Finding Pairwise Intersections of Rectangles in a Query Rectangle[*]

Eunjin Oh[1] and Hee-Kap Ahn[2]

1 **Department of Computer Sience and Engineering, POSTECH, Korea**
`jin9082@postech.ac.kr`

2 **Department of Computer Sience and Engineering, POSTECH, Korea**
`heekap@postech.ac.kr`

Abstract

We consider the following problem: Preprocess a set $\mathcal{S}$ of n axis-parallel boxes in $\mathbb{R}^d$ so that given a query of an axis-parallel box Q in $\mathbb{R}^d$, the pairs of boxes of $\mathcal{S}$ whose intersection intersects the query box can be reported efficiently. For the case that $d = 2$, we present a data structure of size $O(n \log n)$ supporting $O(\log n + k)$ query time, where k is the size of the output. This improves the previously best known result by de Berg et al. which requires $O(\log n \log^* n + k \log n)$ query time using $O(n \log n)$ space. There has been no known result for this problem for higher dimensions, except that for $d = 3$, the best known data structure supports $O(\sqrt{n} + k \log^2 \log^* n)$ query time using $O(n\sqrt{n} \log n)$ space. For a constant $d > 2$, we present a data structure supporting $O(n^{1-\delta} \log^{d-1} n + k \operatorname{polylog} n)$ query time for any constant $1/d \leq \delta < 1$. The size of the data structure is $O(n^{\delta d} \log n)$ if $1/d \leq \delta < 1/2$, or $O(n^{\delta d - 2\delta + 1})$ if $1/2 \leq \delta < 1$.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling

Keywords and phrases Geometric data structures, axis-parallel rectangles, intersection

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2017.60

1 Introduction

Range searching is one of the fundamental problems, which has been studied extensively in computational geometry [2]. Typical problems of this type are formulated as follows. Preprocess a set $\mathcal{I}$ of input geometric objects so that given a query of geometric object Q, the objects in $\mathcal{I} \cap Q$ can be reported or counted efficiently. There are a number of variants of the problem, including checking the emptiness of $\mathcal{I} \cap Q$, finding the minimum (or maximum) weight of the objects in $\mathcal{I} \cap Q$, and computing the sum of the weights of the objects in $\mathcal{I} \cap Q$.

In this paper, we consider a variant of the range searching problem, which is stated as follows. Given a set $\mathcal{S}$ of n axis-parallel boxes in $\mathbb{R}^d$, preprocess $\mathcal{S}$ so that given a query of an axis-parallel box Q in $\mathbb{R}^d$, all the pairs (S, S') of boxes in $\mathcal{S}$ with $S \cap S' \cap Q \neq \emptyset$ can be reported efficiently. The desired running time for the query algorithm is of form $O(f(n) + k(g(n)))$ for some functions $f(n) = o(n)$ and $g(n) = o(n)$, where k is the size of the output. One straightforward way is to compute all boxes of $\mathcal{S}$ intersecting Q and check whether each pair (S, S') of them has their intersection $S \cap S'$ in Q. However, this straightforward algorithm takes $\Omega(n)$ time in the worst case even when $k = 0$.

This problem occurs in a number of real-world applications. For instance, suppose that we are given a collection of personal qualities (or personality traits) of n clients stored in a

[*] Supported by the NRF grant 2011-0030044 (SRC-GAIA) funded by the government of Korea.

database, each of them is represented as an interval of values. A pair of clients is said to be compatible each other if there is a common subinterval over every quality of them. A typical query on such a collection is composed of a range on each of the qualities, which represents a certain criterion of selecting some compatible pairs of clients that match the query criterion.

If we are allowed to use $\Omega(n^2)$ space in the database, we may precompute all compatible pairs in advance and store them to answer queries efficiently. Otherwise, it is desirable to devise a way of storing the data using less amount of space while the query time remains the same or does not increase much. That is, we need to construct a data structure to answer such a query efficiently in both the query time and the size of the data structure. This is the goal of the problem we study in this paper.

Previous Work. There are a few results on this problem [6, 8, 9]. Consider a simpler problem in which input objects are orthogonal line segments. Orthogonal line segments can be considered as degenerate axis-parallel rectangles. Gupta [8] presented a data structure of size $O(n \log^2 n)$ supporting $O(\log^2 n + k)$ query time for this problem, where k is the size of the output and n is the size of the input. Later, the size of the data structure and the query time were improved to $O(n \log n)$ and $O(\log n + k)$, respectively by Rahul et al. [9].

For axis-parallel rectangles in the plane, de Berg et al. [6] presented a data structure of size $O(n \log n)$ that supports $O(\log n \log^* n + k \log n)$ query time. We observe that their data structure can be improved to support $O(\log n + k \log n)$ query time by simply replacing the range searching algorithm in [10] with the one in [1]. For details, see Section 2.2.1.

The algorithm by de Berg et al. [6] does not extend to higher dimensions directly. Using more observations and techniques, they presented a data structure of size $O(n\sqrt{n} \log n)$ supporting $O(\sqrt{n} + k \log^2 n \log^* n)$ query time in $\mathbb{R}^3$. For fat rectangles, the space and query time are improved to $O(\alpha^3 n \log^2 n)$ and $O(\alpha^2(k+1) \log^2 \log^* n)$, respectively, where α is the maximum ratio between the lengths of the longest and the shortest edges of input rectangles.

One might be concerned on the preprocessing time as well as the size of the data structure. In this type of problems, however, queries are supposed to be made in a repetitive fashion and the preprocessing time can be seen as being amortized over the queries to be made later on [3]. Therefore, we focus mainly on the space requirement of the data structure and the query time for the problem as other previous works did.

Our Result. In this paper, we first present a data structure of size $O(n \log n)$ for two-dimensional case that supports $O(\log n + k)$ query time. This improves the data structure of de Berg et al. [6]. Recall that our problem is a generalization of the problem studied by Rahul et al. [9]. Although our problem is more general, our data structure with its query algorithm requires the same storage and running time as theirs.

