Scalable Sweeping-Based Spatial Join

Lars Arge* Octavian Procopiuc† Sridhar Ramaswamy‡
Torsten Suel § Jeffrey Scott Vitter¶

Abstract

In this paper, we consider the filter step of the spatial join problem, for the case where neither of the inputs are indexed. We present a new algorithm, Scalable Sweeping-Based Spatial Join (SSSJ), that achieves both efficiency on real-life data and robustness against highly skewed and worst-case data sets. The algorithm combines a method with theoretically optimal bounds on I/O transfers based on the recently proposed distribution-sweeping technique with a highly optimized implementation of internal-memory plane-sweeping. We present experimental results based on an efficient implementation of the SSSJ algorithm, and compare it to the state-of-the-art Partition-Based Spatial-Merge (PBSM) algorithm of Patel and DeWitt.

*Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by U.S. Army Research Office grant DAAH04–96–1–0013. Email: large@cs.duke.edu.

†Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047. Email: tavi@cs.duke.edu.

‡ Information Sciences Research Center, Bell Laboratories, 600 Mountain Avenue, Box 636, Murray Hill, NJ 07974–0636. Email: sridhar@research.bell-labs.com.

§ Information Sciences Research Center, Bell Laboratories, 600 Mountain Avenue, Box 636, Murray Hill, NJ 07974–0636. Email: suel@research.bell-labs.com.

¶Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047. Part of this work was done while visiting Bell Laboratories, Murray Hill, NJ. Email: jsv@cs.duke.edu.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

1 Introduction and Motivation

Geographic Information Systems (GIS) have generated enormous interest in the commercial and research database communities over the last decade. Several commercial products that manage spatial data are available. These include ESRI's ARC/INFO [ARC93], InterGraph's MGE [Int97], and Informix [Ube94]. GISs typically store and manage spatial data such as points, lines, poly-lines, polygons, and surfaces. Since the amount of data they manage is quite large, GISs are often disk-based systems.

An extremely important problem on spatial data is the *spatial join*, where two spatial relations are combined together based on some spatial criteria. A typical use for the spatial join is the *map overlay* operation that combines two maps of different types of objects. For example, the query "find all forests in the United States that receive more than 20 inches of average rainfall per year" can be answered by combining the information in a landcover map with the information in a rainfall map.

Spatial objects can be quite large to represent. For example, representing a lake with an island in its middle as a non-convex polygon may require many hundreds, if not thousands, of vertices. Since manipulating such large objects can be cumbersome, it is customary in spatial database systems to *approximate* spatial objects and manipulate the approximations as much as possible. One technique is to bound each spatial object by the smallest axis-parallel rectangle that completely contains it. This rectangle is referred to as the spatial object's *minimum bounding rectangle (MBR)*. Spatial operations can then be performed in two steps [Ore90]:

- **Filter Step:** The spatial operation is performed on the approximate representation, such as the MBR. For example, when joining two spatial relations, the first step is to identify all intersecting pairs of MBRs. When answering a window query, all MBRs that intersect the query window are retrieved.
- **Refinement Step:** The MBRs retrieved in the filter step are validated with the actual spatial objects. In a spatial join, the objects corresponding to each intersecting MBR pair produced by the filter step are checked to see whether they actually intersect.

The filter step of the spatial join has been studied extensively by a number of researchers. In the case where spatial indices have been built on both relations, these indices are commonly used in the implementation of the spatial join. In this paper, we focus on the case in which neither of the inputs to the join is indexed. As discussed in [PD96] such cases arise when the relations to be joined are intermediate results, and in a parallel database environment where inputs are coming in from multiple processors.

1.1 Summary of this Paper

We present a new algorithm for the filter step called *Scalable Sweeping-Based Spatial Join* (SSSJ). The algorithm uses several techniques for I/O-efficient computing recently proposed in computational geometry [APR+98, GTVV93, Arg95, AVV98, Arg97], plus the well-known internal-memory plane-sweeping technique (see, e.g., [PS85]). It achieves theoretically optimal worst-case bounds on both internal computation time and I/O transfers, while also being efficient on the more well-behaved data sets common in practice. We present experimental results based on an efficient implementation of the SSSJ algorithm, and compare it to the original as well as an optimized version of the state-of-the-art Partition-Based Spatial-Merge (PBSM) algorithm of Patel and DeWitt [PD96].

The basic idea in our new algorithm is the following: An initial sorting step is performed along the vertical axis, after which the distribution-sweeping technique is used to partition the input into a number of vertical strips, such that the data in each strip can be efficiently processed by an internal-memory plane-sweeping algorithm. This basic idea of partitioning space such that the data in each partition fits in memory, and then solving the problem in each partition in internal memory, has been used in many algorithms, including, e.g., the PBSM algorithm [PD96].

However, unlike most previous algorithms[1], our algorithm only partitions the data along *one* axis. Another important property of our algorithm is its theoretical optimality, which is based on the fact that, unlike in most other algorithms, no data replication occurs.

A key observation that allowed us to greatly improve the practical performance of our algorithm is that in plane-sweeping algorithms not all input data is needed in main memory at the same time. Typically only the so-called *sweepline structure* needs to fit in memory, that is, the data structure that contains the objects that intersect the current sweepline of the plane-sweeping algorithm. During our initial experiments, we observed that on all our realistic data sets of size N, this sweepline structure never grew beyond size $O(\sqrt{N})$.

This observation, which is known as the square-root rule in the VLSI literature (see, e.g., [GS87]), seems to have been largely overlooked in the spatial join literature (with the exception of [BG92]). It implies that for most real-life data sets, we can bypass the vertical partitioning step in our algorithm and directly perform the plane-sweeping algorithm after the initial sorting step. The result is a conceptually very simple algorithm, which from an I/O perspective just consists of an external sort followed by a single scan through the data. We implemented our algorithm such that partitioning is only done when the sweepline structure does not fit in memory. This assures that it is not only extremely efficient on real-life data, but also offers guaranteed worst-case bounds and predictable behavior on highly skewed and worst-case input data[2].

The overall performance of SSSJ depends on the efficiency of the internal plane-sweeping algorithm that is employed. The same is of course also true for PBSM and other spatial join algorithms that use plane-sweeping at the lower level. This motivated us to perform experiments with a number of different techniques for performing the plane-sweep. By using an efficient partitioning heuristic, we were able to decrease the time spent on performing the internal plane-sweep in PBSM by a factor of 4 as compared to the original implementation of Patel and DeWitt [PD96].

The data we used is a standard benchmark data for the spatial join, namely the Tiger/Line data from the US Bureau of Census [Tig92]. Our experiments showed that SSSJ performs at least 25% better than the original PBSM. On the other hand, the improved version of PBSM actually performed about 10% better than SSSJ on the real-life data we used - we believe that this is due to an inefficiency in our implementation of SSSJ as explained in Section 7. To illustrate the effect of data skew on PBSM and SSSJ, we also ran experiments on synthetic data sets. Here, we observe that SSSJ scales smoothly with data size, while both versions of PBSM are very adversely affected by skew.

