THE SPATIAL FILTER REVISITED

Walid G Aref and Hanan Samet

Walid G. Aref
Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
Two Research Way
Princeton, New Jersey 08540
aref@mitl.research.panasonic.com
Phone: (609) 734-7349 Ext. 313
Fax: (609) 987-8827

Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742
hjs@cs.umd.edu
Phone: (301) 405-1755
Fax: (301) 314-9115

ABSTRACT

Database systems make use of a wide variety of spatial access method for efficient storage and indexing of spatial data. Access methods based on spatial occupancy such as the quadtree and the Z-Order are well-suited for a wide collection of data intensive applications (e.g., medical imagery and geographic information systems). In this environment, the spatial join is a fundamental operation for answering queries that involve spatial predicates. Two new algorithms for performing spatial join are presented where they are extensions of two well-known algorithms for performing a spatial join. The first algorithm applies an optimization technique that takes place in the case when either or both input streams of the spatial join are sorted but not necessarily indexed. This case is of practical use in query processing especially when a query is answered by a cascade of spatial join operations.The performance of this algorithm is superior to that of published algorithms. The second algorithm is a further optimization that takes place when both input streams of the spatial join are indexed. It uses on-line estimates of the input streams to achieve better performance.

Keywords: spatial databases, database systems, data structures, design of algorithms, spatial join

1 INTRODUCTION

Representations of spatial data that are based on occupancy are very suitable for a wide variety of data intensive applications (e.g., medical imagery and geographical information systems). Examples of data structures that make use of this representation are the region quadtree [8,16], the bintree [9], and the Z-Order [14]. In these data structures, a spatial object is represented by its internal region. Several researchers have investigated the usage of these structures inside a database environment (e.g., PROBE [13] and SAND [1]). As pointed out in [5], it is straightforward to implement these data structures in a database system because they require common facilities that are already present in almost all database systems (mainly any access method that provides both sequential and direct access, e.g., the B-Tree).

One of the most useful tools for spatial query processing is the *spatial join*. Generally speaking, the spatial join combines entities from two sets into single entities whenever the combination satisfies the spatial join predicate (e.g., if the two entities are within *n* miles from each other). As pointed out in [4], both the CPU and the disk read costs of the spatial join operation are very significant. As a result, extensive research has been conducted on alternative ways of processing the spatial join efficiently (e.g., see [2,3,4,6,13,15]). Becker [3], and Becker, Hinrichs, and Finke [2] propose an algorithm for the efficient evaluation of spatial join for databases of multidimensional point objects. Gunther [6] presents a hierarchical spatial join algorithm applicable to a family of tree-based data structures, termed the generalization tree. Brinkhoff, Kriegel, and Seeger [4] apply a similar idea in the context of the R-tree [7]. In addition, they present several techniques that reduce both the disk read and CPU costs of the spatial join significantly. Rotem [15], and later, Lu and Han [10], suggest precomputing the spatial object pairs satisfying a certain spatial relationship and storing them in *spatial join indices* in order to speed up the spatial join at query runtime. Orenstein and Manola [13] present two algorithms for spatial join where the underlying representation of spatial data is the Z-Order [14]. Since this paper is based on the work of Orenstein and Manola, we include a brief description of their work.

In [13,14], each object is represented by a set of rectangular elements (termed Z-elements) that collectively approximate (and cover) the object. Z-elements are ordered by their key value which is a function of two parameters: the coordinate values of the upper-left corner and the size of the rectangular region corresponding to the Z-element. In addition, each Z-element has an object identifier that indicates the object to which it belongs. In the described implementation, objects are permitted to overlap. As a result, Z-elements from different objects may also overlap. For more details on how to construct the Z-Order from a collection of objects, see [13].

Figure 1 illustrates an example of the spatial join operation. Notice that because of the nature of Z-elements and the way they are constructed, any pair of Z-elements are either equal, disjoint, or are contained in one another but cannot overlap (without containment).

The first algorithm presented in [13] for performing spatial join (which we term *spatial merge*) resembles a merge join. It requires that the two input streams (i.e., the two Z-Order streams) be sorted but does not necessarily require the presence of an index. The second algorithm (which we term *spatial join*) is an optimization of the spatial merge algorithm. It requires that both input streams be indexed (so that random as well as sequential access to the elements in either stream is possible). This is a very modest requirement given the status of current database technology.

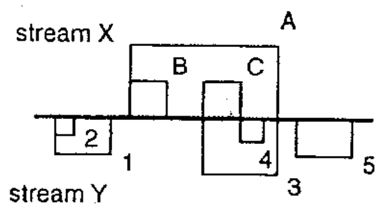


Figure 1: An example a spatial join of two input streams. Stream X is displayed above the horizontal line while stream Y is displayed below the horizontal line. The result of the join is the set of intersecting block pairs (A,3), (A,4), and (C,3).

