Proc. Int. Conf. on Statistical and Scientific Database Management (SSDBM), 2000

Toward Spatial Joins for Polygons*

Hongjun Zhu Jianwen Su Oscar H. Ibarra
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{hongjunz, su, ibarra}@cs.ucsb.edu

Abstract

Efficient evaluation of spatial join is an important issue in spatial databases. The traditional evaluation strategy is to perform a join of "minimum bounding rectangles" (MBR) of the spatial objects (MBR-filter) and evaluate the actual join of the objects using the results of the join on approximations. Improvements to add additional filtering using more accurate approximations were also considered.

In the present paper, we develop efficient algorithms for evaluating joins of "trapezoids" without using MBR's. For the case where there are no intersecting non-horizontal boundaries of trapezoids in the same set, a spatial join of two sets of N trapezoids can be evaluated in $O(N \log_b N+k)$ I/Os, where b is the page size and k the number of trapezoid intersections. For the general case without any assumptions, a join can be done in $O((N+l+k) \log_b N)$ I/Os, where l is the total number of intersections of non-horizontal boundaries within the same set, and N, k, b are the same as above.

The new algorithms can be used to evaluate spatial joins for polygons. One possibility is to decompose polygons into trapezoids and apply a trapezoid join algorithm. In particular, this approach is efficient for "I/O bounded polygons" (each of which can be retrieved in a constant number of I/Os). Given two sets of N I/O bounded polygons, we show that in the case where there are no boundary intersections among polygons of the same set, the join of the two sets can be computed in $O(N \log_b N+k)$ I/Os, and in the case where there is no such assumption, the join takes $O((N + l + k) \log_b N)$ I/Os, where b is the page size, k the number of pairs of intersecting polygons, and l the number of boundary intersections within the same polygon set. Another possibility is to approximate objects by I/O bounded polygons (e.g., 5-corner convex polygons) which are finer than rectangles and use the new algorithms as a filter.

*Support in part by NSF grants IRI-9700370 and IIS-9817432.

1 Introduction

Efficient evaluation of spatial queries is an important issue in spatial databases, constraint databases, geographical information systems [17, 15, 16]. Among spatial operations, spatial join operations, which link together tuples that have overlapping spatial values, are very useful but costly to evaluate. There are two sources that contribute to the complexity of spatial joins. First, similar to join operations in traditional databases, a spatial join involves two relations which are usually very or extremely large. A naive nested loop evaluation will have I/O operations quadratic in the cardinalities of the relations involved. Second, spatial objects are different from traditional data in their data structures and semantics, and the size of an object can be very large. As a result, efficient evaluation of spatial joins is more difficult. The focus of this paper is on efficient evaluation of spatial joins in spatial/constraint databases.

Spatial joins have been well studied in the literature. The traditional approach to evaluating a spatial join employs the following two steps [21]: (1) MBR-filter step, which performs a spatial join on minimum bounding rectangles (MBR's) of the objects; and (2) refinement step, which determines whether the objects discovered by the previous step actually intersect. This two-step approach can be augmented by extra but more refined filter steps. For example, one can use a new extra step which determines intersection of finer approximations (than MBR's) such as convex hulls and minimum n-corner bounding convex polygons [8] or rasters [31] on the results generated by the MBR-filter step. For spatial objects that are polygons, there are no particular reasons to use approximations and multiple steps in the join evaluation. Indeed, if efficient algorithms can be developed, it is desirable to avoid approximations and to evaluate join directly on polygons. However, this remains an interesting open problem. In this paper, we study I/O efficient spatial join algorithms for a subclass of polygons, trapezoids. One immedi-

ate implication is that evaluation of spatial joins for I/O bounded polygons, each of which takes a constant number of I/O's to read, does not have additional I/O cost in terms of big-O. For polygons in the general case, one can decompose polygons into trapezoids or I/O bounded polygons and then use the efficient algorithms to compute a polygon join. Another possibility is to use trapezoids or I/O bounded polygon as finer approximations than MBR's and replace the MBR-filter step with a "trapezoid or I/O bounded polygon based filter step" in the traditional two step spatial join.

In this paper, we give I/O efficient algorithms for spatial joins of trapezoids without using any approximations. The main results are the following. For the case where there are no intersecting non-horizontal boundaries of trapezoids in the same set, we show that evaluation of a spatial join of two sets of N trapezoids can be done within $O(N \log_b N + k)$ I/Os, where b is the page size and k is the number of trapezoid intersections. For the general case without any assumptions, we show that a join can be performed within $O((N+l+k)\log_b N)$ I/Os, where l is the total number of intersections of non-horizontal boundary lines within the same set, and N, k, b are the same as above. The trapezoid algorithm is then used to compute the spatial join of I/O bounded polygons. The basic idea is to decompose I/O bounded polygons into constant number of trapezoids and apply the trapezoid join algorithm on the results. We show that in the case where there are no boundary intersections of polygons within the same input set, the join of two sets of N I/O bounded polygons can be computed in $O(N \log_b N + k)$ I/Os, where k, b are the same as above. In the general case where there is no such assumption, we show that the join takes $O((N + l + k)\log_b N)$ I/Os, where l is the number of boundary intersections within the same polygon set, and k, b are the same as above.

We now briefly summarize the key ideas and techniques developed in this paper. Observe that two trapezoids intersect if and only if (1) their non-horizontal boundaries intersect, or (2) either one contains the other or their boundary intersection involve horizontal lines. This property allows us to consider the two cases separately in evaluating a trapezoid join. Specifically, one case is to determine intersections of non-horizontal boundaries of trapezoids. This step is closely related to the line segment intersection problem [7, 10, 20, 19, 11, 12, 5, 24]. The other case is to find intersections that are caused by containment or horizontal lines.

While many existing algorithms [7, 10, 20, 19, 11, 12, 5, 24] can be the candidates for the first component, the second component seems new and unrelated to known problems. Surprisingly, we show that the containment and horizontal boundary intersection conditions on trapezoids can be reduced to a "dynamic" version of the rectangle join problem. The reduction is based on a mapping from trapezoids to rectangles such that when the non-horizontal boundaries of two trapezoids do not intersect, the trapezoids intersect if and only if the mapped rectangles intersect. The key to obtaining this mapping is the "orderings" of non-horizontal boundaries intersecting a horizontal sweep line $y = \alpha$. The ordering changes when the sweep line changes. We show that this mapping can be computed efficiently.

Therefore, a trapezoid join can be evaluated in the following three steps:

1. Compute non-horizontal boundary intersections,
2. Compute the mapping from trapezoids to rectangles, and
3. Compute rectangle intersections.

Theoretically it is possible to use three algorithms, one for each of the three steps listed above, and run them in an isolated fashion. We argue that it is more desirable to have integrated algorithms with three steps running in parallel and pipelined. Also, the computation of the trapezoid-to-rectangle mapping shares several particular steps of line segment intersection algorithm and also requires that the rectangle join be done in a dynamic fashion. Simply using one algorithm for each step would require many unnecessary and redundant steps.

To achieve the efficiency goal, we extend the algorithm by Mairson and Stolfi [19] (for the case of no non-horizontal boundary intersection) and the algorithm by Bentley and Ottmann [7] (for the general case) for Step 1. We show that Step 2 can be done by modifying Step 1. However, Step 2 also restricts Step 3 to be done in a dynamic way. The new algorithm is needed since the existing algorithms [21, 23, 22, 26, 6, 9, 13, 18, 25, 27, 14, 2, 29] either increase the I/O complexity or cannot be easily tailored to fully utilize the properties of the rectangles generated by Step 2. Detail discussions on the choices are provided in the technical presentation.

The organization of this paper is as follows. Section 2 discusses approaches to evaluate spatial joins of general polygons. Section 3 presents the line segment intersection algorithms. Section 4 develops a new rectangle join algorithm. Section 5 gives the trapezoid join algorithm. Section 6 discusses the join evaluation for I/O bounded polygons. Section 7 concludes the paper.

2 Spatial Joins: Rectangles, Polygons, and Trapezoids

The traditional approach [21] of spatial join evaluation first finds all pairs of objects whose MBR's intersect and then examines for every pair of spatial objects produced if the objects intersect. Consider two polygons t and t' in Fig. 1(i). The traditional algorithm will report the MBR's of t and t' intersect. However, the objects actually do not intersect.