Moreover, our data structure is almost optimal. To see this, observe that our problem can be reduced to the *2D orthogonal range reporting problem.* Given a set $\mathcal{P}$ of points in $\mathcal{R}^2$, the 2D orthogonal range reporting problem asks to preprocess them so that given a query of an axis-parallel rectangle, the points of $\mathcal{P}$ contained in the query rectangle can be reported. To solve this problem using the data structure for our problem, we map each point p in $\mathcal{P}$ to two points lying on p (two degenerate boxes). Then we construct a data structure for our problem on the set of the degenerate boxes for all points in $\mathcal{P}$. The data structure reports the pairs (S, S') of degenerate boxes such that S and S' lie on the same position and are contained in a query rectangle. Therefore, we can answer the 2D orthogonal range reporting problem using the data structure for our problem without increasing the running time. For the 2D orthogonal range reporting problem, it is known that on a pointer machine model, a query time of $O(\text{polylog}\, n + k)$, where k is the size of the output, can only be achieved at the expense of $\Omega(n \log n / \log\log n)$ storage [4]. Moreover, on a pointer machine model, a query time of $o(\log n + k)$ cannot be achieved regardless of the size of the data structure. Therefore, our query time is optimal, and the size of our data structure is almost optimal.

We also consider the problem in higher dimensions $\mathbb{R}^d$. For a constant $d > 2$, we present a data structure that supports $O(n^{1-\delta} \log^{d-1} n + k \log^{d-1} n)$ query time for any constant δ with $1/d \leq \delta < 1$. The size of the data structure is $O(n^{\delta d} \log n)$ if $1/d \leq \delta < 1/2$, or $O(n^{\delta d - 2\delta + 1})$ if $1/2 \leq \delta < 1$. A constant δ shows a trade-off between storage and query time. This is the first result on the problem in higher dimensions.

Preliminaries. We are given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n axis-parallel boxes (hyperrectangles) in $\mathbb{R}^d$ for some constant $d \geq 2$. For any two boxes $S_i, S_j \in \mathcal{S}$, we use $I(i,j)$ to denote the intersection of S_i and S_j.

Our goal is to preprocess $\mathcal{S}$ so that for a query of an axis-parallel box Q, we can report all pairs (S_i, S_j) of boxes in $\mathcal{S}$ with $I(i,j) \cap Q \neq \emptyset$ efficiently. We use $\mathcal{U}(Q)$ and $k(Q)$ to denote the output and the size of the output for a query Q, respectively. We simply use $\mathcal{U}$ and k to denote $\mathcal{U}(Q)$ and $k(Q)$, respectively, if they are understood in context.

Due to lack of space, some of the proofs and details are omitted.

2 Planar Case

In this section, we consider the problem in the plane, that is, we are given a set $\mathcal{S}$ of n axis-parallel rectangles in the plane. We present a data structure of size $O(n \log n)$ that supports $O(\log n + k)$ query time for queries of axis-parallel rectangles. This improves the previously best known data structure with its query algorithm by de Berg et al. [6]. Their data structure has size of $O(n \log n)$ and supports $O(\log^* n \log n + k \log n)$ query time.

2.1 Configurations of Two Intersecting Rectangles

An axis-parallel rectangle has four sides: the top, bottom, left and right sides. We call the top and bottom sides the *horizontal sides*, and the left and right sides the *vertical sides*.

Consider a side ab of a rectangle $S \in \mathcal{S}$ with endpoints a and b. Let $a'b'$ be the segment on ab such that a' and b' are the points closest to a and b, respectively, among all intersection points of ab with input rectangles other than S. We call $a'b'$ the *stretch* of S on ab. Note that ab has no stretch if ab intersects no rectangles of $\mathcal{S} \setminus \{S\}$. There is at most one stretch for each side of a rectangle in $\mathcal{S}$. Let $\mathcal{S}_\ell$ be the set of all stretches of the rectangles of $\mathcal{S}$.

For any pair (S_i, S_j) of rectangles in $\mathcal{S}$ with $I(i,j) \cap Q \neq \emptyset$, it is not difficult to see that the pair belongs to one of the following three cases: (1) Q is contained in one of the two rectangles of the pair, (2) Q contains a corner of $I(i,j)$, or (3) Q contains no corner of $I(i,j)$. Here we provide another way of describing all the cases in terms of stretches so that the query time can be improved without increasing the size of the data structures compared to the one in [6]. Each of these cases can be rephrased into one or two configurations in Observation 1. More precisely, case (1) corresponds to C1, case (2) corresponds to C2 and C3, and case (3) corresponds to C4 and C5 of Observation 1.

▶ **Observation 1** (Five Configurations of Intersections). *For any pair (S_i, S_j) of rectangles in $\mathcal{S}$ with $I(i,j) \cap Q \neq \emptyset$, one of the followings holds. See Figure 1.*

- *C1. S_i or S_j contains Q.*
- *C2. Q contains an endpoint of a stretch of S_i or S_j.*

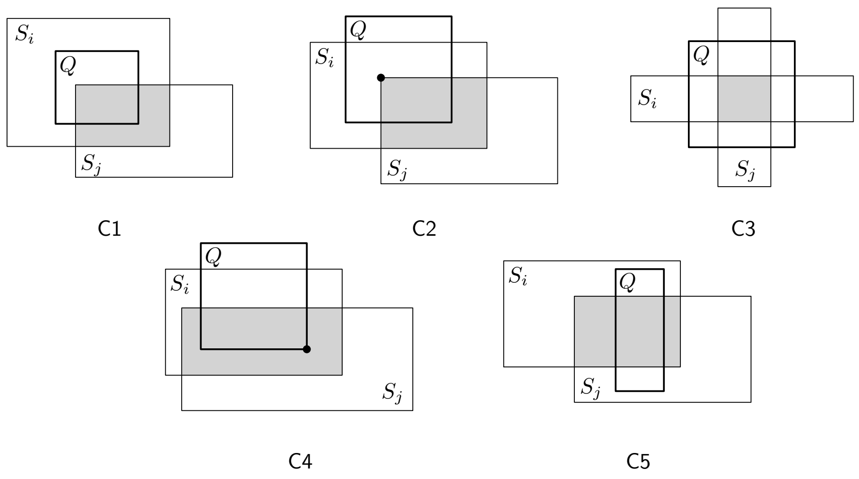

Figure 1 Five configurations of (S_i, S_j) and Q.