In this paper, we present two further optimizations over the above algorithms. The first optimization (which we term the *linear-scan algorithm*) is an enhancement of the first algorithm of [13] (the spatial merge). Similar to the spatial merge, the linear-scan algorithm requires that the two input streams be sorted but does not necessarily require the presence of an index. However, the linear-scan algorithm simulates the effect of an index in order to skip the elements of the input streams that do not contribute to the result of the spatial join and hence reduces the amount of processing induced by the spatial merge. It is important to mention that when we process a query that involves a cascade of spatial join operations, only the first spatial join in the pipeline makes use of the spatial index. The subsequent spatial joins in the pipeline access the

resulting data elements from the first spatial join. The resulting stream of elements is only sorted but is not indexed. In this case, versions of the spatial join algorithms that do not require an index can be useful for the subsequent spatial joins in the pipeline. This gives rise to the importance and applicability of our new algorithm (the linear-scan algorithm).

The second algorithm (which we term the *estimate-based algorithm*) is a further enhancement of the second algorithm of [13] (the spatial filter). Similar to the spatial filter, the estimate-based algorithm requires that both input streams of the spatial join be indexed. Furthermore, the estimate-based algorithm makes use of a minimal preprocessing stage of the underlying spatial database so that it can achieve performance gains at run-time. At each stage of execution, the algorithm computes on-line estimates of the input streams using the data gathered at preprocessing time. The estimates are used as a guide in deciding, in real-time, on how the algorithm proceeds in deciding whether or not to use the index for a given direct access request.

The rest of the paper proceeds as follows. Section 2 describes briefly the two spatial join algorithms in [13]. Section 3 presents the new algorithm for performing spatial join when either one or both input streams are sorted but not indexed. Section 4 presents our second algorithm which assumes that both input streams are indexed. Section 5 contains the results and discussion of our experiments. Section 6 contains concluding remarks.

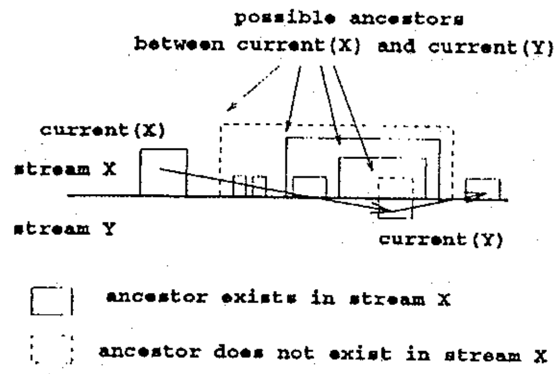
2 BACKGROUND: AN OVERVIEW OF THE SPATIAL JOIN ALGORITHMS

Z-elements can be either square or rectangular blocks [14]. For simplicity, we restrict our presentation to square blocks. We use the term Morton block [11] to distinguish it from a Z-element. However, the techniques presented here apply directly to Z-elements which can assume rectangular shapes. Furthermore, we limit our discussion to a two-dimensional space. All the concepts extend readily to higher dimensions. We use the following terminology: a two-dimensional Morton code [11] is the result of mapping a two-dimensional point into a one-dimensional point by interleaving the bits that represent the values of the coordinates of the two-dimensional point. For example, the Morton code of the point $(10,01)_{base\ 2}$ is $0110_{base\ 2}$. A two-dimensional Morton block is a maximal square block that results from a regular decomposition of space into homogeneous regions (i.e., a quadtree decomposition). A Morton block, say *B*, is represented by the Morton code of *B*'s upper-left corner and its size. We assume that the origin is at the upper-left corner of the space, that the positive *x* direction is to the right, and that the positive y direction is downwards. The key for sorting in each input stream is the Morton code (in ascending order) and size (in descending order) of each block in the stream. The order implied is such that a Morton block *B* appears in the stream before Morton block *C iff* the Morton code of *B* is less than the Morton code of *C*. Furthermore, if two Morton blocks share the same upper-left corner, then the larger one appears first in the stream.

In [13], Orenstein and Manola present two variants of the spatial join algorithm. The first algorithm (which we term *spatial merge*) requires that the two input streams be sorted but not necessarily indexed. The second algorithm (which we term *the spatial filter*) requires that both input streams be indexed (so that random as well as sequential access to the elements in either stream is possible). Below, we briefly describe each of the two algorithms. We present a quadtree variant of the algorithms as we assume that elements are always square blocks (in contrast to rectangular blocks as in the case of the Z-Order [14]).

The spatial merge algorithm resembles the merge join algorithm. However, the spatial merge algorithm is slightly unconventional because input elements of the same stream represent two-dimensional intervals that can be contained in one another. The state of the algorithm is maintained by using a stack for each input stream, the current element in each stream, and a cursor indicating the next element in each stream. The top of each stack contains the current element for the corresponding stream, and the containing elements (from the same sequence) are stored deeper in the stack. A detailed explanation of the spatial merge algorithm can be found in [13].

