Amortized Bounds for Dynamic Orthogonal Range Reporting

Bryan T. Wilkinson*

Abstract

We consider the fundamental problem of 2-D dynamic orthogonal range reporting for 2- and 3-sided queries in the standard word RAM model. While many previous dynamic data structures use $O(\log n/\log\log n)$ update time, we achieve faster $O(\log^{1/2+\epsilon} n)$ and $O(\log^{2/3+\epsilon} n)$ update times for 2- and 3-sided queries, respectively. Our data structures have optimal $O(\log n/\log\log n)$ query time. Only Mortensen [13] had previously lowered the update time convincingly below $O(\log n)$, with 3- and 4-sided data structures supporting updates in $O(\log^{5/6+\epsilon} n)$ and $O(\log^{7/8+\epsilon} n)$ time, respectively. In practice, fast updates are often as important as fast queries, so we make a step forward for an important problem that has not seen any progress in recent years.

We also obtain new results for the special case of 3-sided insertion-only emptiness, showing that the difference in complexity between fully dynamic and partially dynamic 2-D orthogonal range reporting can be significant (i.e., $\Omega(\text{polylog}\, n)$ factor differences). In particular, we achieve $O((\log n\log\log n)^{2/3})$ update time and $O((\log n\log\log n)^{1/3})$ query time. At the other end of our update/query trade-off curve, we achieve $O(\log n/\log\log n)$ update time and $O(\log\log n)$ query time. In contrast, in the pointer machine model, there are only $O(\log\log n)$ factor differences between the complexities of fully dynamic and partially dynamic 2-D orthogonal range reporting.

1 Introduction

We consider various special cases of 2-D dynamic orthogonal range reporting, a fundamental problem in computational geometry. Range reporting has many applications in, for example, databases and information retrieval, since it is often useful to model objects with d attributes as points in $\mathbb{R}^d$, where each dimension represents an attribute. Performing an orthogonal range reporting query then corresponds to filtering objects with inequality filters on attributes. The 2-D orthogonal range reporting problem has been studied for over 30 years, but optimal bounds are still not known. We give the first improved bounds since the work of Mortensen [13] in 2006. In particular, we give data structures with faster updates, maintaining optimal query times. In practice, fast updates are often as important as fast queries, since fast updates allow the efficient maintenance of a larger number of specialized indices, which can then support a more diverse set of queries efficiently. We also give partially dynamic data structures with faster query times than can be achieved by fully dynamic data structures.

*MADALGO, Aarhus University, Denmark, btw@cs.au.dk. Work supported in part by the Danish National Research Foundation grant DNRF84 through the Center for Massive Data Algorithmics (MADALGO) and in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Problems. The *range reporting* problem involves maintaining a set P of n points from $\mathbb{R}^d$ so that given an online sequence of queries of the form $Q \subseteq \mathbb{R}^d$, we can efficiently compute $P \cap Q$. The output size, k, of a range reporting query Q is $|P \cap Q|$. The *range emptiness* problem requires only that the data structure decides whether or not $P \cap Q$ is empty. In *dynamic* range reporting, insertions and deletions of points to and from P may be interspersed with the online sequence of queries. In *incremental* range reporting, the set P is initially empty and there are no deletions (i.e., there are only insertions and queries). In *orthogonal* range reporting, queries are restricted to the set of axis-aligned hypercubes. That is, a query must be of the form $[\ell_1, h_1] \times [\ell_2, h_2] \times \cdots \times [\ell_d, h_d]$. An *s-sided* query range for $d \leq s \leq 2d$, has finite boundaries in both directions along $s - d$ axes and only one finite boundary along $2d - s$ axes. For example, the range $[\ell, r] \times (-\infty, t]$ is a 3-sided 2-D query range.

In the *range minimum query (RMQ)* problem, we maintain an array A of n elements from $\mathbb{R}$ so that given an online sequence of queries of the form $(i, j) \in [n] \times [n]$ such that $i \leq j$, we can efficiently compute $\min\{A[k] \mid i \leq k \leq j\}$. That is, we compute the minimum element in a given query subarray. RMQ can be applied to solve the lowest common ancestor problem in trees and it is often used in solutions to textual pattern matching problems. In the dynamic variant of this problem, an update $(i, v) \in [n] \times \mathbb{R}$ sets $A[i]$ to v. In the *decremental* variant of the problem, the update may not increase $A[i]$. That is, if $A[i]$ contains u prior to the update, then it must be that $v \leq u$.

We consider 2- and 3-sided 2-D orthogonal range reporting. In particular, we consider dynamic reporting as well as incremental emptiness. We also consider RMQ due to its close connection to 3-sided emptiness.

Model. Our model of computation is the standard w-bit word RAM model. We assume that our points have integer coordinates that each fit in a single word, so $P \subseteq [U]^d$, where $w \geq \log U$. Similarly, the bounds of each query range are from $[U]$. We make the standard assumption that $w \geq \log n$ so that an index into our input array fits in a single word.