Thus, our conclusion is that if we can assume that the data is well-behaved, then a simple sort followed by an optimized plane-sweep provides a solution that is very easy to implement, particularly if we can leverage an existing optimized sort procedure, and that achieves a performance that is at least competitive to that of previous, usually more complicated, spatial join algorithms. If, on the other hand, the data cannot be assumed to be well-behaved, then it appears that PBSM, as well as many other proposed spatial-join algorithms, may not fare much better on such data than the simple plane-sweeping solution, as they are also susceptible to skew in the data[3]. In this case, an algorithm with guaranteed bounds on the worst-case running time, such as SSSJ, appears to be a better choice.

The remainder of this paper is organized as follows. In Section 2 we discuss related research in more detail. In Section 3, we describe a worst-case optimal solution to the spatial-join problem. Section 4 discusses the square-

[1] with the exception of the work in [GS87]

[2] We believe that SSSJ should also scale well with the dimension d, though this remains to be demonstrated experimentally. We would expect the sweepline structure to grow as $N^{(d-1)/d}$, which can exceed the size of main memory even for reasonable values of N.

[3] Most previous papers seem to limit their experiments to well-behaved data.

root rule and its implications for plane sweeping algorithms. Section 5 presents the SSSJ algorithm. In Section 6 we describe and compare different implementations of the internal-memory plane-sweeping algorithm. Section 7 presents our main experimental results comparing SSSJ with PBSM. Finally, Section 8 offers some concluding remarks.

2 Related Research

Recall that the spatial join is usually solved in two steps: a filter step followed by a refinement step. These two steps are largely independent in terms of their implementation, and most papers in the literature concentrate on only the filter step. In this section, we discuss the various approaches that have been proposed to solve the filter step. (For the rest of the paper, we will use the term "spatial join" to refer to the filter step of the spatial join unless explicitly stated otherwise.)

An early algorithm proposed by Orenstein [Ore86, OM88, Ore89] uses a space-filling curve, called Peano curve or z-ordering, to associate each rectangle with a set of small blocks, called *pixels*, on that curve, and then performs a sort-merge join along the curve. The performance of the resulting algorithm is sensitive to the size of the pixels chosen, in that smaller pixels leads to better filtering, but also increase the number of pixels associated with each object.

In another transformational approach [BHF93], the MBRs of spatial objects (which are rectangles in two dimensions) are transformed into points in four dimensions. The resulting points are stored in a multi-attribute data structure such as the grid file [NHS84], which is then used for the filter step.

Rotem [Rot91] proposes a spatial join algorithm based on the join index of Valduriez [Val87]. The join index used in [Rot91] partially computes the result of the spatial join using a grid file.

There has recently been much interest in using spatial index structures like the R-tree [Gut85], R^+-tree [SRF87], R^*-tree [BKSS90], and PMR quadtree [Sam89] to speed up the filter step of the spatial join. Brinkhoff, Kriegel, and Seeger [BKS93] propose a spatial join algorithm based on R^*-trees. Their algorithm is a carefully synchronized depth-first traversal of the two trees to be joined. An improvement of this algorithm was recently reported in [HJR97]. (Another interesting technique for efficiently traversing a multi-dimensional index structure was proposed in [KHT89] in a slightly different context.) Günther [Gün93] studies the tradeoffs between using join indices and spatial indices for the spatial join. Hoel and Samet [HS92] propose the use of PMR quadtrees for the spatial join and compare it against members of the R-tree family.

Lo and Ravishankar [LR94] discuss the case where exactly one of the relations does not have an index. They construct an index for that relation on the fly, by using the index on the other relation as a starting point (the *seed*). Once the index is constructed, the tree join algorithm of [BKS93] is used to perform the actual join.

The other major direction for research on the spatial join has focused on the case where neither of the input relations has an index. Lo and Ravishankar [LR95] propose to first build indices for the relations on the fly using spatial sampling techniques and then use the tree join algorithm of [BKS93] for computing the join. Another recent paper [KS97] proposes an algorithm based on a filter tree structure.

Patel and DeWitt [PD96] and Lo and Ravishankar [LR96] both propose *hash*-based spatial join algorithms that use a spatial partitioning function to subdivide the input, such that each partition fits entirely in memory. Patel and DeWitt then use a plane-sweeping algorithm proposed in [BKS93] to perform the join within each partition, while Lo and Ravishankar use an indexed nested loop join.

Güting and Schilling [GS87] give an interesting discussion of plane-sweeping for computing rectangle intersections, and point out the importance of the "square-root rule" for this problem. They are also the first to consider the effect of I/O from an analytical standpoint; the algorithmic bounds they obtain are slightly suboptimal in the number of I/O operations. Their algorithm was subsequently implemented in the Gral system [BG92], an extensible database system for geometric applications, but we are not aware of any experimental comparison with other approaches.

The work in [GS87, BG92] is probably the previous contribution most closely related to our approach. In particular, the algorithm that is proposed is similar to ours in that it partitions the input along a single axis. However, at each level the input is partitioned into only two strips as opposed to $O(\sqrt{m})$ strips in our algorithm, resulting in an additional factor of $O(\log_2 m)$ in the running time.

3 An Optimal Spatial Join Algorithm

In this section, we describe a spatial join algorithm that is worst-case optimal in terms of the number of I/O transfers. This algorithm will be used as a building block in the SSSJ algorithm described in the next section. We point out that this section is based on the results and theoretical framework developed in [APR+98]. The algorithm uses the distribution-sweeping technique developed in [GTVV93] and further developed in [Arg95, AVV98].

Following Aggarwal and Vitter [AV88] we use the following I/O-model: We make the assumption that each access to disk transmits one disk block with B units of data, and we count this as one I/O operation.[4] We denote the total amount of main memory by M. We assume that we are given two sets $P = \{p_i \mid i \in [N_1]\}$ and

[4] In practice there is, of course, a large difference between the performance of random and sequential I/O. The correct way to interpret the theoretical results of this section in a practical context is to assume that the disk block size is large enough to mask the difference between random and sequential I/O.

$Q = \{q_i \mid i \in [N_2]\}$ of rectangles, where $[\nu]$ denotes the set $\{0, 1, \ldots, \nu - 1\}$. For convenience, we define $N = N_1 + N_2$. We use T to denote the number of pairs of intersecting rectangles reported by the algorithm. We use lower-case notation to denote the size of the corresponding upper-case quantities when measured in terms of the number of disk blocks: $n = N/B$, $t = T/B$, $m = M/B$. We define $\log_m n = \max\{1, \frac{\log n}{\log m}\}$. The efficiency of our algorithms is measured in terms of the number of I/O operations that are performed. All bounds reported in this section are provable worst-case bounds.

