I/O-Efficient Spatial Data Structures for Range Queries

Lars Arge Kasper Green Larsen

MADALGO[*], Department of Computer Science, Aarhus University, Denmark
E-mail: `large@madalgo.au.dk,larsen@madalgo.au.dk`

1 Introduction

Range reporting is a one of the most fundamental topics in spatial databases and computational geometry. In this class of problems, the input consists of a set of geometric objects, such as points, line segments, rectangles etc. The goal is to preprocess the input set into a data structure, such that given a query range, one can efficiently report all input objects intersecting the range. The ranges most commonly considered are axis-parallel rectangles, halfspaces, points, simplices and balls.

In this survey, we focus on the planar orthogonal range reporting problem in the external memory model of Aggarwal and Vitter [2]. Here the input consists of a set of N points in the plane, and the goal is to support reporting all points inside an axis-parallel query rectangle. We use B to denote the disk block size in number of points. The cost of answering a query is measured in the number of I/Os performed and the space of the data structure is measured in the number of disk blocks occupied, hence linear space is $O(N/B)$ disk blocks.

Outline. In Section 2, we set out by reviewing the classic B-tree for solving one-dimensional orthogonal range reporting, i.e. given N points on the real line and a query interval $q = [q_1, q_2]$, report all T points inside q. In Section 3 we present optimal solutions for planar orthogonal range reporting, and finally in Section 4, we briefly discuss related range searching problems.

2 The B-tree

A B-tree [10] is constructed in the following manner from a sequence of N points on the real line: First sort the points according to their coordinates and partition them

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

into N/B groups of B consecutive points each. Each group is stored in one disk block and we think of these blocks as the leaves of the B-tree. Each leaf node v is naturally associated to an *x-range* $X_v = (x_v^1, x_v^2]$ where x_v^1 is the largest coordinate stored in the leaf preceding v ($x_v^1 = -\infty$ for the first leaf) and x_v^2 is the largest coordinate of a point in v (we let $x_v^2 = \infty$ for the last leaf). We call the coordinates defining these intervals "splitter" values and we assume without loss of generality that all coordinates are distinct.

To guide searches, we construct a B-ary tree on top of the sorted sequence of leaves. Each internal node v is again associated to an x-range X_v, which is the union of the x-ranges of the leaves in the subtree rooted at v. We augment each internal node with the $O(B)$ splitter values of its children and store the splitter values in $O(1)$ disk blocks.

To answer a query $q = [q_1, q_2]$, we start at the root node of the B-tree and traverse the two paths towards the leaves whose x-ranges contain q_1 and q_2, respectively. These paths are easily determined from the splitter values stored in the nodes along the paths. For each node v on the paths we do the following: We examine the x-range of all children of v, and for each child with an x-range completely inside q, we traverse all the leaves of the corresponding subtree and report all points stored there. Clearly this procedure correctly reports all points in the query range.

Following the two paths towards the leaves containing q_1 and q_2 costs $O(\log_B N)$ I/Os. For every subtree that is traversed because its x-range is inside q, we charge the traversal to the output size, i.e. this costs at most $O(T/B)$ I/Os. Hence we conclude that the B-tree uses linear space and answers queries in $O(\log_B N + T/B)$ I/Os.

When solving higher-dimensional orthogonal range reporting, the aim is to obtain a performance that is comparable to that of the B-tree, i.e. $O(\log_B N + T/B)$ query cost and as close to linear space as possible.

3 Orthogonal Range Queries

Solutions for planar orthogonal range reporting are typically obtained by first obtaining a solution for a restricted type of query ranges, known as 3-sided queries. A 3-sided query $q = (q_1, q_2, q_3)$ asks to report all input points (x, y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$. In Section 3.1, we show how to answer these 3-sided queries in external memory. In Section 3.2 we then extend these results to standard query ranges, which we refer to as 4-sided queries.

3.1 Three-Sided Queries

In the following, we present a data structure for solving *3-sided planar range reporting*: Preprocess a set S of point in the plane such that given a 3-sided query q, we can report all points in S that are inside q.

A solution to the 3-sided range query problem can be obtained using an external *priority search tree* [5]. A basic building block in the external priority search tree is an efficient solution for 3-sided range queries when the number of input points is only $O(B^2)$. More specifically, we assume for now that for a set of $K = O(B^2)$ points, one can construct a linear space data structure such that 3-sided range queries can be answered in $O(1 + T/B)$ I/Os. We refer to such a base data structure as a "B^2–structure". In the following, we first describe the external priority search tree simply assuming the availability of this B^2–structure. We then move on to describe the B^2–structure at the end of the section.