Previous Results. In the pointer machine model, the best known bounds for dynamic orthogonal range reporting are $O(n \log n)$ space, and $O(\log n \log \log n)$ update time, and $O(\log n \log \log n + k)$ query time. These bounds are achieved by augmenting range trees [5] with dynamic fractional cascading [12]. The doubly logarithmic factors can be eliminated for the partially dynamic variants of the problem. The priority search tree [11] solves 3-sided dynamic orthogonal range reporting optimally in the pointer machine model with only linear space, $O(\log n)$ update time, and $O(\log n + k)$ query time.

In the word RAM model, sublogarithmic time operations were first obtained for 3-sided queries. Willard [15] reduces both the update time and the query time of the priority search tree to $O(\log n / \log \log n)$. Let T_u be the time for an update operation and let T_q be the time for a query operation. Alstrup et al. [1] give a cell probe lower bound for 2-D orthogonal range emptiness, showing that $T_q = \Omega(\log n / \log(T_u \log n))$. This lower bound implies a lower bound of $\Omega(\log n / \log \log n)$ on the time per operation (i.e., $\max\{T_u, T_q\}$). These lower bounds hold even when restricted to 2-sided queries. Crucially, they hold for the amortized bounds of fully dynamic data structures as well as for the worst-case bounds of partially dynamic data structures, but not for the amortized bounds of partially dynamic data structures. For some intuition, imagine an adversary forces an extremely expensive insertion. In the fully dynamic case, it can then undo the insertion with a

Problem	Update Time	Query Time
2-sided reporting	$\log^{1/2+\epsilon} n$	$\log n/\log\log n + k$
2-sided reporting	$(\log n \log\log n)^{1/2}$	$\log n + k$
3-sided reporting	$\log^{2/3+\epsilon} n$	$\log n/\log\log n + k$
3-sided reporting	$(\log n \log\log n)^{2/3}$	$\log n + k$
RMQ	$\log^{2/3+\epsilon} n$	$\log n/\log\log n$
RMQ	$(\log n \log\log n)^{2/3}$	$\log n$
2-sided incremental emptiness	$\text{pred} + \log\log n$	$\text{pred} + \log\log n$
3-sided incremental emptiness	$\text{pred} + (\log n \log\log n)^{2/3}$	$\text{pred} + (\log n \log\log n)^{1/3}$
3-sided incremental emptiness	$\text{pred} + \log n/\log\log n$	$\text{pred} + \log\log n$
decremental RMQ	$(\log n \log\log n)^{2/3}$	$(\log n \log\log n)^{1/3}$
decremental RMQ	$\log n/\log\log n$	$\log\log n$

Table 1: Our results

deletion and repeat the expensive operation over and over again. However, in the incremental case, the adversary cannot repeat the operation and the rest of the insertions might require very little work, resulting in low amortized time per insertion.

Despite mounting evidence that the true update and query times for 2-D orthogonal range reporting might be $\Theta(\log n/\log\log n)$, Mortensen [13] showed that updates could in fact be made faster. For 3-sided queries, his data structure supports updates in $O(\log^{5/6+\epsilon} n)$ time. He gives a worst-case deterministic fully dynamic data structure for 4-sided queries that requires $O(n\log^{7/8+\epsilon} n)$ space, $O(\log^{7/8+\epsilon} n)$ update time, and optimal $O(\log n/\log\log n)$ query time. The speed up in update time is achieved by reducing the number of bits required to specify points and packing multiple points into a word for efficient parallel processing. Importantly, Mortensen shows that update time can be reduced convincingly (i.e., by an $\Omega(\text{polylog}\, n)$ factor) below $O(\log n)$ while maintaining optimal $O(\log n/\log\log n)$ query time.

Our Results. We make an initial effort to determine the complexity of updates by first considering the 2- and 3-sided special cases and allowing amortization and randomization. All of our update and query bounds are amortized. The only source of randomization in our data structures originates from our use of randomized dynamic predecessor search data structures [14]. It is possible to eliminate all randomization from our data structures by instead using the deterministic predecessor search data structure of Andersson and Thorup [2], at the expense of doubly logarithmic factors. All of our data structures require only linear space.

Our results are summarized in Table 1. Note that $\epsilon > 0$ is an arbitrarily small constant and $k = |P \cap Q|$ is the output size of a reporting query. Each of our reporting data structures with a query time of the form $O(t(n) + k)$ can be easily adapted to decide emptiness in $O(t(n))$ time. Also, pred is the cost of an update or query to a linear-space dynamic predecessor search data structure containing at most n elements from a universe of size U. These updates and queries can be performed in $O(\log\log U)$ [14] time, $O(\log_w n)$ time [7], or $O(\sqrt{\log n/\log\log n})$ time [2]. For each 3-sided emptiness result there is a corresponding RMQ result due to the close relationship between these two problems.

