A Unified Approach For Indexed and Non-Indexed Spatial Joins

Lars Arge[1⋆], Octavian Procopiuc[1⋆⋆], Sridhar Ramaswamy[2], Torsten Suel[3], Jan Vahrenhold[4⋆⋆⋆], and Jeffrey Scott Vitter[1⋆⋆]

[1] Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. {large,tavi,jsv}@cs.duke.edu
[2] Epiphany, 2300 Geng Road, Palo Alto, CA. sridhar@epiphany.com
[3] Computer and Information Science, Polytechnic University, 6 MetroTech Center, Brooklyn, NY 11201. suel@photon.poly.edu
[4] Westfälische Wilhelms-Universität, Institut für Informatik, 48149 Münster, Germany. jan@math.uni-muenster.de

Abstract. Most spatial join algorithms either assume the existence of a spatial index structure that is traversed during the join process, or solve the problem by sorting, partitioning, or on-the-fly index construction. In this paper, we develop a simple plane-sweeping algorithm that unifies the index-based and non-index based approaches. This algorithm processes indexed as well as non-indexed inputs, extends naturally to multi-way joins, and can be built easily from a few standard operations. We present the results of a comparative study of the new algorithm with several index-based and non-index based spatial join algorithms. We consider a number of factors, including the relative performance of CPU and disk, the quality of the spatial indexes, and the sizes of the input relations. An important conclusion from our work is that using an index-based approach whenever indexes are available does not always lead to the best execution time, and hence we propose the use of a simple cost model to decide when to follow an index-based approach.

1 Introduction

Geographic Information Systems (GIS) have generated considerable interest in the commercial and research database communities over the last decade. A GIS typically supports the management and manipulation of spatial data types such as points, lines, polylines, polygons, and surfaces. Since the amount of data that needs to be managed is often quite large, GISs are usually disk-based systems.

One of the fundamental operations on spatial data is the *spatial join*, which combines two relations based on some spatial criterium. The most common case is the *spatial overlay join* where the *intersect* predicate is used for joining the input relations. Spatial objects can be quite complex, and their accurate representation can require a large amount of memory. Since manipulating such large objects can be cumbersome and expensive, it is customary to *approximate* spatial objects and manipulate the approximations as much as possible. The most common technique is to bound each spatial object by the smallest

⋆ Supported in part by National Science Foundation grants EIA–9870734 and EIA-9972879.
⋆⋆ Supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9522047 and EIA–9870734.
⋆⋆⋆ This work was done while a Visiting Scholar at Duke University.

axis-parallel rectangle that completely contains it, called the *minimal bounding rectangle* (MBR). Spatial overlay joins can then be performed in two steps [28]:

– **Filter Step:** The spatial operation is first performed on the MBR representation, i.e., the first step is to identify all intersecting pairs of MBRs.
– **Refinement Step:** The exact representations of the objects corresponding to each such pair of MBRs are used to validate the results.

In this paper, we focus on the filter step. (For the rest of the paper, we use the term "spatial join" to refer to the filter step of the spatial join, unless explicitly stated otherwise.) This step has been studied extensively by a number of researchers, and most existing work can be broadly categorized into two approaches. The first approach relies on the existence of a spatial index structure (e.g., an R-tree or R*-tree) that is traversed during the join process. The second approach does not use any existing index structures, but instead uses techniques such as partitioning, sorting, or on-the-fly construction of indexes.

The first contribution of this paper is a simple spatial join algorithm, called *Priority Queue-Driven Traversal* (or *PQ* for short), that combines the index-based and non-index based approaches. The algorithm can be built from a few standard operations, and can process indexed as well as non-indexed inputs. It can also easily be extended to handle certain multi-way joins.

Our second contribution is to show that using an index-based spatial join whenever indexes are available does not always yield the fastest execution time. This is mainly due to the difference between the performance of sequential and random I/O. We propose the use of a cost model that explicitly takes this difference into account when choosing which approach to use.

The third contribution is an extensive comparative study of several index-based and non-index based spatial joins. We present experiments on three different hardware platforms representing the typical range in CPU and disk performance of current workstations. Our experiments use real-world data sets scaling up to tens of millions of spatial objects. Most previous studies consider only input sizes in the tens or at most hundreds of thousands. We consider both the number of disk block accesses and the actual execution time, thus quantifying the effect of random versus sequential disk accesses, as well as the quality of the index structures, on the performance. In contrast, most previous work focuses on only one of the two measures.

The remainder of this paper is organized as follows. In Section 2, we provide a brief summary of related work on spatial joins. In Section 3, we describe the algorithms we considered for the comparative study. Section 4 describes our new algorithm PQ. Section 5 describes our experimental platform, and in Section 6 we present and discuss the experimental results. Finally, Section 7 offers some concluding remarks.

2 Previous Work

Early Work. Orenstein [29] uses a transformational approach based on space-filling curves, and then performs a sort-merge join along the curve to solve the join problem. In another transformational approach [6], the MBRs of two-dimensional spatial objects are transformed into points in four dimensions. These points are stored in a multi-attribute data structure such as the grid file [27], which is then used to perform the join. An efficient algorithm for the rectangle intersection problem based on plane-sweeping was proposed by Güting and Schilling [13], who observed that real data sets from VLSI applications tend to obey a so-called *square-root rule*, i.e., in a set of N rectangles there are only $O(\sqrt{N})$ rectangles that intersect a given vertical or horizontal line. Rotem [32] proposes a spatial join algorithm based on the join index of Valduriez [37] and a grid file.