We solve the two-dimensional join problem in two steps. We first solve the simpler one-dimensional join problem of reporting all intersections between two sets of intervals. We then use this as a building block in the two-dimensional join, which as based on the distribution sweeping technique.

3.1 One-dimensional Case: Interval Join

In the one-dimensional join problem, each interval $r \in P$ or $r \in Q$ is defined by a lower boundary $r_{\min}$ and an upper boundary $r_{\max}$. The problem is to report all intersections between an interval in P and an interval in Q. We assume that at the beginning of the algorithm, P and Q have already been sorted into one list L of intervals by their lower boundaries, which can be done in $O(n \log_m n)$ I/O operations using, say, the optimal sorting algorithm from [AV88]. We will show that the following algorithm then completes the interval join in $O(n + t)$ I/O operations.

Algorithm *Interval_Join:*

(1) Scan the list in order of increasing lower boundaries, maintaining two initially empty lists L_P and L_Q of "active" intervals from P and Q. More precisely, for every interval r in L, do the following:

   (a) If $r \in P$, then add r to L_P, and scan through the entire list L_Q. If an interval q in L_Q intersects with r, output the intersection and keep q in L_Q; otherwise delete q from L_Q.

   (b) If $r \in Q$, then add r to L_Q, and scan through the entire list L_P. If an interval p in L_P intersects with r, output the intersection and keep p in L_P; otherwise delete p from L_P.

In order to see that this algorithm correctly outputs all intersections exactly once, we observe that pairs $p \in P$ and $q \in Q$ that intersect can be classified into two cases: (i) p begins before q and (ii) q begins before p. (Coincident intervals are easily handled using a tie-breaking strategy.) Step (1)(a) reports all intersections of an interval from P with currently "active" intervals from Q, thus handling case (ii), while step (1)(b) similarly handles case (i).

In order to establish the bound of $O(n+t)$ I/O operations for the algorithm, we need to show that the lists L_P and L_Q can be efficiently maintained in external memory. As it turns out, it suffices if we keep only a single block of each list in main memory. To add an interval to a list, we add it to this block, and write the block out to disk whenever it becomes full. To scan the list for intersections, we just read the entire list and write out again all intervals that are not deleted. We will refer to this simple implementation of a list as an *I/O-list*.

To see that this scheme satisfies the claimed bound, note that each interval in L is added to an I/O-list only once, and that in each subsequent scan of an I/O-list, the interval is either permanently removed, or it produces an intersection with the interval that initiated the scan. Since all output is done in complete blocks, this results in at most $n + t$ reads and $n + t$ writes to maintain the lists L_P and L_Q, plus another t writes to output the result, for a total of $2n + 3t$ I/O operations in the algorithm. Thus, we have the following Lemma.

Lemma 3.1 *Given a list of intervals from P and Q sorted by their lower boundaries, Algorithm* Interval_Join *computes all intersections between intervals from P and Q using $O(n + t)$ I/O transfers.*

3.2 The Two-dimensional Case: Rectangle Join

Recall that in the two-dimensional join problem, each rectangle $r \in P$ or $r \in Q$ is defined by a lower boundary $r^x_{\min}$ and upper boundary $r^x_{\max}$ in the x-axis, and by a lower boundary $r^y_{\min}$ and upper boundary $r^y_{\max}$ in the y-axis. The problem is to report all intersections between a rectangle in P and a rectangle in Q. We will show that the following algorithm performs $O(n \log_m n + t)$ I/O operations, and thus asymptotically matches the lower bound implied by the sorting lower bound of [AV88] (see also [AM]). It can be shown that the algorithm is also optimal in terms of CPU time. We again assume that at the beginning of the algorithm, P and Q have already been sorted into one list L of rectangles by their lower boundaries in the y-axis, which can be done in $O(n \log_m n)$ I/O operations.

Algorithm *Rectangle_Join:*

(1) Partition the two-dimensional space into k vertical strips (not necessarily of equal width) such that at most $2N/k$ rectangles start or end in any strip, for some k to be chosen later (see Figure 1).

(2) A rectangle is called *small* if it is contained in a single strip, and *large* otherwise. Now partition each large rectangle into exactly three pieces, two *end pieces* in the first and last strip that the rectangle intersects with, and one *center piece* in between the end pieces. We then solve the problem in the following two steps:

   (a) First compute all intersections between a center piece from P and a center piece from Q, and all intersections between a center piece from P and a small rectangle from Q, or a center piece from Q and a small rectangle from P.

(b) In each strip, recursively compute all intersections between an end piece or small rectangle from P and an end piece or small rectangle from Q.

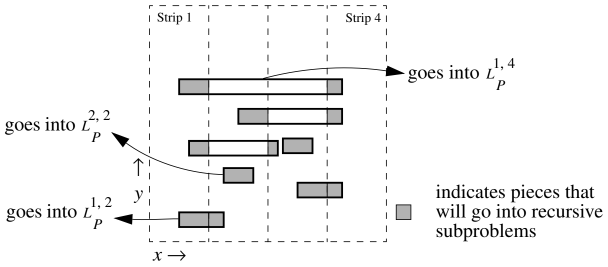

Figure 1: An example of the partitioning used by the two-dimensional join algorithm. Here, $k = 4$, and each strip has no more than three rectangles that will be handled further down in the recursion. (Such rectangles are shown as shaded boxes.) The I/O-lists that rectangles will get added into are shown for some of the rectangles.

We can compute the boundaries of the strips in Step (1) by sorting the x-coordinates of the end points, and then scanning the sorted list. In this case, we have to be careful to split the sorted list into several smaller sorted lists as we recurse, since we cannot afford to sort again in each level of the recursion; the same is also the case for the list L. (In practice, the most efficient way to find the strip boundaries will be based on sampling.) The recursion in Step (2)(b) terminates when the entire subproblem fits into memory, at which point we can use an internal plane-sweeping algorithm to solve the problem. Note that the total number of input rectangles at each level of the recursion is at most $2N$, since every interval that is partitioned can result in at most two end pieces.

What remains is to describe the implementation of Step (2)(a). The problem of computing the intersections involving center pieces in (2)(a) is quite similar to the interval join problem in the previous section. In particular, any center piece can only begin and end at the strip boundaries. This means that a small rectangle r contained in strip i intersects a center piece s going through strip i if and only if the intervals $(r^y_{\min}, r^y_{\max})$ and $(s^y_{\min}, s^y_{\max})$ intersect. Thus, we could compute the desired intersections by running k interval joins along the y-axis, one for each strip.