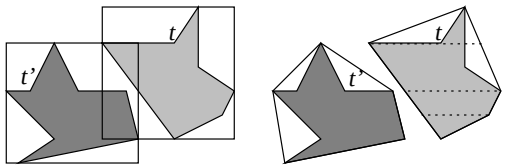

Figure 1. MBR's and 5-corner approximations

An improvement [8] is to add an extra step to detect the intersections of n-corner (e.g., 5-corner) convex polygons (or other approximations) for those whose MBR's intersect. Fig. 1(ii) shows that the 5-corner approximations for t and t' do not intersect. In this way, we know t and t' do not intersect without actually examining t and t'.

An idea to further improve the filter effectiveness is to combine the MBR-filtering with n-corner approximation intersection testing into a single direct join algorithm for n-corner approximations. For example, we can represent a 5-corner convex polygon by four "trapezoids". A trapezoid is a four corner convex polygon with two horizontal boundaries. Fig. 1(ii) shows a decomposition of the 5-corner approximation of t into four trapezoids. A focus of this paper is on the efficient evaluation of spatial joins on trapezoids without using any approximations.

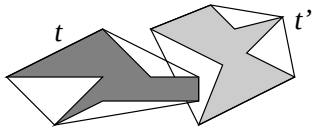

Figure 2. 5-corner approximations

However, these approaches using approximations require the approximations to be computed in advance and the approximations themselves take extra storage space. In addition, since intersections of two approximations do not imply object intersections, further operations need to be performed on actual objects. A better alternative can evaluate join operations directly on polygon objects, if efficient algorithms exist. For example, Fig. 2 shows two polygons that do not intersect although their 5-corner approximations do. One idea is to decompose polygons into trapezoids and use an efficient trapezoid join algorithm to evaluate polygon joins. Such an algorithm will still work as long as the size of each polygon involved is not too large, e.g., "I/O bounded" (retrieval of one polygon can be done in a constant number of I/Os). However, in the general case, since a polygon could be decomposed into a large (unbounded) number of trapezoids, it remains open if a polygon join can be evaluated efficiently.

3 Line Orderings and Line Segments Intersections

The first step of a trapezoid join is to detect non-horizontal boundary intersections. This problem is closely related to line segment intersection [7, 10, 20, 12, 5] in computational geometry. When there are no intersections among the non-horizontal boundaries of the same trapezoid set, the problem is closely related to the red/blue line segment intersection problem [19, 11, 24], a variant of line segment intersection. In this section, we show that some in-memory algorithms for these two problems can be extended to detect intersections between non-horizontal boundaries of trapezoids in trapezoid join. In particular, the extended algorithms can produce the orderings of non-horizontal boundaries along a horizontal line which is critical in computing the trapezoid-to-rectangle mapping.

Let r, s be two sets of N trapezoids, b the page size, and k the total number of intersections between non-horizontal boundaries of trapezoids in r and of trapezoids in s. When non-horizontal boundaries of trapezoids in the same set do not intersect, the red/blue line segment intersection algorithm of Mairson and Stolfi [19] is extended in this section to find all intersections of non-horizontal boundaries of trapezoids in r and s in $O(N \log_b \frac{N}{b} + k)$ I/Os. For the general case, we extend the algorithm for line segment intersection by Bentley and Ottmann [7] for finding all intersections of non-horizontal boundaries of trapezoids in r and s in $O((N + k + l) \log_b \frac{N}{b})$ I/Os, where l is the total number of intersections between non-horizontal boundary lines of trapezoids in the same set. Although both extensions apply the plane sweeping technique as their respective in-memory versions, the primary difference is that the extended algorithms use B-trees as the intermediate data structures rather than balanced binary search trees.

It is important to note that one key piece of information used in the trapezoid join algorithm is the ordering of non-horizontal boundaries along a horizontal sweep line. Roughly, all boundary lines intersecting the sweep line can be ordered based on the x-coordinates of their intersection points along the sweep line. The sweep

line moves from bottom up, and the ordering of the boundaries changes only when the sweep line encounters an endpoint of a line segment or an intersection point. An important property of the two extended algorithms mentioned above is that we can obtain the ordering of non-horizontal boundaries at any position of the sweep line during their execution. This is a key for the second step of the trapezoid join algorithm.

More specifically, for a fixed position of the sweep line ℓ, all boundary lines intersecting ℓ are ordered by the intersection points with the line ℓ (from left to right). Since the ordering changes when the sweep line moves, there is a list of orderings of line segments. This list of ordering is critical for the trapezoid join because it is used in the second step of trapezoid join to map trapezoids into rectangles that preserve certain topological relationships. Before we discuss the details of the line segment intersection algorithm, we formalize the notion of an "ordering" and a (sorted) list of orderings.

Let S be a set of line segments. An *S-sequence* is a sequence of distinct line segments (not necessarily all) in S. In particular, an empty sequence (denoted by $\emptyset$) contains no line segment.

Definition 3.1 Let S be a set of line segments and α a real number. An *ordering of S at α*, denoted by $order(S, \alpha)$, is defined as an S-sequence $l_1, ..., l_k$ $(k \geq 0)$ such that:

- $l_1, ..., l_k$ are all line segments in S that intersect the horizontal line $y{=}\alpha$, and
- for all $0 \leq i < j \leq k$, if $(p_x^i, p_y^i), (p_x^j, p_y^j)$ are the intersection points of $y{=}\alpha$ with l_i, l_j (respectively), then $p_x^i \leq p_x^j$.

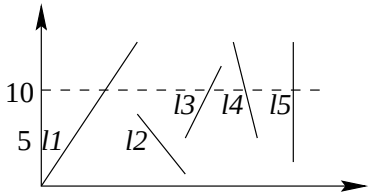

Figure 3. An ordering of S at 10

Intuitively, an ordering of S at α is a sequence of line segments in S sorted by the intersection points with the line $y{=}\alpha$. If two lines have exactly the same intersection point, the order is arbitrary. Fig. 3 shows a set of line segments $S{=}\{\ell_1, ..., \ell_5\}$. The dashed line is $y{=}10$, and the ordering of S at 10 is then $\langle \ell_1, \ell_3, \ell_4, \ell_5 \rangle$. In the following, we capture the changes to the ordering during sweeping.

Definition 3.2 Let S be a set of line segments. An *ordering list* of S, denoted by $OrdList(S)$, is defined as a sequence of pairs $(\alpha_1, T_1), \ldots, (\alpha_k, T_k)$ for some positive integer k such that the following conditions are all true.

- $\alpha_1 < \cdots < \alpha_k$,
- T_i is an ordering of S at α_i for each $1 \leq i \leq k$ and T_i is different from T_{i+1} for each $1 \leq i < k$, and
- for each real number $\alpha \notin \{\alpha_1, ..., \alpha_k\}$, let ℓ be the largest i such that $\alpha_i < \alpha$ if it exists or 0 otherwise. Then, the following are true:
  1. if $\ell = 0$, $T_1 = order(S, \alpha)$,
  2. if $\ell = k$, $T_k = order(S, \alpha)$, and
  3. otherwise, either T_ℓ or $T_{\ell+1}$ equals $order(S, \alpha)$.

For example,the following is an ordering list of the set S shown in Fig. 3:

$$
\begin{array}{l}
(-1, \emptyset) \\
(\ 0, \langle \ell_1 \rangle), \\
(\ 1, \langle \ell_1, \ell_2 \rangle), \\
(\ 2, \langle \ell_1, \ell_2, \ell_5 \rangle), \\
(\ 5, \langle \ell_1, \ell_2, \ell_3, \ell_4, \ell_5 \rangle), \\
(\ 8, \langle \ell_1, \ell_3, \ell_4, \ell_5 \rangle), \\
(14, \langle \ell_1, \ell_4, \ell_5 \rangle), \\
(20, \emptyset).
\end{array}
$$

We now discuss how to extend the in-memory algorithms for line segment intersection problem to detect non-horizontal boundary intersections of trapezoids. In order to be used in the trapezoid join algorithm, the extended algorithm must have the property of reporting the ordering lists of non-horizontal boundaries in the trapezoid sets. We develop an external algorithm lintersect extended from the algorithm MS by Mairson and Stolfi [19] for the case of no non-horizontal boundary intersections in the same set, and another external algorithm extended from the line segment intersection algorithm BO of Bentley and Ottmann [7] for the general case where non-horizontal boundaries in the same set may intersect.

