Cascaded Spatial Join Algorithms with Spatially Sorted Output

Walid G. Aref and Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742
Phone: 301-405-1755, Fax: 301-314-9115
E-mail: aref,hjs@umiacs.umd.edu

Abstract

The spatial join is an operation that combines entities from two spatial data sets into a single entity whenever the combination satisfies the spatial join condition (e.g., if they overlap in space). Linear quadtrees and the Z-order are two widely-used data structures for organizing spatial databases. They are based on the concept of recursive decomposition of space. For spatial databases using recursive decomposition, algorithms for performing spatial join already exist in the literature. For efficiency purposes, some of these algorithms require that the two input streams of the spatial join be indexed, while other algorithms require that the two input streams be sorted, but not necessarily indexed. These algorithms suffer from the fact that they produce spatially un-ordered output, i.e., their output is not sorted. As a result, they cannot be used efficiently to answer complex spatial queries that involve cascading spatial join operations in their query evaluation pipelines. A new optimization is presented that can be applied for a wide variety of spatial join algorithms that would result in producing spatially ordered output, i.e., sorted. This optimization enhances the applicability of these algorithms significantly so that they can be efficiently used to answer spatial database queries that involve multiple cascading spatial join operations.

1 Introduction

One of the most useful tools for spatial query processing is the *spatial join*. Generally speaking, the spatial join combines entities (e.g., tuples in a relational database management system and also termed *elements* here) from two sets (termed *streams*) into a single entity in a resulting stream whenever the combined entity satisfies the spatial join condition (e.g., if the two entities intersect each other). In the context of data representations where the spatial attribute is one that describes the space occupied by the entity (i.e., tuple), the spatial join primarily means that the two entities are joined *iff* they overlap in space. For example, applying the spatial join to the two maps in Figures 1a and 1b yields the object pairs (3,4), (3,5), and (3,6).

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee
GIS 96 Rockvillle MD USA
Copyright 1997 ACM 0-89791-874-6/96/11 ..$3.50

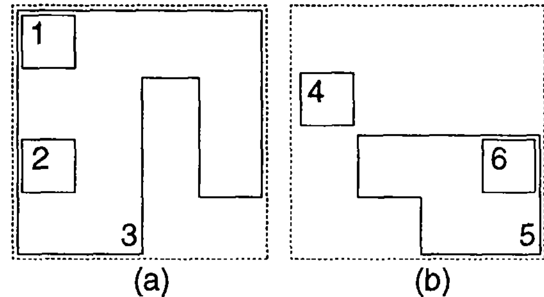

Figure 1: Two sample maps.

Many algorithms for performing spatial join already exist in the literature (e.g., [5, 4, 6, 7, 9, 16, 19]). Our focus in this paper in on algorithms that assume that the underlying spatial database is organized based on recursive decomposition of space, e.g., quadtrees [8, 11, 20], and the Z-Order [17, 16]. Orenstein and Manola [15, 16] present two algorithms for spatial join (overlap) using the Z-Order as their underlying organization of the spatial database. The first algorithm, termed the *spatial merge*, requires that the two input streams be sorted but does not necessarily require the presence of an index. The second algorithm, termed the *spatial filter*, is an optimization of the spatial merge. The spatial filter yields significant execution time speedups over the spatial merge at the added expense of requiring that both input streams be indexed (e.g., as a result of sorting on the basis of the spatial attribute 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 (e.g., a B-tree, ISAM, etc.). The speedups are a direct result of using the index to directly access the input streams.

Aref and Samet [2] present two new spatial join algorithms that are optimizations over the spatial merge and the spatial filter algorithms. The first algorithm, termed the linear-scan spatial join algorithm, avoids processing every element in the two streams by just scanning the irrelevant intervening elements between corresponding positions in the two streams as it has no index. The second algorithm, termed the estimate-based spatial join algorithm, uses online estimates of the input streams to decide whether to use the index for a direct access request or a linear scan.

All the four spatial join algorithms (namely, the spatial merge, the spatial filter, the linear-scan, and the estimate-

based algorithms), require that the input streams must be sorted (spatially ordered, e.g., according to the Z-order), while the output they produce is not sorted. This is not a problem if the spatial join is only performed once. However, in most applications (e.g., in a geographic information system), it is usually the case that the spatial query requires the execution of a cascade of spatial join operations (i.e., the output of one operation serves as the input to the next operation). Thus, use of any one of these algorithms means that only the first spatial join is efficiently executed. All subsequent spatial joins will require either to spatially sort their input (or build a temporary index) before performing them, or to use a nested loop join algorithm with a spatial intersection predicate, since none of the above spatial join algorithms works properly for unsorted input streams.

In this paper we present an algorithmic improvement (or optimization) that works properly for all of the above four spatial join algorithms. When this new optimization is augmented to any of the four spatial join algorithms, the augmented algorithm will produce output tuples that are sorted spatially (e.g., using the Z-order sort order). Therfore, any of the modified spatial join algorithms can then be applicable in spatial query pipelines that involve multiple cascaded spatial joins without the need to sort the output at each intermediate stage.

The rest of this paper is organized as follows. Section 2 covers background material that is needed for the presentation of the algorithms. Then, we present the new optimization that we propose in this paper and show how it is applicable to some of the existing spatial join algorithms, namely, the spatial merge (Section 3), the spatial filter (Section 4), and the linear-scan algorithm (Section 5). Section 6 summarizes our findings and draws some concluding remarks. The Appendix contains detailed specifications of the new modified algorithms so that readers can understand the important subtleties that are involved in their implementation.