For 2- and 3-sided reporting, we obtain data structures with optimal query time and update times of the form $O(\log^\gamma n)$ for $\gamma < 1$. Mortensen [13] previously gave a 3-sided data structure with $\gamma = 5/6 + \epsilon$. We improve the exponent to $\gamma = 2/3 + \epsilon$ for 3-sided queries and even further to $\gamma = 1/2 + \epsilon$ for 2-sided queries. It is plausible that an exponent of $\gamma = 1/2$ is optimal.

We circumvent the lower bound of Alstrup et al. [1] by considering amortized solutions to partially dynamic problems. We give an optimal 2-sided data structure with operations that run in only $O(\text{pred} + \log\log n)$ time. We finally give 3-sided incremental emptiness data structures. At one end of our update/query trade-off curve, we obtain update and query times that are both of the form $O(\log^\gamma n)$ for $\gamma < 1$, showing that there can be $\Omega(\text{polylog}\, n)$ factor differences between the complexities of fully dynamic and partially dynamic reporting problems. In contrast, in the pointer machine model, there are only $O(\log\log n)$ factor differences between the complexities of the fully dynamic and partially dynamic problems. At the other end of our trade-off curve, we obtain queries that run in only $O(\text{pred} + \log\log n)$ time. These latter bounds could plausibly be optimal.

Our Approach. Mortensen [13] develops a framework which essentially uses a high-fanout priority search tree to reduce the problem of 3-sided range reporting to the same problem, but with fewer points, of which multiple can be packed into a single word. Once points are packed into words, we encounter a situation that can be modeled very closely to the external memory model. An important divergence from the work of Mortensen [13] is that we start by explicitly designing external memory data structures. This both simplifies our presentation and allows reuse of previous external memory results. We both reuse the framework of Mortensen [13] and extend it to consider the more special case of 2-sided queries and incremental emptiness. In the latter case, it turns out that using a high-fanout range tree yields better results than the high-fanout priority search tree of Mortensen [13].

2 Preliminaries

Dynamic and Static Axes. An *axis* is a set of coordinates from which points and query ranges obtain their coordinates along some dimension. Axes may also specify restrictions on how points are assigned coordinates. A *standard axis* is an axis with coordinates from $[U]$ that accommodates at most n points with distinct coordinates. Our geometric problems have two standard axes.

A standard technique in static orthogonal range reporting is rank space reduction. Instead of using the full coordinates of points, we use their x- and y-ranks. Thus, each point is specified by ranks from $[n]$ instead of coordinates from $[U]$. To handle a query, we must perform a predecessor search for each side of the query range in order to reduce the query range to rank space. In the dynamic case, rank space reduction as such does not work since inserting a new point might require updating the ranks of all of the other points. To encapsulate this problem, Mortensen [13] introduces the concept of a *dynamic axis*. A dynamic axis has coordinates from the set of nodes of a linked list such that one node u is less than another node v if u occurs prior to v in the linked list. There is a bijection between points and linked list nodes. Initially, a new point has no node, so an update specifies its position along the axis with a pointer to its predecessor node. A new node is inserted to the linked list after the predecessor node. Query coordinates are specified by pointers to linked list nodes. A dynamic axis of size u can accommodate at most u points/nodes.

It is easy to reduce a standard axis to a dynamic axis via predecessor search. This is why some

of our results include pred terms. In some cases, in choosing pred $= O(\sqrt{\log n / \log\log n})$, the pred terms are dominated by the other terms of the update/query times. In these cases, we have omitted the pred terms.

In solving the problem encapsulated by dynamic axes, we will end up with some form of reduction of coordinates to a smaller set of integers (though not to the set of ranks as in rank space reduction). A *static axis* of size u has coordinates from $[u]$ and accommodates at most u points with distinct coordinates.

3-Sided Emptiness and RMQ. For any set of points P, a 3-sided range of the form $[\ell, r] \times (-\infty, t]$ is empty if and only if the lowest point of P in the slab $[\ell, r] \times (-\infty, +\infty)$ has y-coordinate less than or equal to t. Assume we have an array A of n elements from $[U]$. We construct a specific set P of points such that for each $i \in [n]$, there is a point $(i, A[i]) \in P$. Then, the result of an RMQ (i, j) on A is the y-coordinate of the lowest point of P in the slab $[i, j] \times (-\infty, +\infty)$. So, both types of queries can be solved by finding the lowest point in a query vertical slab. This is precisely how our emptiness data structures operate. Consider an update (i, v) which sets $A[i]$, initially containing u, to v. We update P by deleting (i, u) and inserting (i, v). So, an RMQ update is at most twice as expensive as a 3-sided emptiness update. In the case of decremental RMQ, we are guaranteed that $v \leq u$. Assuming we have a 3-sided incremental emptiness data structure that can handle multiple points with the same x-coordinate, it is sufficient to update P by only inserting (i, v) without first deleting (i, u). It is easy to modify our incremental emptiness data structures to handle multiple points with the same x-coordinate by, in this case, implicitly deleting (i, u). Thus, (decremental) RMQ can thus be solved by a 3-sided (incremental) emptiness data structure with a static x-axis of size n and a standard y-axis. In order to reduce the standard y-axis to a dynamic axis, we require predecessor search along the y-axis only for updates, since an RMQ is specified with only two x-coordinates and no y-coordinates.