For efficient evaluation of trapezoids, line segment intersection algorithms to be used for non-horizontal boundary intersection should keep the I/O complexity as low as possible. More importantly, we must be able to obtain the ordering lists of non-horizontal boundaries during the execution. The algorithms of [11, 24, 4] do not seem to provide the ordering lists easily. We use the algorithm MS because we can obtain the ordering lists from it and it has low (in-memory) complexity. For the general case, the algorithms of [12, 20] use randomization and do not have optimal worst case complexity. The in-memory algorithms of [10, 5] and external algorithm of [4] have lower complexity but we cannot obtain the ordering information from them. For this reason, we use the algorithm BO.

We first present the algorithm lintersect, which extends MS and uses the plane sweeping technique [28] for the case where there are no non-horizontal boundary intersections in the same set. We then give a brief description of an algorithm that extends BO for the general case.

For the case of no non-horizontal boundary intersection, we let r and s be two sets of line segments. In lintersect, the sweep line is horizontal and moves vertically from small to larger value during the execution of the algorithm. The sweep line is initially below the lowest endpoint of all line segments in r and s. A line segment becomes *active* if it intersects the current sweep line. Two *active* sets are used to keep track of the active line segments in r and s, respectively. Initially, both active sets are empty. A line segment is inserted into the corresponding active set when the sweep line reaches its lower endpoint. It becomes *dead* and is deleted from the active set when the sweep line meets its upper endpoint. For each position α of the sweep line, the active sets of r and s maintain $order(r, \alpha)$ and $order(s, \alpha)$, respectively. Because there are no intersections between line segments from the same set, the active set only changes when a line segment becomes active or dead. In MS, balanced binary search trees are used to maintain the active sets. In lintersect, however, we use B-trees in order to achieve I/O efficiency. Each entry in a node of a B-tree stores an active line segment. Inserting, deleting, and searching of a line segment in an active set can be done in $O(\log_b \frac{N}{b})$ I/Os. In addition, for every entry e corresponding to line segment ℓ in a B-tree, we maintain a link to the node containing the entry for the successor of ℓ and a similar link for the predecessor of ℓ. Adding these links does not increase the I/O complexity of the insertion, deletion, and search operations on B-trees but allows to locate the successor and predecessor of ℓ at any position of the sweep line in $O(1)$ I/Os.

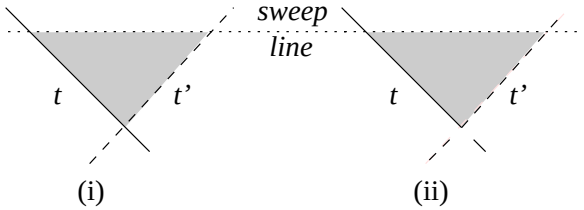

Figure 4. A cone: before and after it is broken

Now we discuss how the algorithm finds intersections between line segments. We only give the basic idea here, details can be found in [19]. The algorithm ensures that when a line segments ℓ is about to be deleted from the active set, all intersections of ℓ with line segments in the opposite set are reported. To help understand the algorithm, the concept "cone" is introduced in [19]. When two active line segments $t \in r$ and $t' \in s$ intersect, the region bounded by t, t' and the current sweep line (which must be above the intersection point of t and t') forms a cone. Fig. 4(i) shows an example. The *Cone Invariant* says that no active line segment begins in a cone. It was proved in [19] that if the Cone Invariant holds, all active line segments ℓ in r (or s) intersecting an arbitrary line segment ℓ' in the opposite set will occur in consecutive position in the ordering of r (or s) at the upper endpoint of ℓ'. To ensure that the Cone Invariant holds, whenever a new line segment is to be inserted into the active set, we must check if it lies in any cones; and if it does, we must break all those cones by truncating the line segments that form the cones to just above their intersection points. Fig. 4(ii) shows such a situation. Meanwhile, all intersections that forms these cones must be reported. Also because the Cone Invariant holds, the reporting of intersections on an active line segment ℓ becomes simple. When ℓ is about to be removed from the active set, we follow the ordering preserved in the opposite active set, report its intersections with active line segments in the opposite set, starting from its successor or predecessor in the ordering of the opposite set, whichever intersects ℓ, until the sweep line meets the first line segment that does not intersect ℓ.

The following example illustrates the execution of algorithm lintersect.

Example 3.3 Let r and s be two sets of line segments. Part of r and s are shown in Fig. 5, where A, B are in r and a, b in s. When the sweep line reaches the lower endpoint of b (Fig. 5), because b lies in the cone formed by A, a, and the current sweep line, the cone must be broken and the intersection of A and a is reported.

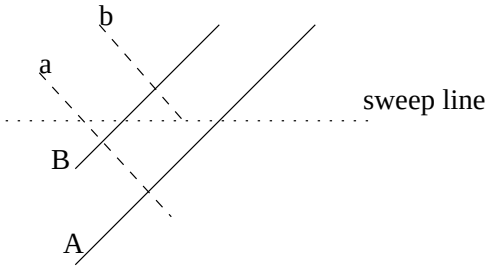

Figure 5. Two sets of line segments

When the sweep line encounters the upper endpoint of a, the successor and predecessor of a in the opposite set are examined. In this case, the successor, B, intersects a and is reported. The algorithm then checks the successor of B, which is A. Because both A and a are truncated when the cone formed by them is broken in previous processing, they do not intersect now. The algorithm then deletes a from the active set and moves to the next position. ∎

The algorithm lintersect uses the same idea as MS except that it uses B-trees for active sets. The correctness of lintersect follows the proof in [19]. We now consider the I/O complexity of lintersect.

Theorem 3.4 Let r, s be two sets of N line segments. The I/O complexity of lintersect is $O(N \log_b N + k)$, where k the number of pairs of rectangles that intersect.

Proof: (Sketch) Assume that all endpoints of line segments in r and s are sorted by their y coordinates. The sweeping takes $O(\frac{N}{b})$ I/Os. Because B-trees are used to maintain the active sets, all insertion and deletion operations during the sweeping takes $O(N \log_b \frac{N}{b})$ I/Os. It follows from [19] that at each lower endpoint of a line segment ℓ, breaking the cones takes $O(\log_b \frac{N}{b} + c_\ell)$ I/Os, where c_ℓ is the number of intersections that form these cones. Also, at each upper endpoint of a line segment ℓ, locating the successor and predecessor of ℓ in the opposite set takes $O(\log_b \frac{N}{b})$ I/Os, and reporting all the intersections takes $O(k_\ell)$ I/Os, where k_ℓ is the number of intersections reported. Therefore, the total I/O complexity of this algorithm is $O(\log_b \frac{N}{b} + k)$ I/Os, where k is the total number of line intersections. ∎

The computation of the trapezoid-to-rectangle mapping needs the ordering lists of non-horizontal boundaries of trapezoids. The extended algorithm lintersect can be used in the trapezoid join because it has the following property.

Lemma 3.5 Let r, s be two sets of line segments. For each real number α, there exists an execution instant τ in lintersect such that the orderings $order(r, \alpha)$ and $order(s, \alpha)$ are stored in the B-trees of lintersect at time τ. Moreover, for all real numbers α, α' such that $\alpha < \alpha'$, if the orderings at α, α' are stored at times τ, τ' (respectively), then $\tau < \tau'$.

From the description of the algorithm, we know that during the execution of lintersect, two B-trees are built to maintain the orderings of r and s at any position of the sweep line. In particular, when a line segment becomes active, the ordering changes and the B-tree is modified based on the change. Therefore each possible ordering is stored at some point of the execution. Furthermore, the orderings are "sorted" according to the time of their occurrences.

Finally we briefly discuss the general case where boundary lines in the same set may intersect. For this case we extend the plane sweeping based algorithm BO by Bentley and Ottmann [7] for line segment intersection.

Let r, s be two sets of trapezoids and r_l, s_l the sets of non-horizontal boundary lines of r and s, respectively. To find the intersections between non-horizontal boundary lines of trapezoids in r and s, we extend BO and apply it on r_l and s_l. Like BO, the extended algorithm applies the plane sweeping technique. It maintains line segments of r and s that intersect the current sweep line in two intermediate data structures called "active sets", respectively. The basic idea of the extended algorithm is the same as BO. But in BO, the active sets are maintained by balanced binary search trees, while in the extended algorithm, B-trees are used for I/O efficiency. It is not hard to verify that the I/O complexity of the extended algorithm is $O((N + k + l) \log_b \frac{N}{b})$, where k is the number of intersections between non-horizontal boundary lines from different sets, and l is the number of intersections of non-horizontal boundary lines within the same trapezoid set.