2 Background

2.1 Spatial Data Representation

Spatial data consists of objects which are permitted to overlap. In the Z-Order data structure, each object is represented by a set of restricted rectangular elements (termed *Z-elements*) that collectively approximate (and cover) the object. The restricted rectangular elements result from a recursive decomposition of the space into two equal-sized blocks while cycling the splits through the different coordinate axes corresponding to the dimensions of the space.[1] The decomposition is such that the space (i.e., the blocks) is recursively decomposed until each block is totally covered by one or more objects or is not covered by any of the objects. The Z-elements consist of the blocks that are covered by the objects and not the blocks that are empty. For example, Figures 2a and 2b are the Z-elements corresponding to the maps in Figures 1a and 1b, respectively. In this example, we have cycled through the axes in the order $xyxy\cdots$. Note that it is impossible for a block corresponding to a Z-element to be covered partially by one object and partially by another object. In other words, block i is not considered to be totally covered if one half of the block is covered by object j and one half by object k.

[1] The result is a collection of blocks the length of whose sides are either all equal or differ by a factor of two; hence the use of the qualifier *restricted*.

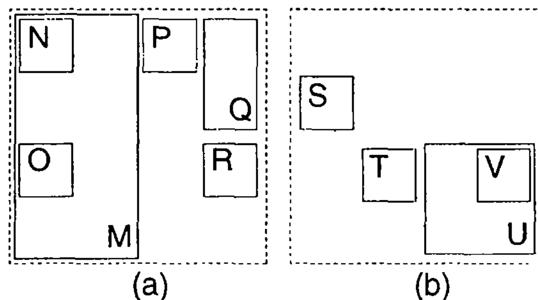

Figure 2: Result of decomposing the maps in Figure 1 into their Z-elements.

The algorithms of Orenstein and Manola make use of what is termed a *Z-order* [13] to order the Z-elements (i.e., the blocks that cover the objects). The Z-order is a linearization of space where each of the blocks corresponding to the Z-elements is represented by a unique value which is then stored in a sorted list or some other data structure (possibly with an indexing capability). In this case, the value is a function of two parameters: the coordinate values of the upper-left corner and the size of the restricted rectangular region corresponding to the Z-element. In addition, each Z-element has an object identifier that indicates the identity of the object to which it belongs. For more details on how to construct the Z-order for a collection of objects and the performance issues related to redundancy and accuracy when using them to approximate the objects, see [14, 16].

In our presentation of the new optimization and the modified algorithms, we also make use of a Z-order. For simplicity, we restrict our domain to two dimensions with Z-elements whose corresponding blocks are square. These Z-elements result from a recursive decomposition of the space into four square blocks (i.e., a quadtree-like decomposition [20]). The resulting elements are special cases of Z-elements and are termed *Morton elements* [12]. For example, Figures 3a and 3b are the Z-elements corresponding to the maps in Figures 1a and 1b, respectively. The Morton elements are ordered on the basis of their size and the result of mapping the point at the upper-left corner of each block into an integer. This is achieved by interleaving the bits that represent the values of the x and y coordinates of the two-dimensional point. The result of the interleaving process is termed a *Morton code* [12]. Note that the techniques presented here apply directly to arbitrary Z-elements which can assume restricted rectangular shapes in two and higher dimensions.

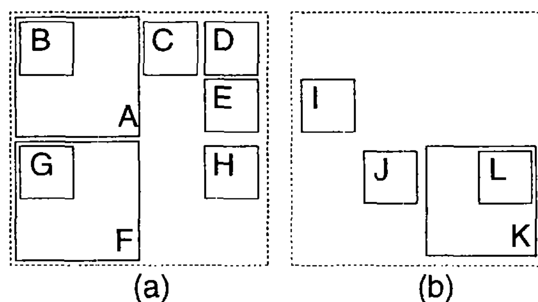

Figure 3: Result of decomposing the maps in Figure 1 into their Morton elements.

2.2 The Spatial Join Operation

When computing the spatial join of Z-element (including Morton element) decompositions of two maps, we report the pairs of overlapping blocks and then obtain the appropriate object identifiers by use of lookup operations. For example, the spatial join of the Morton elements in Figures 3a and 3b consists of the block pairs (A,I), (F,J), (H,L), and (H,K) corresponding to the object pairs (3,4), (3,5), (3,6), and (3,5), respectively. Notice that because of the nature of the Z-elements and the recursive decomposition process by which they are constructed, any pair of corresponding Z-elements either in the input or output streams are either equal or are contained in one another; however, they cannot overlap without one Z-element being totally contained in another Z-element.

One of the drawbacks of decomposing spatial objects into a collection of Z-elements is that the same pair of objects could be reported as overlapping many times. This was the case in the spatial join of Morton element decompositions of the maps in Figures 3a and 3b where the object pair (3,5) was reported twice. In fact, if objects i and j are represented by n_i and n_j Z-elements, respectively, then, depending on the extent of the overlap between the objects, the pair (i, j) could be reported as often as $n_i + n_j - 1$ times. This problem of multiple reportings arises in the implementations of many other spatial operations as well. For example, suppose we want to retrieve all the objects in a given region. Clearly, each object should be reported once. Eliminating the multiply reported values using more efficient techniques than simply sorting the output is a serious issue (see [1, 3] for a discussion of how to deal with this problem when the spatial objects are line segments and rectangles).

3 The Modified Spatial Merge Join Algorithm

For brevity, we explain directly our modified version of the spatial merge algorithm of Orenstein and Manola [15, 16]. In contrast to their algorithm, the modified spatial merge join algorithm produces sorted tuples based on some spatial order, which in our case is the Morton- or Z-Order.

The modified spatial merge algorithm requires that the two input streams be sorted but not necessarily indexed. We present quadtree variants of the algorithm as we assume that Z-elements are always Morton elements (i.e., square blocks in contrast to restricted rectangular blocks as in the more general case of the Z-Order [13, 17]).