The second algorithm of Orenstein and Manola (the spatial filter) for performing spatial join assumes that both input streams are sorted and indexed. The underlying index permits both sequential and direct access of elements in the index. The algorithm behaves exactly like the spatial merge algorithm except when advancing to the next element in the stream. In particular, all the changes are encapsulated inside one routine, namely routine Advance, which in the case of the spatial merge algorithm advances to the next element in either one of the two input streams. We term the second version of this routine SpatialFilterAdvance. Instead of advancing to the next element in stream X, SpatialFilterAdvance(X,Y) uses information from the stream Y to directly access stream X, using the index on X, to skip elements in stream X that are not relevant to the spatial join operation (i.e., ones that do not overlap with any elements in stream Y and therefore cannot contribute to the result of spatial join). Figure 2 illustrates this process. Given the current elements of each stream, say current(X) and current(Y), the algorithm


possible ancestors
between current(X) and current(Y)
current(X)
stream X
stream Y
current(Y)
ancestor exists in stream X
ancestor does not exist in stream X
Figure 2: Possible parents between current(X) and current(Y).

uses current(Y) to directly access stream X thereby skipping the unneeded elements that lie between current(X) and current(Y) that do not contribute to the result. One problem that the algorithm has to resolve is that there may exist some blocks in stream X that start past current(X) and that contain current(Y) (and hence contribute to the result of the join) that may be skipped if the algorithm uses only current(Y) to directly access the elements in stream X. In order to overcome this problem, the spatial filter algorithm generates all the possible ancestors (i.e., containing blocks) of current(Y) that lie between current(X) and current(Y). Not all the possible ancestors of current(Y) exists in the stream X. In fact, it could be that none of them may exist. For example, in Figure 3 four possible parents of current(Y) are computed but only two of them really exist in the stream. The algorithm needs to advance the cursor of stream X so that it points to the largest ancestor of current(Y) that exists in stream X. If such an ancestor exists, then routine SpatialFilterAdvance assigns to current(X) the value of this ancestor and the spatial join algorithm proceeds from there. After computing the possible ancestors of current(Y), the algorithm directly accesses stream X searching for one ancestor of current(Y) at a time. This requires a number of direct accesses to stream X equal to the number of computed ancestors between zhi(current(X)) and zlo(current(X)) in the worst case. The algorithm returns after finding the first largest ancestor. For example, in Figure 3 the algorithm returns after directly accessing the second largest parent of current(Y) that exists in stream X as the largest possible parent was directly accessed and was not found. If none of the parents is found, then the algorithm assigns to current(X) the smallest element past current(Y) in stream X. Obviously, the algorithm requires that the input streams be indexed so that both sequential and random access are possible. A detailed explanation of the algorithm can be found in [13].

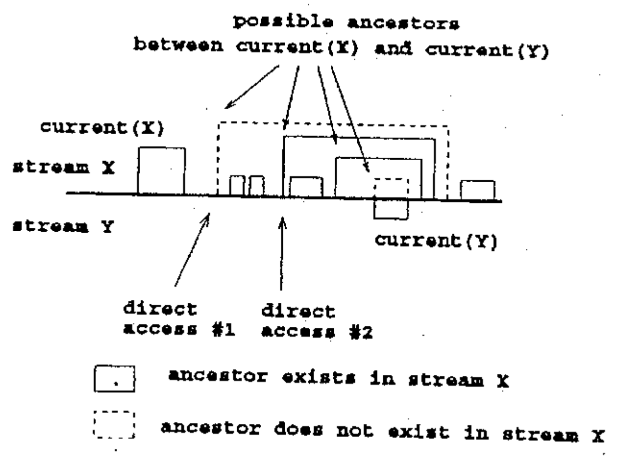

possible ancestors between current(X) and current(Y)
current(X)
stream X
stream Y
current(Y)
direct access #1
direct access #2
ancestor exists in stream X
ancestor does not exist in stream X

Figure 3: Possible parents between current(X) and current(Y).

3 A NEW ALGORITHM FOR NON-INDEXED SORTED STREAMS

A pipelined architecture for processing queries is commonly used in database systems. A query can be answered by a pipeline that is composed of a cascade of one or more operations. The first operation in the pipeline operates on the original input streams and its result is passed to the second operation in the pipeline using common buffers. The major problem in such an architecture is that the underlying indexing capabilities are effective only for the first operations in the pipeline. Later operations in the pipeline have to operate on non-indexed spatial data unless temporary indexes are built. In other words, if the input stream is indexed, then only the first operations in the pipeline can benefit from the index. The same problem arises when operating on spatial data.

