. The binary searches also tell us which child/children of r represent ranges that contain y_1 and y_2, and we continue the top-down traversal by descending into these nodes.

It now suffices to show, for each node v visited at each level ℓ, how to locate the start and end positions of S_v in L_ℓ, how to compute i_v, j_v, c_v and d_v, and which child/children of v we shall visit at level $\ell+1$. Let u be the parent of v, and we assume that v is the c^{th} child of u. Let s be the start position of S_u in $L_{\ell-1}$, which was computed when we visited u. We observe that, to compute the start and end positions of S_v, it suffices to compute the numbers of entries of S_u that are in the range $[1..c-1]$ and $[1..c]$, respectively. Thus the start and end positions of S_v in L_ℓ are $s+\texttt{range\_count}(S_u, 1, |S_u|, 1, c-1)+1$ and $s+\texttt{range\_count}(S_u, 1, |S_u|, 1, c)$, respectively. Positions i_v and j_v can also be computed by performing operations on S_u using the identities $i_v = \texttt{rank}_c(S_u, i_u - 1) + 1$ and $j_v = \texttt{rank}_c(S_u, j_u)$. Finally, to compute c_v and d_v, and to determine the child/children of v that we visit at level $\ell+1$, we perform binary searches on the ranges represented by the at most $4d$ children of v using $O(\lg d) = O(\lg\lg n)$ time. Since we perform a constant number of `rank`, `select` and `range_count` operations on a sequence of small integers at each level of T_y, and there are $O(\frac{\lg n}{\lg\lg n})$ levels, we can answer an orthogonal range counting query over P in $O((\frac{\lg n}{\lg\lg n})^2)$ time. □

Finally, we support update operations to achieve our main result:

Theorem 3. *Under the word RAM model with word size $w = \Omega(\lg n)$, there is a data structure using $O(n)$ words of structural information plus space for the coordinates that can maintain a dynamic set, P, of n points on the plane to answer orthogonal range counting queries in $O((\frac{\lg n}{\lg\lg n})^2)$ worst-case time. A point can be inserted to or deleted from P in $O((\frac{\lg n}{\lg\lg n})^2)$ amortized time.*

Proof. To support update operations, we only show how to insert a point into P; deletions can be handled similarly. To insert a point with coordinates $< x, y >$ into P, we first insert x into T_x in $O(\lg n)$ time. Inserting y into T_y will either cause a leaf of T_y to split into two, or create a new leaf that is either the leftmost leaf or the rightmost. Without loss of generality, we assume that a leaf is split. Then this leaf can be located by performing a top-down traversal, similar to the process required to support range counting. Let q be the parent of this leaf, and let ℓ' be the level number of the level right above the leaf level. Then the start and end positions of S_q in $L_{\ell'}$ can also be located in the above process, using $O((\frac{\lg n}{\lg\lg n})^2)$ time in total. We first consider the case that the split of this leaf will not cause q to split. In this case, since q has one more child, we insert one more entry into S_q for the new child, and increase the values of at most $|S_q|$ entries in S_q by 1. This can be done by performing at most $|S_q|$ insertions and deletions over S_q, which costs $O(d \times \frac{\lg n}{\lg\lg n}) = O(\frac{\lg^{1+\epsilon} n}{\lg\lg n})$ time. Since this insertion also causes the length of the sequence S_v for the ancestor, v, of q at each level to increase by 1, one integer is inserted into each string L_ℓ for $\ell = 0, 1, \cdots, \ell' - 1$. The exact position where we should perform the insertion can be determined using tree T_x for L_0, and by performing one `rank` operation on $L_0, L_1, \cdots, L_{\ell'-2}$ for $L_1, L_2, \cdots, L_{\ell'-1}$, respectively, during the top-down traversal. The exact value to be inserted to each L_ℓ is an appropriate child number. So far we have spent $O((\frac{\lg n}{\lg\lg n})^2)$ time in total.

Each insertion may however cause a number of internal nodes to split. Let v be an internal node that is to split, and let v_1 and v_2 be the two nodes that v is to be split into, where v_1 is a left sibling of v_2. Let f be the number of the level that contains v. Then the splitting of v requires us to replace the substring, S_v, of L_f by two substrings S_{v_1} and S_{v_2}. Since points corresponding to these substrings are sorted by their x-coordinates, this is essentially a process that splits one sorted list into two sorted lists. Thus, we can perform a linear scan on S_v, and perform one insertion and one deletion for each entry of S_v. This costs $O(|S_v| \times \frac{\lg n}{\lg\lg n})$ time. This would have been messy, but fortunately the following two invariants of weight-balanced B-trees allow us to bound the above time in the amortized sense: First, if v is k levels above the leaf level, then $|S_v| < 2d^k$. Second, after the split of node v, at least $d^k/2$ insertions have to be performed below v before it splits again. Hence we can amortize the above $O(|S_v| \times \frac{\lg n}{\lg\lg n}) = O(2d^k \times \frac{\lg n}{\lg\lg n})$ cost over $d^k/2$ insertions, which is $O(\frac{\lg n}{\lg\lg n})$ per insertion.