A Morton element B is represented by the Morton code of B's upper-left corner and its size.[2] The key for sorting in each input stream is the Morton code (in ascending order) and size (in descending order) of each Morton element in the stream. The Z-order for the Morton elements is such that Morton element B appears in the stream before Morton element C if the Morton code of B is less than the Morton code of C. Furthermore, if two Morton elements share the same upper-left corner, then the larger one appears first in the stream. For example, the Z-order for the Morton elements in Figure 1a is A, B, C, D, E, F, G, H.

The modified spatial merge algorithm resembles a merge join algorithm. The difference is that in the modified spatial merge algorithm elements of the same input stream represent two-dimensional intervals that can be contained in one another. The modified spatial merge algorithm is given in the Appendix using procedures Control, NextElement, EnterElement, ExitElement, and Advance. The basic difference between the modified spatial merge algorithm and the spatial merge algorithm of Orenstein and Manola is in the timing when an output tuple is reported by the algorithm. By selecting the proper timing for reporting the output tuples during the proper execution of the algorithm, this results in the reported tuples being produced in sorted order without extra effort.

In the following, we explain the necessary data structures and the control structure of the algorithm. The control structure is particularly important as it is the same for all of the spatial join algorithms discussed in this paper. Procedures Control, NextElement, EnterElement, and ExitElement make up the control structure. Procedure Advance is used in conjunction with NextElement to obtain the next Morton elements from the input streams. It differs for each of the spatial join algorithms. In the case of the spatial merge algorithm, it simply produces the next Morton element in the stream corresponding to its first argument.

The control structure is responsible for maintaining the state of the spatial join algorithm. This is done by using a stack for each input stream, and pointers to the current Morton element in each stream. The top of each stack contains the most recent Morton element for the corresponding stream, while the containing Morton elements (from the same stream) are stored deeper in the stack (as they are encountered first according to our ordering which places them earlier in the stream). The area of space spanned by each Morton element B ranges between the value of the Morton code of B's upper-left corner, denoted by *zlo(B)*, and that of B's lower-right corner, denoted by *zhi(B)*.

In each iteration of the spatial join, the state of one stream is updated. As Morton elements from the input streams are scanned during the merge, Morton elements are pushed (i.e., entered) into and popped (i.e., removed) from the stacks of the corresponding streams. The spatial join behaves like a plane-sweep [18] in the metric space of Morton codes of the upper-left corners of the Morton elements (i.e., by their *zlo* values). Therefore, Morton elements from both streams that overlap are processed in consecutive iterations of the algorithm (since Morton elements are ordered by their *zlo* values).

The main step in each iteration of the algorithm is a four-way comparison between the zlo values of the current Morton elements in each stream and the zhi values of the top Morton elements in the stacks of the streams. The minimum of these four values is used to control the action. In order for the comparisons to work, the stacks are initialized to contain MaxMortonElement which is a unit-sized Morton element whose Morton code is larger than the maximum possible value (called MaxMortonCode in the code) in the space in which the spatial join is executed. MaxMortonElement also serves as the current value in each of the input streams once they have been exhausted (set by procedure NextElement).

We illustrate our explanation of the algorithm with the aid of Figure 4 which traces the steps of the modified spatial merge algorithm when used to compute the spatial join of the streams X and Y given by the Morton elements in Figures 3a and 3b corresponding to the maps in Figures 1a and 1b, respectively. In the figure, the leftmost and rightmost values in the stack entries correspond to the top and bottom, respectively, of the stack. The stacks are initialized to W which corresponds to MaxMortonElement. W is also used as the final Morton element once the input

[2] 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.

streams have been exhausted. zlo(W) and zhi(W) are set to MaxMortonCode+1. Figure 4 is organized in such a way that the entry for step i indicates the state resulting from taking the action indicated at this step. This action is based on the value produced by a four-way comparison using the current values of the two input streams and the contents of the top entries in the stacks that resulted from the execution of step $i-1$.

Morton element R from stream i is entered (i.e., pushed) on the stack of stream i when the sweep encounters position $zlo(R)$ such that the four-way comparison yields $zlo(R)$ as the minimum value. Note that upon entering Morton element R on the stack of stream i we already know that Morton element R is contained in Morton element S at the top of the stack of stream i since $zlo(R)$ is less than $zhi(S)$ as a result of the minimum four-way comparison. Thus there is no need to remove any members of the stack of stream i at this point. Processing continues with the next Morton element in stream i. For example, this is the case when we push B on the stack of stream X in step 2 in Figure 4. We continue after advancing current(X) to C.

A Morton element is removed (i.e., popped) from the stack of stream i when the sweep encounters Morton elements S and T at positions $zlo(S)$ and $zlo(T)$ in streams i and j, respectively, such that the four-way comparison results in the zhi value of Morton element U at the top of the stack of stream i being the minimum value. This means that Morton element U can be removed from the stack of stream i as Morton element S cannot be contained in Morton element U. For example, this is the case when we encounter J as the current element of stream Y in step 5 in Figure 4 – that is, we pop I from the stack of stream X.

Upon inserting an element U into the stack of stream i, the algorithm checks if U overlaps with any of the Morton elements in the stack of stream j, and reports all the overlapping pairs that are found. These pairs constitute the result of the spatial join. Processing continues with a new element at the top of the stack of stream i. Continuing our example in Figure 4, this means that in step 4 we check if the pushed element I overlaps with A, the only member of the stack of stream X that is in the space in which the spatial join is executed, and find that it does. We report (A,I) as the overlapping pair and continue. Note that we need not check if Morton element U overlaps with Morton element T (the current Morton element of stream j) because at this time, by virtue of our minimum value four-way comparison, we already know that $zhi(U)$ is less than $zlo(T)$ (e.g., in step 4 in Figure 4 I need not be tested for overlap with C). Recall that no overlap between two Morton elements V and W is possible if $zhi(V)$ is less than $zlo(W)$.