- *C3. A stretch of S_i and a stretch of S_j cross Q in different directions.*
- *C4. $I(i,j)$ contains a corner of Q.*
- *C5. $I(i,j)$ and Q cross each other.*

We consider the configurations one by one in our query algorithm. We first report all pairs satisfying C1 (simply, all *C1-pairs*), then we report all pairs satisfying C2 (simply, all *C2-pairs*), and so on. There might be a pair (S_i, S_j) of input rectangles that belongs to more than one configuration. To avoid reporting the same pair more than once, we give a priority order to the configurations such that our algorithm reports a pair exactly once in the configuration of the highest priority. Since there are only five configurations and we can check in constant time whether a pair belongs to a configuration or not, this does not increase the asymptotic time complexity of our algorithm.

2.2 Reporting All Pairs, except C5-pairs

We first show how to construct data structures for finding all pairs (S_i, S_j) of input rectangles with $I(i,j) \cap Q \neq \emptyset$. except C5-pairs. In Section 2.3, we show how to find all C5-pairs.

2.2.1 Data Structures

We construct four data structures for four different problems: the *orthogonal segment intersection problem*, the *point enclosure problem*, the *orthogonal range reporting problem*, and the *rectangle crossing problem*. There has been a fair amount of work on these problems. We observe that the last problem reduces to the 3D orthogonal range reporting problem with a four-sided query box, which has also been studied well. Thus we borrow data structures for these four problems after slightly modifying them to achieve our purpose.

Orthogonal Segment Intersection Problem: SegInt. The *orthogonal segment intersection problem* asks to preprocess horizontal input segments so that given a query of a vertical segment, the horizontal input segments intersected by the query can be computed efficiently. Chazelle [3] gave a data structure called the *hive-graph* to solve this problem efficiently. The hive-graph is a planar orthogonal graph with $O(N)$ cells, each of which has a constant number of edges on its boundary, where N is the number of the input segments.

The query algorithm first finds the cell of the hive-graph containing an endpoint of the query segment and traverses the hive-graph along the query segment from the endpoint to the other endpoint. The edges of the hive-graph encountered during the traversal are the horizontal input segments intersected by the query. In this way, the algorithm finds all such segments in order sorted along the query. The query algorithm takes constant time per output segment, excluding the time for the point location.

In our problem, we construct two hive-graph data structures, one for the horizontal sides of the rectangles of $\mathcal{S}$ and one for the vertical sides of the rectangles of $\mathcal{S}$. Then for each endpoint of the stretches of $\mathcal{S}_\ell$, we find the cells of the hive-graphs that contain the endpoint in the preprocessing phase. This saves the time for point locations in our query algorithm. Specifically, we can find the sides of the rectangles of $\mathcal{S}$ crossed by a stretch ℓ of $\mathcal{S}_\ell$ in the sorted order along ℓ from one endpoint of ℓ. This takes constant time per output side. We denote this data structure by SegInt.

Point Enclosure Problem: PtEnc and EPtEnc. The *point enclosure problem* asks to preprocess input rectangles so that all input rectangles containing a query point can be computed efficiently. Chazelle [3] gave a data structure for this problem. We construct this data structure on $\mathcal{S}$ in the preprocessing time, and denote the data structure by PtEnc. It has size of $O(n)$ and allows us to find all rectangles of $\mathcal{S}$ containing a query rectangle in $O(\log n + K)$ time, where K is the size of the output in this subproblem. Moreover, it allows us to check whether there exists such a rectangle in $O(\log n)$ time.

In our query algorithm, we consider this problem for two different purposes: finding all rectangles of $\mathcal{S}$ containing a *corner* of Q, and finding all rectangles of $\mathcal{S}$ containing an *endpoint* of a stretch of $\mathcal{S}_\ell$. We perform the former task at most four times in our query algorithm since Q has four corners. Thus we simply use PtEnc for this task. However, we will perform the latter task $\Theta(k)$ times in the worst case, which takes $\Omega(k \log n)$ time. Here k is the size of the output in our query algorithm. Note that we have the endpoints of the stretches of $\mathcal{S}_\ell$ in the preprocessing phase, and therefore the latter task can be done in the preprocessing phase.

To do this, we show how the data structure by Chazelle [3] works. Its primary structure is a balanced binary search tree on the rectangles of $\mathcal{S}$ with respect to the x-coordinates of their vertical sides. Each node of the binary search tree corresponds to a vertical line, and it is augmented by the hive-graph on the set of the rectangles of $\mathcal{S}$ intersecting its corresponding vertical line. The query algorithm finds $O(\log n)$ nodes of the binary search tree, and then searches on the hive-graphs associated with the nodes. This takes $O(\log n + K)$ time due to fractional cascading, where K is the size of the output in this subproblem.

This means that we consider $O(\log n)$ hive-graphs and spend $O(\log n)$ time to find the cells containing a query point on the hive-graphs. By finding such cells in the preprocessing phase, we can save the $\log n$ term in the running time of the query algorithm. Note that we need $O(n \log n)$ space to store the cells containing endpoints of the stretches of $\mathcal{S}_\ell$. Then we can find all rectangles of $\mathcal{S}$ containing an endpoint of a stretch of $\mathcal{S}_\ell$ in $O(1+K)$ time, where K is the size of the output. Note that $O(1+K) = O(K)$ since each endpoint is contained in at least two rectangles of $\mathcal{S}$, and thus $K > 1$. We denote this data structure (PtEnc associated with pointers for the endpoints of the stretches) by EPtEnc.

Orthogonal Range Reporting Problem: RecEnc. We want to preprocess all endpoints of the stretches of $\mathcal{S}_\ell$ so that the endpoints contained in a query rectangle can be computed efficiently. Chazelle [3] presented a data structure for this problem that has $O(n \log n / \log\log n)$ size and supports $O(\log n + K)$ query time, where K is the size of the output. We denote this data structure by RecEnc.