Many queries with spatial predicates can be answered using a cascade of spatial join operations. The result of the first spatial join operation is passed to the next spatial join in the pipeline, and so on. Only the first spatial join in the pipeline can take advantage of the fact that the input streams are indexed, while, the next spatial joins in the pipeline will have to operate on non-indexed data. In order to apply Orenstein and Manola's optimizations for spatial join (i.e., in order to use the spatial filter algorithm rather than the spatial merge algorithm), the input streams have to be indexed. If we want to apply the spatial

filter for each spatial join in the pipeline, then the query processor has to build a temporary index from the results of each spatial join.

Building temporary spatial indexes during query processing has its tradeoffs. On the one hand, building temporary indexes to organize intermediate results helps speed-up the execution of the spatial operations since they operate on structured data (inside the temporary index) rather than on an unorganized collection of data items. On the other hand, the space and time costs of building the temporary index may exceed its benefits.

As a compromise between the two approaches (building a temporary index and applying the optimized spatial join algorithm vs. applying the non-optimized spatial join algorithm, i.e., the spatial merge algorithm, directly on the non-indexed data), we propose a semi-optimized spatial join algorithm, which we term *linear-scan spatial join*, that operates on sorted but non-indexed streams. This algorithm fits very well in the pipeline as a later operation since it does not require an index. However, its performance is much better than the spatial merge algorithm.

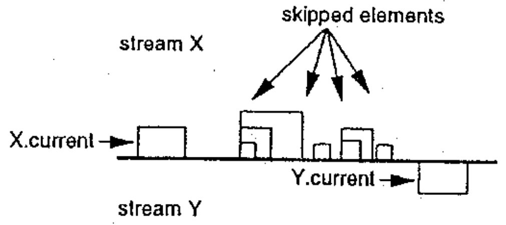


Figure 4: Example illustrating the appropriateness of random access to the B-tree since it results in skipping many elements that do not contribute to the result of the spatial join.

The idea is simple. We would like to simulate the spatial filter algorithm (the indexed version of spatial join) although we do not have an index. More specifically, we want to avoid the random access of nodes in the stream, thus relaxing the requirement of indexing on the part of input streams. The new algorithm behaves almost exactly like the spatial filter. The only difference is in routine Advance. We term it LinearScanAdvance.

LinearScanAdvance(X,Y) uses current(Y) as a guide to advance the cursor in the stream X. First, current(X) is advanced to the next element in stream X. If current(Y) happens to be past current(X), then we can advance stream X so that current(X) points to the first element that contains current(Y). If such an element does not exist, then current(X) is set to the first element in stream X that is greater than or equal to current(Y). Notice that this resembles exactly what algorithm indexed-spatial join (i.e., the spatial filter) does except in one point. Once routine LinearScanAdvance decides that it wants to directly access an ancestor, it just performs a linear scan until it reaches that ancestor (instead of directly accessing the ancestor through an index).

The number of disk page reads for this algorithm is the same as the one for the spatial merge (i.e., the non-indexed version). The performance gain results from the fact that the spatial merge will have to process each of these elements through its main loop, which is CPU intensive, while they do not contribute to any output tuples of the join. Alternatively, our version detects these elements and excludes them by sequentially scanning and skipping all of them so that they will not get processed by the main loop of the algorithm. The performance gain from this simple optimization is substantial. Experimental results are shown in Section 5. The listing of the algorithm is omitted for brevity.

4 FURTHER OPTIMIZATION OVER THE SPATIAL FILTER ALGORITHM

The linear-scan algorithm enhances over spatial merge. It is interesting to know if the spatial filter can be improved upon. The main advantage of the spatial filter is its ability to skip the elements in the input stream that do not contribute to the join result (e.g., Figure 4). On the other hand, the linear-scan must examine elements that do not contribute to the final join result, although it need not process them as is the case for the spatial merge. In contrast, the main drawback of the spatial filter is that it must perform a direct access to the index on stream *i* for each of the possible ancestors of the current element of stream *j* that it

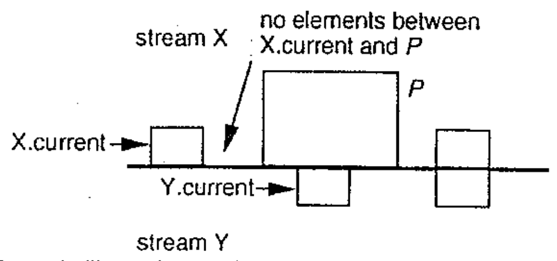


Figure 5: Example illustrating the inappropriateness of a random access to the B-tree since *P* is very close to current(X). A linear scan from current(X) to *P* will result in a lower execution time.

examines even if an ancestor *P* of the current element of stream *j* is very close to the current element of stream *i* (e.g., P in Figure 5 for streams X and Y corresponding to streams *i* and *j*, respectively). In such a case, a linear scan from the current element of stream *i* to *P* would be faster than one (or possibly more if other larger ancestors were possible) direct access operations to *P* using the index.