4 The Modified Spatial Filter Algorithm

The second algorithm is the modified spatial filter. It is a modified version of the spatial filter join algorithm of Orenstein and Manola [15, 16]. It requires that both input streams be indexed (so that random as well as sequential access to the elements in either stream is possible).

As mentioned before, the key to the spatial merge algorithm is that procedure SpatialMergeAdvance processes the elements of the two streams in Z-order. This could be rather wasteful as the elaborate control structure is invoked for every element in the two streams. In contrast, the modified spatial filter algorithm, encoded by procedure SpatialFilterAdvance given in the Appendix, assumes the presence of an index for each of the two streams (e.g., a B-tree, B+-tree, ISAM, etc.).

We illustrate our explanation of the algorithm with the aid of Figure 5 which traces the steps of the modified spatial filter algorithm when used to compute the spatial join of the streams X and Y given by the Morton elements in Figures 3a and 3b corresponding to the maps in Figures 1a and 1b, respectively. It is interpreted in the same way as Figure 4. Note that along with each step in Figure 5 we also indicate the corresponding step in the trace given in Figure 4.

SpatialFilterAdvance differs from SpatialMergeAdvance in that instead of simply advancing to the next element in stream i and continuing processing from there, SpatialFilterAdvance advances to the next element in stream i and uses information from stream j to directly access stream i (via the index on i) thereby skipping elements in stream i that do not overlap the current element of stream j and thus cannot contribute to the result of the spatial join operation. Figure 6 is an example where elements A and B in stream X are skipped while current(X) is reset to C. This situation also occurs in steps 1, 5, and 7 of Figure 5 where elements B, D and E, and G, respectively, are skipped.

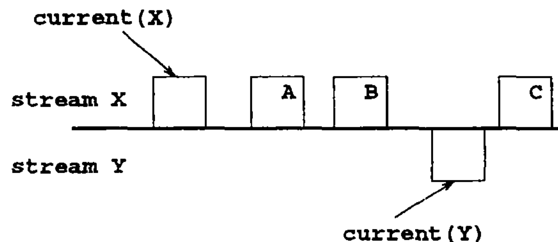

Figure 6: Morton elements A and B are skipped after processing current(X) with processing resumed at C.

It should be clear that SpatialFilterAdvance cannot skip any elements in stream i after executing the initial advance in stream i if the current element of stream i starts at a point beyond the current element of stream j or overlaps with it. For example, in Figure 5 step 2 illustrates the case that J, the current element of stream Y, starts at a point beyond C, the current element of stream X. Continuing this example, steps 8 and 11 demonstrate the case that the current element of stream Y (i.e, K and L, respectively) overlaps the current element of stream X (i.e., H). Also, at the same instance, no elements can be skipped if the current element of stream i is on the stack of stream j or is contained in one of the elements of this stack.

In the rest of the discussion of the modified spatial filter algorithm in this section, we consider the case that procedure SpatialFilterAdvance has just advanced the current element of stream i. This means that it uses Q, the current element of stream j, to directly access stream i to find the next element to be processed in stream i. If Q is not found in the index corresponding to stream i, then the element R in stream i with the smallest index value (i.e., $zlo(R)$) greater than that of Q (i.e., $zlo(Q)$) is returned as the new current element of stream i. One problem that must be resolved by the modified spatial filter algorithm is that there may exist some Morton elements in stream i that start at a position that is past the current element of stream i and that contain the current element of stream j (hence contributing to the result of the spatial join) that might be erroneously skipped if the algorithm uses only the current element of stream j to directly access the elements in stream i. For example, in

step	current(X)	current(Y)	stack of X	stack of Y	action
start	A	I	W	W	
1	B	I	A W	W	push A on stack of X
2	C	I	B A W	W	push B on stack of X
3	C	I	A W	W	pop B from stack of X
4	C	J	A W	I W	push I on stack of Y and report pair (A,I)
5	C	J	A W	W	pop I from stack of Y
6	C	J	W	W	pop A from stack of X
7	D	J	C W	W	push C on stack of X
8	D	J	W	W	pop C from stack of X
9	E	J	D W	W	push D on stack of X
10	E	J	W	W	pop D from stack of X
11	F	J	E W	W	push E on stack of X
12	F	J	W	W	pop E from stack of X
13	G	J	F W	W	push F on stack of X
14	H	J	G F W	W	push G on stack of X
15	H	J	F W	W	pop G from stack of X
16	H	K	F W	J W	push J on stack of Y and report pair (F,J)
17	H	K	W	J W	pop F from stack of X
18	H	K	W	W	pop J from stack of Y
19	H	L	W	K W	push K on stack of Y
20	W	L	H W	K W	push H on stack of X and report pair (H,K)
21	W	W	H W	L K W	push L on stack of Y and report pair (H,L)
22	W	W	W	L K W	pop H from stack of X
end	W	W	W	L K W	X and its stack are empty

Figure 4: Trace of the application of the spatial merge algorithm to the Morton elements in Figures 3a and 3b.

Figure 5 step 5 illustrates the case that D, the current element of stream X, terminates at a point before J, the current element of stream Y, while F in stream X starts at a point past D and does contain J.