Rectangle Crossing Problem: RecCross and RecInt. We want to preprocess the stretches in $\mathcal{S}_\ell$ so that all stretches crossing a query rectangle can be computed efficiently. De Berg et al. [6] also considered this problem. To do this, they reduce to this problem to the orthogonal range reporting problem in three dimensional space as follows. Let $[a,b] \times [c,d]$ be a query rectangle. The query rectangle is crossed by a vertical stretch $x_1 \times [y_1, y_2]$ if and only if $x_1 \in [a,b]$, $y_1 \in [-\infty, c]$, and $y_2 \in [d, \infty]$. Using this observation, they map each vertical stretch $x_1 \times [y_1, y_2]$ to the point (x_1, y_1, y_2) in $\mathbb{R}^3$. Then we can find all vertical stretches crossing the query rectangle by finding all points contained in the orthogonal region $[a,b] \times [-\infty, c] \times [d, \infty]$. Similarly, we can do this for horizontal stretches. However, they did not use the fact that a query is unbounded: it is four-sided in $\mathbb{R}^3$. In this case, we can use a more efficient algorithm given by Afshani et al. [1] instead of the one in [10]. The algorithm by Afshani et al. takes $O(\log n + K)$ time for four-sided query boxes using a data structure of $O(n \log n / \log\log n)$ size, where K is the size of the output. We denote this data structure by RecCross. This data structure is of size $O(n \log n / \log\log n)$ and allows us to find all vertical (or horizontal) stretches of $\mathcal{S}_\ell$ crossing a query rectangle in $O(\log n + K)$ time, where K is the size of the output.

A rectangle S of $\mathcal{S}$ intersects a query rectangle Q if and only if (1) Q crosses a side of S, (2) Q contains a corner of S, or (3) Q is contained in S. To find all rectangles of $\mathcal{S}$ intersecting a query rectangle, we use RecCross for case (1), use RecEnc for case (2), and use PtEnc for case (3). We call the combination of these data structures RecInt. We can find all rectangles of $\mathcal{S}$ intersecting Q in $O(\log n + K)$ time using RecInt, where K is the size of the output in this subproblem.

2.2.2 Query Algorithms.

Assume that we have the data structures of size $O(n \log n)$ described in Section 2.2.1. Then, we can find all pairs (S_i, S_j) of $\mathcal{S}$ with $I(i,j) \cap Q \neq \emptyset$, except C5-pairs, in $O(\log n + k)$ time.

Reporting C1-pairs of Q. We can find the C1-pairs of Q in $O(\log n + k(Q))$ time. A pair of rectangles in $\mathcal{S}$ is a C1-pair of Q if one rectangle in the pair contains all four corners of Q and the other rectangle intersects Q.

We find the rectangles of $\mathcal{S}$ containing all four corners of Q by finding all rectangles of $\mathcal{S}$ containing each corner of Q using PtEnc. Note that there are $O(k(Q) + 1)$ rectangles that contain a corner of Q simply because every pair of the rectangles containing the corner is in $\mathcal{U}(Q)$. (We need "+1" since it is possible that there is just one rectangle containing the corner, but $k(Q)$ is zero.) Thus, we can compute such rectangles in $O(\log n + k(Q))$ time. Let $\mathcal{S}_1$ denote the set of all rectangles containing all four corners of Q.

Let $\mathcal{S}_2$ denote the set of all rectangles intersecting Q. If $\mathcal{S}_1$ is not empty, we find all rectangles of $\mathcal{S}_2$ in $O(\log n + K)$ time using RecInt, where K is the number of such rectangles. Since $\mathcal{S}_1$ is not empty, K is at most $k(Q)$. We report every pair (S_1, S_2) with $S_1 \in \mathcal{S}_1$ and $S_2 \in \mathcal{S}_2$ as a C1-pair of Q, which takes $O(\log n + k(Q))$ time. It is clear that we report all C1-pairs of Q in this way.

Reporting C2-pairs of Q. We can find the C2-pairs of Q in $O(\log n + k(Q))$ time. A pair of rectangles in $\mathcal{S}$ is a C2-pair of Q if Q contains an endpoint of a stretch ℓ of one of them and the other intersects $\ell \cap Q$. We find all stretches of $\mathcal{S}_\ell$ whose endpoints are in Q in $O(\log n + k(Q))$ time using RecEnc. The number of such stretches is $O(k(Q))$ because each endpoint of the stretches of $\mathcal{S}_\ell$ is contained in at least two rectangles of $\mathcal{S}$ and there are at most four stretches from one rectangle of $\mathcal{S}$.

For each such stretch ℓ, we want to find all rectangles S of $\mathcal{S}$ with $S \cap \ell \cap Q \neq \emptyset$. Such rectangles S satisfy one of the followings: $\ell \cap Q$ is intersected by the boundary of S or $\ell \cap Q$ is contained in S. For the former case, we use SegInt. Starting from the endpoint of ℓ contained in Q, we traverse the hive-graph along ℓ until we escape from Q or we arrive at the other endpoint of ℓ. We find all rectangles S whose sides intersect $\ell \cap Q$ in time linear to the number of such rectangles using SegInt. For the latter case, we compute all rectangles containing the endpoint of ℓ that is also in Q in time linear to the number of such rectangles using EPtEnc. Therefore, for each stretch ℓ with an endpoint in Q, we can find all rectangles of $\mathcal{S}$ intersecting $\ell \cap Q$ in time linear to the number of such rectangles.

By applying this procedure for every stretch with an endpoint in Q, we can find all C2-pairs of Q in $O(k(Q))$ time, excluding the time for finding all such stretches. Therefore, we can compute all C2-pairs of Q in $O(\log n + k(Q))$ time in total.