Spatial Index-Based Approaches. Several join algorithms have been proposed that use spatial index structures such as the R-tree [14], R^{+}-tree [34], R^{*}-tree [7], or PMR quad-tree [33]. Brinkhoff, Kriegel, and Seeger [8] propose an algorithm based on R^{*}-trees that performs a carefully synchronized depth-first traversal of the two trees to be joined. An optimized version of this algorithm was described in [16]. Günther [12] studies the tradeoffs between using join indexes and spatial indexes for the spatial join. He concludes that a join index approach is better for low join selectivities, while for higher join selectivities, spatial indexes perform better. Hoel and Samet [15] propose to use PMR quad-trees for the spatial join and compare it against members of the R-tree family. Lo and Ravishankar [21] discuss the case where only one of the relations has an index. They construct an index for the other relation on the fly, by using the existing index as a starting point (or *seed*). Afterwards, the tree join algorithm of [8] is used to perform the actual join. Another algorithm for the case where only one relation has an index was recently proposed by Mamoulis and Papadias [24], who also discuss how to perform multiple joins occurring in more complex spatial queries.

Non Index-Based Approaches. Recently a lot of work has focused on the case where neither of the input relations has an index. Lo and Ravishankar [22] propose to first build indexes using spatial sampling techniques, and then use the tree join algorithm of [8] to compute the join. Another recent paper [20] proposes an algorithm based on a filter tree structure. Patel and DeWitt [30] and Lo and Ravishankar [23] both propose *hash*-based algorithms that use a spatial partitioning function to subdivide the input such that each partition fits in memory. Patel and DeWitt then use a standard plane-sweeping technique to perform the join for each partition, while Lo and Ravishankar use an indexed nested loop join. Arge et al. [4] propose an algorithm based on plane-sweeping and partitioning along a single axis that guarantees an asymptotically optimal number of disk accesses in the worst case. The algorithm is essentially an improved version of the algorithm of Güting and Schilling [13]. As shown in [4], for common data sets the partitioning steps are never executed, and the algorithm thus reduces to an initial sort followed by a plane sweep.

3 Description of the Algorithms

In this section, we describe the previously known algorithms that we compare in our study. We implemented two non-index based algorithms, the *Partition-based Spatial Merge Join* (*PBSM*) of Patel and DeWitt [30] and the *Scalable Sweeping-based Spatial Join* (*SSSJ*) of Arge et al. [4], as well as an index-based algorithm, the synchronized R-tree traversal (*ST*) of Brinkhoff et al. [8].

3.1 Scalable Sweeping-based Spatial Join (SSSJ)

The Scalable Sweeping-based Spatial Join (*SSSJ*) algorithm [4] combines an optimized internal memory plane-sweep algorithm with a partitioning along a single dimension that makes the algorithm provably robust against worst-case data distributions.

Plane-Sweeping. We first briefly describe the internal memory plane-sweep algorithm, which is used as a component in all four algorithms that we implemented. A plane-sweep algorithm solves a two-dimensional geometric problem by moving a vertical or horizontal *sweep-line* across the data, processing each object as it is reached by the sweep-line (see, e.g., [31]). Clearly, for any pair of intersecting rectangles, there exists a horizontal line that passes through both rectangles, and thus only rectangles located on the same sweep-line (or rather, the intervals corresponding to their projections onto that line) need to be tested for intersection. This observation is used in plane-sweeping to reduce the join problem to a (dynamic) one-dimensional interval intersection problem. Arge et al. [4] experimentally compared four internal memory data structures for storing the intervals corresponding to rectangles cut by the same sweep-line, including two methods called `Striped-Sweep` and `Forward-Sweep`. `Forward-Sweep` has been used in several previous implementations of spatial join algorithms (see, e.g., [8, 30]), while `Striped-Sweep` was shown in [4] to be by a factor of 2 to 5 faster than the other methods for most real-life data sets. We refer to [4] for a detailed discussion of these algorithms and their performance.

Structure of SSSJ. After initially sorting the two sets of input MBR's based on y-coordinates, a plane-sweep is performed by reading both sorted inputs sequentially while maintaining two internal memory interval data structures. This approach works efficiently as long as the data structures do not grow beyond the size of the available internal memory. It was observed by Arge et al. [4] that even for very large real-life data sets, the maximum size of the data structures will be relatively small. To handle cases where the structures do not fit in memory, *SSSJ* combines the plane-sweep approach with an I/O-optimal algorithm based on the distribution sweeping technique [5, 11]. In all experiments performed for this study the data structures were always significantly smaller than the available internal memory, and thus *SSSJ* essentially consists of a sorting step followed by a single scan over the data.

Implementation. Our implementation of SSSJ is the same as that in [4], and is based on external memory multiway mergesort and the internal memory algorithm `Striped-Sweep`. For data sets of the size we used, and excluding the output of the intersections, *SSSJ* performs two sequential read passes, one non-sequential read pass (while merging), and two sequential write passes over the data.