In order to overcome this problem, the modified spatial filter algorithm generates all the possible ancestors (i.e., containing Morton elements) of the current element of stream j that lie between the current elements of streams i and j (i.e., Morton elements A in stream j such that *zhi(current(i))* is less than *zlo(A)*).[3] Of course, not all the possible ancestors of the current element of stream j exist in stream i. In fact, it could be that none of them may exist. For example, in Figure 7 four possible ancestors of `current(Y)` are generated but only two of them are found in the stream. A similar situation occurs in step 5 of Figure 5 where F and an element analogous to J are generated for stream X with D as its current element, and of course only F is found in stream X.

Procedure `SpatialFilterAdvance` must set the current element of stream i to point at the largest ancestor of the current element of stream j that is found in stream i past the current element of stream i (e.g., in step 5 of Figure 5 F is the largest ancestor in stream X of J, the current element of stream Y, that is past D, the current element of stream X). If no such ancestor exists, then the current element of stream i is set to the first element in stream i that overlaps with or lies past the current element of stream j (i.e., *zlo(current(i))* $\geq$ *zlo(current(j))*). For example, in step 7 of Figure 5 `current(X)` is set to H as no ancestor of `current(Y)` (i.e., J) exists past the previous value of `current(X)` (i.e., G). Computing the set of possible ancestors of the current element of stream j does not require any accesses to the index as this process only involves the calculation of Morton codes of the ancestors.

[3] We could have also used the less restrictive test "*zlo(current(i))* < *zlo(A)*". However, we make use of the fact that no ancestor of *current(j)* can start between *zlo(current(i))* and *zhi(current(i))* to restrict the search, and thus compare *zhi(current(i))* with *zlo(A)* in the code.

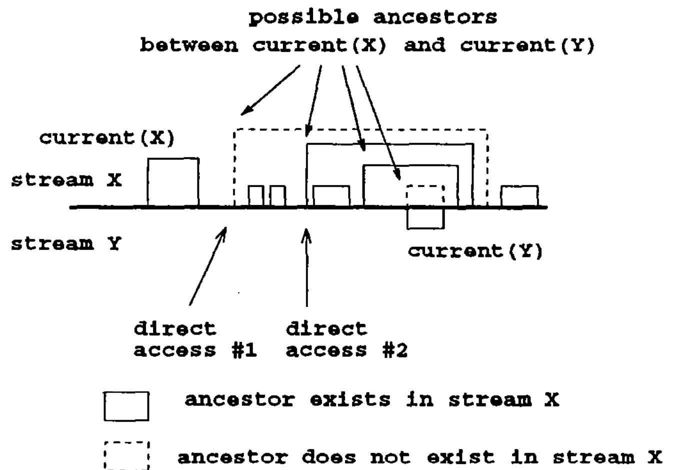

Figure 7: Possible ancestors between current(X) and current(Y).

Once the possible ancestors of the current element of stream j have been computed, procedure `SpatialFilterAdvance` directly accesses stream i searching for the largest ancestor of the current element of stream j one-by-one. In the worst case, this requires a number of direct accesses to stream i equal to the number of computed ancestors between *zhi(current(i))* and *zlo(current(j))*. As soon as the algorithm finds the largest ancestor in stream i, it can stop the search and return. For example, in Figure 7, `SpatialFilterAdvance` returns after directly accessing the second largest possible ancestor of `current(Y)` in stream X as the largest possible ancestor was directly accessed but was not found.

step	current(X)	current(Y)	stack of X	stack of Y	action
start	A	I	W	W	
1 (1)	B	I	A W	W	push A on stack of X and set current(X) to B candidate in Y: I
	C	I	A W	W	random access on X yields C
2 (4)	C	J	A W	W	push I on stack of Y and report pair (A,I)
3 (5)	C	J	A W	W	pop I from stack of Y
4 (6)	C	J	W	W	pop A from stack of X
5 (7)	D	J	C W	W	push C on stack of X and set current(X) to D candidates in Y: F's analog and J
	F	J	C W	W	random access on X yields F
6 (12)	F	J	W	W	pop C from stack of X
7 (13)	G	J	F W	W	push F on stack of X and set current(X) to G candidate in Y: J
	H	J	F W	W	random access on X yields H
8 (16)	H	K	F W	J W	push J on stack of Y and report pair (F,J)
9 (17)	H	K	W	J W	pop F from stack of X
10 (18)	H	K	W	W	pop J from stack of Y
11 (19)	H	L	W	K W	push K on stack of Y
12 (20)	W	L	H W	K W	push H on stack of X and report pair (H,K)
13 (21)	W	W	H W	L K W	push L on stack of Y and report pair (H,L)
14 (22)	W	W	W	L K W	pop H from stack of X
end	W	W	W	L K W	X and its stack are empty

Figure 5: Trace of the application of the spatial filter algorithm to the Morton elements in Figures 3a and 3b. Step numbers within parentheses correspond to the step numbers in the trace of the spatial merge algorithm in Figure 4.

5 The Modified Linear-Scan Spatial Join Algorithm

Similarly, the modified linear-scan spatial join is an enhancement over the linear-scan spatial join of Aref and Samet [2], where the modified linear-scan algorithm guarantees that the output of the join is spatially sorted according to the Morton- or the Z-order.

The spatial filter algorithm results in significant speedups over the spatial merge algorithm. Unfortunately, it requires that the two streams be indexed while producing an output that is not indexed. This is problematic in a pipelined architecture for query processing (e.g., [10, 21]) where the query is answered by a pipeline composed of a cascade of one or more operations. The first operation in the pipeline operates on the original input stream and its result is passed to the second operation in the pipeline using common buffers. Thus in such an environment the underlying indexing capabilities are effective only for the first operations in the pipeline. Subsequent operations in the pipeline must operate on non-indexed spatial data unless temporary indexes are built, which is expensive.