Reporting C3-pairs of Q. We can find the C3-pairs of Q in $O(\log n + k(Q))$ time. A pair of rectangles in $\mathcal{S}$ is a C3-pair of Q if two stretches, one from each rectangle, cross Q in different directions. Let $\mathcal{S}_v$ be the set of the rectangles of $\mathcal{S}$ whose vertical stretches cross Q. Let $\mathcal{S}_h$ be the set of the rectangles of $\mathcal{S}$ whose horizontal stretches cross Q.

We first check whether $\mathcal{S}_v$ or $\mathcal{S}_h$ is empty in $O(\log n)$ time using RecCross. If both of them are nonempty, we compute $\mathcal{S}_v$ and $\mathcal{S}_h$ in $O(\log n + k(Q))$ time using RecCross. The size of $\mathcal{S}_v$ and $\mathcal{S}_h$ is $O(k(Q))$ since every rectangle of $\mathcal{S}_v$ intersects every rectangle of $\mathcal{S}_h$ in Q. Then we report the pairs (S, S') with $S \in \mathcal{S}_v$ and $S' \in \mathcal{S}_h$ as the C3-pairs in $O(\log n + k(Q))$ time in total.

Reporting C4-pairs of Q. We can report the C4-pairs of Q in $O(\log n + k(Q))$ time. A pair of rectangles in $\mathcal{S}$ is a C4-pair of Q if the intersection of the rectangles contains a corner of Q. We first check whether there exists a rectangle of $\mathcal{S}$ containing a corner of Q in $O(\log n)$ time using PtEnc. Again, the number of the rectangles of $\mathcal{S}$ containing a corner of Q is $O(k(Q))$ as every pair of such rectangles is in $\mathcal{U}(Q)$.

If there exists such a rectangle, we find all such rectangles in $O(\log n + k(Q))$ time using PtEnc. Then we report all pairs consisting of such rectangles. We do this for each of the other corners of Q. Then we can report all C4-pairs in $O(\log n + k(Q))$ time.

2.3 Reporting C5-pairs

We have shown how to find all pairs consisting of two rectangles of $\mathcal{S}$ intersecting each other in Q, except for the C5-pairs. There might be some pairs of rectangles that belong to both C5 and one of the other configurations. As mentioned earlier, this can be checked in constant time per pair of rectangles. Since we use a priority order over the configurations, we assume that they have already been reported by the algorithm for the configurations other than C5.

A pair of rectangles in $\mathcal{S}$ is a C5-pair of a query rectangle Q if the intersection of the rectangles and Q cross each other. In the following, we show how to find and report the C5-pairs of Q not belonging to any other configuration such that the horizontal sides of the intersection intersect the vertical sides of Q. The C5-pairs not belonging to any other configuration such that the vertical sides of the intersection intersect the horizontal sides of Q can be found analogously.

One-dimensional segment tree. We construct a *one-dimensional segment tree* T of $\mathcal{S}$ with respect to the x-axis as follows [5]. The segment tree has a balanced binary search tree on the x-projections of the rectangles of $\mathcal{S}$ as a primary structure. Each node v of the balanced

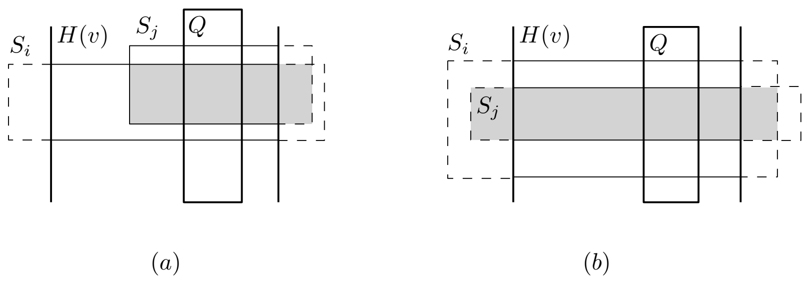

Figure 2 (a) A canonical node v of (i, j, Q). It holds that $S_i \in \mathcal{S}_C(v)$, but $S_j \in \mathcal{S}_B(v)$. (b) A canonical node v of (i, j, Q). It holds that both S_i and S_j are in $\mathcal{S}_C(v)$.

binary search tree corresponds to a closed vertical slab $H(v)$. We say that a rectangle S *crosses* $H(v)$ if S intersects $H(v)$ and no vertical side of S is contained in $H(v)$. Let $\mathcal{S}_C(v)$ be the set of the rectangles of $\mathcal{S}$ that cross $H(v)$ but do not cross $H(u)$ for the parent u of v in T. There are $O(\log n)$ nodes v with $S \in \mathcal{S}_C(v)$. Moreover, the union of $H(v)$'s for all such nodes v contains S. Let $\mathcal{S}_B(v)$ be the set of the rectangles of $\mathcal{S}$ whose left or right side is contained in the interior of $H(v)$. Note that $\mathcal{S}_B(v)$ is empty for every leaf node v. For a rectangle $S \in \mathcal{S}$, there are at most two nodes v of T with $S \in \mathcal{S}_B(v)$ at each level of T, and each such node lies on one of the two paths of T from the root to two leaf nodes w, w' with the left side of S contained in $H(w)$ and the right side of S contained in $H(w')$. We use $\mathcal{S}(v)$ to denote the union of $\mathcal{S}_C(v)$ and $\mathcal{S}_B(v)$. For each node v of T, we store $\mathcal{S}_B(v)$ and $\mathcal{S}_C(v)$. The binary search tree together with the sets $\mathcal{S}_B(\cdot)$ and $\mathcal{S}_C(\cdot)$ forms the segment tree of $\mathcal{S}$.

Canonical nodes of a C5-pair. Consider any C5-pair (S_i, S_j) of Q. Note that there are $O(\log n)$ nodes v of T such that $S_i, S_j \in \mathcal{S}(v)$ and $I(i, j) \cap Q \cap H(v) \neq \emptyset$. If we traverse T and find C5-pairs at each node, then the same pair is found at $O(\log n)$ nodes of T, and therefore the total running time is $\Omega(k \log n)$ for k pairs in the worst case. Instead, we use a number of *canonical nodes* (to be defined below) such that there is a unique canonical node of (i, j, Q) in T for any C5-pair. We will show how to find the canonical nodes and report all C5-pairs efficiently in the subsequent sections. See Figure 2.