3.2 Partition-based Spatial Merge Join (PBSM)

Partition-based Spatial Merge Join (*PBSM*) [30] is a hash-join algorithm that consists of a partitioning step followed by a plane-sweep step. In the partitioning step the objects from both input sets are distributed to a number of *partitions* such that each partition is likely to fit into internal memory. Then intersections within each partition are computed using the `Forward-Sweep` algorithm.

Since a copy of each input rectangle is assigned to each partition that it intersects, it can be difficult to compute a priori the number p of partitions that are needed. To avoid overfull partitions in the case of clustered data, a larger number $t \gg p$ of *tiles* is created to which the rectangles are distributed. The algorithm then assigns the t tiles to the p partitions by enumerating the tiles in row-major order and applying a hash function (e.g., round-robin); see [30] for details. Excluding the reporting of the intersections, *PBSM* usually performs two sequential read passes and one non-sequential write pass over the data.

Implementation. Our implementation of *PBSM* followed closely that of Patel and DeWitt [30]. Although we estimated the available internal memory very conservatively, we observed several partitions exceeding the internal memory size. Handling these partitions caused a fairly large number of page faults, which slowed down the internal memory part of the algorithm. We were able to alleviate this problem by increasing the number of tiles from 32×32 (as suggested by Patel and DeWitt) to 128×128, and this number of tiles was used throughout all experiments.

3.3 Synchronized R-tree Traversal (ST)

One of the most widely studied spatial join algorithms using R-trees was proposed by Brinkhoff et al. [8], and it has been used as a benchmark in several recent experimental studies [16, 21, 23, 30]. The main idea is to perform a synchronized depth-first traversal of two trees storing the MBR's of the two data sets. For each pair of nodes in the tree whose bounding rectangles intersect, the algorithm computes all pairs of children whose bounding rectangles intersect, and then recurses. Finally, the intersections are reported once the search reaches the leaves.

If the MBR's in the two sets to be joined are distributed in approximately the same way, then all nodes from both trees are involved in the join operation. Assuming that each node occupies exactly one page, the total number of nodes in the R-trees is clearly a lower bound on the number of page requests for dense data sets. We refer to this number as the “optimal” number of page accesses. Recently, Huang, Jing, and Rundensteiner [16] proposed an algorithm based on breadth-first traversal that is reported to take approximately the same amount of CPU time as *ST*, while performing an almost optimal number of I/O operations (if a sufficiently large buffer pool is available).

Implementation. Since ST usually visits R-tree nodes more than once, it benefits from the use of a buffer pool storing previously touched nodes. In other spatial join experiments buffer pools occupying between 0.5 MB and 1 MB [8, 16, 24] or between 2 MB and 24 MB [30] have been used. Having 24 MB of internal memory available (see Section 5.1), we decided to give *ST* as much advantage as possible by using a buffer pool of size 22 MB (the remaining 2 MB were used for internal memory computations). Pages were replaced using the *least recently used* (LRU) policy. Following the recommendations of Brinkhoff et al. [8], we computed intersections between MBR's in two R-tree nodes using the `Forward-Sweep` algorithm, while considering only rectangles overlapping the intersection of the MBR's of the nodes under consideration.

For all R-tree experiments, we used packed R-trees that had been bulk-loaded using the Hilbert heuristic [17]. The maximum fanout was set to 400, corresponding to a page size of 8192 bytes (see Section 5.1). Following recommendations by DeWitt et al. [10], we were careful not to pack all nodes to 100% of capacity, since that might result in too much overlap between bounding rectangles on the same level, and thus decrease the quality of the index. Instead, we filled each node to 75% and included additional rectangles only if they did not increase the area already covered by the node by more than 20%. For our data sets, the resulting trees had an average packing ratio of around 90%.

4 Priority Queue-Driven R-tree Traversal (PQ)

In this section, we describe our new spatial join algorithm called *Priority Queue-Driven Traversal* (*PQ*). The main advantage of this algorithm is that it combines the index-based and non-index-based approaches in a way such that inputs in either representation can be processed using the same algorithm. The algorithm incorporates aspects of the plane-sweep approach of *SSSJ* as well as the tree traversal idea of *ST*.

Structure of PQ. A non-indexed input is processed by *PQ* in essentially the same way as in *SSSJ*; the MBR's are first sorted and then feed into a plane-sweep algorithm. If an input has a spatial index structure, such as an R-tree, the algorithm will exploit this structure and directly extract the data in sorted order according to the direction of the plane-sweep. The extracted data is directly fed into the plane-sweep algorithm. The other input to the sweep can be extracted in the same way from another index structure or read from a sorted non-indexed input.

PQ can be thought of as an extension of *SSSJ* to the case of indexed inputs: it utilizes the same sorting and plane-sweep components as SSSJ, but adds an "index adapter" which extracts data from a spatial index structure in sorted order. In the following we describe how this extraction is performed by means of a tree traversal. A somewhat similar way of traversing the indexed data in sorted order was proposed by Kitsuregawa, Harada, and Takagi [19] in the context of joining two relations indexed by a k-d-tree. Here we present a conceptually simpler algorithm based on a priority queue.