This suggests that an ideal solution would be one that combines the linear-scan and spatial filter algorithms by selecting the appropriate one on the basis of an ``estimate" of the closeness of the "next" relevant element in stream *i*. Such an algorithm would perform a direct access for the case in Figure 4 and a linear scan for the case in Figure 5. The result is termed an *estimate-based* spatial join algorithm. Both input streams must be indexed in order for this optimization to be applicable.

The key issue in using an estimate-based spatial join algorithm is what is a proper estimate? Such an estimate must be simple and fast to compute. Our estimate does not take the disk read cost into account directly -- that is, it is only in terms of the CPU cost. However, we can argue that our CPU estimate is also proportional to and reflects the disk read cost. A detailed analysis is omitted due to lack of space. Let N_{ij} be the number of elements in stream *i* that lie between the current elements of streams *i* and *j*. N_{ij} is an attainable upper bound for the number of elements that must be examined during a linear scan in the search for the ancestors of the current element of stream *j*. Therefore, the average number of elements that are examined is $\lceil N_{ij}/2 \rceil$.

The cost of directly accessing the index depends on the nature of the index. Assume that the index is a B-tree of depth *d* with a maximum of N_B elements per B-tree node. The CPU cost results from descending *d* B-tree nodes, and at each node performing a binary search to locate which pointer to use in order to descend to the next level. This results in $\log_2 N_B$ element comparisons per node and a total of $d \log_2 N_B$ element comparisons in order to directly access one ancestor. If the algorithm computes *p* ancestors between the current elements of streams *i* and *j*, then, on the average, it will find the first largest ancestor in stream *i* after $\lceil p/2 \rceil$ direct accesses to the B-tree. Therefore, the average total CPU cost for locating the first largest ancestor using direct access is $\lceil dp/2 \rceil \log_2 N_B$ element comparisons. Thus a reasonable initial estimate is one that uses the linear-scan algorithm if $\lceil dp/2 \rceil \log_2 N_B$ is greater than $\lceil N_{ij}/2 \rceil$. The value of *p* is computed while generating the possible ancestors of the current element of stream *j*, while *d* and N_B are system parameters frequently known in advance.

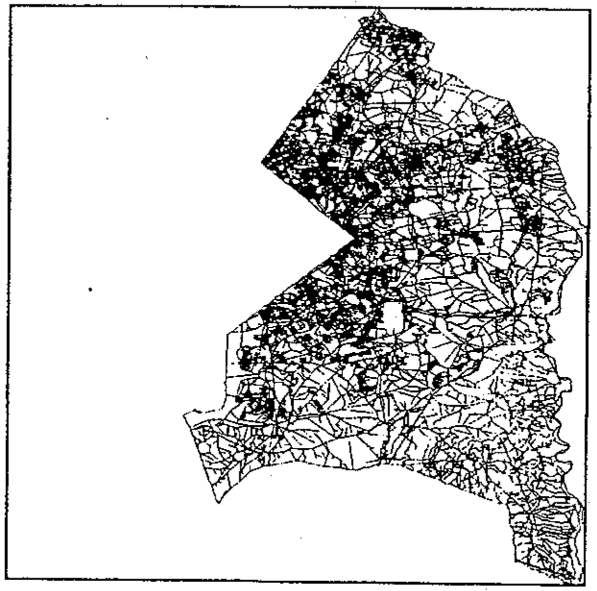


Figure 6: Part of a map sheet of Prince George's county in Maryland.

One approximation to $\lceil N_{ij}/2 \rceil$, and the one we use, is based on a uniform distribution of Morton elements in the underlying space. Let A_S be the area measured in unit-sized elements of the space occupied by the underlying database, and N_i be the number of Morton elements in stream *i*. Moreover, recalling that *zlo(current(j))* is the Morton code of the upper-left corner of the Morton element corresponding to the current element of stream *j*, and *zhi(current(i))* is the Morton code of the lower-right corner of the Morton element corresponding to the current element of stream *i* we have *zlo(current(Y)) - zhi(current(X))* unit-sized Morton elements lying between the current elements of streams *i* and *j*. Thus N_{ij}= *(zlo(current(j)) - zhi(current(i)))* N_i / A_S. Usually the distribution of Morton elements in space is not uniform and thus the estimate N_i may not be accurate. A closer approximation can be obtained by assuming a piecewise uniform distribution of Morton elements in the underlying space. For example, we could subdivide the space into a 16X16 grid of cells, say N_G, and store a number such as N_B for each of the cells in

N_G. Now, we compute N_{ij} by accessing the cell in N_G containing the *zlo* values of the current elements of streams *i* and *j*.

5 EXPERIMENTAL RESULTS

We tested our algorithms by running a number of experiments on a SUN SPARC I workstation. All the spatial join algorithms were written in C++. We made use of the following software: a buffer manager, a B-tree index, a Morton block manipulation library, and a Morton block index (a tailored B-tree index for storing and retrieving Morton blocks with an appropriate interface). Our software maintains its own page buffer pool. The buffer manager used a least recently used (LRU) page replacement policy. We also implemented and used a quadtree decomposition algorithm to construct the Morton blocks corresponding to any given two-dimensional object.