Structure. An external priority search tree consists of a base B-tree $\mathcal{T}$ on the x-coordinates of the points in S. Recall from Section 2 that $\mathcal{T}$ stores all points in the leaves and that internal nodes store only splitter values. For the external priority search tree, we keep the splitter values in the internal nodes, but also move the points from the leaves into the internal nodes by the following recursive procedure: Start at the root node v of $\mathcal{T}$ and collect for each child v_i of v, the B points with highest y-coordinates (if existing) amongst all points stored in the leaves of the subtree rooted at v_i. We move these points out of their leaves and store the $O(B^2)$ points collected for all children of v in a B^2–structure. This B^2–structure is stored at v. We finally recurse on all $O(B)$ children of v. Observe that when this procedure terminates at the leaves, all points stored in the subtree rooted at a node v still have an x-coordinate in the x-range of v. Overall the external priority search tree uses linear space since $\mathcal{T}$ uses linear space and since each point is stored in precisely one B^2–structure.

Query. To answer a 3-sided query $q = (q_1, q_2, q_3)$ we start at the root of $\mathcal{T}$ and proceed recursively to the appropriate subtrees: When visiting a node v we first query the B^2–structure and report the relevant points. Then we advance the search to some of the children of v. The search is advanced to a child v_i if it is either along the search path for q_1 or q_2, or if the entire set of points corresponding to v_i in the B^2–structure were reported. The query procedure reports all points in the query range, since if we do not visit a child v_i with an x-range completely contained in the interval $[q_1, q_2]$, it means that at least one of the points in the B^2–structure corresponding to v_i is not in q. Since all points corresponding to v_i have an x-coordinate in the query range, there must be at least one of the points that have a y-coordinate smaller than q_3. This in turn means that none of the points in the subtree rooted at v_i can be in q since they all have even smaller y-coordinates.

That we use $O(\log_B N + T/B)$ I/Os to answer a query can be seen as follows. In each internal node v of $\mathcal{T}$ visited by the query procedure we spend $O(1 + T_v/B)$ I/Os, where T_v is the number of points reported at v. There are $O(\log_B N)$ nodes visited on the search paths in $\mathcal{T}$ to the leaf containing q_1 and the leaf containing q_2, and thus the number of I/Os used in these nodes adds up to $O(\log_B N + T/B)$. Each remaining visited internal node v in $\mathcal{T}$ is not on the search paths but it is visited because $\Theta(B)$ points corresponding to v were reported in its parent. Thus the cost of visiting these nodes adds up to $O(T/B)$.

Theorem 1. *An external priority search tree on a set of N points in the plane uses linear space and answers 3-sided range queries in $O(\log_B N + T/B)$ I/Os.*

Three-Sided Queries on $O(B^2)$ Points. In the following, we describe the B^2–structure. The data structure is based on a *sweep line* approach commonly used in solving geometric problems.

Structure. We start by sorting the $K = O(B^2)$ input points according to their x-coordinates. We then partition this sequence into $O(K/B)$ groups of $\Theta(B)$ consecutive points. We store the points of each group in one disk block. We think of the blocks as being *ordered* in the natural way (i.e. according to the x-coordinates of the points inside). Then we conceptually sweep a horizontal line from $y = -\infty$ to ∞ and merge the groups as the sweep line passes through the input points. More specifically, we start by marking all groups as *active*. We then raise the sweep line from $y = -\infty$ and each time it passes through a point we check whether there are any two consecutive active groups for which both groups contain less than $B/2$ points above the sweep line. If this is the case, we form a new active group and fill it with the (at most B) points that are above the sweep line in the two groups. We then store the points in the newly formed group in one disk block and mark the two old groups as *inactive*. This process continues until no point remains above the sweep line. Since we start out with $O(K/B)$ active groups, and we reduce the total number of active groups by one each time we create one new group, it follows that the total number of groups formed is $O(K/B)$. Thus the stored disk blocks occupy only linear space.

