Incremental Distance Join Algorithms for Spatial Databases*

Gísli R. Hjaltason and Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
grh@cs.umd.edu and hjs@cs.umd.edu

Abstract

Two new spatial join operations, *distance join* and *distance semi-join*, are introduced where the join output is ordered by the distance between the spatial attribute values of the joined tuples. Incremental algorithms are presented for computing these operations, which can be used in a pipelined fashion, thereby obviating the need to wait for their completion when only a few tuples are needed. The algorithms can be used with a large class of hierarchical spatial data structures and arbitrary spatial data types in any dimensions. In addition, any distance metric may be employed. A performance study using R-trees shows that the incremental algorithms outperform non-incremental approaches by an order of magnitude if only a small part of the result is needed, while the penalty, if any, for the incremental processing is modest if the entire join result is required.

1 Introduction

The spatial join operation is similar to the join operaion in relational databases. It is defined on two sets of objects, and computes a subset of the Cartesian product of the two sets, determined by a spatial predicate, which prescribes a certain spatial relationship between the objects in the result. The most common spatial predicate is *intersect*, i.e., the geometry of the objects are required to intersect [1, 7, 8, 19, 21, 22, 23]. A generalization of this is *within*, where the objects are required to lie within some distance of each other [25, 30]. Other spatial predicates have been considered as well, and general methods to compute a spatial join proposed [4, 14]. Some of these methods involve special *join indexes* [14, 25].

In this paper, we define a "distance join" operation, which computes a subset of the Cartesian product of sets A and B, and specifies an *order* on the result, based on distance. The distance is usually defined in terms of spatial attributes, but this need not be the case. When the distance of the resulting pairs is limited to a range, we have a generalization of a spatial join based on a within predicate. The "distance semi-join" is a useful special case of the distance join which for each object in A finds the nearest object in B. Figure 1 defines the distance join and distance semi-join operations using a syntax loosely adapted from SQL-92, including the `STOP AFTER` clause extension proposed in [10]. The `WHERE` and `STOP AFTER` clauses, specifying limits on the distance and/or the number of result tuples, are optional. These basic queries could be made more complicated by adding further selection conditions in the `WHERE` clause.

```
SELECT *
FROM R1, R2, distance(R1.s1, R2.s2) d
[WHERE d >= <dmin> AND d <= <dmax>]
ORDER BY d
[STOP AFTER <n>]
```

(a)

```
SELECT *, min(d)
FROM R1, R2, distance(R1.s1, R2.s2) d
[WHERE d >= <dmin> AND d <= <dmax>]
GROUP BY R1.s1
ORDER BY d
[STOP AFTER <n>]
```

(b)

Figure 1: Definition of (a) distance join and (b) distance semi-join using SQL.

The distance join and distance semi-join have numerous useful applications in spatial databases. For example, given a spatial database of rivers and cities, we can use partial computation of them to "find the city nearest to any river", "find the city nearest to any river, such that the city has a population of more than 5 million", and "find cities within 5 miles of any river". The distance semi-join is useful as a clustering operation. For example, suppose we are given two relations consisting of the locations of stores and of warehouses, respectively, and for each store we wish to determine the closest warehouse. This is achieved by taking the distance semi-join of the stores relation with the warehouse relation. The distance semi-join works by reporting the (store,warehouse) pairs in order of distance. Note that once we have determined the closest warehouse to a particular

*This work was supported in part by the National Science Foundation under Grant IRI-9712715 and the Department of Energy under Contract DEFG0295ER25237.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '98 Seattle, WA, USA
© 1998 ACM 0-89791-995-5/98/006...$5.00

store, that store does not participate in other tuples with the remaining warehouses.

Computing the complete distance semi-join yields a clustering of the stores. In fact, for point data, the result partitions the space in a manner analogous to a discrete Voronoi diagram, i.e., each point in the stores relation is associated with the closest point in the warehouse relation (thus, in the terminology of Voronoi diagrams, the locations of the warehouses are the sites). The attractiveness of this analogy lies in providing users a mechanism to perform a geometric operation such as the Voronoi diagram using a data base primitive without having to invoke a special purpose algorithm from a geometric library to perform the operation. Note that this operation is not symmetric. In particular, the result of computing the distance semi-join of the warehouse relation and the stores relation is that for each warehouse, we get the closest store.