However, this direct solution does not guarantee the claimed bound, since a center piece spanning a large number of strips would have to participate in each of the corresponding interval joins. We solve this problem by performing all these interval joins in a single scan of the rectangle list L. The key idea is that instead of using two I/O-lists L_P and L_Q, we maintain a total of $(k+2)(k+3)$ I/O-lists $L_P^{i,j}$ and $L_Q^{i,j}$ with $0 \leq i \leq j \leq k+1$ (refer again to Figure 1). The algorithm for Step (2)(a) then proceeds as follows:

Algorithm *Rectangle_Join* **(continued):**

(2a) Scan list L in order of increasing value of r^y_{min}. For every interval r in L do the following:

(i) If $r \in P$ and r is small and contained in strip i, then insert r into $L_P^{i,i}$. Perform a scan of every list $L_Q^{h,j}$ with $h < i < j$, computing intersections and deleting every element of the list that does not intersect with r. Also, write out r lazily to disk for use in the recursive subproblem in strip i.

(ii) If $r \in P$ and r is large and its center piece consists of strips $i, i+1, \ldots, j$, then insert r into $L_P^{i-1,j+1}$. Perform a scan of every list $L_Q^{i',j'}$ with $i < j'$ and $j > i'$, computing intersections and deleting every element of a list that does not intersect with r. Also, write out the end pieces of r lazily to disk for use in the appropriate recursive subproblems.

(iii) If $r \in Q$ and r is small, do (i) with the roles of P and Q reversed.

(iv) If $r \in Q$ and r is large, do (ii) with the roles of P and Q reversed.

Lemma 3.2 *Algorithm* Rectangle_Join *outputs all intersections between rectangles in P and rectangles in Q correctly and only once.*

Proof: (sketch) We first claim that if a rectangle is large, then Step (2)(a) reports all the intersections between the large rectangle's center piece and all other rectangles in the other set. To see this, we classify all the (p, q)-pairs that intersect, and where p is large, into two cases: (α) $q^y_{\min} < p^y_{\min}$ and (β) $p^y_{\min} < q^y_{\min}$. (Equalities are easily handled using a tie-breaking strategy.) Steps (2)(a)(i) and (ii) clearly handle all intersections from case (α) because the currently "active" intervals are stored in the various I/O-lists, and Steps (2)(a)(i) and (ii) intersect p with all of the lists that intersect with its center piece. Steps (2)(a)(iii) and (iv) similarly handle case (β). The case where the interval from q is large follows from symmetry.

In order to avoid reporting an intersection multiple times at different levels of the recursion, we keep track of intervals whose endpoints extend beyond the "current boundaries" of the recursion and store them in separate distinguished I/O-lists. (There are at most $2k$ such lists.) By never comparing elements from distinguished lists of P and Q, we avoid reporting duplicates. □

To show a bound on the number of I/Os used by the algorithm, we observe that as in the interval join algorithm from the previous section, each small rectangle and each center piece is inserted in a list exactly once. A rectangle in a list produces an intersection every time it is scanned, except for the last time, when it is deleted. This analysis requires that each of the roughly k^2 I/O-lists has exclusive use of at least one block of main memory, so the partitioning factor k of the distribution sweep should be chosen to be at most $\sqrt{m}$. Thus, the cost of Step (2)(a) is linear in the input size and the number of intersections produced in this step, and the total cost over the $O(\log_k n = O(\log_m n))$ levels of recursion is $O(n \log_m n + t)$.

Note that the sublists of L created by Step (2)(a) for use in the recursive computations inside the strips are in sorted order. Putting everything together, we have the following theorem.

Theorem 3.3 *Given a list of rectangles from P and Q sorted by their lower boundaries in one axis, Algorithm* Rectangle_Join *reports all intersections between rectangles from P and Q using $O(n \log_m n + t)$ I/O transfers.*

4 Plane Sweeping and the Square-Root Rule

As discussed in the introduction, the overall efficiency of many spatial join algorithms is greatly influenced by the plane-sweeping algorithm that is employed as a subroutine. Several efficient plane-sweeping algorithms for rectangle intersection have been proposed in the computational geometry literature. Of course, these algorithms were designed under the assumption that the data fits completely in main memory. In this section we will show that under certain realistic assumptions about the input data, many of the internal-memory algorithms can in fact be applied to input sets that are much larger than the available memory.

4.1 Plane Sweeping

The rectangle intersection problem can be solved in main memory by applying a technique called *Plane Sweeping*. Plane sweeping is one of the most basic algorithmic paradigms in computational geometry (see, e.g., [PS85]). Simply speaking, a plane-sweeping (or sweepline) algorithm attempts to solve a geometric problem by moving a vertical or horizontal *sweepline* across the scene, processing objects as they are reached by the sweepline. Clearly, for any pair of intersecting rectangles there is a horizontal line that passes through both rectangles. Hence, a plane-sweeping algorithm for rectangle intersection only has to find all intersections between rectangles located on the same sweepline, thus reducing the problem to a (dynamic) one-dimensional interval intersection problem.

A typical plane-sweeping algorithm for rectangle intersection uses a dynamic data structure that allows insertion, deletion, and intersection queries on intervals. Rectangles are inserted into the structure as they are reached by the sweepline, at which time a query for intersections with the new rectangle is performed, and are removed after the sweepline has passed over them. Many optimal and suboptimal dynamic data structures for intervals have been proposed; important examples are the interval tree [Ede83], the priority search tree [McC85], and the segment tree [Ben77].

4.2 The Square-Root Rule

In most implementations of plane-sweeping algorithms, the maximum amount of memory ever needed is determined by the maximum number of rectangles that are intersected by a single horizontal line. For most "real life" input data this number, which we will refer to as the *maximum overlap* of a data set, is actually significantly smaller than the total number of rectangles. This observation has previously been made by other researchers (see [GS87] and the references therein), and is known as the "square-root rule" in the VLSI literature. That is, for a data set of size N, the number of rectangles intersected by any horizontal or vertical line is typically $O(\sqrt{N})$, with a moderate multiplicative constant determined by the data set.

We consider the standard benchmark data for spatial join, namely the US Bureau of the Census TIGER/Line [Tig92] data. These data files consist of polygonal line entities representing physical features such as rivers, roads, railroad lines, etc. We used the data for the states of New Jersey, Rhode Island, Connecticut and New York. Table 1 shows the maximum number of rectangles that are intersected by any horizontal line, for the road and hydrographic features of the four data sets. The number of spatial objects in each data set is given along with the maximum number of the corresponding MBR rectangles that overlap with any horizontal line. Note that the maximum overlap for the TIGER data appears to be consistent with the square-root rule.

Thus, if a data set satisfies the square-root rule, then we can use plane sweeping to solve the spatial join problem on inputs that are much larger than the available amount of memory, as long as the data structure used by the plane sweep never grows beyond the size of the memory.

	Road	Max. Overlap Road	Hydro	Max. Overlap Hydro
Rhode Island	68277	878	7012	54
Connecticut	188642	932	28775	145
New Jersey	414442	1600	50853	156
New York	870412	2649	156567	362

Table 1: Characteristics of road and hydrographic data from TIGER/Line data.