Each group g formed by the above procedure, including the initial groups, have a natural associated *active y-interval* $[y_1, y_2]$ where y_1 is the y-coordinate of the point that caused g to be formed (we let $y_1 = -\infty$ for the initial groups) and y_2 is the y-coordinate of the point that caused g to be inactive ($y_2 = -\infty$ for the last group). Each group also has an associated x-range, which is simply the range containing the x-coordinates of the points in the group.

Query. To answer the 3-sided query $q = (q_1, q_2, q_3)$, we consider the groups whose active y-interval include q_3, that is, the groups that were active when the sweep line was at $y = q_3$. To answer the query, we assume for now that we can find the subset of these groups whose x-ranges intersect the interval $[q_1, q_2]$. If there are k such groups, we know that at least $k-2$ of them have an x-range completely inside $[q_1, q_2]$. Since we merge two consecutive groups when both have less than $B/2$ points above the sweep line, it follows that the query range contains $\Omega((k-2)B)$ points. Thus we can afford to spend $O(1)$ I/Os for each intersected group to retrieve the corresponding disk block and report the subset of points that are also in the query range.

What remains is to describe how we find the intersected groups. For this, recall that the total number of groups formed is only $O(K/B) = O(B)$. Thus we can store $O(1)$ "catalog" blocks containing the active y-intervals and x-ranges of all the groups plus pointers to the corresponding disk blocks. We thus answer the query by first reading these catalog blocks to find the desired groups. We then follow the stored pointers to the corresponding disk blocks and finish as described above. This completes the description of the B^2–structure.

3.2 Four-Sided Queries

In this section we show how the results for 3-sided queries can be extended to solve 4-sided queries. This extension is based on the classic data structures known as *range trees* [8].

Structure. The data structure consists of a balanced binary tree $\mathcal{T}$ with the N input points stored in sorted order of x-coordinates in the leaves. Each node is naturally associated to an x-range as in Section 2. For each internal node v of $\mathcal{T}$, we store two data structures for answering 3-sided queries on the points stored in the leaves of the subtree rooted at v, one for query ranges of the form $[q_1, \infty) \times [q_2, q_3]$ and one for query ranges of the form $(-\infty, q_1] \times [q_2, q_3]$. Note that such data structures can be obtained from the data structure presented in Section 3.1 by a simple geometric transformation of the input points. Since each point is stored in at most two linear space 3-sided data structures for each node on a root-to-leaf path in $\mathcal{T}$, we conclude that the space usage is $O(N/B \cdot \log N)$.

Queries. To answer a 4-sided query $q = [q_1, q_2] \times [q_3, q_4]$ we start at the root node v of $\mathcal{T}$. If q_1 and q_2 are contained in the x-range of the same child of v, we recursively visit that child. If this procedure ends in a leaf node, we answer the query simply by examining the associated point. Otherwise, let v be the internal node for which q_1 and q_2 lies in the x-range of different children. At v, the query decomposes into two 3-sided queries, one of the form $[q_1, \infty) \times [q_3, q_4]$ on the 3-sided data structure stored for the left child of v, and one of the form $(-\infty, q_2] \times [q_3, q_4]$ on the 3-sided data structure stored for the right child of v. By blocking the tree $\mathcal{T}$ appropriately, finding the node v can be done in $O(\log_B N)$ I/Os, hence the total query cost is at most $O(\log_B N + T/B)$ I/Os.

Space Improvements. The solution for 4-sided queries presented above can be slightly improved. The idea is to change from a binary range tree, to a tree of degree α for a parameter $\alpha > 2$. This decreases the height of the tree to $\log N / \log \alpha$ and thus the space becomes $O(N/B \cdot \log N / \log \alpha)$. Increasing the degree raises another problem however; at the node v where q_1 and q_2 lie in the x-range of two different children there might be up to α children whose x-range is intersected by $[q_1, q_2]$. Thus the query decomposes into two 3-sided queries and up to α one-dimensional queries. By elegant ideas, the one-dimensional queries can be solved jointly in $O(\alpha + T/B)$ I/Os [5]. Thus by setting $\alpha = \log_B N$ one arrives at a data structure using $O(N/B \cdot \log N / \log \log_B N)$ space and answering queries in $O(\log_B N + T/B)$ I/Os. Surprisingly, this space bound has been shown to be optimal for any query time of the form $O(\log_B^c N + T/B)$, where $c > 0$ is any arbitrary constant [12, 5].