▶ **Definition 2.** For a rectangle Q and a pair (S_i, S_j) of the rectangles of $\mathcal{S}$ with $I(i, j) \cap Q \neq \emptyset$, a node v of T is called the *canonical node* of (i, j, Q) if the left side of Q is contained in $H(v)$ and both S_i and S_j are in $\mathcal{S}(v)$ satisfying $S_i \in \mathcal{S}_C(v)$ or $S_j \in \mathcal{S}_C(v)$.

Note that not every canonical node of some triple (i, j, Q) defines a C5-pair of Q, though $I(i, j) \cap Q \neq \emptyset$. However, there is a canonical node of (i, j, Q) in T for each C5-pair of Q.

▶ **Lemma 3.** *For any C5-pair (S_i, S_j) of Q, there is a canonical node of (i, j, Q) in T.*

Proof. Consider the C5-pairs of Q such that the horizontal sides of the intersection intersects the vertical sides of Q. The other cases can be analyzed in a similar way. Let p be the intersection between the left side of Q and the top side of $I(i, j)$. Then there is a path π from the root node to some leaf node u with $p \in H(u)$ in T. Consider a node w in π. Since p lies on the left side of Q, the slab $H(w)$ contains the left side of Q. Moreover, $H(w)$ intersects both S_i and S_j.

We claim that there is a canonical node of (i, j, Q) in π. By the construction of the segment tree, $S_i \in \mathcal{S}_B(v)$ for the root node v and $S_i \notin \mathcal{S}_B(u)$ for the leaf node u of π. Thus, there is a node w_i of π with $S_i \in \mathcal{S}_C(w_i)$. For a node w closer to the root node than w_i,

$S_i \in \mathcal{S}_B(w)$. For a node w' closer to the leaf node than w_i along π, $S_i \notin \mathcal{S}(w')$. This also holds for S_j, so there is a node w_j of π with $S_j \in \mathcal{S}_C(w_j)$. Without loss of generality, we assume that w_i lies between the root node and w_j (including them) along π. Then we have $S_i \in \mathcal{S}_C(w_i)$ and $S_j \in \mathcal{S}(w_i)$. Since w_i is in π, $H(w_i)$ contains the left side of Q. Therefore, s_i is a canonical node of (i, j, Q) in π. ◀

We need the following lemma to bound the total number of canonical nodes for Q over all pairs of rectangles in $\mathcal{S}$ by $O(k(Q))$.

▶ **Lemma 4.** *For any rectangle Q and any pair (S_i, S_j) of the rectangles of $\mathcal{S}$ with $I(i,j) \cap Q \neq \emptyset$, there is at most one canonical node of (i, j, Q) in T.*

▶ **Corollary 5.** *The total number of canonical nodes for a query rectangle Q is $O(k(Q))$.*

Our general strategy is the following. Given a query rectangle Q, we find a set of nodes of the segment tree T that contains the canonical node of (i, j, Q) for every C5-pair (S_i, S_j) not belonging to any other configuration in $O(\log n + k(Q))$ time. The size of this set is $O(k(Q))$. For each such node v, we find all C5-pairs (S_i, S_j) such that v is a canonical node of (i, j, Q) in time linear to the number of the output.

2.3.1 Finding all Canonical Nodes for C5-pairs

In this subsection, we present data structures and their query algorithms to find a set of canonical nodes of (i, j, Q) with $I(i, j) \cap Q \neq \emptyset$ for a query rectangle Q. This set contains all canonical nodes of (i, j, Q) for every C5-pair (S_i, S_j) not belonging to any other configuration. We show how to find such C5-pairs such that the horizontal sides of $I(i, j)$ intersect the vertical sides of Q. Similarly, we can find the other C5-pairs such that the vertical sides of $I(i, j)$ intersect the horizontal sides of Q.

Data Structures. For each node v of T and each rectangle S in $\mathcal{S}(v)$, we trim S as follows. We first remove the parts of S lying outside of $H(v)$. Then we remove the parts of the resulting rectangle that lie outside of the smallest horizontal slab enclosing $\bigcup_{S' \in \mathcal{S}_C(v)} S'$ with $S \cap S' \neq \emptyset$. We call the resulting rectangle the *trimmed rectangle* for (S, v). Let $\mathcal{L}$ be the set of the horizontal sides of all trimmed rectangles for all nodes of T. Note that $|\mathcal{L}| = O(n \log n)$.

We construct the hive-graph on $\mathcal{L}$, which allows us to report all horizontal sides of $\mathcal{L}$ intersecting a query vertical segment ℓ in sorted order along ℓ in $O(\log n + K)$ time, where K is the size of output [3]. Since the size of $\mathcal{L}$ is $O(n \log n)$, the hive-graph has $O(n \log n)$ size. We make each segment in $\mathcal{L}$ to point to the rectangle in $\mathcal{S}$ from which the segment comes.

Query Algorithm. Given a query rectangle Q, our query algorithm finds all sides of $\mathcal{L}$ intersecting the left side of Q using the hive-graph on $\mathcal{L}$. Then for each such side, our query algorithm marks the node of T pointed by the side as a canonical node in $O(\log n + k)$ time due to the following lemmas.

▶ **Lemma 6.** *The query algorithm finds the canonical node of (i, j, Q) for every C5-pair (S_i, S_j) not belonging to any other configuration.*

▶ **Lemma 7.** *The number of the sides of $\mathcal{L}$ intersecting the left side of Q is $O(k(Q))$.*

▶ **Lemma 8.** *Given a query rectangle Q, we can find a set of k nodes of T containing all canonical nodes for C5-pairs not belonging to any other configuration in $O(\log n + k)$ time.*

2.3.2 Handling Each Canonical Node to Find All C5-pairs

Let $\mathcal{V}$ be the set of all nodes we found in Section 2.3.1. For each node $v \in \mathcal{V}$, we show how to find all $\mathsf{C5}$-pairs (S_i, S_j) not belonging to any other configuration such that v is a canonical node of (i, j, Q). Here, we consider only the case that $S_i \in \mathcal{S}_C(v)$ and $S_j \in \mathcal{S}(v)$. The other case that $S_j \in \mathcal{S}_B(v)$ and $S_i \in \mathcal{S}(v)$ can be handled analogously. Moreover, we consider only the $\mathsf{C5}$-pairs such that the horizontal sides of $I(i, j)$ intersect the vertical sides of Q. The other case can be handled analogously.