4 Rectangle Join Revisited

Section 3 covers non-horizontal boundary intersections of trapezoids. The remaining cases of trapezoid intersection are trapezoid containment and intersections involving horizontal boundaries. It seems that determining the intersections of trapezoids can be extended from the line segment intersection algorithm presented in Section 3. Surprisingly, in Section 5, we show that trapezoid containment and intersections involving horizontal boundaries can be reduced to a restricted version of rectangle join using the ordering lists of non-horizontal boundaries. This has a cost. The existing algorithms [21, 23, 22, 26, 6, 9, 13, 18, 25, 27, 14, 2, 29] either lead to high complexity or cannot be pipelined with the line segment intersection algorithm. For this reason, we develop a new rectangle join algorithm rjoin which can be extended for the restricted rectangle join. We show that rjoin evaluates a join of two sets of N rectangles in $O(N \log_b N + k)$ I/Os, where b is the page size and k is the number of rectangle intersections.

The rectangle join algorithm used in trapezoid join should process rectangles incrementally from bottom up so that it can be pipelined with the previous steps of trapezoid join. Recall that a trapezoid join has three steps. The first step computes the non-horizontal boundary intersections, and the last two steps map trapezoids into rectangles and compute rectangle intersections (respectively). Theoretically it is possible to compute the three steps separately. However, it is more desirable to have an integrated algorithm that pipelines all three steps. As we will show later in Section 5, the computation of the trapezoid-to-rectangle mapping can be pipelined with the first step of trapezoid join and generate the rectangles incrementally in the direction of sweeping (bottom up). To be pipelined with these steps, the rectangle join algorithm should

also process rectangles in the same order.

We now examine existing rectangle join algorithms. The algorithms of [21, 23, 22, 26, 6, 9, 13, 25, 18, 27, 14] cannot guarantee optimal worst case I/O complexity. In the worst cases, the I/O complexity could be quadratic even when the number of rectangle intersections is small. The algorithms in [1] and [29] have desirable I/O complexity. But both algorithms require index structures on the x-projections of rectangles to be constructed before processing the join. This makes pipelining impossible and causes additional I/Os and storage space. The I/O efficient algorithm in [2] does not use any index structures. However, it requires all rectangles to be sorted both vertically and horizontally before the join, thus cannot be pipelined with the line segment intersection algorithm. Sorting also requires additional I/Os and storage space.

In this section, we present a new rectangle join algorithm rjoin. Unlike other algorithms, rjoin does not require any index structure or pre-computation. It applies plane sweeping technique, processes one rectangle at a time, thus can be pipelined with the line segment intersection algorithms and the computation of the trapezoid-to-rectangle mapping to solve trapezoid join efficiently. The key technique of algorithm rjoin is that it maps rectangles into 2-dimensional points. The y-coordinates of these points change when the sweep line moves. Rectangle intersections are detected by 3-sided range searches performed on the points corresponding to rectangles in the opposite set.

We now give a brief description of the basic idea of rjoin. Let r and s be two sets of rectangles. The algorithm rjoin uses a horizontal sweep line that initially lies below the lowest rectangle in r and s. During the execution of the algorithm, the sweep line moves from bottom up. Rectangles intersecting the current sweep line are mapped into two 2-dimensional points and stored in intermediate data structures called *past sets*. More importantly, when the sweep line leaves the upper boundary of a rectangle t, the y-coordinates of the points corresponding to t change and a 3-sided range search is performed to find rectangles in the opposite set that intersect t.

We first define the mapping between rectangles and points.

Let t be a rectangle. In the remainder of this section, we denote by $t.x$ the projection of t on the x axis (x-projection), and $t.y$ the projection of t on the y axis (y-projection). Moreover, $t.x^L, t.x^U$ denote the lower, upper boundary of $t.x$ (respectively), and $t.y^L$, $t.y^U$ refer to the lower, upper boundary of $t.y$ (respectively).

Let $\mathbf{B}$ be the set of rectangles in plane and $\mathbb{R}$ the set of real numbers. We denote $\mathbb{R} \cup \{+\infty\}$ by $\mathbb{R}^+$.

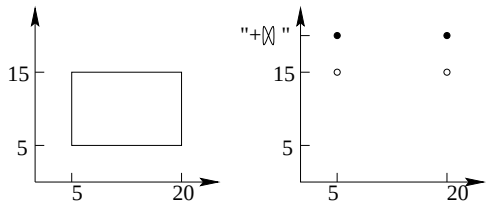

Figure 6. A rectangle and its mapped points

Definition 4.1 We define a partial mapping $f : \mathbf{B} \times \mathbb{R} \rightarrow ((\mathbb{R} \times \mathbb{R}^+) \times (\mathbb{R} \times \mathbb{R}^+))$ such that for every $t \in \mathbf{B}$ and $\alpha \in \mathbb{R}$,

1. If $\alpha < t.y^L$, $f(t, \alpha)$ is undefined;
2. If $t.y^L \leq \alpha \leq t.y^U$, $f(t, \alpha) = ((t.x^L, +\infty), (t.x^U, +\infty))$; and
3. If $\alpha > t.y^U$, $f(t, \alpha) = ((t.x^L, t.y^U), (t.x^U, t.y^U))$.

The following example illustrates the mapping.

Example 4.2 The left part of Fig. 6 shows a rectangle t whose x-projection and y-projection are $[5, 20]$ and $[5, 15]$, respectively. When α is less than 5, $f(t, \alpha)$ is undefined. There is no point corresponding to t in the right part of Fig. 6. When α is in between 5 and 15, the two points corresponding to t are $(5, +\infty)$ and $(20, +\infty)$, the darker ones in the right part of Fig. 6. When α is greater than 15, the two points corresponding to t are $(5, 15)$ and $(20, 15)$, the lighter ones in the right part of Fig. 6. ∎

The mapping f has the following property. Intuitively, the property says that if two rectangles t and t' intersect, then when α equals the upper boundary of one of them, say t, one of the points mapped from t' must lies in the region between the vertical boundary lines of t and above the lower boundary of t.

Theorem 4.3 Let t, t' be two rectangles in $\mathbf{B}$. t intersects t' if and only if the following is true:

1. $f(t', t.y^U)$ is defined and $= ((a, b), (c, b))$ for some $a, c \in \mathbb{R}$ and $b \in \mathbb{R}^+$, $b \geq t.y^L$, and either $a \in [t.x^L, t.x^U]$ or $c \in [t.x^L, t.x^U]$, or
2. $f(t, t'.y^U)$ is defined and $= ((a, b), (c, b))$ for some $a, c \in \mathbb{R}$ and $b \in \mathbb{R}^+$, $b \geq t'.y^L$, and either $a \in [t'.x^L, t'.x^U]$ or $c \in [t'.x^L, t'.x^U]$.

Proof: (Sketch) We first consider the only if direction. Since t intersects t', their y-projections overlap. Because t, t' are symmetric in the statement of the theorem, we assume w.o.l.g. that $t.y^U$ is less than or equal to $t'.y^U$. It follows that $t.y^U$ must be greater than (or equal to for the degenerated case) $t'.y^L$. By Definition 4.1, $f(t', t.y^U)$ is defined and equal to $((t'.x^L, +\infty), (t'.x^U, +\infty))$. Obviously, $+\infty > t.y.L$.

Because t intersects t', their x-projections overlap too. Two x-projections overlap if and only if one contains at least one endpoint of the other. If the x-projection of t contains one endpoint of the x-projection of t', say $t'.x^L$, then $t'.x^L \in [t.x^L, t.x^U]$. Condition (1) is satisfied.

If the x-projection of t' contains at least one endpoint of the x-projection of t, say $t.x^L$, then $t.x^L \in [t'.x^L, t'.x^U]$. From the assumption, we know that $t'.y^U$ is greater than $t.y^U$. It follows that $f(t, t'.y^U)$ is defined and equal to $((t.x^L, t.y^U), (t.x^L, t.y^U))$. Because t intersects t', $t.y^U$ must be greater than or equal to $t'.y^L$. Condition (2) holds.