Theorem 2. *There exists a data structure that uses $O(N/B \cdot \log N/\log\log_B N)$ space and answers 4-sided range queries in $O(\log_B N + T/B)$ I/Os on a set of N points in the plane.*

Linear Space. There also exists a number of linear space solutions for 4-sided queries. The simplest solution is a generalization of the kd-tree [9] structure to external memory [7]. This data structure stores each input point exactly once and answers queries in $O(\sqrt{N/B} + T/B)$ I/Os. This query time has been shown to be optimal when each point can be stored only once [13].

Dynamization. In the above, we have only focused on a static set of input points that we must preprocess into a data structure. If insertions and deletions of points are also to be supported, several new ideas are needed to make the above solutions efficient.

For 3-sided queries, Arge et al. [5] showed how to dynamize the external priority search tree such that insertions and deletions can be supported in $O(\log_B N)$ I/Os while maintaining the same space and query cost. Using the range tree idea, this also gave a dynamic data structure for 4-sided queries, which supports insertions and deletions in $O(\log_B N \log N/\log\log_B N)$ I/Os.

The external memory kd-tree can be dynamized to support insertions and deletions in $O(\log_B^2 N)$ I/Os, while maintaining $O(\sqrt{N/B} + T/B)$ query I/Os and still storing each point exactly once. This update cost can be improved to $O(\log_B N)$ using the O-tree structure [13].

3.3 Higher Dimensions

For d-dimensional orthogonal range reporting, i.e. d-dimensional points and d-dimensional axis-parallel query rectangles, one can adapt the O-tree such that it answers queries in $O((N/B)^{1-1/d} + T/B)$ I/Os. This is optimal if points can be stored only once [13]. If one is willing to spend super-linear space, the best known data structure uses $O(N/B \cdot (\log N/\log\log_B N)^{d-1})$ space and answers queries in $O(\log_B N(\log N/\log\log_B N)^{d-2} + T/B)$ I/Os [1]. The space bound is optimal if the query cost is $O(\log_B^c N + T/B)$, where $c > 0$ is any arbitrary constant [1]. For three-dimensions, there is also another tradeoff with optimal $O(\log_B N + T/B)$ query cost, using $O(N(\log N/\log\log_B N)^3)$ space [1].

4 Other Range Searching Problems

In many applications, the input data set does not consist of points, but rather of line segments, axis-parallel rectangles etc. Here we mention two fundamental problems where the input consists of a set of two-dimensional axis-parallel rectangles.

In the *rectangle stabbing problem*, we are to support reporting all rectangles containing a query point. Very surprisingly, it has been shown that this problem cannot be solved in near-linear space and $O(\log_B N + T/B)$ query cost. In fact, the best possible query time with $O(N/B \cdot \log^c N)$ space is just $O(\log N/\log\log N + T/B)$ I/Os, where $c > 0$ is an arbitrary constant [6]. This is an interesting example where it is not possible to take advantage of blocking.

In the *rectangle-rectangle reporting problem*, a query is specified by an axis-parallel rectangle and the goal is to report all rectangles intersecting it. The PR-tree [4] is a data structure that stores each input rectangle exactly once and supports queries in $O(\sqrt{N/B} + T/B)$ I/Os. Since a point is a special case of a rectangle, this bound is obviously optimal when rectangles can be stored only once.

For further details and discussion of other range searching problems, we refer the reader to surveys [3, 11, 14, 7].

References

[1] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting in three and higher dimensions. In *Proc. 50th IEEE Symposium on Foundations of Computer Science*, pages 149–158, 2009.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[3] L. Arge. External memory geometric data structures, 2005. ”Lecture notes available at http://www.cs.au.dk/˜large/ioS06/ionotes.pdf”.

[4] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:30, 2008.

[5] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. 18th ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.

[6] L. Arge, V. Samoladas, and K. Yi. Optimal external-memory planar point enclosure. In *Proc. 12th European Symposium on Algorithms*, pages 40–52, 2004.

[7] M. J. Atallah and M. Blanton, editors. *Algorithms and theory of computation handbook: general concepts and techniques.* Chapman & Hall/CRC, 2 edition, 2010.

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

[9] M. de Berg and O. Cheong and M. van Kreveld and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 3rd edition, 2008. Chapter 10.

[10] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[12] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM*, 49(1):35–55, 2002.

[13] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory*, pages 257–276, 1997.

[14] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.