In other words, if the input streams are indexed, then only the first operation in the pipeline can benefit from the index. This makes it difficult to apply the spatial filter algorithm at latter stages in the query pipeline, unless we are willing to rebuild the index after each operation. Moreover, since the output of the spatial filter is not sorted, we cannot even use the spatial merge join algorithm at the latter stages in the query pipeline. Alternatively, we can use the modified spatial filter algorithm just for the first spatial join, and then use the modified spatial merge algorithm for all subsequent spatial joins in the cascade.

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 cost of building the temporary index may exceed its benefits.

In this section, we present a new algorithm, termed *the modified linear-scan spatial join algorithm*, which is an enhancement of an algorithm, termed linear-scan spatial join, that is proposed by the authors in [2]. In addition to the savings in execution time resulting from the linear-scan spatial join, the modified linear-scan spatial join produces its output in spatial order (i.e., sorted), and hence the same algorithm can be applied progressively in further stages of a query pipeline.

Assume that the spatial filter algorithm has just advanced the current element of stream i. In this case, we observe that the only time procedure `SpatialFilterAdvance` makes use of the index is when it tries to set the current element of stream i to the largest ancestor of the current element of stream j that is found in stream i. Of course, this process is speeded up by use of the index. However, we could also avoid the need for the index by using a linear scan of stream i starting at the current element of stream i. Recall, that computing the set of possible ancestors does not require use of the index.

The above observation is used as the basis of the modified linear-scanspatial join algorithm encoded by procedure `LinearScanAdvance`, given in the Appendix. We illustrate our explanation of the algorithm with the aid of Figure 8 which traces the steps of the modified linear-scan algorithm when used to compute the spatial join of the streams `X` and `Y` given by the Morton elements in Figures 3a and 3b corresponding to the maps in Figures 1a and 1b, respectively. It is interpreted in the same way as Figure 5. Note that along with each step in Figure 5 we also indicate the corresponding step in the trace given in Figure 4. Figure 8 is almost

unchanged from Figure 5. The only changes occur in steps 1, 5, and 7 where current(X) is set to the appropriate next element by use of a linear scan in Figure 8 while this is done with a random access in Figure 5.

There are two differences between LinearScanAdvance and SpatialFilterAdvance. Again, assume that the algorithms have just advanced the current element of stream i. The first is the absence of the use of the index in determining the largest ancestor in stream i of the current element of stream j (e.g., in steps 1, 5, and 7 in Figure 8). Recall, that in SpatialFilterAdvance a random access was used to check if the possible ancestors of the current element of stream j, in decreasing order of Morton element block size, are present in stream i. As soon as one was found, say A, procedure SpatialFilterAdvance sets the current element of stream i to A and exits.

Procedure LinearScanAdvance also scans the possible ancestors of the current element of stream j in decreasing order of Morton element block size. In addition, it also simultaneously scans the Morton elements in stream i in increasing order, each time resetting the value of the current element of stream i, while testing if a particular ancestor of the current element of stream j is present (e.g., in step 5 in Figure 8 when checking stream X for the presence of ancestor F of J in stream Y, current(X) is advanced from D to E and from E to F). The scan continues for a particular ancestor A as long as the Morton code value of the current element of stream i (i.e., *zlo(current(i))*) is less than A's Morton code value (i.e., *zlo(A)*). Whenever this test fails (i.e., A is not found in stream i), the scan continues with the next smaller ancestor of the current element of stream j.

The second difference is that if none of the possible ancestors of the current element of stream j is found in stream i at a position past the current element of stream i, then the scan of stream i must be continued until encountering the first element in stream i that lies past the current element of stream j (i.e., *zlo(current(i))* > *zlo(current(j))*). For example, this is the case in steps 1 and 7 in Figure 8 where we scan stream X to C and H which are past I and J, the respective current elements of stream Y.

The number of disk page read operations for the modified linear-scan algorithm is the same as for the modified spatial merge algorithm. The modified linear-scan algorithm has superior performance because the spatial merge algorithm must process each element in a stream (i.e., enter it into the stream's stack and remove it from the stack) even when it does not contribute to any output tuples of the spatial join. Instead, the modified linear-scan algorithm detects these elements and excludes them by sequentially scanning and skipping each one of them so that they will not get processed by the controlling loop of the algorithm (i.e., entered and removed from the stack's of each stream by procedure Control). The performance gain and a comparison between the algorithms can be found in [2]. Notice that it is difficult to compare the number of disk page read operations between the linear-scan and the spatial filter algorithms. The reason is that they will differ depending on the structure of the data. In particular, they depend on the separation between the current elements of the two streams. For more details, see [2]. However, the contribution here is that both algorithms are now adjusted so that they produce spatially sorted blocks, and hence the algorithms can be used anywhere in a query pipeline.

6 Conclusions

Providing sorted output for the spatial join operation is of significant importance to spatial query processing of complex spatial queries. Since the modified spatial join algorithms, presented in this paper, produce spatially sorted output, they can be used in an environment where the queries are cascaded without the need to sort the output after every intermediate operation as is the case when using the spatial merge algorithm.

Spatial join algorithms have a variety of requirements and characteristics. Some need indexed input and produce unindexed unsorted output, while others need only sorted input but produce sorted output, etc. We plan to model and study the cost of alternative query evaluation plans that contain cascaded spatial joins and study the effect of the characteristics and requirements of the various spatial join algorithms on the overall cost of the plans.

7 Acknowledgements

The second author gratefully acknowledges the support of the National Science Foundation under Grant IRI-92-16970.

References

[1] W. G. Aref and H. Samet. Uniquely reporting spatial objects: Yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, August 1992.

[2] W. G. Aref and H. Samet. The spatial filter revisited. In *6th international Symposium on Spatial Data Handling*, Edinburgh, Scotland, September 1994.