The main idea in our traversal algorithm is to run a horizontal sweep-line through the nodes of the R-tree. To do so, we maintain a priority queue initially containing the bounding rectangle of the root of the tree. We advance the sweep-line by extracting the rectangle with minimum lower y-coordinate from the priority queue. If this rectangle is a bounding rectangle of an internal node of the R-tree, then we load all bounding rectangles of its children from disk and insert them in the priority queue. If the rectangle is a bounding rectangle stored within a leaf (i.e., the MBR of some spatial object), then we feed it into the plane-sweep algorithm, which performs the actual join. Figure 1 shows the basic structure of the algorithm for extracting elements in sorted order.

```
Algorithm Extract_Next_Item:
/* We have a priority queue $P$ for bounding rectangles organized by their lower $y$-coordinate.
Initially, $P$ contains only the bounding rectangle of the root. */

    while $P$ is not empty
        Extract the minimum element $r$ from $P$
        if $r$ is an internal node of the tree
            Read the children of $r$ and insert them into $P$
        else
            Return($r$)
    endwhile
    Return("End of input")
```

Fig. 1. Algorithm for extracting the next item from the index in sorted order.

We point out that while the version of *PQ* described here will always access all nodes of the index structure, the algorithm can be modified so that it only visits those parts that can result in intersections; the details of this (slightly more complicated) version are omitted from this paper. This modification is important for cases where one of the relations is very sparse or localized (see the discussion in Section 6.3), but has no influence on performance for any of the experiments in this paper.

The presentation of *PQ* given here assumes that the priority queue never grows larger than the amount of internal memory available. Note, however, that *PQ* can be modified to handle overflow gracefully by using an external priority queue [2, 9], and that it can also be combined with the partitioning step along one dimension that *SSSJ* performs in the case of an overflow of the interval data structure. We omit these details here since they are only needed for unusual worst-case input distributions.

One key feature of *PQ* is that it touches each node of the R-tree at most once. Thus, the algorithm achieves an "optimal" number of page accesses to the tree (provided that the size of the priority queue never grows beyond the available internal memory). Since *PQ* can process both indexed and non-indexed inputs, it can also easily be extended to multi-way intersection joins. For example, a 3-way intersection join can be performed by feeding the output of a two-way join directly into another join with a third (indexed or non-indexed) input.[1]

Implementation. *PQ* use the same internal memory components as *SSSJ* (see Section 3.1). For the priority queue, we chose the heap-based implementation provided by the *C++ Standard Template Library* (STL) [26]. To optimize the performance and reduce the space requirements of the priority queue, we actually maintained two priority queues: one for the bounding rectangles of the internal nodes and one for the data rectangles in the leaves. Since the only information needed for processing an internal node is its position on disk and the lower y-coordinate of its bounding box, we maintained the internal nodes by storing tuples of the form $(y, \text{page ID})$ in the first queue. For data MBR's however, we need to store four coordinates and an ID. During the algorithm, the next MBR to be processed can be found by comparing the first elements of the two queues.

As the priority queues grow larger, the individual access operations (*add* and *extract_min*) get noticeably slower, even though each operation on a heap storing N objects takes at most $O(\log N)$ comparisons. To increase the performance of the priority queue, we therefore used the following strategy: Whenever we loaded an R-tree leaf from disk, we sorted its rectangles by their lower y-coordinates and inserted only the first rectangle from this sorted sequence into the priority queue. Whenever we extracted a rectangle corresponding to a leaf from the queue, we added the next rectangle from that leaf (if any) to the queue. This technique does not significantly decrease the total space requirement, since all data rectangles of a given leaf have to be loaded into internal memory in order to perform the initial sort. However, by reducing the size of the priority queue, we save $O(\lg B)$ time per priority queue operation.

[1] However, for more complicated multi-way joins which do not correspond to n-way intersections, it is not clear how to extend the algorithm in an elegant fashion; see [25] for a discussion of such cases.

5 Experimental Platforms

In this section, we describe the experimental set-up for our studies, providing detailed information on the hardware, software, and data sets that were used.

5.1 Hardware Platforms

To cover a wide range of CPU speeds and disk transfer rates, we performed experiments on three different system configurations—refer to Table 1. The first system (also used in [4, 16]) is a combination of a relatively slow processor and a fast disk. The second system has a fast processor (slightly faster than the one used in [24]) and a disk with high transfer rate but relatively slow average access time. The third is a state-of-the-art workstation, with both a fast processor and a fast disk. All machines were equipped with 64 MB of internal memory and the amount of free internal memory was at least 24 MB.

	Workstation (Model)	CPU (MHz)	Hard Disk (Model)	Size (GB)	Buffer (KB)	Read (ms)	Throughput (peak, MB/s)
1	SUN Sparc 20	50	ST-32550N (Barracuda)	2.1	512	8.0	10
2	SUN Ultra 10	300	ST-34342A (Medalist)	4.3	128	12.5	33.3
3	DEC Alpha 500	500	ST-34501W (Cheetah)	4.4	512	7.7	40

Table 1. Hardware configurations used in our experiments.