5 Scalable Sweeping-Based Spatial Join

We now describe the SSSJ algorithm, which is obtained by combining the theoretically optimal *Rectangle_Join* algorithm presented in Section 3 with an efficient plane-sweeping technique. The resulting SSSJ performs an initial sort, and then directly attempts to use plane-sweeping to solve the join problem. The vertical partitioning step in *Rectangle_Join* is only performed if the sweeping structure used by the plane-sweep grows beyond the size of the main memory.

Alternatively, we could use random sampling to es-

Scalable Sweeping-Based Spatial Join:

Sort sets P and Q on their lower y-coordinates
Initiate an internal-memory plane-sweep
if the plane-sweep runs out of main memory

perform one level of partitioning using *Rectangle_Join* and recursively call SSSJ on each subproblem

Figure 2: SSSJ algorithm for computing the join between two sets P and Q of rectangles.

timate the overlap of a data set with guaranteed confidence bounds, and then use this information to decide whether the input needs to be partitioned; the details are omitted due to space constraints. In our implementation we followed the slightly less efficient approach describes above. After the initial sort we simply start the internal plane-sweeping algorithm, assuming that the sweepline structure will fit in memory. During the sweep we monitor the size of the structure, and if it reaches a predefined threshold we abort the sweep and call *Rectangle_Join*.

Given the large main memory sizes of current workstations, we expect that in most cases, we can process data sets on the order of several hundred billion rectangles with the internal plane-sweeping algorithm.[5] However, if the data is extremely large, or is highly skewed, then SSSJ will invoke the vertical partitioning at a moderate increase in running time.

In most cases, SSSJ will skip the vertical partitioning, and our spatial join algorithm is reduced to an initial external sorting step, followed by a scan over the sorted data (during which the internal plane-sweeping algorithm is run). We believe that this observation is conceptually important for two reasons. First, it provides a very simple and insightful view of the structure and I/O behavior of our spatial join algorithm. Second, it allows for a simple and fast implementation, by leveraging the performance of the highly tuned sorting routines offered by many database systems. There has been considerable work on optimized database sorts in recent years (see, e.g,, [Aga96, DDC+97, NBC+94]), and it appears wise to try to draw on these results.

6 Fast Plane-Sweeping Methods

As mentioned already, the overall efficiency of many spatial join algorithm is greatly influenced by the internal-memory join algorithm used as a subroutine. In this section we describe and compare several internal-memory plane-sweeping algorithms. We present the algorithms in Subsection 6.1 and report the results of experiments with the TIGER/line data set in Subsection 6.2.

[5] We estimate that the TIGER/Line data for the *entire* United States will be no more than 50 million rectangles.

6.1 Algorithms

Recall from Section 4.1 that the most common plane-sweeping algorithm for the rectangle intersection problem is based on an abstract data structure for storing and querying intervals. We implemented several versions of this data structure. In the following we first give a generic description of the plane-sweeping algorithm, and then describe the different data structure implementations. We also describe the plane-sweeping algorithm used in PBSM, which we also implemented for comparison. This algorithm was proposed in [BKS93], and does not use an interval data structure.

As before, assume that we have two sets $P = \{p_i \mid i \in [N_1]\}$ and $Q = \{q_i \mid i \in [N_2]\}$ of rectangles, sorted in ascending order by their lower boundary in the y-axis. We want to find intersections between P and Q by sweeping the plane with a horizontal sweepline. For a rectangle $r = (r^x_{\mathrm{min}}, r^x_{\mathrm{max}}, r^y_{\mathrm{min}}, r^y_{\mathrm{max}})$, we call $[r^x_{\mathrm{min}}, r^x_{\mathrm{max}}]$ the interval of r, r^y_{min} the starting time of r, and r^y_{max} the expiration time of r. Let D be an instance of the generic data structure $\mathcal{D}$ that supports the following operations on rectangles and their associated intervals:

(1) **Insert**(D, r) inserts a rectangle into D.

(2) **Delete**(D, y) removes from D all rectangles r with expiration time $r^y_{\mathrm{max}} < y$.

(3) **Query**(D, r) reports all rectangles in D whose interval overlaps with that of r.

Figure 3 shows the pseudocode for the resulting algorithm *Sweep_Join_Generic*.

Algorithm *Sweep_Join_Generic:*
$/*$ Head(P) and Head(Q) denote the current first elements in the sorted lists P and Q of rectangles, and D_P and D_Q are two initially empty data structures. $*/$

```
repeat until P and Q are empty
    Let p = Head(P) and q = Head(Q)
    if p^y_min < q^y_min
        Insert(D_P, p)
        Delete(D_Q, p^y_min)
        Query(D_Q, p)
        Remove p from P
    else
        Insert(D_Q, q)
        Delete(D_P, q^y_min)
        Query(D_P, q)
        Remove q from Q
```

Figure 3: Generic plane-sweeping algorithm for computing the join between two sets P and Q of rectangles.

We implemented three different versions of $\mathcal{D}$ and obtained three different versions of the generic algorithm:

Tree_Sweep where $\mathcal{D}$ is implemented as an interval tree data structure, *List_Sweep* where $\mathcal{D}$ is implemented using a single linked list, and *Striped_Sweep* where $\mathcal{D}$ is implemented by partitioning the plane into vertical strips and using a separate linked list for each strip. Finally, we refer to the algorithm used in PBSM as *Forward_Sweep*. In the following we discuss each of these algorithms.

Algorithm Tree_Sweep uses a data structure that is essentially a combination of an interval tree [Ede83] and a skip list [Pug90]. More precisely, we used a simplified dynamic version of the interval tree similar to that described in Section 15.3 and Exercise 15.3–4 of [CLR90], but implemented the structure using a randomized skip list instead of a balanced tree structure. (Another, though somewhat different, structure combining interval trees and skip lists has been described in [Han91].) Our reason for using a skip list is that it allows for a fairly simple but efficient implementation while matching (in a probabilistic sense) the good worst-case behavior of a balanced tree. With this data structure, the expected time for an insertion can be shown to be $O(\log N)$, while query and deletion operations take time $O(T \log N)$, where T is the number of rectangles reported or deleted during the operation. The worst-case running time of this algorithm is $O(N \log N + T \log N)$, which is at most a $\log N$ factor away from the optimal $\Theta(N \log N + T)$ bound. However, on real-life data, the intersection query time is usually $O(\log N + T)$, as most intersecting rectangles are typically close to each other in the tree. Thus, we would not expect significant improvements from more complicated, but asymptotically optimal data structures [McC85, Ede83].