[3] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *The International Conferance in Knowledge Management*, pages 347–354, Gaithersburg, MD, November 1994.

[4] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proceedings of the 9th International Conference on Data Engineering*, pages 190–197, Vienna, Austria, April 1993.

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

[6] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 237–246, Washington, DC, May 1993.

[7] Volker Gaede. Geometric information makes spatial query processing more efficient. In *Proc. 3rd ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, pages 45–52, Baltimore, Maryland, USA, 1995.

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

step	current(X)	current(Y)	stack of X	stack of Y	action
start	A	I	W	W	
1 (1)	B	I	A W	W	push A on stack of X and set current(X) to B candidate in Y: I
	C	I	A W	W	scan X to C
2 (4)	C	J	A W	W	push I on stack of Y and report pair (A,I)
3 (5)	C	J	A W	W	pop I from stack of Y
4 (6)	C	J	W	W	pop A from stack of X
5 (7)	D	J	C W	W	push C on stack of X and set current(X) to D candidates in Y: F's analog and J
	E	J	C W	W	scan X to E
	F	J	C W	W	scan X to F
6 (12)	F	J	W	W	pop C from stack of X
7 (13)	G	J	F W	W	push F on stack of X and set current(X) to G candidate in Y: J
	H	J	F W	W	scan X to H
8 (16)	H	K	F W	J W	push J on stack of Y and report pair (F,J)
9 (17)	H	K	W	J W	pop F from stack of X
10 (18)	H	K	W	W	pop J from stack of Y
11 (19)	H	L	W	K W	push K on stack of Y
12 (20)	W	L	H W	K W	push H on stack of X and report pair (H,K)
13 (21)	W	W	H W	L K W	push L on stack of Y and report pair (H,L)
14 (22)	W	W	W	L K W	pop H from stack of X and
end	W	W	W	L K W	X and its stack are empty

Figure 8: Trace of the application of the linear-scan algorithm to the Morton elements in Figures 3a and 3b. Step numbers within parentheses correspond to the step numbers in the trace of the spatial merge algorithm in Figure 4.

[9] O. Günther. Efficient computation of spatial joins. In *Proceedings of the 9th International Conference on Data Engineering*, pages 50–59, Vienna, Austria, April 1993.

[10] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

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

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

[13] J. A. Orenstein. Algorithms and data structures for the implementation of a relational database system. Technical Report SOCS-82-17, School Comput. Sci., McGill Univ., Montreal, Quebec, Canada, 1983.

[14] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 294–304, Portland, OR, June 1989.

[15] J. A. Orenstein and F. A. Manola. Spatial data modeling and query processing in PROBE. Technical Report CCA-86-05, Computer Corporation of America, Cambridge, MA, October 1986.

[16] 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, May 1988.

[17] J. A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 181–190, Waterloo, Canada, April 1984.

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

[19] Doron Rotem. Spatial join indices. In *Proceedings of the 5th International Conference on Data Engineering*, pages 500–509, Kobe, Japan, April 1991.

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

[21] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, Washington, DC, May 1986.

8 Appendix: Detailed Implementations of the Spatial Join Algorithms

Each spatial join algorithm is given by the collection of procedures `Control`, `NextElement`, `EnterElement`, `ExitElement`, and `Advance`. Procedures `Control`, `NextElement`, `EnterElement`, and `ExitElement` are the same for each of the algorithms that we describe. The difference is in the encoding of procedure `Advance` which produces the next elements in the input streams. We differentiate between the different algorithms through the use of

procedures SpatialMergeAdvance, SpatialFilterAdvance, ScanCascadeAdvance, and EstimateBasedAdvance.