The page size on Machines 2 and 3 was 8 KB, while Machine 1 had a page size of 4 KB. In order to obtain comparable results, we used 8 KB per R-tree node in all experiments. Thus, on Machine 1 we always requested two blocks per I/O-operation.

5.2 Software Environment

We implemented the algorithms in `C++` using the *Transparent Parallel I/O Programming Environment* (TPIE) [3], a templated library that supports high-level, yet efficient implementations of external memory algorithms.

In TPIE, the actual page transfers between disk and internal memory is performed by a so-called Block Transfer Engine (BTE). One implementation employs the `read` and `write` system calls, which improve the performance of purely stream-based algorithms like *PBSM* or *SSSJ*. Hence, we used this BTE in our experiments with these algorithms, and in order to take advantage of the sequential disk access pattern, we used a logical page size of 512 KB. For our R-tree implementation, however, we chose a BTE that performs memory-mapped I/O operations using `mmap` system calls, thus bypassing the operating system's buffer cache in a way similar to using a raw disk device (as done in many commercial database systems [36, p. 535]).

We compiled all programs using the `GNU C++` compiler (version 2.8), with `-O2` level of optimization.

5.3 Data Sets

The TIGER/Line data set from the US Bureau of the Census [35] is one of the standard benchmarks for spatial databases. Its current distribution consists of six CD-ROMs of data. We extracted the hydrographic and road features of the entire United States and created six data sets of different sizes (see Table 2). The two smallest sets consist of the state of New Jersey (NJ) and New York (NY), respectively. These sets were also used in our previous experiments with *PBSM* and *SSSJ* [4]. The data on the first disk (DISK1) covers 15 states from the Eastern US. The data on disks 4-6 (DISK4-6) covers the Western half of the US, while the data on disks 1-3 (DISK1-3) covers the Eastern half. Our largest data set is obtained from all six disks (DISK1-6).

The sizes given in Table 2 refer to files containing the MBRs of each feature. Each MBR occupies 20 bytes (16 bytes for the corner coordinates, 4 bytes for the ID), and each output item is a pair of IDs corresponding to overlapping MBRs.

Category		NJ	NY	DISK1	DISK4-6	DISK1-3	DISK1-6
"Road"	Objects	414,442	870,412	6,030,844	11,888,474	17,199,848	29,088,173
	Data	7.9 MB	16.6 MB	115.0 MB	226.7 MB	328.0 MB	554.8 MB
	R-tree	8.3 MB	17.7 MB	122.8 MB	245.8 MB	352.5 MB	598.4 MB
"Hydro"	Objects	50,853	156,567	1,161,906	3,446,094	3,967,649	7,413,353
	Data	1.0 MB	3.0 MB	22.1 MB	65.7 MB	75.6 MB	141.4 MB
	R-tree	1.1 MB	3.3 MB	25.0 MB	74.6 MB	85.5 MB	160.2 MB
Output	Objects	130,756	421,110	3,197,520	8,554,133	9,378,642	17,938,533
	Data	1.0 MB	3.2 MB	24.4 MB	65.3 MB	71.6 MB	136.9 MB

Table 2. Bounding rectangles of the TIGER/Line 97 data sets.

In addition to the disk space required to hold the original data and the spatial index, we need scratch space for temporary files created during the preprocessing. Since bulk loading an R-tree requires a sorting step, we had to store both the unsorted and the sorted stream of rectangles on the local disk. Together with the spatial index, the overall space requirement was a little more than three times the size of the original data set, and therefore we were unable to construct the R-trees for the largest data sets on Machine 1.

6 Experimental Results

In this section, we present some of the results of an extensive experimental study of the performance of the four algorithms described in the previous sections. We measured I/O cost, internal computation time, and memory requirements. We consider two different measures of the I/O cost: the total number of I/O operations performed, and the actual time taken by the I/O operations. The first set of experiments, discussed in the next section, considers the internal memory requirements of the new *PQ* join algorithm and verifies that its data structures indeed fit in internal memory. In Section 6.2 we compare the performance of the two index-based algorithms (*PQ* and *ST*). Finally, in Section 6.3 we compare the running times of all four algorithms.

6.1 Memory Requirements of PQ

The space requirements of *PQ* on the different data sets are shown in Table 3. The space requirement is measured as the size of the sweep-line data structures plus the size of the priority queues. The latter includes the actual STL priority queues as well as the buffers needed to hold the currently active sorted lists of MBR's as described in Section 4. We see that even though the priority queue is significantly larger than the sweep-line structure, it nevertheless easily fits in memory even on the largest data set. In particular, the size of the priority queue is always less than 1% of the total data set.

Data Structure	NJ	NY	DISK1	DISK4-6	DISK1-3	DISK1-6
Priority Queue	0.32	0.76	1.44	2.72	3.65	4.99
Sweep Structure	0.09	0.10	0.12	0.15	0.17	0.20
Total	0.41	0.86	1.56	2.87	3.82	5.19

Table 3. Maximal memory usage (in MB) for the PQ Join algorithm

6.2 Comparison of Indexed Joins

A common measure for the I/O efficiency of index-based algorithms is the number of pages requested by read or write operations. In this section, we consider the I/O efficiency of the index-based join algorithms *ST* and *PQ* under this measure and compare the results to the actual running times of the algorithms.