Algorithm List_Sweep uses a simple linked-list data structure. To decrease allocation and other overheads and improve locality, each element of the linked list can hold up to 16 rectangles. Insertion is done in constant time, while query and deletion both take time linear in the number of elements in the list in the worst case. Thus, the worst case running time of the algorithm is $O(N^2)$ ($O(N\sqrt{N})$ if we assume that the square root rule applies).

Algorithm Striped_Sweep uses a data structure based on a simple partitioning heuristic. The basic idea is to divide the domain into a number of vertical strips of equal width and use one instance of the linked list structure from *List_Sweep* in each strip. Intervals are stored in each strip that they intersect. The key parameter in this data structure is the number of strips used. By using s strips, we hope to achieve an improvement of up to a factor of s in query and deletion time. However, if there are too many strips, many intervals will intersect more than one strip, thus increasing the size of the data structure and slowing down the operations. We experimented with a number of values for the number of strips to determine the optimum. Insertions can be done in constant time, while searching and deletion both can be linear in the worst case. However, we expect this algorithm to perform very well for real-life data sets that have many small rectangles and that are not extremely clustered in one area.

Algorithm Forward_Sweep is the plane-sweeping algorithm employed by Patel and DeWitt in PBSM, and was first proposed in [BKS93]. This algorithm is somewhat similar in structure to *List_Sweep*, except that it does not use a linked list data structure to store intervals encountered in the recent past, but scans forward in the sorted lists for intervals that will intersect the current interval in the future; see Figure 4 for the structure of the algorithm. The worst-case running time of this algorithm is $\Theta(N^2)$ ($O(N\sqrt{N})$ if we assume that the square root rule applies).

Algorithm *Forward_Sweep:*
/∗ Head(P) and Head(Q) denote the current first elements in the sorted lists P and Q of rectangles. ∗/

> **repeat** until P and Q are empty
>> Let p = Head(P) and q = Head(Q)
>> **if** $p^y_{\min} < q^y_{\min}$
>>> Remove p from P
>>> Scan Q from the current position and report all rectangles that intersect p. Stop when the current rectangle $r \in Q$ satisfies $r^y_{\min} > p^y_{\max}$.
>>
>> **else**
>>> Remove q from Q
>>> Scan P from the current position and report all rectangles that intersects q. Stop when the current rectangle $r \in P$ satisfies $r^y_{\min} > q^y_{\max}$.

Figure 4: Algorithm *Forward_Sweep* used by PBSM for computing the join between P and Q.

6.2 Experimental Results

In order to compare the four algorithms we conducted experiments joining the road and hydrographic line features from the states of Rhode Island, Connecticut and New Jersey. The input data was already located in main memory at the start of each run. The experiments were conducted on a Sun SparcStation 20 with 32 megabytes of main memory (we were thus unable to fit the New York data into internal memory). To decrease the cost of deletion operations, we introduced a *lazy factor* l and only actually performed the deletion operation every lth time it was called. We chose $l = 10$ in *Tree_Sweep* and *List_Sweep* and $l = 5$ in *Striped_Sweep*. We varied the number of strips in *Striped_Sweep* from 4 to 256.

Table 2 compares the running times of the four plane-sweeping algorithms (excluding the sorting times). We can clearly see that *Striped_Sweep* outperforms the other algorithms by a factor of 4 to 5. The algorithm achieves the best performance for 64 to 128 strips; beyond this

Algorithm	RI	CT	NJ
Tree_Sweep	2,28	8.57	16.65
Forward_Sweep	1.18	12.0	15.8
List_Sweep	1.30	10.19	14.83
Striped_Sweep(4)	0.72	4.02	6.91
Striped_Sweep(8)	0.56	2.72	4.91
Striped_Sweep(16)	0.48	1.97	3.73
Striped_Sweep(32)	0.43	1.57	3.18
Striped_Sweep(64)	0.42	1.39	3.06
Striped_Sweep(128)	0.45	1.36	2.91
Striped_Sweep(256)	0.54	1.61	3.25

Table 2: Performance comparison of the four plane-sweeping algorithms in main memory (times in seconds).

point, the performance slowly degrades due to increased replication. (For $k = 256$, we get a replication rate of more than 40% on the smallest data set, and more than 20% on the other two sets.) Algorithms *Forward_Sweep* and *List_Sweep* are similar in performance, which is to be expected given their similar structure. Maybe a bit surprising is that *List_Sweep* actually outperforms *Forward_Sweep* slightly on the larger data sets, even though it has additional overheads associated with maintaining a data structure. Finally, *Tree_Sweep* is slower than *Forward_Sweep* and *List_Sweep* on small (RI) and thin (NJ) sets, and faster on wide (CT) and large (NY) sets.[6]

We point out that *Tree_Sweep* is the only algorithm that has a good worst-case behavior. While *Striped_Sweep* is the fastest algorithm on the tested data sets, *Tree_Sweep* is useful because it offers reasonably good performance even on very skewed data. As discussed in the next section we therefore decided to use both of them in our practically efficient, yet skew resistant, SSSJ algorithm.

7 Experimental Results

In this section, we compare the performance of the SSSJ and PBSM algorithms. We implemented the SSSJ algorithm along with two versions of the PBSM algorithm, one that follows exactly the description of Patel and DeWitt [PD96] and one that replaces their internal-memory plane-sweeping procedure with *Striped_Sweep*. We refer to the original and improved PBSM as QPBSM and MPBSM, respectively.

We begin by giving a sketch of the PBSM algorithm. We then describe the details of our implementations, and compare the performance of SSSJ against that of QPBSM and MPBSM on TIGER/Line data sets. Finally, we compare the performance of the three algorithms on some artificial worst-case data sets that illustrate the robustness of SSSJ.

[6]The claim for the New York data set was verified with additional runs on a different machine with larger main memory.

7.1 Sketch of the PBSM Algorithm

The filter step of PBSM consists of a decomposition step followed by a plane-sweeping step. In the first step the input is divided into p partitions such that each partition fits in memory. In the second step, each partition is loaded into memory and intersections are reported using an internal-memory plane-sweeping algorithm.

To form the partitions in the first step, a spatial partitioning function is used. More precisely, the input is divided into tiles of some fixed size. The number of tiles is somewhat larger than the number of partitions p. To form a partition, a tile-to-partition mapping scheme is used that combines several tiles into one partition. An input rectangle is placed in each partition it intersects with.

The tile-to-partition mapping scheme is obtained by ordering the tiles in row-major order, and using either round robin or hashing on the tile number. Figure 5 illustrates the round-robin scheme, which was used in our implementations of PBSM. Note that an input rectangle can appear in more than one partition, which makes it very difficult to compute a priori the number of partitions such that each partition fits in internal memory. Instead, an estimation is used that does not take duplication into account.