The spatial join algorithms make use of the following data types. The principal data type is a Morton element element represented by a record of type element. StateOfInput identifies the state of each input stream and is a record of three fields: sequence, nest, and current. sequence is a list of Morton element elements. nest is a stack of Morton elements. current is the current Morton element. output is a list of pairs of elements (corresponding to overlapping Morton elements) consisting of fields element1 and element2 which are pointers to records of type element.

```
procedure Control(LeftSequence, RightSequence, Result);
/* Perform the spatial join operation by using a variant of merge join.
  The two input streams are assumed to be sorted but not necessarily
   indexed. A stack is used to maintain information about the nest-
ing of
   Morton blocks in each input stream. */
begin
  reference pointer list LeftSequence, RightSequence;
  reference pointer output Result;
  pointer StateOfInput L,R;
  integer event;
  global pointer element MaxMortonElement;
  /* Initialize the left and right streams and their state variables.
     MaxMortonCode is the maximum possible value for the
     Morton code in the space in which the spatial join is executed.
     MaxMortonElement is a constant unit-sized Morton element that
     lies outside the space in which the spatial join is executed.
     The stacks are initialized to contain this element, and it is
     also the last element in each stream. */
  MaxMortonElement:=create(element);
  zlo(MaxMortonElement):=MaxMortonCode+1;
  zhi(MaxMortonElement):=MaxMortonCode+1;
  L:=create(StateOfInput);
  R:=create(StateOfInput);
  sequence(L):=LeftSequence;
  clear(nest(L));
  push(nest(L),MaxMortonElement);
  current(L):=first(sequence(L)); /* Assume sequence is non-empty */
  sequence(R):=RightSequence;
  clear(nest(R));
  push(nest(R),MaxMortonElement);
  current(R):=first(sequence(R)); /* Assume sequence is non-empty */
  while not(current(L)=MaxMortonElement and
          top(nest(L))=MaxMortonElement) and
       not(current(R)=MaxMortonElement and
          top(nest(R))=MaxMortonElement) do
    begin
     /* The main loop of spatial join scans each element of both
        streams. zlo(B) returns the Morton code of the upper-left
        corner of B. zhi(B) returns the Morton code of the lower-
        right corner of B. */
     event:=min(zlo(current(L)), zhi(top(nest(L))),
               zlo(current(R)), zhi(top(nest(R))));
     /* depending on the minimum event, perform the appropriate
        action. */
     if event=zlo(current(L)) then /* enter current(L) into L's */
     EnterElement(L,R,Result)   /* stack and advance to next element.*/
     else if event=zlo(current(R)) then /* enter current(R) into R's */
     EnterElement(R,L,Result) /* stack and advance to next element.*/
    else if event=zhi(top(nest(L))) then /* delete elements from L's stack */
       ExitElement(L,R,Result) /* and produce the output pairs. */
    else if event=zhi(top(nest(R))) then /* delete elements from L's stack */
       ExitElement(R,L,Result); /* and produce the output pairs. */
    end;
end;

pointer element procedure NextElement(X);
/* Return a pointer to the next element in stream X.  If the stream
  is empty, then return a pointer to MaxMortonElement which is a
  global constant. */
begin
  reference pointer StateOfInput X;
  return(if eof(sequence(X)) then MaxMortonElement
        else next(sequence(X)));
end;

procedure EnterElement(X,Y,Result);
/* Enter the current element of stream X (current(X)) into X's
  stack (nest(X)). */
begin
  reference pointer StateOfInput X,Y;
  reference pointer output Result;
  push(nest(X),current(X));
  ReportPairs(top(nest(X)),nest(Y),Result);
  Advance(X,Y);  /* advance to the next element in stream X. */
end;

procedure ExitElement(X,Y,Result);
/* Remove the top element of the X's stack after comparing it with all
  the elements of Y's stack. The elements that overlap are reported as
   output of the spatial join. */
begin
  reference pointer StateOfInput X,Y;
  reference pointer output Result;
  pop(nest(X));
end;

procedure SpatialMergeAdvance(X,Y);
/* Assign to the current element of X the next element of stream X.
  Y is unchanged in this version of the procedure. */
begin
  reference pointer StateOfInput X,Y;
  current(X):=NextElement(X);
end;

procedure SpatialFilterAdvance(X,Y);
/* Assign to current(X) an element in stream X that overlaps current(Y)
  or with some element in nest(Y). Skip the elements in stream X that
  lie between current(X) and current(Y) that do not correspond to any
   output pair of the spatial join. */
begin
  reference pointer StateOfInput X,Y;
  pointer element c;
  stack of element candidates;
  Boolean CoverFound;
  current(X):=NextElement(X);
  if zhi(current(X))$"ge$zlo(current(Y)) then
    return  /* current(X) overlaps with or is past current(Y) */
  else if StackContains(nest(Y),current(X)) then
    return  /* current(X) is one of the elements in nest(Y) or
           is contained in one of the elements in nest(Y) */
  else /* Check for nodes in stream X that contain current(Y) */
    begin
     clear(candidates); /* Initialize the stack */
     c:=current(Y);
     while zhi(current(X));zlo(c) do /* Calculate the ancestors of */
       begin                    /* current(Y) between current(X) */
      push(candidates,c);      /* and current(Y). No I/O is involved.*/
      c:=parent(c);  /* parent(c) is the Morton element that contains c
                      and is twice the size of c. */
       end;
    /* Search stream X using the zlo field value for each of the computed
       ancestors starting with the larger ones. This involves one direct
        access to stream X for each ancestor. */
     CoverFound:=false;
     while not(empty(candidates)) and not(CoverFound) do
     begin  /* randac(sequence(X),k) performs random access to stream X
              using key k to search the index. If k is not found in the
             index, then return the smallest key greater than k in the
         index. This could be MaxMortonElement if we are at the end
              of the stream. */
       c:=randac(sequence(X),zlo(top(candidates)));
       if same element(c,top(candidates)) then
         begin
          CoverFound:=true;
          current(X):=c;
         end
       else pop(candidates);
      end;
     if not(CoverFound) then current(X):=c;
       /* At this point, current(X) is assigned to one of the follow-
ing values:
       (a) the largest ancestor of current(Y) found in stream X past
       current(X), (b) or in the case where no ancestor is found, the
     smallest key greater than or equal to current(Y). In either case,
       the irrelevant elements in stream X are skipped. */
     return;
    end;
end;
```

```
procedure ScanCascadeAdvance(X,Y);
begin
  reference pointer StateOfInput X,Y;
  pointer element c;
  stack of element candidates;
  Boolean CoverFound;
  current(X):=NextElement(X);
  if zhi(current(X))$"ge$zlo(current(Y)) then
    return  /* current(X) overlaps with or is past current(Y) */
  else if StackContains(nest(Y),current(X)) then
    return  /* current(X) is in nest(Y) */
  else /* Check for nodes in stream X that contain current(Y) */
    begin
      clear(candidates);
      c:=current(Y);
      while zhi(current(X))¡zl'o(c) do
        begin
          push(candidates,c);
       c:=parent(c); /* parent(c) is the Morton element that contains c
                        and has double the size of c */
        end;
      CoverFound:=false;
      while not(empty(candidates)) and not(CoverFound) do
      begin /* Perform the linear scanning until an ancestor is found */
          if zlo(current(X))¡zlo(top(candidates)) then
            current(X):=NextElement(X)
          else if same element(current(X),top(candidates)) then
            CoverFound:=true
          else pop(candidates);
        end;
      if not(CoverFound) then
        while not(current(X)=MaxMortonElement)
              and zlo(current(X))¡zlo(current(Y)) do
          current(X):=NextElement(X);
      return;
    end;
end;
```