We conducted our experiments using both real and synthetic data sets. The real data sets consist of the road networks in the data of the U.S. Bureau of the CensusTiger/Line file [12] for representing the roads and other geographic features in the U.S. (e.g., see Figure 6). For each line segment in the Tiger database, we constructed the line's minimum bounding rectangle. The synthetic data sets are collections of randomly generated rectangles. Each rectangle is decomposed into the Morton blocks that are inside it. The blocks are then inserted into the Morton block index (the B-tree) along with blocks from the decomposition of the other rectangles.

Map Name	Data Coverage	Number of Line-segments	Total Number of Morton Blocks
Falls Church	0.43	587	24330
Bedford	0.34	1644	27725
Williamsburg	0.19	2112	20686
Franklin	0.35	1685	25708
Baltimore	0.54	13952	87301
Prince George's	0.82	44392	148826
Washington DC	0.56	18517	94934

Table 1: Parameter values for the Tiger data files used in our experiments. The size of all the maps is normalized to 512X512.

The synthetic data sets are characterized by the following parameters: the area of the underlying space A_S, (e.g., 512X512 pixels), the average area of the rectangles A_R in the data set (measured in square pixels), and the total coverage of the rectangles C_{RS} (measured in percent). This is the ratio between the total area of all the rectangles in the index and the area of the underlying space. This measure is an indication of the degree of denseness of the underlying space. The Tiger/Line maps are characterized by the total coverage, the number of line segments, and the number of Morton blocks. Table 1 gives the values of these parameters for the Tiger/Line maps that we used in the experiments.

For the synthetic data sets, each input stream of the spatial join is characterized by assigning one value to each of the above parameters. We call the triplet of parameters A_S, A_R, and C_{RS} a parameter setting *PS*. In order to test a spatial join algorithm, we need to specify the parameter setting of each input stream, e.g., the two input streams have parameter settings PS_1 and PS_2, respectively. To make our results more robust, for a given parameter setting *PS* of A_S, A_R, and C_{RS}, we generate 10 sets of random rectangles satisfying the same parameter setting. We call the 10 sets a master data set, and denote it by MD_{PS}. To test any version of the spatial join algorithms described in this paper, say algorithm *FOO_JOIN*, for a pair of parameter settings PS_1 and PS_2, we ran the following experiment: each set of rectangles in MD_{PS1} is joined with each set of rectangles in MD_{PS2} using algorithm *FOO_JOIN*. The overall CPU time is then summed and averaged (by dividing it by 100, which is the total number of times the *FOO_JOIN* algorithm was executed for this experiment). For our experiments, we fixed the area of the underlying space A_S to be always 512X512. Other parameters that we used are given in Table 2.

Parameter Name	Constant Value
Disk page size	1024 bytes
Space size	512X512 pixels
Morton block size	12 bytes (includes an object identifier)
B-tree pointer size	4 bytes
B-tree fanout	64
B-tree buffer size	16 pages

Table 2: Constant parameters used throughout the experiments.

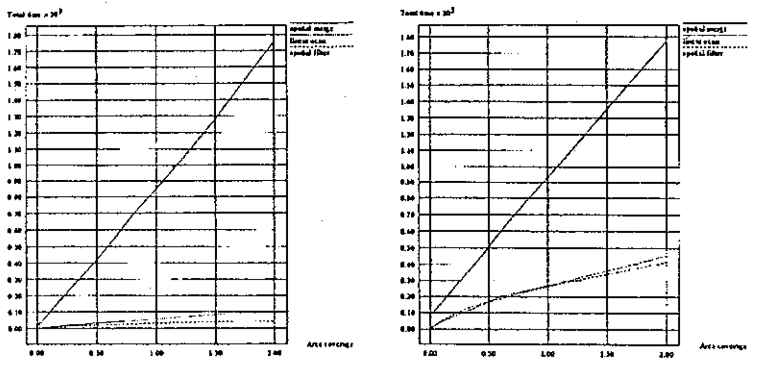
Table 3 gives the average number of elements (i.e., Morton blocks) per input stream for some of the master data sets that we used in the experiments. Notice that the total number of Morton blocks is proportional to the total number of objects in the database. Since the area of the rectangles is fixed, more rectangles need to be inserted into a stream in order to increase its data coverage. This in turn increases the number of Morton Blocks in the stream. Also notice that as the area of the rectangles increases, the number of Morton blocks decreases. This is a result of the data coverage being fixed, which means that as the area of the rectangles increases, less rectangles are needed to achieve the required coverage. Therefore, the overall number of Morton blocks decreases as well since there are less rectangles in the stream.