Notation. We borrow and extend the notation of Mortensen [13] to specify various different 2-D range reporting problems. A dynamic problem is specified by $T^s(t_x : u_x, t_y : u_y)$ where $T \in \{\mathrm{R}, \mathrm{R_I}, \mathrm{E}, \mathrm{E_I}\}$; $d \leq s \leq 2d$; $t_x, t_y \in \{\mathrm{d}, \mathrm{s}\}$; and $1 \leq u_x, u_y \leq n$. Depending on T, the problem is either s-sided dynamic reporting (R), s-sided incremental reporting ($\mathrm{R_I}$), s-sided dynamic emptiness (E), or s-sided incremental emptiness ($\mathrm{E_I}$). For each axis $a \in \{x, y\}$, depending on t_a, the axis is either dynamic (d) or static (s). The axis has size u_a. We assume without loss of generality that 2-sided queries are of the form $(-\infty, r] \times (-\infty, t]$ and 3-sided queries are of the form $[\ell, r] \times (-\infty, t]$.

We say a data structure has performance (Update : T_u, Query : T_q, Space : S) if it requires $O(T_u)$ update time, $O(T_q)$ query time (or $O(T_q + k)$ query time for reporting problems), and $O(S)$ words of space. We originally defined n as $|P|$, but we redefine it now to an upper bound on $|P|$ that we know in advance and for which we have the guarantee that $w \geq \log n$. We give the performances of data structures as functions of the sizes of their axes rather than of $|P|$. Our final data structures have axes of size n. We can eliminate the requirement of knowing n in advance and simultaneously ensure that performance is a function of $|P|$ rather than n by initially building a data structure with axes of constant size, rebuilding the data structure with axes twice as large whenever the data structure fills up, and rebuilding the data structure with axes half as large whenever the data structure is only a quarter full (except when the axes are already at their initial constant size).

When considering external memory data structures, we denote the input size by N and the output size by K, in keeping with the conventions of the external memory literature. A block can

hold B elements and internal memory can hold M elements. In the external memory setting, we say a data structure has performance (Update : T_u, Query : T_q, Space : S) if it requires $O(T_u)$ I/Os for an update, $O(T_q)$ I/Os for a query (or $O(T_q + K/B)$ I/Os for a reporting query), and $O(S)$ blocks of space.

3 Data Structures

3.1 Dynamic Reporting

An important technique we will apply involves performing some form of reduction of the coordinates of small sets of $u \leq n$ points in such a way that we can describe each point using only $O(\log u)$ bits. Thus, since $w \geq \log n$, we can pack $O(\log n / \log u)$ of these points into a single word. Using standard word operations and table lookups, we can then operate on multiple points at unit cost. A similar situation arises in the external memory model: multiple points fit in a block and we can operate on all of the points in all of the blocks in internal memory at no cost. In fact, it is possible to simulate external memory algorithms to solve problems on packed words. For this reason, we begin with the design of an external memory reporting data structure, which we intend to simulate in the word RAM.

Lemma 1. *For any $f \in [2, B]$, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{s} : N)$ *with performance* (Update : $(f/B)\log_f N$, Query : $\log_f N + (f/B)K$, Space : N/B).

Proof. The data structure is a modified I/O tournament tree (I/O-TT) [10]. The I/O-TT is an I/O-efficient priority queue data structure, but we adapt it to answer 3-sided range reporting queries. The I/O-TT stores elements, each of which consists of a key and a priority. We map a point to an element so that the point's x-coordinate is the element's key and the point's y-coordinate is the element's priority.

The I/O-TT is a static binary tree on x-coordinates where each node is associated with a set of at most B points and an update buffer of size B[1]. The sets of points are in heap order by y-coordinate and a non-root node may only contain points if its parent is full. Updates are initially sent to the root node. An update is inserted into a node's update buffer if it cannot be resolved in the node directly. When a node's update buffer is filled, the updates are flushed down to the node's children (by x-coordinate) in $O(1/B)$ amortized I/Os per update. The total cost of an update is thus $O((1/B)h)$ I/Os, where h is the height of the I/O-TT. A query to the I/O-TT involves finding the point with minimum y-coordinate. This point is in the root and can thus be found in $O(1)$ I/Os. The I/O-TT requires $O(N/B)$ blocks of space.

Our first modification is to use the I/O priority search tree [3] query algorithm to handle 3-sided reporting queries instead of priority queue queries. Ignoring the update buffers, the I/O-TT is essentially an I/O priority search tree with fanout of 2 instead of B. The query algorithm of the I/O priority search tree performs $O(1)$ I/Os in $O(h + K/B)$ nodes. However, this query algorithm is only correct if the updates in the buffers of all of these nodes, as well as all of their ancestors, have been flushed. We flush these $O(h + K/B)$ buffers as the query algorithm proceeds. These flushing operations may cascade to descendant nodes, but all I/Os performed while cascading are

[1] In the original description of the I/O-TT these sets have size M instead of B, but it is easy to see that such large sets are not necessary to achieve the desired bounds.

charged to update operations. Therefore, there are only $O(h + K/B)$ additional I/Os that cannot be charged to update operations.