For each node v, we spend $O(1+k(v))$ time, where $k(v)$ is the number of $\mathsf{C5}$-pairs (S_i, S_j) such that v is an canonical node of (i, j, Q). Once we do this for every node in $\mathcal{V}$, we can obtain all $\mathsf{C5}$-pairs for the canonical nodes of (i, j, Q) not belonging to any other configuration in $O(k)$ time, excluding the time for computing all such canonical nodes.

Data Structures and Preprocessing. While computing the set $\mathcal{V}$ in Section 2.3.1, we obtain all rectangles $S_j \in \mathcal{S}(v)$ for each node $v \in \mathcal{V}$ such that a horizontal side of the trimmed rectangle for (S_j, v) intersects the left side of Q. Moreover, a horizontal side of the trimmed rectangle for (S_j, v) intersects the left side of Q if there is a rectangle $S_i \in \mathcal{S}_C(v)$ such that (S_i, S_j) belongs to $\mathsf{C5}$, but does not belong to any other configuration and the vertical sides of $I(i, j)$ intersect the horizontal sides of Q.

Therefore, we can find all such $\mathsf{C5}$-pairs as follows. For each such rectangle S_j, we find all pairs (S_i, S_j) for $S_i \in \mathcal{S}_C(v)$ such that $\mathsf{y}(S_j)$, $\mathsf{y}(S_i)$ and $\mathsf{y}(Q)$ contain a common point, where $\mathsf{y}(A)$ is the y-projection of a set $A \subseteq \mathbb{R}^2$. Let S'_j be the trimmed rectangle for (S_j, v). Note that for such rectangle, $\mathsf{y}(S'_j)$, $\mathsf{y}(S_i)$ and $\mathsf{y}(Q)$ also contain a common point.

Thus there are two cases: $\mathsf{y}(S'_j) \cap \mathsf{y}(Q)$ contains an endpoint of $\mathsf{y}(S_i)$, or $\mathsf{y}(S'_j) \cap \mathsf{y}(Q)$ is contained in $\mathsf{y}(S_i)$. For the first case, a horizontal side of S_i intersects $Q \cap S'_j$. For the second case, a horizontal side of S'_j is contained in S_i. We maintain two data structures, one for finding the pairs of the first case and the other for finding the pairs of the second case.

The first one is as follows. For each node v of T, we maintain two sorted lists of the rectangles in $\mathcal{S}_C(v)$, one with respect to their top sides and the other with respect to their bottom sides. We make each rectangle S_j in $\mathcal{S}_B(v)$ to point to the rectangle in $\mathcal{S}_C(v)$ with highest bottom side (and highest top side) lying below the top side (and bottom side) of S'_j. Similarly, we make S_j to point to the rectangle in $\mathcal{S}_C(v)$ with lowest top side (and lowest bottom side) lying above the bottom side (and top side) of S'_j.

For the second one, we use a *partially persistent data structure* of a linked list. Once we update a linked list and destroy the old versions, we cannot search any element on an old version any longer. But a partially persistent data structure allows us to access any version at any time by keeping the changes on the linked list. Driscoll et al. [7] presented a general method to make a data structure based on pointers partially persistent. Using their method, we can construct a partially persistent data structure of a linked list.

In our case, the linked list has rectangles in $\mathcal{S}_C(v)$ as its elements. We consider a y-coordinate as a time stamp. A rectangle $S_i \in \mathcal{S}_C(v)$ is appended to the linked list at time t_1 and deleted from the linked list at time t_2, where t_1 is the y-coordinate of the top side of S_i and t_2 is the y-coordinate of the bottom side of S_i. Each insertion and deletion can be done in constant time, which is subsumed in the preprocessing time. For each horizontal side of S'_j, we need an extra pointer that points to the first element of the persistent data structure at time t, where t is the y-coordinate of the horizontal side. The size of the partially persistent data structure is linear to the size of $\mathcal{S}(v)$. Due to the partially persistent data structure and the pointers associated with the horizontal sides of the rectangles of $\mathcal{S}(v)$, we can find all rectangles in S_i containing a horizontal side of S'_j in $O(1 + K)$ time, where K is the number of such S_i's.

Query Algorithm. Given a query rectangle Q, for each node v in $\mathcal{V}$ and each rectangle $S_j \in \mathcal{S}(v)$ such that the trimmed rectangle for (S_j, v) intersects the left side of Q, we want to find all C5-pairs (S_i, S_j) with $S_i \in \mathcal{S}_C(v)$ such that $\mathsf{y}(S_j)$, $\mathsf{y}(S_i)$ and $\mathsf{y}(Q)$ contain a common point. Recall that there are two cases: $\mathsf{y}(S'_j) \cap \mathsf{y}(Q)$ contains an endpoint of $\mathsf{y}(S_i)$, or $\mathsf{y}(S'_j) \cap \mathsf{y}(Q)$ is contained in $\mathsf{y}(S_i)$.

To find the C5-pairs (S_i, S_j) belonging to the first case, we do the followings. We search the sorted list of the rectangles in $\mathcal{S}_C(v)$ with respect to their top sides starting from the rectangle of $\mathcal{S}_C(v)$ with lowest top side lying above the bottom side of S'_j if the bottom side of S'_j intersects Q. Note that we can obtain the starting point using the pointer that the bottom side of S'_j has. We stop searching the sorted list when we reach the top side of S'_j or the top side of Q. Similarly, we search the sorted list of the rectangles in $\mathcal{S}_C(v)$ with respect to their bottom sides starting from the rectangle that the bottom side of S'_j points to until we reach the top side of S'_j or the top side of Q if the top side of S'_j intersects Q. In this way, we can find all rectangles S_i in $\mathcal{S}_C(v)$ belongs to the first case in $O(1 + K)$ time, where K is the number of such rectangles.