Let u be the parent of v. The above split may also increase the values of up to $|S_u|$ entires of S_u by 1, which cost $O(|S_u| \times \frac{\lg n}{\lg\lg n})$ time. By the same argument as above, we have $|S_u| < 2d^{k+1}$, and we can amortize the above $O(|S_u| \times \frac{\lg n}{\lg\lg n}) = O(2d^{k+1} \times \frac{\lg n}{\lg\lg n})$ cost over $d^k/2$ insertions, which is $O(\frac{d\lg n}{\lg\lg n})$ per insertion. Since the insertion into a leaf may cause its $O(\frac{\lg n}{\lg\lg n})$ ancestors to split, and each split charges $O(\frac{d\lg n}{\lg\lg n})$ amortized cost for this insertion, splitting these $O(\frac{\lg n}{\lg\lg n})$ internal nodes after an insertion requires $O(\frac{\lg^{2+\epsilon} n}{(\lg\lg n)^2}) = O(\lg^{2+\epsilon} n)$ amortized time.

To further speed up the process in the previous paragraph, we observe that the bottleneck is the $O(|S_u| \times \frac{\lg n}{\lg\lg n})$ time required to change the values of up to $|S_u|$ entires of S_u after v is split. Since these entries are all in S_u, which is a contiguous subsequence of L_{f-1}, we apply Lemma 8. There is one more technical detail: this lemma requires that $|S_u| > 5L/\lg(4d)$ where $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg\lceil \lg n \rceil} \rceil$. By an invariant maintained by a weight-balanced B-tree, $|S_u| > d^{k+1}/2$, since u is $k+1$ levels above the leaf level. Hence $|S_u| > 5L/\lg(4d)$ is true for all $k > \log_{d/2}(5L/\lg(4d))$, and the floor of the right-hand side is a constant number. Let k_0 denote this constant. Hence if v is up to k_0 levels above the leaf level, we use the approach in the previous paragraph to update S_u. Since each insertions can only cause a constant number of internal nodes that are up to k_0 levels above the leaf level to split, this incurs $O(\frac{k_0 d\lg n}{\lg\lg n}) = O(\lg^{1+\epsilon} n)$ amortized cost per insertion. If v is more than k_0 levels above the leaf level, then we use Lemma 8 to update L_{f-1} in $O(|S_u|)$ time. By the analysis in the previous paragraph, the cost of splitting v can be amortized over $d^k/2$ insertions, which is $O(d)$ per insertion. Since each insertions can potentially cause $O(\frac{\lg n}{\lg\lg n})$ nodes that are more than k_0 levels above the leaf level to split, this also incurs $O(\frac{d\lg n}{\lg\lg n}) = O(\lg^{1+\epsilon} n)$ amortized cost per insertion. Therefore, each insertion can be supported in $O((\frac{\lg n}{\lg\lg n})^2) + O(\lg^{1+\epsilon} n) = O((\frac{\lg n}{\lg\lg n})^2)$ amortized time.

We finish our proof by pointing out that the succinct global rebuilding approach of He and Munro [8] can be applied here to handle the change of the value of $\lceil \lg n \rceil$, which affects the choices of the value for d. □

5 Concluding Remarks

We have presented three new dynamic range counting structures, and to obtain these results, we designed two data structures for range sum queries, which are of independent interest. We have also developed new techniques. Our approach of deamortization on a two-dimensional array in Section 2.1 is interesting. Our attempt on combining wavelet trees and range trees in Section 4.2 is the first that combines these two very powerful data structures, and we expect to use the same strategy to solve other problems.

References

1. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. 32(6), 1488–1508 (2003)
2. Bentley, J.L.: Multidimensional divide-and-conquer. Commun. ACM 23(4), 214–229 (1980)
3. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
4. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)
5. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM 57(1) (2009)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
7. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410(43), 4414–4422 (2009)
8. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: SPIRE, pp. 334–346 (2010)
9. He, M., Munro, J.I.: Succinct representations of dynamic strings. CoRR abs/1005.4652 (2010)
10. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554 (1989)
11. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
12. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry: Theory and Applications 42(4), 342–351 (2009)
13. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: STOC, pp. 40–46 (2007)
14. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)