Our second modification is to increase the fanout of the I/O-TT to f. To ensure that we can still flush updates from a node to its children efficiently, we must reduce the sizes of the sets of points and update buffers to B/f. In this way, we can store these sets for all of a node's children in $O(1)$ blocks and thus flush an update buffer in $O(f/B)$ amortized I/Os per update. As a side-effect, the I/O priority search tree query algorithm then visits $O(h + (f/B)K)$ nodes. Since the height of the I/O-TT is now $O(\log_f N)$, we are done. □

Let $m \geq 2$ be the smallest constant such that any of our external memory data structures can operate with $M = mB$. Let $\delta > 0$ be a sufficiently small constant to be determined later.

Lemma 2. *For any $u \in [n^{\delta/2m}]$ and $f \in [2, (\delta/2m) \log n / \log u]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{s} : u)$ *with performance* $(\mathrm{Update} : 1 + f \log^2 u / \log n, \mathrm{Query} : \log_f u, \mathrm{Space} : u)$.

Proof. We simulate the data structure of Lemma 1 in the word RAM. Since an element (a point or block pointer) requires at most $2 \log N$ bits, if $N = u$, we can store $(\delta/2) \log n / \log u$ elements in $\delta \log n$ bits in a single word. We designate a single word to act as our simulated main memory containing up to $M = (\delta/2) \log n / \log u$ elements. The rest of our actual main memory acts as our simulated external memory: it is divided into blocks of $B = (\delta/2m) \log n / \log u$ elements such that $M = mB$. Since $u \leq n^{\delta/2m}$, each block can hold at least one element. A constant number of standard word operations can transfer a simulated block into or out of our simulated main memory.

The update and query algorithms of Lemma 1 may perform arbitrary manipulations of the elements in main memory between I/Os. Since the algorithms are finite and do not depend on the input size, there are only a constant number of different manipulations of the elements in main memory. Since our simulated main memory can be described in exactly $\delta \log n$ bits, we use a global lookup table containing n^δ entries of $\delta \log n$ bits to support constant-time simulations of each of these manipulations. Since our final data structures reuse these lookup tables for many instances of our intermediate data structures (such as this one), we do not include the space for global lookup tables in the space bounds for our intermediate data structures. The lookup tables can be built in time polynomial in their number of entries. We set δ sufficiently small so that they can be built in $O(n)$ time. Whenever we rebuild our final data structures to handle constant factor changes in n, we can also afford to rebuild our global lookup tables.

Substituting $N = u$ and $B = (\delta/2m) \log n / \log u$ into Lemma 1 (and converting from space consumption in blocks to space consumption in words) gives the desired bounds. We also need to add a constant term to the running times of the update and query algorithms, since they may read from and write to simulated main memory a constant number of times without performing any simulated I/Os. In the external memory model, these reads and writes to main memory are free, but they are not free in our simulation. Also, since we need to report each output point individually, the $(f/B)K$ term in the query time simply becomes k. □

We now extract and slightly generalize two techniques from the framework of Mortensen [13] and encapsulate them in the following two lemmata. The first technique involves converting a data structure with a static axis to a data structure with a dynamic axis. It can also be used as a space reduction technique.

Lemma 3 (Lemma 24 of Mortensen [13])**.** *Given a data structure for $T^s(t_x : u, t_y : u)$ for $u \in [n]$ with performance* $(\mathrm{Update} : T_u, \mathrm{Query} : T_q, \mathrm{Space} : u \operatorname{polylog} u)$ *where $t_a = \mathrm{s}$, for some $a \in \{x, y\}$,*

and queries have only one finite boundary along the other axis, then there exists a data structure for the same problem with $t_a = \mathrm{d}$ *and performance* $(\text{Update} : \log\log u + T_u, \text{Query} : \log\log u + T_q, \text{Space} : u)$.

The second technique involves converting a data structure that can only handle $u \leq n$ points to a data structure that handles n points. This is achieved using a u-ary priority search tree.

Lemma 4 (Lemma 23 of Mortensen [13])**.** *Given a data structure for* $T^s(\mathrm{s} : u, \mathrm{d} : u)$ *for* $u \in [n]$ *with performance* $(\text{Update} : T_u, \text{Query} : T_q, \text{Space} : u)$ *where queries have only one finite boundary along the* y*-axis, then there exists a data structure for* $T^s(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : (\log n/\log u)(\log\log n + T_u), \text{Query} : (\log n/\log u)(\log\log n + T_q), \text{Space} : n)$.