Now for the if direction. Suppose that t and t' satisfy Condition (1). Obviously, the x-projection of t intersects the x-projection of t'. Because $f(t', t.y^U)$ is defined, by Definition 4.1, $t.y^U \geq t'.y^L$. If it is the case that $t.y^U \leq t'.y^U$, then the y-projection of t' also intersects the y-projection of t, t and t' intersect. Otherwise, $f(t', t.y^U) = ((t'.x^L, t'.y^U), (t'.x^U, t'.y^U))$. By the condition we know that $t'.y^U \geq t.y^L$. Thus the y-projection of t' intersects the y-projection of t, t intersects t'. The case when Condition (2) holds can be proved in a similar way. ∎

Based on the above discussions, the algorithm rjoin can be easily developed. We illustrate rjoin with the following example.

Example 4.4 Let r and s be two sets of rectangles. Parts of r and s are shown in Fig. 7, where A, B, C are in r and a, b, c are in s. Initially, the sweep line is below the lowest rectangle, B, and the past sets of r and s contain no points. Fig. 7(ii) and (iii) illustrate

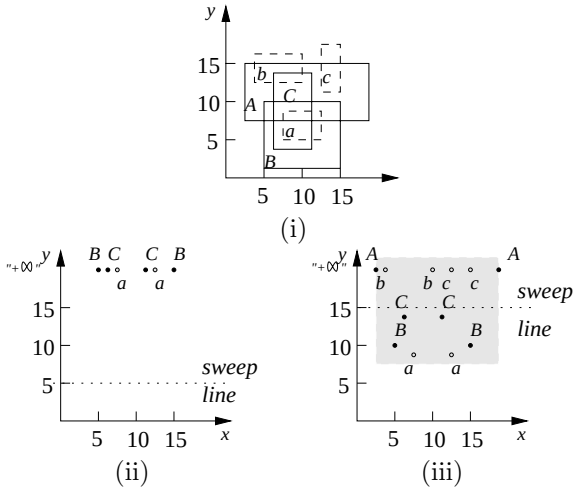

Figure 7. Rectangle sets and their past sets

the points mapped from the rectangles in Fig. 7(i) when the sweep line is at two different positions.

During the execution of rjoin, the sweep line moves up. When the sweep line moves to the lower boundary of B ($y = 1$), the two points defined by $f(B, 1)$, $(5, +\infty)$ and $(15, +\infty)$, are inserted into the past set of r.

When the sweep line reaches the lower boundary of a ($y = 5$), the points mapped from B and C are already in the past sets of r. Two points defined by $f(a, 5)$ are now inserted into the past set of s. Fig. 7(ii) shows the points currently stored in the past sets of r and s (respectively), where the dark ones are for r and the light ones are for s.

When the sweep line moves to the upper boundary of A ($y = 15$), it is above rectangles B, C, and a. The points mapped from these rectangles are different from those in Fig. 7(ii). Now these points are all below the sweep line $y = 15$, as shown in Fig. 7(iii). Rectangles A, b, and c intersect the sweep line, and the points mapped from them locate on $y = +\infty$. A 3-sided range search is performed at this moment to find all points (x, y) that satisfy $2 \leq x \leq 19$ and $y \geq 7$ (i.e., points that locate within the shaded region in Fig. 7(iii)). As a result, a, b, and c are reported.

The sweep line keeps moving up and the algorithm completes when the sweep line leaves the highest upper boundary of rectangles in r and s. ∎

The correctness of algorithm rjoin follows from Theorem 4.3.

We now consider the I/O complexity of rjoin. A key factor in the performance of rjoin is the performance of the insertion, update, and 3-sided range search operation on the past sets of r and s. To achieve I/O efficiency, we must organize the past sets in such a way that these operations can be implemented efficiently. We decide to use the external priority search tree developed in [3]. The external priority search tree is a tree structure designed to solve 3-sided range search problem. It is shown in [3] that an external priority search tree occupies $O(\frac{N}{b})$ pages and it supports insertions and deletions in $O(\log_b N)$ I/Os and 3-sided range searches in $O(\log_b N + k)$ I/Os, where N is the number of points, b is the page size, and k is the number of points in the query results.

Theorem 4.5 Let r, s be two sets of N rectangles. The I/O complexity of rjoin is $O(N \log_b N + k)$, where k is the number of pairs of intersecting rectangles.

Proof: (Sketch) The algorithm rjoin sweeps through all rectangles in r and s and executes procedures for each rectangle encountered. The I/O complexity of rjoin is determined by the number of rectangles and the processing of each rectangle.

The sweep accesses each rectangle in r and s at most twice (one for each horizontal boundary). The number of I/Os for this phase is $O(\frac{N}{b})$. And the total number of executions of each operation (insertion, update, and 3-sided range search) is $O(N)$.

Each past set contains at most $2N$ points during the execution. The past sets are implemented by external priority search trees. It is shown in [3] that the insertion operation on an external priority search tree takes $O(\log_b N)$ I/Os. And one can also show that the update operation also takes no more than $O(\log_b N)$ I/Os. Thus all executions of insertion and update operations take $O(N \log_b N)$ I/Os.

For each rectangle t, a 3-sided range search is also executed. This takes $O(\log_b N + k_t)$ I/Os according to [3], where k_t is the number of intersections discovered by the range search. We observe that one pair of intersection is discovered by at most two executions of a 3-sided range search. This is because range searches are only executed at the upper boundaries of rectangles. Let k be the total number of rectangle intersections. It follows that $\Sigma_{t \in r \cup s} k_t = O(k)$. Therefore, all executions of the range searches take $O(N \log_b N + k)$ I/Os.

In conclusion, the total number of I/Os needed by rjoin is $O(N \log_b N + k)$, where k is the number of intersections. ∎

5 Spatial Join for Trapezoids

In the previous two sections, we introduced the key techniques for the first (computation of non-horizontal boundary intersections) and third (computation of rectangle intersections) steps of trapezoid join. In this section, we introduce the technique of computing the trapezoid-to-rectangle mapping and show how to use the line segment intersection algorithms introduced in Section 3 to compute the mapping based on the ordering lists obtained from the execution of the algorithms. We combine all these techniques together into efficient trapezoid join algorithms. The main results of this section are the following. In the case where there are no intersecting non-horizontal boundaries of trapezoids in the same set, the join of two sets of N trapezoids can be computed in $O(N \log_b N + k)$ I/Os, where b is the page size and k is the number of trapezoid intersections. In the general case the problem can be evaluated in $O((N + l + k) \log_b N)$ I/Os, where l is the number of intersections of non-horizontal boundaries within the same trapezoid set, and b, k are as above.

The key technique of trapezoid join is to reduce the trapezoid join problem into the rectangle join problem. Basically, trapezoids are mapped into rectangles that preserve the ordering of non-horizontal boundary lines such that trapezoid containment and intersections involving horizontal boundaries can be determined by checking intersections of the corresponding rectangles. These rectangles are called the "rectangle representations" of trapezoids.

In this section, we first assume that there are no intersecting non-horizontal boundary lines of trapezoids in the same set and give details of the trapezoid join algorithm tjoin. We then give a brief description of the algorithm for the general case. In the remainder of this section, we fix the following notations. Let r, s be two sets of N trapezoids and r_l, s_l the sets of non-horizontal boundaries of r, s (respectively). We use t^L and t^R to denote the left and right (respectively) non-horizontal boundaries of a trapezoid t.

Definition 5.1 A *pre-representation* of r, s is a mapping from $r_l \cup s_l$ to $\mathbb{R}$ such that for all lines ℓ and ℓ' in $r_l \cup s_l$, if ℓ' intersects the horizontal line that overlaps the lower endpoint (x_0, y_0) of ℓ at (x_0', y_0) and $x_0' \theta x_0$, where $\theta \in \{>, <, =\}$, then $f(\ell') \theta f(\ell)$.

We now give the definition of "rectangle representations" of trapezoids.

Definition 5.2 Let f be a pre-representation and t a trapezoid in $r \cup s$. The *rectangle representation of t under f wrt r and s*, denoted by $\bar{f}(r \cup s, t)$, is the rectangle defined as follows:

1. The y-projection of $\bar{f}(r \cup s, t)$ is the same as the y-projection of t, and
2. The x-projection of $\bar{f}(r \cup s, t)$ is $[f(t^L), f(t^R)]$.