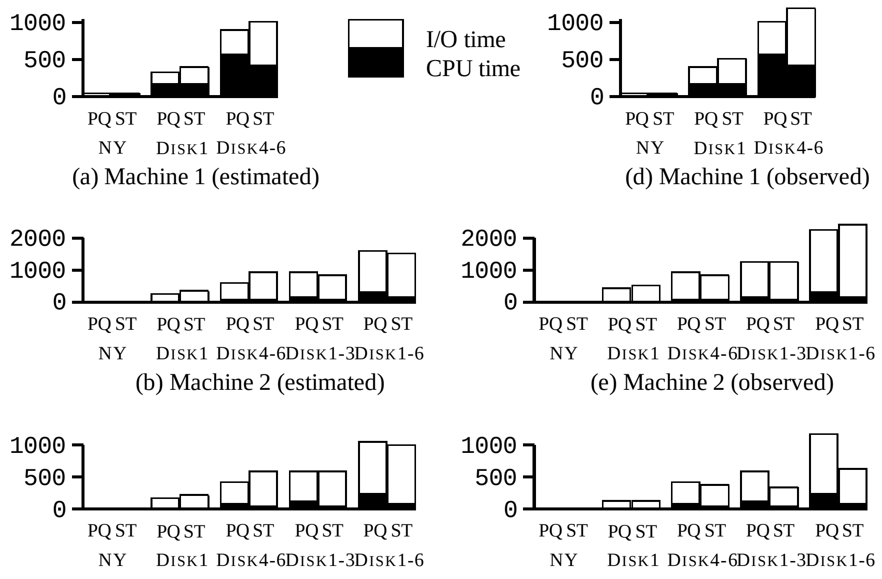
Page Accesses. In Table 4, we show the number of pages requested by *ST* and *PQ*. We give the total number of page requests as well as the average number of requests per R-tree node. These numbers are independent of the machine used, since internal memory and logical page sizes are identical for all machines. The "lower bound" refers to the number of pages occupied by the indexes.

Method	Requests	NJ	NY	DISK1	DISK4-6	DISK1-3	DISK1-6
Lower Bound	Total	1,198	2,706	18,917	41,011	56,061	97,096
	Avg.	1.00	1.00	1.00	1.00	1.00	1.00
PQ Join	Total	1,198	2,706	18,917	41,011	56,061	97,096
	Avg.	1.00	1.00	1.00	1.00	1.00	1.00
ST Join	Total	1,196	2.704	27,001	66,937	63,823	112,323
	Avg.	1.00	1.00	1.43	1.63	1.14	1.16

Table 4. Number of pages requested during joining.

As expected, the number of page requests for *PQ* is optimal. In fact, *PQ* is guaranteed to be optimal as long as its data structures fit into internal memory. The numbers for *ST*, on the other hand, vary quite widely. There are two factors that affect the number of page accesses of *ST*: the heuristic for restricting the search space and the depth first-search traversal mechanism. The first factor decreases the set of pages that need to be touched, while the second results in many pages being requested more than once. For the small data sets (NJ and NY), the entire indexes fit in the buffer pool, so no page needs to be requested more than once *from disk*. Furthermore, we can directly see the positive effect of restricting the search space since the number of pages requested by *ST* is actually slightly less than the "lower bound". As the trees become larger than the buffer pool, the number of page requests increases significantly, with each page being requested between 1.14 and 1.63 times on the average.

Estimated Running Times. The simple method of counting the number of disk accesses has been used in several papers on index-based spatial joins in order to compare the performance of spatial join algorithms; see, e.g., [8, 16, 24]. The estimate for the running time is commonly obtained by multiplying the number of page requests by the average disk block read access time, and then adding the measured internal computation time. In Figure 2(a)–(c) we show the resulting estimated running times for *ST* and *PQ* on all three machines. Here and in the following, we suppress the results for NJ for reasons of readability. The total CPU cost is the sum of the amounts of time spent in *user* and *system* mode as reported by the `getrusage` function call. This time was added to the I/O cost estimated as described above.



(a) Machine 1 (estimated)
(d) Machine 1 (observed)
(b) Machine 2 (estimated)
(e) Machine 2 (observed)
(c) Machine 3 (estimated)
(f) Machine 3 (observed)

Fig. 2. Join costs (in seconds) for all machines: estimated (a)–(c) and observed (d)–(f).

Using the estimated running times, there is no clear winner between the two algorithms. On Machine 1, with the slowest processor and a relatively fast disk, *PQ* seems to have a slight advantage. On the two faster machines, however, *PQ* does significantly more CPU work, while *ST* spends more time on I/O. The higher CPU cost of *PQ* is mainly due to the various internal memory data structures.

Actual Running Times. The above estimates are based on the assumption that page requests are random. While this may be true in some situations (e.g., index structures that are built in an ad-hoc fashion, or database servers that handle multiple interfering requests), it is not clear to what degree it applies to packed or bulk-loaded spatial index structures where neighbors in the index are often located closely together on the disk.