We note that $\mathrm{R}^2(\mathrm{s} : u, \mathrm{s} : u)$ is a special case of $\mathrm{R}^3(\mathrm{s} : u, \mathrm{s} : u)$, so the data structure of Lemma 2 also solves $\mathrm{R}^2(\mathrm{s} : u, \mathrm{s} : u)$. Since 2-sided queries have only one finite boundary along the x-axis, we can apply Lemma 3 to the data structure of Lemma 2 in order to obtain a dynamic y-axis, which then allows us to apply Lemma 4 (with $u = 2^{(\delta/2m)((1/f)\log n \log\log n)^{1/2}}$) to handle n points. Finally, another application of Lemma 3 converts the still static x-axis into a dynamic axis and we obtain the following theorem.

Theorem 5. *For any* $f \in [2, \log n/\log\log n]$*, there exists a data structure for* $\mathrm{R}^2(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : (f \log n \log\log n)^{1/2}, \text{Query} : (f \log n \log\log n)^{1/2} + \log_f n, \text{Space} : n)$.

We obtain the 2-sided reporting data structures of Table 1 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 2\epsilon$, or alternatively setting $f = 2$. All of the techniques we have seen so far carry through for both 2- and 3-sided queries, except for the first application of Lemma 3. Therefore, in order to obtain results for 3-sided queries we only need some way to support a dynamic y-axis for 3-sided queries.

We proceed by designing an external memory data structure for online list labelling which we will use to augment our original external memory data structure with a dynamic y-axis. In the online list labelling problem, we maintain an assignment of labels from some universe of totally ordered labels to linked list nodes so that the labels are monotonically increasing along the list. Assume the linked list has at most n nodes. For a universe of size $O(n)$, an insertion or deletion requires that $\Theta(\log^2 n)$ worst-case nodes are relabelled [4]. However, for a universe of size $O(n^2)$, an insertion or deletion can be limited to relabelling only $O(\log n)$ amortized nodes (this is a folklore modification of [9]). In the external memory setting, we consider a linked list to be a linked list of blocks, where each block contains an ordered array of elements with unique ids. A pointer to a specific element is then its id along with a pointer to its containing block. An insertion is specified by the element to be inserted and a pointer to its intended predecessor. A deletion is specified by a pointer to the element to be deleted. A relabelling consists of a triple (i, ℓ, ℓ'), where i is the id of the element being relabelled, ℓ is its old label, and ℓ' is its new label.

Lemma 6. *There exists an external memory data structure for online list labelling of up to* N *elements with labels from a universe of size* $O(N^2)$ *that, upon each update, reports* $O(\log N)$ *amortized relabellings in* $O(1 + (1/B)\log N)$ *amortized I/Os.*

Proof. Upon an insertion or deletion, we load the target block in a single I/O and add or remove the element. We maintain block sizes of $\Theta(B)$ elements (except when there are too few elements to fill a single block) by splitting and merging adjacent blocks whenever they become a constant

factor too large or too small. Each split or merge can be charged to $\Omega(B)$ updates. Each split or merge causes an insertion to or a deletion from the linked list of blocks. We maintain an online list labelling data structure for a polynomially large universe [9] to assign the $O(N/B)$ blocks labels from a universe of size $O((N/B)^2)$ with $O(\log(N/B))$ amortized relabellings per update. If an element has rank r in the block with label ℓ, then its label is $B\ell + r - 1$, which is bounded by $O(N^2)$. For each block relabelled, we can report all of the relabellings for all of its elements in $O(1)$ I/Os. □

Lemma 7. *For any $f \in [2, B]$, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$ *with performance* $(\text{Update} : 1 + (f/B)\log_f N \log N, \text{Query} : \log_f N + (f/B)K, \text{Space} : N/B)$.

Proof. We maintain the online list labelling data structure of Lemma 6 on the y-axis list of our data structure of type $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$, using x-coordinates as the unique ids. We build the data structure D of Lemma 1 on an asymmetric $N \times N'$ grid, where $N' = O(N^2)$, instead of an $N \times N$ grid. The bounds of the data structure increase by only constant factors, but we obtain the requirement that points must be a constant factor larger to store the larger y-coordinates. Queries and updates to our data structure include y-axis pointers, which we convert to y-coordinates in $[N']$ in constant I/Os using the online list labelling data structure. We then forward the operations on to D, including the given x-coordinates and our computed y-coordinates. Upon an update, the online list labelling data structure reports $O(\log N)$ relabellings in $O(1 + (1/B)\log N)$ I/Os. In a scan through the relabellings, we convert each relabelling of the form (i, ℓ, ℓ') to a deletion of (i, ℓ) from D and an insertion of (i, ℓ') to D. So, our update time increases by an $O(\log N)$ factor. □

We can now simulate the data structure of Lemma 7 similarly to the simulation of Lemma 2.

Lemma 8. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m))\log n/\log u]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$ *with performance* $(\text{Update} : 1 + f\log^3 u/\log n, \text{Query} : \log_f u, \text{Space} : u)$.