Note that a pre-representation of r, s does not necessarily map a non-horizontal boundary ℓ to the x-coordinate of ℓ's lower endpoint. The importance of pre-representations is that they preserve the orderings of r_l and s_l (respectively) at the lower boundary line of every trapezoid. In particular, it can be shown that a pre-representation f has the following property. For every non-horizontal boundary $\ell \in r_l \cup s_l$ with lower endpoint (x_0, y_0), if $order(r_l \cup \{\ell\}, y_0)$ is $\ell_1, ..., \ell_k$ for some $k \geq 1$, then $f(\ell_1) \leq \cdots \leq f(\ell_k)$. Similar property holds for $order(s_l \cup \{\ell\}, y_0)$.

Recall that two trapezoids intersect if and only if their non-horizontal boundaries intersect, one contains the other, or their boundaries intersect but non-horizontal boundaries do not intersect. The following lemma shows that the last two cases of trapezoid intersection can indeed be captured by the intersection of their rectangle representations.

Lemma 5.3 Let f be a pre-representation of r, s and t, t' two trapezoids in r, s (respectively) whose non-horizontal boundaries do not intersect. Then t intersects t' if and only if $\bar{f}(r \cup s, t)$ intersects $\bar{f}(r \cup s, t')$.

Proof: (Sketch) Let the lower bounds of the y-projections of t and t' be y_0 and y'_0, respectively.

We first consider the if direction. If $\bar{f}(r \cup s, t)$ intersects $\bar{f}(r \cup s, t')$, by Definition 5.2 the y-projections of t and t' must intersect. Without loss of generality, we assume that y_0 is less than y'_0.

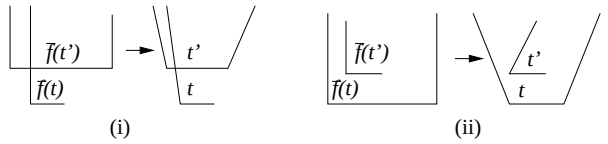

Figure 8. Rectangle representations and their corresponding trapezoids

It is also true that the x-projection $\bar{f}(r \cup s, t).x$ of $\bar{f}(r \cup s, t)$ intersects the x-projection $\bar{f}(r \cup s, t').x$ of $\bar{f}(r \cup s, t')$. Two intervals intersect if one contains the endpoint of the other. Assume that $\bar{f}(r \cup s, t').x$ contains one endpoint of $\bar{f}(r \cup s, t).x$, without loss of generality, let it be the lower endpoint (Fig. 8(i)). By Definition 5.2, the intersection point of t^L and the horizontal line $y = y'_0$ must be between the lower endpoints of t'^L and t'^R on $y = y'_0$, as shown in Fig. 8(i). Then t^L must intersect the lower boundary of t', thus t intersect t'. Now consider the case where $\bar{f}(r \cup s, t).x$ contains one endpoint of $\bar{f}(r \cup st').x$, without loss of generality, let it be the lower endpoint (Fig. 8(ii)). By Definition 5.2, on the horizontal line $y = y'_0$, the lower endpoint of t'^L must lie between the intersection point of t^L and $y = y'_0$ and the intersection point of t^R and $y = y'_0$ (Fig. 8(ii)). Thus t intersects t'.

Now we consider the only if direction.

For the case where t contains t', it is easy to see that the y-projection of $\bar{f}(r \cup s, t)$ contains the y-projection of $\bar{f}(r \cup s, t')$. The non-horizontal boundaries of t must intersect the horizontal line $y = y'_0$. And on the line $y = y'_0$, the lower endpoints of t'^L and t'^R must lie between the intersection points of $y = y'_0$ and t^L, t^R. By Definition 5.2, $f(t^L)$ is less than or equal to $f(t'^L)$ and $f(t'^R)$, and $f(t^R)$ is greater than or equal to $f(t'^L)$ and $f(t'^U)$. The x-projection of $\bar{f}(r \cup s, t)$ contains the x-projection of $\bar{f}(r \cup s, t')$. Therefore, $\bar{f}(r \cup s, t)$ intersects $\bar{f}(r \cup s, t')$. The case where t' contains t can be proved similarly.

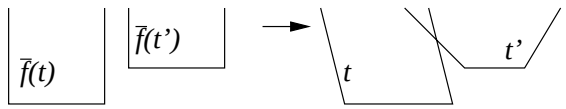

Figure 9. Rectangle representations and the corresponding trapezoids

Now we consider the case where the boundary lines of t and t' intersect. We prove this by contradiction. Assume that $\bar{f}(r \cup s, t)$ does not intersect $\bar{f}(r \cup s, t')$. If their y-projections do not intersect, by Definition 5.2, the y-projections of t and t' do not intersect. This contradicts the fact that t intersects t'. Therefore, the y-projections of $\bar{f}(r \cup s, t)$ and $\bar{f}(r \cup s, t')$ must intersect. Without loss of generality, we assume that y_0 is less than y'_0. It follows that both t^L and t^R intersect the horizontal line $y = y'_0$. If the x-projections of the two rectangles do not intersect, then on $y = y'_0$, either the intersection points of t^L and t^R with $y = y'_0$ are to the left of the lower endpoint of t'^L, or both of them are to the right of the lower endpoint of t'^R. Without loss of generality, assume it is the first case (Fig. 9). Because t intersects t', it must be true that t^R intersects t'^L, This conflicts the fact that the non-horizontal boundaries of t and t' do not intersect. Therefore, the assumption that $\bar{f}(r \cup s, t)$ does not intersect $\bar{f}(r \cup s, t')$ is not correct, $\bar{f}(r \cup s, t)$ and $\bar{f}(r \cup s, t')$ intersect. ∎

We now discuss the trapezoid join algorithm tjoin. Recall that tjoin has the following three steps:

1. Compute non-horizontal boundary intersections,
2. Compute the mapping from trapezoids to rectangles, and
3. Compute rectangle intersections.

Step 1 can be done using the line segment intersection algorithm lintersect introduced in Section 3. Since Step 2 computes a pre-representation of r, s and generates the rectangle representations of trapezoids, Step 3 can be done using the rectangle join algorithm rjoin developed in Section 4, except that the inputs are now pipelined from Step 2. The major remaining problem is Step 2.

Step 2 is to compute the pre-representation of r, s and the rectangle representations of trapezoids incrementally from bottom up. We show below that this can be done efficiently. For simplicity, we first give a conceptual procedure that computes the pre-representation using the ordering lists of r_l and s_l. Then we give the actually computation which is modified from the algorithm lintersect. The computation can access the data structures (B-trees) used in lintersect to incrementally obtain the ordering information and generate the mapping from non-horizontal boundaries to real numbers in the direction of the sweeping in lintersect.

Assume that the ordering lists *OrdList*(r_l) and *OrdList*(s_l) of r_l and s_l are available. We compute a pre-representation f of r, s using plane sweeping technique. The sweep line is a horizontal line that initially lies below the lowest endpoints of all boundaries in $r_l \cup s_l$. The sweeping can be implemented by sorting all boundaries in $r_l \cup s_l$ lexicographically by the y-coordinates and then the x-coordinates of their lower

endpoint (in the actually computation, sorting the x-coordinates is not necessary). Let $\ell_1, \ldots, \ell_n$ be this sorted list of lines in $r_l \cup s_l$, $n \geq 0$.