Data Coverage	Area of Rectangle	Space Area	Number of Rectangles	Average Number of Morton Blocks
0.01	64	512X512	40	950
0.10	64	512X512	409	9385
0.20	64	512X512	819	18937
1.00	64	512X512	4096	94239
1.50	64	512X512	6144	142086
2.00	64	512X512	8192	189496
1.00	128	512X512	2048	69813
1.00	512	512X512	512	40010
1.00	1024	512X512	256	29096
1.00	2048	512X512	128	21754
1.00	16384	512X512	16	8404

Table 3: Average number of elements in an input stream with a given parameter setting in synthesized data sets. The area of the rectangles is measured in square pixels.

For synthetic data sets, we compared our first spatial join algorithm (the linear-scan spatial join) with the spatial merge and spatial filter algorithms for different data coverage values. Figure 7 gives the comparison results when the data coverage of the first input stream is fixed at 1% and 10%, respectively, while the data coverage of the second input stream varies from 1% -- 200%. Other data coverage values show similar trend. Although the linear-scan algorithm is not as efficient as the the spatial filter algorithm, it performs much better than the spatial merge algorithm (9 and 20 times faster). The performance gain is substantial. This result is very promising especially in light of the fact that the linear-scan spatial join algorithm operates on non-indexed but sorted data which makes it more applicable than the spatial filter algorithm in a pipelined query processing architecture.

For real data sets, Table 4 compares the spatial merge algorithm with the linear-scan spatial join algorithm using the Tiger/Line data files. Since their geographical locations are different, we normalized them so that they cover the same area as far as the join is concerned.



(a) Area coverage = 1% (1st stream). (b) Area coverage = 10% (1st stream).

Figure 7: A comparison of the linear-scan join, spatial merge, and spatial join algorithms. The data coverage of the first input stream of the join is (a) 1%, (b) 10%. The x-axis corresponds to the data coverage of the second input stream.

	FC	BF	WB	FR	BA	PG	DC
FC	0.68	1.49	1.64	1.20	1.44	1.20	1.13
BF	1.48	0.70	1.88	1.39	1.84	1.48	1.33
WB	1.66	1.87	0.71	1.69	3.60	2.50	2.31
FR	1.18	1.36	1.72	0.67	2.18	1.27	1.27
BA	1.46	1.84	3.70	1.97	0.69	1.65	1.85
PG	1.21	1.50	2.58	1.31	1.62	0.70	1.20
DC	1.11	1.30	2.36	1.27	1.86	1.22	0.69

Table 4: Ratio of the execution time of the spatial merge over the linear-scan spatial join algorithms using the Tiger/Line files.

The table computes the ratio of the execution time of the spatial merge algorithm divided by the execution time of the linear-scan algorithm. From the table, we can deduce that the linear-scan algorithm performs better in all but

the diagonal cases (i.e., when a map is joined with itself). The performance gain varies from 11% to 270% in favor of the linear-scan algorithm. This gain is related to savings in the CPU time of the linear-scan algorithm (see [4] for a similar discussion on the expensive CPU cost in spatial joins). As mentioned in Section 3, the number of disk page reads of the two algorithms is exactly the same. The performance gain results from the fact that while the spatial merge join processes each element in the two input streams uniformly, the linear-scan algorithm spends less CPU-time on the elements detected by it that do not contribute to the final result of spatial join. An interesting observation is that only on the diagonal of the table does the spatial merge algorithm perform better. The reason is that when applying spatial join to two identical data sets (which is the case for the diagonal of the table), each element in one stream will be joined with its corresponding element in the other stream. In this case, the algorithm can not jump ahead and thus reduces to a spatial merge with the additional overhead of needlessly trying to compute parents and skip elements which does not pay off in the case of two similar input streams.

We also compared the performance of our second algorithm with the spatial filter algorithm for both synthetic and real data. For synthetic data, Figure 8 gives the comparison results when the data coverage of the first input stream is fixed at 1% and 10%, respectively, while the data coverage of the second input stream varies from 1% -- 100%.The figures show that the estimate-based algorithm does not perform much better than the spatial filter algorithm. Although dominating in all cases, the estimate-based algorithm has a performance gain of only 8%–20% which is relatively insignificant. For real data, Table 5 gives the ratio of the execution time of the spatial filter algorithm divided by the estimate-based algorithm. The performance gain is only 0%-- 32% which again is relatively insignificant. Thus we see that both the spatial filter and the linear-scan algorithm are quite efficient and it is difficult to improve on them.

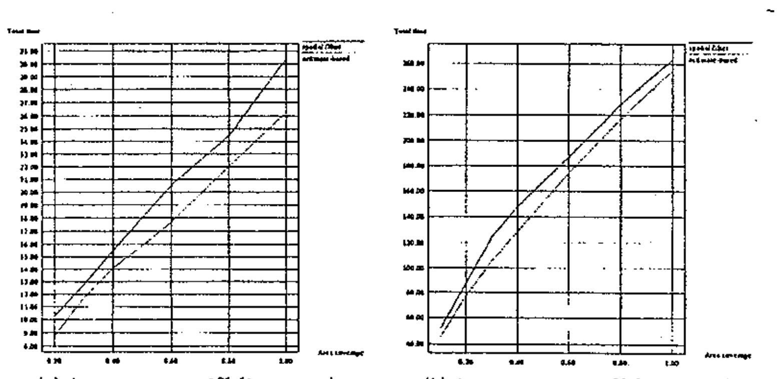

(a) Area coverage = 1% (1st stream). (b) Area coverage = 10% (1st stream).