To find the C5-pairs (S_i, S_j) belonging to the second case, we do the followings. Recall that a horizontal side ℓ of S'_j is contained in Q. Moreover, ℓ is also contained in S_i since $\mathsf{y}(S'_j) \cap \mathsf{y}(Q)$ is contained in $\mathsf{y}(S_i)$. We search the partially persistent data structure at time t, where t is the y-coordinate of ℓ. Starting from the pointer that ℓ points to, we traverse the linked list at time t. All rectangles we encounter are the rectangles containing ℓ. This takes $O(1 + K')$ time, where K' is the number of such rectangles.

Note that both K and K' are at least $k(v)$, where $k(v)$ is the number of the output pairs (S_i, S_j) in $\mathcal{U}(Q)$ such that the canonical node of (i, j, Q) is v. Therefore, we spend $O(1 + k(v))$ time for each node $v \in \mathcal{V}$. Note that $k(v)$ is at least one for every node $v \in \mathcal{V}$ by the construction of $\mathcal{V}$. Once we do this for every node in $\mathcal{V}$, we can obtain $\mathcal{U}(Q)$ in $O(1 + k(Q))$ time in total.

▶ **Lemma 9.** *Given a query rectangle Q, we can find all C5-pairs in $O(\log n + k(Q))$ time.*

Therefore, we have the following theorem.

▶ **Theorem 10.** *We can construct a data structure of size $O(n \log n)$ on a set $\mathcal{S}$ of n axis-parallel rectangles so that for a query axis-parallel rectangle Q, the pairs (S_i, S_j) of $\mathcal{S}$ with $S_i \cap S_j \cap Q \neq \emptyset$ can be reported in $O(\log n + k)$ time, where k is the size of the output.*

3 Higher Dimensional Case

In this section, we consider a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n axis-parallel boxes (hyperrectangles) in $\mathbb{R}^d$ for a constant $d > 2$. Let $\delta \in \mathbb{R}$ be any constant with $1/d \leq \delta < 1$. We present a data structure that supports $O(n^{1-\delta} \log^{d-1} n + k \operatorname{polylog} n)$ query time. The size of the data structure is $O(n^{\delta d} \log n)$ if $1/d \leq \delta < 1/2$, or $O(n^{\delta d - 2\delta + 1})$ if $1/2 \leq \delta < 1$. There has been no known result for this problem in higher dimensions, except that for $d = 3$, the best known data structure has size of $O(n\sqrt{n} \log n)$ and supports $O(\sqrt{n} + k \log^2 \log^* n)$ query time.

Due to lack of space, we gives only a sketch of our data structure and algorithm. For each index $1 \leq t \leq d$, we construct n^δ intervals on the x_t-axis. Consider the x_t-projections of the x_t-facets of $\mathcal{S}$ and choose every $\lfloor n^{1-\delta} \rfloor$th projection. Then we have n^δ projections that define n^δ intervals. Let T_t be the set of such intervals. A *grid cell* is a d-tuple $(v_1, \ldots, v_d)$ of intervals $v_t \in T_t$ for $1 \leq t \leq d$. Note that there are $O(n^{\delta d})$ grid cells. We define the canonical grid cell of a box B, not necessarily in $\mathcal{S}$ in $\mathbb{R}^d$ to be the grid cell containing the corner of B with minimum x_t-coordinates for all $1 \leq t \leq d$.

We construct a data structure for each interval on the x_t-axis, and store a Boolean value to each grid cell. The total size of the data structures is $O(n^{\delta d}\log n)$ if $0 < \delta < 1/2$, or $O(n^{\delta d-2\delta+1})$ if $1/2 \le \delta < 1$.

Given a query box Q, we observe that the canonical grid cell of $I(i,j)$ is contained in Q, or $I(i,j)\cap Q$ intersects a grid cell intersecting the boundary of Q for a pair (S_i,S_j) with $I(i,j)\cap Q \neq \emptyset$. We find all pairs of the first case in $O(k\log^{d-1} n)$ time. For the second case, we reduce this problem to the $(d-1)$-dimensional problem and obtain a recurrence equation on the running time of our algorithm. Then we obtained the following theorem.

▶ **Theorem 11.** *We can construct data structures on a set $\mathcal{S}$ of n axis-parallel boxes in $\mathbb{R}^d$ so that for a query axis-parallel box Q for a constant d, the pairs (S_i,S_j) of $\mathcal{S}$ with $S_i\cap S_j\cap Q \neq \emptyset$ can be reported in $O(n^{1-\delta}\log^{d-1} n + k\log^{d-1} n)$ time for any constant $1/d \le \delta < 1$, where k is the size of the output. The size of the data structure is $O(n^{\delta d}\log n)$ if $1/d \le \delta < 1/2$, or $O(n^{\delta d-2\delta+1})$ if $1/2 \le \delta < 1$.*

References

1 Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceddings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009)*, pages 149–158, 2009.

2 Pankaj K Agarwal, Jeff Erickson, et al. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

3 Bernard Chazelle. Filtering search: A new approach to query answering. *SIAM Journal on Computing*, 15(3):703–724, 1986.

4 Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

5 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, 2008.

6 Mark de Berg, Joachim Gudmundsson, and Ali D. Mehrabi. Finding pairwise intersections inside a query range. In *Proceedings of the 14th Algorithms and Data Structures Symposium (WADS 2015)*, pages 236–248, 2015.

7 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

8 Prosenjit Gupta. Algorithms for range-aggregate query problems involving geometric aggregation operations. In *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC 2005)*, pages 892–901, 2005.

9 Saladi Rahul, Ananda Swarup Das, K. S. Rajan, and Kannan Srinathan. Range-aggregate queries involving geometric aggregation operations. In *Proceedings of the 5th International Workshop on Algorithms and Computation (WALCOM 2011)*, pages 122–133, 2011.

10 Sairam Subramanian and Sridhar Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1995)*, pages 378–387, 1995.