We compute a pre-representation f for $\ell_1, \ldots, \ell_n$ as following:

- Set $f(\ell_1)$ to be the x-coordinate of the lower endpoint of ℓ_1.
- When the sweeping line encounters the lower endpoint (x_i, y_i) of $\ell_i \in r_l \cup s_l$ $(1 < i \leq n)$, we look for the predecessors of ℓ_i in $order(r_l, y_i)$ and in $order(s_l, y_i)$, let the predecessors be $pred(r_l, y_i, \ell_i)$ and $pred(s_l, y_i, \ell_i)$, respectively. Note that the orderings $order(r_l, y_i)$ and $order(s_l, y_i)$ can be easily obtained from $OrdList(r_l)$ and $OrdList(s_l)$). We also search in $order(r_l, y_i)$ and in $order(s_l, y_i)$ for the first boundaries $succ^+(r_l, y_i, \ell_i)$ and $succ^+(s_l, y_i, \ell_i)$ that are after ℓ_i and the mapping of which have been computed. Note that not all these lines can be found in the orderings of r_l and s_l. We then computed $f(\ell_i)$ based on the following rule:
  - If there exists a line ℓ' in the set $\{succ^+(r_l, y_i, \ell_i), succ^+(s_l, y_i, \ell_i), pred(r_l, y_i, \ell_i), pred(s_l, y_i, \ell_i)\}$, such that x_i is the x-projection of the intersection point of ℓ' and the horizontal line $y = y_i$, then $f(\ell_i) = f(\ell')$.
  - If none of the four lines above exist, We set $f(\ell_i)$ to be the x-coordinate of the lower endpoint of ℓ_i.
  - If only $pred(r_l, y_i, \ell_i)$ and $pred(s_l, y_i, \ell_i)$ exist, let $f(\ell_i) =$

    $$\tfrac{4}{3} \max\{f(pred(r_l, y_i, \ell_i)), f(pred(s_l, y_i, \ell_i))\}.$$

  - If only $succ^+(r_l, y_i, \ell_i)$ and $succ^+(s_l, y_i, \ell_i)$ exist, let $f(\ell_i) =$

    $$\tfrac{1}{3} \min\{f(succ^+(r_l, y_i, \ell_i)), f(succ^+(s_l, y_i, \ell_i))\}.$$

  - If all four lines exist, let $f(\ell_i)$ be the sum

    $$\tfrac{2}{3} \max\{f(pred(r_l, y_i, \ell_i)), f(pred(s_l, y_i, \ell_i))\} + \tfrac{1}{3} \min\{f(succ^+(r_l, y_i, \ell_i)), f(succ^+(s_l, y_i, \ell_i))\},$$

    when ℓ_i is the left boundary of a trapezoid, and

    $$\tfrac{1}{3} \max\{f(pred(r_l, y_i, \ell_i)), f(pred(s_l, y_i, \ell_i))\} + \tfrac{2}{3} \min\{f(succ^+(r_l, y_i, \ell_i)), f(succ^+(s_l, y_i, \ell_i))\},$$

    when ℓ_i is the right boundary of a trapezoid.

The computation stops when there is no boundary starts above the sweeping line.

It is easy to see that the above computation generates a pre-representation f of r, s. The following example illustrates this procedure.

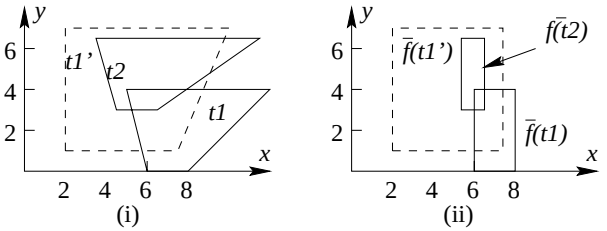

Figure 10. The trapezoids in r and s

Example 5.4 Parts of r and s are shown in Fig. 10(i), where t_1, t_2 are trapezoids in r and t'_1 are in s. The computation starts from the leftmost boundary with the lowest endpoint, t_1^L. We set $f(t_1^L)$ to be 6, the same as the x-projection of the lower endpoint of t_l^L. The next boundary to be considered is t_1^R. It only has a predecessor in $order(r_l, 0)$, thus $f(t_l^R) = \frac{4}{3} \times 6 = 8$.

Because t'^L_1 only has a successor in $order(r_l, 1)$, which is t_l^L, $f(t'^L_1) = \frac{1}{3} f(t_1^L) = \frac{1}{3} \times 6 = 2$. The successor and predecessor of t'^R_1 in $order(r_l, 1)$ are t_1^R and t_1^L (respectively), therefore, $f(t'^R_1) = \frac{2}{3} f(t_1^L) + \frac{1}{3} f(t_1^R) = \frac{2}{3} \times 6 + \frac{1}{3} \times 8 = 6\frac{2}{3}$.

We continue compute the mapping for t_2^L then t_2^R, and get $f(t_2^L) = \frac{1}{3} f(t'^L_1) + \frac{2}{3} \min\{f(t_1^L), f(t'^R_1)\} = \frac{1}{3} \times 2 + \frac{2}{3} \times 6 = 4\frac{2}{3}$, and $f(t_2^R) = \frac{2}{3} \max\{f(t_1^L), f(t'^L_1)\} + \frac{1}{3} \min\{f(t_1^R), f(t'^R_1)\} = \frac{2}{3} \times 6 + \frac{1}{3} \times 6\frac{2}{3} = 6\frac{2}{9}$. ∎

Given a pre-representation f, we can then generate the rectangle representations of trapezoids in r and s. For example, Fig. 10(ii) shows the rectangle representations of trapezoids in Fig. 10(i) using the pre-representation computed in Example 5.4.

Now we consider the actual computation of a pre-representation f in tjoin. The first step of tjoin uses lintersect to compute the non-horizontal boundary line intersections. By Lemma 3.5, the ordering lists of r_l and s_l can be obtained from the execution of lintersect. So we can actually compute a pre-representation by modifying lintersect. Basically, we obtain the ordering information from the two B-trees for r, s (respectively) in lintersect which are uses to capture the changes of orderings and compute the pre-representation incrementally.

Recall that lintersect also uses plane sweeping technique. The sweep line moves from bottom up. B-trees are used to maintain the ordering of boundaries that intersect the current sweep line. Moreover, each boundary ℓ in the B-tree is associated with links to the nodes that contains the processor and successor (respectively) of ℓ in the current ordering. We can then find the predecessor and successor of a given boundary ℓ in no more than $O(\log_b \frac{N}{b})$ I/Os, and find all lines lie before (or after) it following the links.

In order to compute a pre-representation f, we mod-

ify lintersect in the following way. When the sweep line encounters the lower endpoint of a boundary ℓ, we compute $f(\ell)$ before lintersect insert ℓ into the corresponding B-tree. Once the computation and other operations in lintersect complete, we insert ℓ into corresponding B-tree along with $f(\ell)$. To compute $f(\ell)$, we search for the successor and predecessor of ℓ in both B-trees and compute $f(\ell)$ based on the conceptual procedure presented earlier. Note that the successor of ℓ in both B-tree must have been mapped, because we compute the mapping of every boundary before it is inserted into corresponding B-tree. By doing so, we computation of pre-representation is pipelined with lintersect and it is easy to see that every computation takes only $O(\log_b \frac{N}{b})$ I/Os.

Theorem 5.5 For each trapezoid $t \in r$ and each $t' \in s$, t intersects t' if and only if the pair t, t' is reported by tjoin. Furthermore, the I/O complexity of tjoin is $O(N \log_b N + k)$, where k is the number of pairs of trapezoids that intersect.

Proof: (Sketch) Let $t \in r$, $t' \in s$ be two arbitrary trapezoids. We first show that t intersects t' if and only if the pair (t, t') is reported by the algorithm tjoin.

Consider the only if direction. If a non-horizontal boundary line of t intersects a non-horizontal boundary line of t', by the proof in [19] and the description of tjoin, it is easy to see that t and t' will be reported by the steps of lintersect. Otherwise, t and t' either have containment relation ship, or their intersection involves horizontal boundary lines. Then, by Lemma 5.3, $rect(r \cup s, t)$ must intersect $rect(r \cup s, t')$. And it follows from Theorem 4.3 that tjoin will report t and t'.

For the if direction, suppose that two trapezoids, $t \in r$ and $t' \in s$, are reported by tjoin. By the description of the algorithm we know that t and t' must be reported by the steps of lintersect or by a 3-sided range search. If the pair is reported in the first case, by the proof in [19] and the description of tjoin, it is straightforward that the non-horizontal boundary lines of t and t' must intersect, therefore t intersects t'. If the pair is reported by a 3-sided range search, by Theorem 4.3, $rect(t)$ intersects $rect(t')$. It follows from Lemma 5.3 that t intersects t'.

Now we consider the I/O complexity of the algorithm tjoin. The algorithm applies plane sweeping technique, pipelines the steps in lintersect, computation of representative mapping, and the steps in rjoin together. Each pair of intersecting trapezoids will be discovered by no more that 4 times, either by lintersect, or by rjoin.