Figure 8: A comparison of our estimate-based spatial join algorithm and the spatial filter algorithm. The data coverage of the first input stream of the join is (a) 1%, (b) 10%. The x-axis corresponds to the data coverage of the second input stream.

	FC	BF	WB	FR	BA	PG	DC
FC	1.00	1.25	1.29	1.22	1.12	1.20	1.00
BF	1.26	1.00	1.22	1.13	1.13	1.13	1.15
WB	1.29	1.32	1.04	1.31	1.21	1.19	1.20
FR	1.24	1.25	1.28	1.12	1.16	1.15	1.15
BA	1.16	1.16	1.19	1.16	1.03	1.13	1.14
PG	1.15	1.16	1.22	1.18	1.15	1.03	1.11
DC	1.17	1.19	1.18	1.15	1.14	1.12	1.00

Table 5: Comparison between the spatial filter and the estimate-based spatial join using the Tiger/Line files.

6 CONCLUSIONS

Two new spatial join algorithms are presented in this paper which are proposed optimizations over the spatial merge and spatial filter algorithms. Our first algorithm, the linear-scan spatial join runs around 9-20 times faster than the spatial merge algorithm. The importance of the linear-scan algorithm lies in the fact that it addresses a very common need in a pipelined query processor. It assumes that the input streams are sorted but not necessarily indexed. The

only existing alternative in this case (i.e., the spatial merge algorithm) is 9--20 times slower. Our second algorithm, the estimate-based spatial join, an optimization of the spatial filter algorithm, uses estimates and some preprocessing parameters in order to enhance the performance of the spatial filter algorithm. As shown in the experiments, the spatial filter algorithm proves to be very efficient and that using estimates does not enhance the performance significantly enough (a maximum of 32%) to pay off the burden of using estimates and preprocessing parameters. Therefore we recommend the use of the spatial filter algorithm instead.

7 ACKNOWLEDGEMENTS

The support of the National Science Foundation under Grant IRI 9216970 is gratefully acknowledged. The first author conducted most of this research while at The University of Maryland, College Park.

REFERENCES

[1] W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Gunther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symp., SSD'91. Lecture Notes in Computer Science 525*, pages 299--318. Springer-Verlag, Berlin, 1991.

[2] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proc. of the 9th Intl. Conf. on Data Engr.*, pages 190--197, Vienna, Austria, April 1993.

[3] L. A. Becker. *A New Algorithm and a Cost Model for Join Processing with Grid Files*. PhD thesis, University of Siegen, July 1992.

[4] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of the 1993 ACM SIGMOD*, pages 237--246, Washington, DC, May 1993.

[5] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905--910, December 1982.

[6] O. Gunther. Efficient computation of spatial joins. In *Proc. of the 9th Intl. Conf. on Data Engr.*, pages 50--59, Vienna, Austria, April 1993.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD*, pages 47--57, Boston, June 1984.

[8] A Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303--337. Academic Press, New York, 1971.

[9] K. Knowlton. Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes. In *Proceedings of the IEEE 68*, number 7, pages 885--896, July 1980.

[10] W. Lu and J. Han. A new algorithm for computing joins with grid files. In *Proc. of the 8th Intl. Conf. on Data Engr.*, pages 284–292, Tempe, Arizona, February 1992.

[11] G.M. Morton. A computer oriented geodetic data base, and a new technique in file sequencing. Tech. Report Unpublished, IBM Ltd., Ottawa, Canada, 1966.

[12] Bureau of the Census: 1990 technical documentation. Tiger/line precensus files. Tech. report, US Bureau of Census, Washington, DC, 1989.

[13] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Engr.*, 14(5):611--629, May 1988.

[14] J. A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proc. of the 3rd ACM PODS*, pages 181--190, Waterloo, Canada, April 1984.

[15] Doron Rotem. Spatial join indices. In *Proc. of the 5th Intl. Conf. on Data Engr.*, pages 500--509, Kobe, Japan, April 1991.

[16] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

SDH 94

International Geographical Union Commission
on Geographic Information Systems

agi

Association for Geographic Information

SDH 94

Sixth International Symposium
on
Spatial Data Handling

5th - 9th September 1994
Edinburgh, Scotland, UK

ADVANCES IN GIS RESEARCH

PROCEEDINGS, VOLUME 1

EDITORS

Thomas C. Waugh

and

Richard G. Healey