Proof. We simulate the data structure of Lemma 7 as in Lemma 2, except that elements now require a constant factor more bits due to the polynomially large universe used by the online list labelling data structure. In internal memory, the y-axis list is a standard linked list. To support conversions from pointers to y-axis list nodes to simulated external memory pointers, we store in each y-axis list node the external memory element's unique id (x-coordinate) and a pointer to the simulated block containing the element. Whenever there is a split or a merge, we must update $O(B)$ pointers from the y-axis list to simulated blocks. These updates can be charged to the $\Omega(B)$ updates that are required to cause a split or a merge. When a point is reported by the simulated data structure, its x-coordinate is included. To obtain the associated y-axis list node, we simply maintain (in $O(1)$ time per update) an array of size u containing, for each x-coordinate, the associated y-axis list node. □

Applications of Lemmata 4 and 3 to the data structure of Lemma 8 with $\log u = (\delta/O(m))((1/f)\log n \log\log n)^{1/3}$ yield the following theorem.

Theorem 9. *For any $f \in [2, \log n/\log\log n]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : f^{1/3}(\log n \log\log n)^{2/3}, \text{Query} : f^{1/3}(\log n \log\log n)^{2/3} + \log_f n, \text{Space} : n)$.

We obtain the 3-sided reporting and RMQ data structures of Table 1 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 3\epsilon$, or alternatively setting $f = 2$.

3.2 Incremental Emptiness

We will require a data structure of Mortensen [13] that solves a problem called *colored predecessor search* in a linked list L with colored nodes. The data structure supports the following operations:

- insert(u, v, c): inserts node u with color c after node v
- delete(u): deletes node u
- change(u, c): changes the color of u to c
- predecessor(u, c): returns the last node of color c that is not after u

Lemma 10 (Theorem 15 of Mortensen [13])**.** *There exists a linear-space data structure for colored predecessor search in a linked list L that supports all operations in $O(\log\log|L|)$ time.*

We will also require a solution to a problem called *subsequence predecessor search* in which we maintain a primary linked list L and a set of secondary linked lists $\mathcal{S}$. For the purposes of this problem, let $n = |L| + \sum_{S\in\mathcal{S}} |S|$ be the total size of all lists. Each node u in a secondary list $S \in \mathcal{S}$ is associated with a primary node $p(u)$, such that mapping S with function p yields a subsequence of L. For any primary node v, there may be multiple nodes u from different secondary lists for which $p(u) = v$. We require the following operations:

- $\text{insert}_\text{p}(u, v)$: inserts primary node u after v in L
- $\text{delete}_\text{p}(u)$: deletes primary node u from L (only if there is no secondary node v with $p(v) = u$)
- $\text{insert}_\text{s}(u, S)$: inserts a new secondary node v with $p(v) = u$ to secondary list S (preserving the subsequence ordering of S)
- $\text{delete}_\text{s}(u)$: deletes secondary node u from its secondary list
- predecessor(u, S): returns the last secondary node v in secondary list S such that $p(v)$ is not after primary node u in L
- secondaries(u): returns all secondary nodes v such that $p(v) = u$ for primary node u
- primary(u): returns $p(u)$ for secondary node u

Lemma 11. *There exists a data structure for subsequence predecessor search that requires $O(n)$ space, $O(\log\log n)$ time for updates and predecessor queries, $O(1+k)$ time to report the k secondary nodes associated with a primary node, and $O(1)$ time to find the primary node associated with a secondary node.*

Proof. We construct an aggregate doubly linked list A containing, for each primary node u in order, pointers to all secondary nodes v such that $p(v) = u$ followed by a pointer to u. We also store pointers from primary and secondary nodes to their corresponding aggregate nodes. We color each node of A with the list of the node to which it stores a pointer. For each aggregate node pointing to a secondary node u, we also store an extra pointer to $p(u)$ which does not affect the aggregate node's color. We build the colored predecessor search data structure of Lemma 10 on A, which has size n. We can then support a subsequence predecessor query in secondary list S with a colored

predecessor query in A with color S. Reporting the secondary nodes of a primary node requires a walk in A. Reporting $p(u)$ for secondary node u requires following two pointers. It is a simple exercise to verify that all of the update operations can be supported using a constant number of colored predecessor search operations and a constant amount of pointer rewiring. □

We begin with an efficient data structure for 2-sided incremental emptiness.

Theorem 12. *There exists a data structure for* $\mathrm{E}_{\mathrm{I}}^2(\mathrm{d}:n,\mathrm{d}:n)$ *with performance* (Update : $\log\log n$, Query : $\log\log n$, Space : n).