By Theorems 3.4 and 4.5, we can easily show that all the executions of the operations in the sweeping take $O(N \log_b N + k)$ I/Os, where k is the number of intersections. ∎

Now we briefly discuss the trapezoid join algorithm for the case where intersecting non-horizontal boundary lines of trapezoids in the same set may intersect. Similar to tjoin, the algorithm evaluates trapezoid join in three steps. The first step checks non-horizontal boundary intersections, the second step maps trapezoids into rectangles, and the third step compute rectangle intersections. However, there are following differences.

First, since there are intersections between non-horizontal boundary lines of trapezoids in the same set, the algorithm lintersect does not work. However, we can use the algorithm extended from the line segment intersection algorithm of [7] (as shown in Section 3) can be used. Same as lintersect, this extended algorithm can detect all intersections between non-horizontal boundaries of trapezoids from different set. Also like lintersect, it can generate the ordering lists of non-horizontal boundaries of trapezoids. But in this

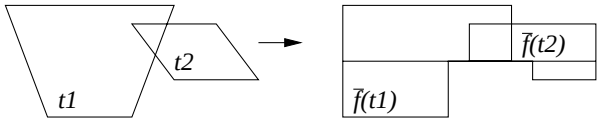

Figure 11. Trapezoids and their rectangles

case, the orderings of line segments change not only when the sweep line encounters new line segments or leaves line segments, but also at the intersection point of two line segments in the same set. For example, considering two trapezoids t_1 and t_2 in the same set r (Fig. 11(i)). The right boundary of t_1 is before both non-horizontal boundaries of t_2 before it intersects the left boundary of t_2. After the intersection, the right boundary of t_1 lies between the two non-horizontal boundaries of t_2.

This change affects the second step of the join. Now a trapezoid may be represented by one or more rectangles. Each of the rectangle represents a change of ordering caused by intersections within the same set. The number of rectangles representing a trapezoid t is determined by the number of non-horizontal boundary lines intersecting t that are from the same set as t. For example, each of t_1 and t_2 in Fig. 11(i) is represented by two rectangles (Fig. 11(ii)). Lemma 5.3 can be extended for this case, i.e., if two trapezoids do not have non-horizontal boundary intersection, then they intersect if and only if some rectangles representing them intersect. This resembles spatial joins for rectilinear polygons [30].

The I/O complexity of this algorithm is higher than tjoin. Let b be the page size and k the number of trape-

zoid intersections. The line segment join algorithm now takes $O((N + k_1 + l) \log_b N)$, where l is the number of intersections of non-horizontal boundary lines within the same trapezoid set, and k_1 is the number of intersections of non-horizontal boundary lines from different trapezoid set, Each trapezoid may now be transformed into more than one rectangles, which makes the total I/O cost of rjoin to be $O((N + l) \log_b N + k_2)$, where k_2 is the total number of intersections discovered by rjoin. We can show that $k_1 + k_2 = O(k)$. Thus this algorithm takes no more than $O((N + l + k) \log_b N)$ I/Os.

6 I/O Bounded Polygons

In this section, we use the trapezoid join algorithms introduced in the previous section to compute spatial joins of "I/O bounded polygons." Intuitively, each I/O bounded polygon can be retrieved with a constant number of I/Os. We show that the join problem of two sets of N I/O bounded polygons can be evaluated in $O(N \log_b N + k)$ I/Os, when there are no boundary intersections between polygons of the same set, and in $O((N + l + k) \log_b N)$ I/Os in the general case, where N is the total number of polygons, b the page size, k the number of pairs of intersecting polygons, and l the number of boundary intersections within the same polygon set.

A polygon is said to be *I/O bounded* if it occupies a constant number of disk page. Thus, it takes $O(1)$ I/Os to access each I/O bounded polygon. Let r and s be two sets of N I/O bounded polygons. To compute the spatial join of r and s, we first decompose each polygon in r and s into trapezoids. This step can be done in no more than $O(N)$ I/Os. It is easy to see that the step results in $O(N)$ trapezoids, since polygons in r and s are all I/O bounded. We then apply the trapezoid join algorithms introduced in Section 5 on the trapezoids. Let k be the number of intersections between polygons in r and s. It can be shown that the number of I/Os required in the join step is $O(N \log_b N + k)$. The I/O complexity of the join evaluation for I/O bounded polygons is summarized in the following theorem.

Theorem 6.1 Let r and s be two sets of N I/O bounded polygons, and k the total number of pairs of polygons from r and s (respectively) that intersect. In the case where there is no boundary intersections of polygons within each of r and s, the join of r and s can be computed within $O(N \log_b N + k)$ I/Os. In the general case, the join can be computed within $O((N + l + k) \log_b N)$ I/Os, where l is the number of boundary intersections within the same polygon set.

7 Conclusions

We developed efficient evaluation algorithms for spatial joins of trapezoids and showed that they can be applied to I/O bounded polygons. This is a significant extension from our earlier algorithms on rectilinear polygons [30]. There are many interesting problems that remain open. For example, it is not clear if the trapezoid join algorithm for the general case can be improved. More generally, it is interesting to develop I/O efficient join algorithms for other types of objects, such as polygons in the general case. It is also very useful to compare the different approaches to spatial join evaluation. For instance, trapezoids can also used as approximations of polygons. It will be interesting to compare different approximations on their approximation quality, effectiveness of filtering (number of false hits), and I/O performance of their join algorithms.

On the other hand, in addition to intersection, other join predicates such as distance and direction related ones are also very useful operations in spatial databases. By studying joins for different types of objects, it might be possible to extend the techniques for joins with other kind of predicates.

References

[1] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data structures*, 1995.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter. Scalable sweeping-based spatial join. In *Proc. Int. Conf. on Very Large Data Bases*, 1998.

[3] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexibility and optimal range search indexing. In *Proc. ACM Symp. on Principles of Database Systems*, 1999.

[4] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Third Annual European Symposium on Algorithms*, 1995.

[5] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. AXM Symp. Computational Geometry*, pages 211–219, 1993.

[6] L. Becker, K. Hinriches, and U. Finke. A new algorithm for computing joins with grid files. In *Proc. Int. Conf. on Data Engineering*, pages 190–197, 1993.

[7] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computer*, 28:643–647, 1979.

[8] T. Brinkhoff, H-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1994.

[9] T. Brinkhoff, H-P. Kriegel, and B. Seeger. Efficient processing of spatial join using R-trees. In *Proc. Int. Conf. on Data Engineering*, 1993.

[10] B. Chazelle and H. Edelsbrunner. An optimal algorithm for itnersecting line segments in plane. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 590–600, 1988.

[11] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Technical report, Department of computer science, university of Illinois, Urbana, IL, 1989.

[12] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete Computational Geometry*, 4:387–421, 1989.

[13] O. Günther. Efficient computation of spatial joins. In *Proc. Int. Conf. on Data Engineering*, pages 50–59, 1993.

[14] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proc. Int. Conf. on Very Large Data Bases*, 1997.

[15] M. Kreveld, editor. *Geographic Information systems*. Utrecht University, 1995.

[16] G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 1999.

[17] R. Laurini and A. D. Thompson, editors. *Fundamentals of Spatial Information Systems*. Acedemic Press, 1992.

[18] M-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[19] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In *Theoretical foundations of computer graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 307–325. Springer-Verlag, 1988.

[20] K. Mulmuley. A fast planar partition algorithm. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 580–589, 1988.

[21] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1986.

[22] J. A. Orenstein. Redundancy in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 294–305, 1989.

[23] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image datebase application. *IEEE Transactions on Software Engineering*, 14(5):611–629, 1988.

[24] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. In *3rd Workshop, Algorithms and data structures*, volume 709, 1993.

[25] J. M. Patel and D. J. DeWitt. Partition based spatial-merge joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[26] D. Rotem. Spatial join indices. In *Proc. Int. Conf. on Data Engineering*, pages 500–509, 1991.

[27] K. C. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *Proc. Int. Conf. on Very Large Data Bases*, 1996.

[28] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 208–215, 1976.

[29] H. Zhu, J. Su, and O. H. Ibarra. An index structure for spatial joins in linear constraint databases. In *Proc. Int. Conf. on Data Engineering*, 1999.

[30] H. Zhu, J. Su, and O.H. Ibarra. Extending rectangle join algorithms for rectilinear polygons. In *Int. Conf. on Web-Age Information Systems*, 2000. to appear.

[31] G. Zimbrao and J. M. Souza. A raster approximation for the processing of spatial joins. In *Proc. Int. Conf. on Very Large Data Bases*, pages 558–569, 1998.