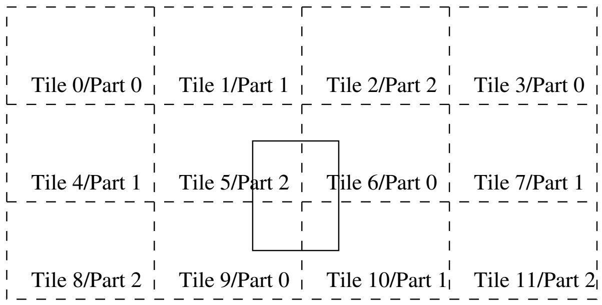

Figure 5: Partitioning with 3 partitions and 12 tiles using the round-robin scheme. The rectangle drawn with a solid line will appear in all three partitions.

7.2 Implementation Details

We implemented the three algorithms using the *Transparent Parallel I/O Programming Environment* (TPIE) system [Ven94, Ven95, VV96] (see also http://www.cs.duke.edu/TPIE/). TPIE is a collection of templated functions and classes to support high-level yet efficient implementations of external-memory algorithms. The basic data structure in TPIE is a *stream*, representing a list of objects of an arbitrary type. The system contains I/O-efficient implementations of algorithms for scanning, merging, distributing, and sorting streams, which are building blocks for our algorithms. This made the implementation relatively easy and facilitated modular design. The input data consists of two streams of rectangles, each rectangle being a structure containing the coordinates of the lower left corner and of the upper right corner, and an ID, for a total of 40 bytes. The output consists of a stream containing the IDs of each pair

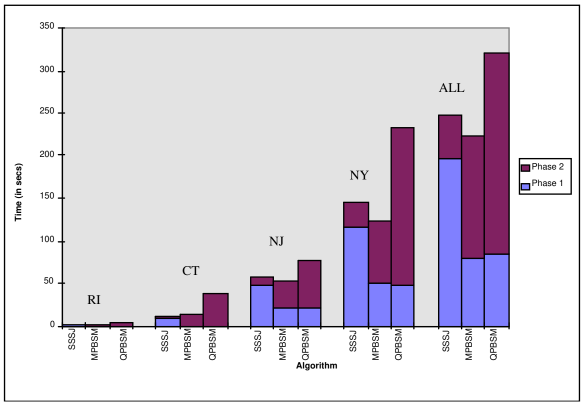

Figure 6: Running times for TIGER data. For MPBSM and QPBSM, Phase 1 consists of the partitioning step, while for SSSJ, it consists of the initial sorting step. The remaining steps are contained in Phase 2.

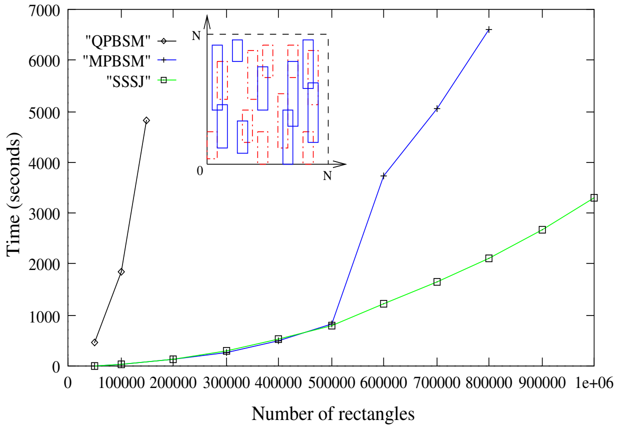

Figure 7: Running times for `tall_rect`

of intersecting rectangles.

As described in Section 6.1, SSSJ first performs a sort. We used TPIE's external-memory merge sorting routine in our implementation. Then SSSJ tries to perform the internal-memory plane-sweeping directly, and reverts to *Rectangle_Join* to partition the data only if the sweep runs out of main memory. We used *Striped_Sweep* as the default internal memory algorithm for the first sweep attempt, and switched to *Tree_Sweep* once partitioning has occurred. In *Rectangle_Join*, we used random sampling to determine the strip boundaries, thus avoiding an additional sort of the data along the x dimension. The I/O lists used by *Rectangle_Join* were implemented as TPIE streams.

The PBSM implementations, QPBSM and MPBSM, consist of three basic steps: A partitioning step, which uses 1024 tiles,[7] and then for each generated partition an internal-memory sorting step and finally a plane-sweeping step. The two PBSM programs differ in the way they perform the plane-sweep: QPBSM uses the *Forward_Sweep* of Patel and DeWitt, while MPBSM uses our faster *Striped_Sweep*.

7.3 Experiments with TIGER Data

We performed our experiments on a Sun SparcStation 20 running Solaris 2.5, with 32 megabytes of internal memory. In order to avoid network activity, we used a local disk for the input files as well as for scratch files. We put no restrictions on the amount of internal memory that QPBSM and MPBSM could use, and thus the virtual-memory system was invoked when needed. However, the amount of internal memory used by SSSJ was limited to 12 megabytes, which was the amount of free memory on the machine when the program was running. For all experiments, the logical block transfer size used by the TPIE streams was set to 48 times the physical disk block size of 4 kilobytes, in order to achieve a high transfer rate.

Figure 6 shows the running times of the three programs on the TIGER data sets. It can be seen that both SSSJ and MPBSM clearly outperform QPBSM: SSSJ performs at least 25% better than QPBSM, while MPBSM performs approximately 10% better than SSSJ. The gain in performance over QPBSM is due to the use of *Striped_Sweep* in SSSJ and MPBSM. As the maximum overlap of the five data sets is relatively small, SSSJ only runs the TPIE external-memory sort and the internal-memory plane-sweep.

From an I/O perspective, the behavior of MPBSM and SSSJ is as follows. SSSJ first performs one scan over the data to produce sorted runs and then another scan to merge the sorted runs (in the TPIE merge sort), before scanning the data again to perform the plane-sweep. MPBSM, on the other hand, first distributes the data to the partitions using one scan, and then performs a second scan over the data that sorts each partition in internal memory and performs the plane-sweep on it. Thus, MPBSM has a slight advantage in our implementation because it makes one less scan of the data on disk.

A more efficient implementation of SSSJ would feed the output of the merge step of the TPIE sort directly into the scan used for the plane-sweep, thus eliminating one write and one read of the entire data. We believe that such an implementation would slightly outperform MPBSM. While conceptually this is a very simple change, it is somewhat more difficult in our setup as it would require us to open up and modify the TPIE merge sort. In general, such a change might make it more difficult to utilize existing, highly optimized external sort procedures.

7.4 Experiments with Synthetic Data

In order to illustrate the effect of skewed data distributions on performance, we compared the behavior of the three algorithms on synthetically generated data sets. We generated two data sets of skewed rectangles, following a procedure used in [Chi95]. Each data set contains two

[7] This is the value used by Patel and DeWitt in their experiments. They also found that increasing the number of tiles had little effect on the overall execution time.