To investigate whether such issues significantly affect performance, we considered the actual running times of *ST* and *PQ*. We made sure that no other processes were running on the machines, and measured the overall running time using the `gettimeofday` function call. The CPU usage time was determined as before, but the I/O cost was now determined by calculating the difference between the overall running time and the CPU usage time. The measured running times of *PQ* and *ST* are shown in Figure 2(d)–(f). Note that they are significantly different from the estimated times in Figure 2(a)–(c). In particular, on Machine 3, which has both the fastest CPU and disk, *ST* is significantly faster than *PQ* on the larger data sets, while the estimated times are nearly equal.

To explain the above behavior, we note that most R-tree bulk-loading algorithms—including the one we used—construct an index structure in a sequential bottom-up fashion that causes all children of a node to be allocated sequentially. Thus, if there is only one process allocating pages, it is most likely that the children will be laid out sequentially on disk, and, in the best case, may even reside on the same track. *ST* traverses the trees in a depth first-search manner, which means that all leaf nodes having the same parent are loaded consecutively. Since the leaves are by far the largest part of the tree, *ST* could perform significant amounts of sequential I/O on the bulk-loaded trees. PQ, on the other hand, basically performs random I/Os that do not depend on the way the tree is layed out on disk. As the sweep-line of *PQ* is advancing, nodes are read more or less randomly from the different parts of the tree that intersect with the sweep-line. This effect becomes more pronounced as the size of the tree increases. However, we expect that the performance of *ST* will degrade on systems running multiple processes with interfering requests, whereas the behavior of *PQ* should be roughly the same. An indication of such an effect can be found in the relative performance of *ST* on Machine 2. The on-disk buffer on this disk is significantly smaller than that of the other machines (128 KB vs. 512 KB), and on this machine we do not observe the same relative advantage of *ST* over *PQ*. Finally, note that the running times on Machine 1 are mainly determined by the internal computation times, since this machine has a fast disk and a fairly slow processor.

6.3 Running Times for all Algorithms

We now compare the measured running times of all four algorithms, the index-based *ST* and *PQ* algorithms and the non-index-based *PBSM* and *SSSJ* algorithms. Since *SSSJ* and *PBSM* access the data in a highly sequential fashion, we did not include them in the comparison between estimated and measured running times. The sequential access should give these algorithms an advantage relative to *PQ* and *ST*. On the other hand, *SSSJ* and *PBSM* access the data multiple times.

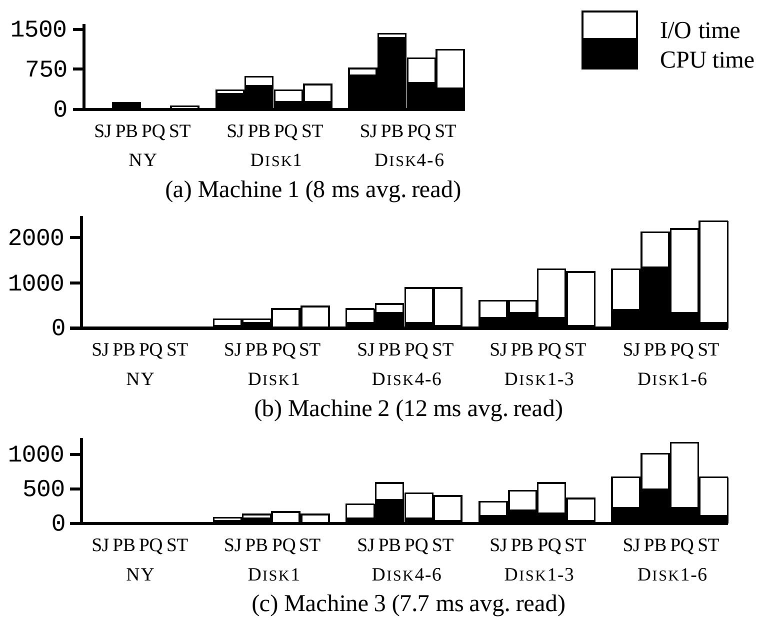


Fig. 3. Observed join costs (in seconds) for all machines.

The results are shown in Figure 3. With the exception of one experiment, *SSSJ* (SJ) always outperforms all other algorithms in terms of total running time even though it performs the largest number of I/Os. This difference in performance can be explained by taking the difference between random and sequential I/O into account. If, based on the disc specifications, we assume that a random read takes on average 10 times as much time as a sequential read and that a sequential write takes on average 1.5 times as much time as a sequential read, *SSSJ* performs the equivalent of $3n + (2n) * 1.5 = 6n$ sequential reads (*SSSJ* performs a total of 3 reads and 2 writes of the entire data), while *PQ* performs the equivalent of $10n$ sequential reads.

Note however, that in some cases index-based algorithms may not have to read the entire input data. This can happen when the join is performed between small localized portions of the input data sets, e.g., when joining hydrographic features from the state of Minnesota and road features of the entire United States. Here, *SSSJ* will still sort both data sets, though only a small clustered portion of the road relation needs to participate in the join. In such a case, index-based algorithms such as *ST* or *PQ*, which only traverse the relevant parts of an index, may be significantly more efficient.