Proof. Without loss of generality, we handle 2-sided queries of the form $(-\infty,r]\times(-\infty,t]$. It is sufficient to find the lowest point in the query range and compare its y-coordinate to t. Since y-coordinates are linked list nodes, we require the list order data structure of Dietz and Sleator [6] to compare them. The lowest point with x-coordinate at most r is the minimal point whose x-coordinate is the predecessor of r. A point $p=(p_x,p_y)\in P$ is minimal if and only if there does not exist another point $(p'_x,p'_y)\in P\setminus\{p\}$ such that $p'_x<p_x$ and $p'_y<p_y$. We maintain the colored predecessor search data structure of Lemma 10 on the x-axis list. We color nodes associated with minimal points one color and all other nodes another color. We can thus find the minimal point whose x-coordinate is the predecessor of r in $O(\log\log n)$ time. A newly inserted point may be a minimal point, which can result in many, say k, previously minimal points becoming non-minimal. For each of these k points we must execute a color change operation, which requires $O(k\log\log n)$ time. However, in the incremental setting, a point p can only transition from minimal to non-minimal at most once, so we can charge the $O(\log\log n)$ cost of the color change of p to the insertion of p. □

Given that 2-sided incremental emptiness queries can be supported very efficiently, we can use an alternative to Lemma 4 to handle n points given a data structure for only $u\le n$ points: a u-ary range tree instead of a u-ary priority search tree. The range tree requires superlinear space; however, we can reduce space to linear once again with an application of Lemma 3.

Lemma 13. *For any* $u\in[n^{\delta/O(m)}]$ *and* $f\in[2,(\delta/O(m))\log n/\log u]$, *there exists a data structure for* $\mathrm{E}_{\mathrm{I}}^3(\mathrm{s}:n,\mathrm{d}:n)$ *with performance* (Update : $(\log n/\log u)(\log\log n+f\log^3 u/\log n)$, Query : $\log\log n+\log_f u$, Space : $n\log n/\log u$)

Proof. We build a static u-ary range tree on x-coordinates. In each internal node of the range tree, we build 2 auxiliary data structures on the points stored in the node. Each of these data structures has its own y-axis list corresponding to a subsequence of the original y-axis list. The total size of all y-axis lists is $O(nh)$, where h is the height of the range tree. We build the subsequence predecessor search data structure of Lemma 11 using the original y-axis list as the primary list and all other y-axis lists as secondary lists. This data structure requires $O(nh)$ space. Now, given a query or update, we can translate it in $O(\log\log(nh))=O(\log\log n)$ time into the coordinate space of a specific auxiliary data structure. Also, we can convert a point in the coordinate space of a specific auxiliary data structure to our original coordinate space in $O(1)$ time.

One of our auxiliary data structures is that of Theorem 12, which handles 2-sided ranges of the form $(-\infty,r]\times(-\infty,t]$ and $[\ell,\infty)\times(-\infty,t]$. The other is a data structure which handles 3-sided ranges that are aligned to the boundaries of the x-ranges of the node's children. We call these *aligned* queries. This data structure for aligned queries is the data structure of Lemma 8 built on the lowest points in each of the node's chidren. An aligned query is empty if and only if it contains

none of the lowest points in each of the node's children. Since a node has u children, the axes of our data structure for aligned queries have size u. All of these auxiliary data structures require a total of $O(nh)$ space.

Given an insertion of a point p, we insert p into the 2-sided data structures of all $O(h)$ nodes of the range tree to which p belongs at a cost of $O(h \log\log n)$. If p becomes the lowest point in the child of some node u, we must delete the old lowest point from the data structure of Lemma 8 in u and insert p. This introduces another term of $O(hf \log^3 u / \log n)$ to our update time.

Given a query of the form $[\ell, r] \times (-\infty, t]$, we find the lowest common ancestor (LCA) in the range tree of the leaf corresponding to x-coordinate ℓ and the leaf corresponding to x-coordinate r. Using a linear-space data structure, this LCA operation requires only $O(1)$ time [8]. In the resulting node u, we can decompose the query into an aligned query and two 2-sided queries in children of u. Our range is empty if and only if all 3 of these subranges are empty. The 3 emptiness queries to the auxiliary data structures require $O(\log\log n + \log_f u)$ time. Since the height of the range tree is $O(\log_u n) = O(\log n / \log u)$, we are done. □

An application of Lemma 3 to the data structure of Lemma 13 yields the following theorem.

Theorem 14. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m)) \log n / \log u]$, there exists a data structure for $\mathrm{E}^3_{\mathrm{I}}(\mathrm{d} : n, \mathrm{d} : n)$ with performance* (Update : $(\log n / \log u)(\log\log n + f \log^3 u / \log n)$, Query : $\log\log n + \log_f u$, Space : n).

We obtain the 3-sided incremental emptiness and decremental RMQ data structures of Table 1 by setting $u = 2^{(\log n \log\log n)^{1/3}}$ and $f = 2$, or alternatively setting $u = 2^{(\log\log n)^2}$ and $f = \log^{\epsilon'} n$ for a sufficiently small $\epsilon' > 0$.

References

[1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–544, 1998.

[2] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM (JACM)*, 54(3), 2007.

[3] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999.

[4] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[5] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM (CACM)*, 23(4):214–229, 1980.

[6] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[7] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[8] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[9] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 417–431, 1981.

[10] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Annual IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 169–176, 1996.

[11] E. M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[12] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[13] C. W. Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM Journal of Computing*, 35(6):1494–1525, 2006.

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

[15] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal of Computing*, 29(3):1030–1049, 2000.