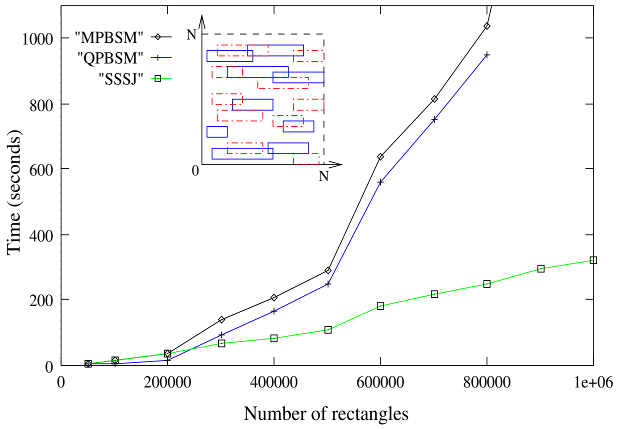

Figure 8: Running times for `wide_rect`

sets of $N/2$ rectangles each, placed in the $[0, N] \times [0, N]$ square. In order to guarantee that the reporting cost does not dominate the searching cost, the rectangles are chosen such that the total number of intersections between rectangles from the two sets is $O(N)$. The first data set, called `tall_rect`, consists of long and skinny vertical rectangles which result in a large maximum overlap. To construct the data, we used a fixed width h for each rectangle ($h = 10$ in the experiments), and chose the height uniformly in $[0, N/2]$. We also chose the x and y coordinates of the lower left corner uniformly in $[0, N - h]$ and $[0, N/2]$, respectively. The second data set, called `wide_rect`, can be obtained by rotating the `tall_rect` data set by 90 degrees. It has the same number of intersections, but a small expected maximum overlap.

Figure 7 shows the running times of the three algorithms on the `tall_rect` data set. The performance of MPBSM degrades quickly, due to the replication of each input rectangle in several partitions. QPBSM performs even worse because of its $O(N^2)$ worst-case plane-sweep: The rectangles are tall, and as we are scanning in the y direction, the time for each query becomes $\Theta(N)$. This problem can be somewhat alleviated if we increase the number of partitions. However, this would further increase replication.

Figure 8 shows the running times on the `wide_rect` data set. In this data set the maximum overlap is small (constant), which is advantageous for all three programs. However, the PBSM implementations still suffer from excessive replication.

8 Conclusions and Open Problems

In this paper, we have proposed a new algorithm for the spatial join problem called Scalable Sweeping-Based Spatial Join (SSSJ), which combines efficiency on realistic data with robustness against highly skewed and worst-case data. We have also studied the performance of several internal-memory plane-sweeping algorithms and their implications for the overall performance of spatial joins.

In our future work, we plan to compare the performance of SSSJ against tree-based methods [BKS93]. We also plan to study the problem of higher-dimensional joins. In particular, three-dimensional joins would be interesting since they arise quite naturally in GIS. Finally, a question left open by our experiments with internal-memory plane-sweeping algorithms is whether there exists a simple algorithm that matches the performance of *Striped_Sweep* but that is less vulnerable to skew.

Acknowledgements

We would like to thank Jignesh Patel for his many clarifications on the implementation details of PBSM.

References

[Aga96] Ramesh C. Agarwal. A super scalar sort algorithm for RISC processors. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 240–246, 1996.

[AM] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry. Manuscript, 1998.

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 685–694, 1998.

[ARC93] ARC/INFO. *Understanding GIS—the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.

[Arg95] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS Technical Report RS-96-28, University of Aarhus.

[Arg97] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, Lecture Notes in Computer Science 1340, 1997.

[AV88] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AVV98] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica (to appear in special issues on Geographical Information Systems)*, 1998. Extended abstract appears in Proc. of Third European Symposium on Algorithms, ESA'95.

[Ben77] J. L. Bentley. Algorithms for Klee's rectangle problems. Dept. of Computer Science, Carnegie Mellon Univ., unpublished notes, 1977.

[BG92] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, 1992.

[BHF93] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *International Conference on Data Engineering*, pages 190–198, 1993. IEEE Computer Society Press.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, 1993.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, 1990.

[Chi95] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[DDC$^+$97] A. C. Arpaci Dusseau, R. H. Arpaci Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 243–254, 1997.

[Ede83] H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.

[GS87] R. H. Güting and W. Schilling. A practical divide-and-conquer algorithm for the rectangle intersection problem. *Information Sciences*, 42:95–112, 1987.

[GTVV93] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[Gün93] O. Günther. Efficient computation of spatial joins. In *International Conference on Data Engineering*, pages 50–60, 1993. IEEE Computer Society Press.

[Gut85] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM-SIGMOD Conf. on Management of Data*, pages 47–57, 1985.

[Han91] E. N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proceedings of Algorithms and Data Structures (WADS '91)*, volume 519 of *LNCS*, pages 153–164. Springer, 1991.

[HJR97] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proc. IEEE International Conf. on Very Large Databases*, pages 396–405, 1997.

[HS92] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large linear segment databases. In *Proc. ACM SIGMOD Conf.*, page 205, 1992.

[Int97] Intergraph Corp. *MGE 7.0,* "http://www.intergraph.com/iss/products/mge/mge-7.0.htm", 1997.

[KHT89] M. Kitsuregawa, L. Harada, and M. Takagi. Join strategies on kd-tree indexed relations. In *International Conference on Data Engineering*, pages 85–93, 1989. IEEE Computer Society Press.

[KS97] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 324–335, 1997.

[LR94] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 209–220, 1994.

[LR95] M.-L. Lo and C. V. Ravishankar. Generating seeded trees from data sets. In *Proc. International Symp. on Large Spatial Databases*, 1995.

[LR96] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 247–258, 1996.

[McC85] E.M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[NBC$^+$94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 233–242, 1994.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984.

[OM88] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611–629, 1988.

[Ore86] J. A. Orenstein. Spatial query processing in an object-oriented database system. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, 1986.

[Ore89] J. A. Orenstein. Redundancy in spatial databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):294–305, June 1989.

[Ore90] J. A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):343–352, June 1990.

[PD96] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 259–270, 1996.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[Rot91] D. Rotem. Spatial join indices. In *International Conference on Data Engineering*, pages 500–509, 1991. IEEE Computer Society Press.

[Sam89] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison Wesley, MA, 1989.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: A dynamic index for multi-dimensional objects. In *Proc. IEEE International Conf. on Very Large Databases*, 1987.

[Tig92] Tiger/line files (tm), 1992 technical documentation. Technical report, U. S. Bureau of the Census.

[Ube94] M. Ubell. The montage extensible datablade architecture. In *Proc. SIGMOD Intl. Conf. on Management of Data*, 1994.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[Ven94] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[Ven95] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1995 with subsequent revisions. Available via WWW at `http://www.cs.duke.edu/TPIE`.

[VV96] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, 1996.