In summary, *PQ* suffers in performance because it naively chooses to use an index whenever it one available. From the above arguments, we can see that, for the given disk configuration, it is advantageous to use the index only when the join involves less than 60% of the leaf nodes. An estimate of this number can be obtained using, e.g., the spatial histograms developed in [1]. Using such a cost-based approach to choose between the index-based and non-index based algorithms, *PQ* should have the best overall execution time in most cases.

We also comment on the relationship between our experiments and a similar set of experiments performed by Patel and DeWitt [30]. Their experiments, which compare *PBSM* and *ST*, were conducted on a machine with a relative CPU/disk performance similar to our Machine 1. Our results match their observation that for such a configuration, the index-based *ST* is faster than the non-index based PBSM (see Figure 3(a)). (On the other hand, if the time spent on constructing the index is taken into consideration, the tree-based join *ST* is slower than *PBSM*.) Furthermore, our experiments extend their results in two ways. First, we demonstrate that the relative performance of spatial join algorithms depends heavily on the size of the data sets and the relative performance of the CPU and the I/O subsystems. Second, we show how index layout on disk can significantly influence the performance. The latter relates to whether one should take index loading time into consideration when comparing index-based spatial join algorithms. As we have seen, *ST* benefits from the layout produced by a good bulk-loading algorithm, and its performance may degrade if the R-tree is updated frequently after bulk loading.[2] Thus, it seems fair to take into account the costs for building or periodic rebuilding. On the other hand, since bulk loading essentially consists of (external) sorting of the data, there would have been no possibility of improving over the sorting based *SSSJ*, unless the cost of building or periodic rebuilding is amortized over several spatial join operations.

[2] Note, however, that Kim and Cha [18] have recently described how to locally reorganize the tree during updates to maintain a good layout of sibling nodes.

7 Conclusions and Open Problems

In this paper, we presented a simple algorithm that unifies the index based and non-index based spatial join approaches. Under reasonable assumptions about the input data, our algorithm is guaranteed to perform an optimal number of I/O operations. We also presented the results of an extensive set of experiments on real-life data that shows that it is important to take into account the difference between sequential and random I/O when designing spatial join algorithms for massive data sets.

The performance of the index-based algorithms depends heavily on the properties of the spatial index structure. Not surprisingly, tightly packed space-efficient index structures perform better than structures that achieve a lower space utilization or that do not map adjacent leaves of the tree to consecutive locations on disk. It remains an open problem to incorporate these properties of bulk-loaded index structures into testbeds and performance models for spatial join algorithms.

Acknowledgments

We thank Jignesh Patel for clarifications on implementation details of PBSM.

References

1. S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 13–24, 1999.
2. 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.
3. L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickeremesinghe. *TPIE User Manual and Reference (edition 0.9.01a).* Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.
4. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. Intl. Conf. on Very Large Databases*, pages 570–581, 1998.
5. 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 *ACM-SIAM Symp. on Discrete Algorithms*, pages 685–694, 1998.
6. L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 190–197, 1993.
7. 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*, pages 322–331, 1990.
8. 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*, pages 237–246, 1993.
9. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.
10. D. J. DeWitt, N. Kabra, J. M. Patel, and J.-B. Yu. Client-server Paradise. In *Proc. Intl. Conf. on Very Large Databases*, pages 558–569, 1994.
11. M. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 714–723, 1993.
12. O. Günther. Efficient computation of spatial joins. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 50–59, 1993.
13. R. H. Güting and W. Schilling. A practical divide-and conquer algorithm for the rectangle intersection problem. *Information Sciences*, 42:95–112, 1987.

14. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.
15. E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large linear segment databases. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 205–214, 1992.
16. Y.-W. Huang, N. Jing, and E. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proc. Intl. Conf. on Very Large Databases*, pages 396–405, 1997.
17. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 47–57, 1993.
18. K. Kim and S. K. Cha. Sibling clustering of tree-based spatial indexes for efficient spatial query processing. In *Proc. ACM Intl. Conf. Information and Knowledge Management*, pages 398–405, 1998.
19. M. Kitsuregawa, L. Harada, and M. Takagi. Join strategies on KD-tree indexed relations. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 85–93, 1989.
20. N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 324–335, 1996.
21. 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.
22. M.-L. Lo and C. V. Ravishankar. Generating seeded trees from data sets. In *Proc. Intl. Symp. on Spatial Databases, LNCS 951*, pages 328–347, 1995.
23. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 247–258, 1996.
24. N. Mamoulis and D. Papadias. Integration of spatial join algorithms for joining multiple inputs. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 1–12, 1999.
25. N. Mamoulis, D. Papadias, and Y. Theodoridis. Processing and optimization of multiway spatial joins using R-trees. In *Proc. Symp. on Principles of Database Systems*, pages 44–55, 1999.
26. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
27. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.
28. J. A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 343–352, 1990.
29. J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Engineering*, 14(5):611–629, May 1988.
30. 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.
31. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, Berlin, 2nd edition, 1988.
32. D. Rotem. Spatial join indices. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 500–509, 1991.
33. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
34. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. Intl. Conf. on Very Large Databases*, pages 507–518, 1987.
35. *TIGER/Line*$^{\text{TM}}$ *Files, 1997 Technical Documentation*. Washington, DC, September 1998. `http://www.census.gov/geo/tiger/TIGER97D.pdf`.
36. U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.
37. P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.