The *clustering join* [33] is similar to the distance semi-join with the difference being that the clustering join is symmetric. An algorithm for computing the clustering join is also given in [33]. However, that algorithm is not well suited for spatial data that resides in d-dimensional Euclidean space. The reason is that [33] deals with more general objects—such as patterns, strings, trees, graphs, etc.—whose internal structure is unknown as far as the algorithm is concerned. The only knowledge about the objects comes from a distance measure that returns the distance between two objects. Furthermore, the distance measures are assumed to be expensive to compute, so that the overall goal is to compute as few distances as possible. In contrast, spatial data allows the use of spatial indexes which in effect summarize the data and enable avoiding many distance calculations (which, however, are not necessarily the most expensive component of query algorithms involving distances).

In this paper we present incremental algorithms for computing the distance join and distance semi-join in the sense that the pairs resulting from the corresponding operation are reported one-by-one. This enables the query processor to use the algorithms in a pipelined fashion. Furthermore, the algorithms aim to deliver results as soon as possible. Such "fast first" pipelined join methods have recently become a focus of attention [3, 34]. They have become important in enabling the development of more user friendly and interactive interfaces to database systems [16]. Recent proposals for extending SQL [10] also benefit greatly from the presence of such algorithms.

A variation of our incremental distance join algorithm can be used to compute intersecting pairs [31], closest pair [6], and all nearest neighbors [2, 11, 32] in a set of objects. While our incremental distance join algorithm may not always be competitive with some of the above algorithms in terms of computational complexity, it may nevertheless be a reasonable alternative given that a spatial data structure has already been built. In addition, unlike most of these methods, it is not limited to point or rectangle objects.

The rest of this paper is organized as follows. Section 2 describes the incremental algorithms for computing the distance join and distance semi-join. Section 3 describes the environment in which we perform our experiments, and Section 4 presents the results. Section 5 concludes with a number of future tasks.

2 Incremental Distance Join Algorithms

In this section we describe our incremental distance join algorithm. Although our algorithm is general in the sense that it can be used with most spatial data structures, for concreteness we present it in the context of the R-tree. Also, performance tests were conducted with R-trees (see Section 4). The rest of this Section is organized as follows. Section 2.1 reviews the R-tree. Section 2.2 describes the basic incremental algorithm for the distance join, followed by an outline of a number of methods for extending its functionality as well as improving its performance. Section 2.3 presents modifications to the basic algorithm to enable it to compute the distance semi-join operation.

2.1 R-trees

The R-tree [15] (see Figure 2) is one of many proposed spatial data structures. It is an object hierarchy in the form of a balanced structure inspired by the B^+-tree [12]. Each R-tree node contains an array of *(key, pointer)* entries where *key* is a hyper-rectangle that minimally bounds the data objects in the subtree pointed at by *pointer*. In an R-tree leaf node, the *pointer* is an object identifier (e.g., a tuple ID in a relational system), while in a non-leaf node it is a pointer to a child node on the next lower level. The maximum number of entries in each node is termed its *node capacity* or *fan-out* and may be different for leaf and non-leaf nodes. The node capacity is usually chosen so that a node fills up one (or a small number of) disk pages. R-trees can be used to index a space of arbitrary dimension and arbitrary spatial objects rather than just points.

As described above, R-tree leaf nodes contain a minimal bounding rectangle and an object identifier for each object in the node, i.e., the geometric description of the objects is stored external to the R-tree itself. Another possibility is to store the actual object, or only its geometric description, in the leaf instead of the bounding rectangle. This is usually only useful if the object representation is relatively small (e.g., similar in size to a bounding rectangle) and is fixed in length. If the entire object data (i.e., all relevant attributes) are stored in the leaf nodes, then the object identifiers need not be stored. The disadvantage of this approach is that objects will not have a fixed address, as some objects must be moved upon each R-tree node split.

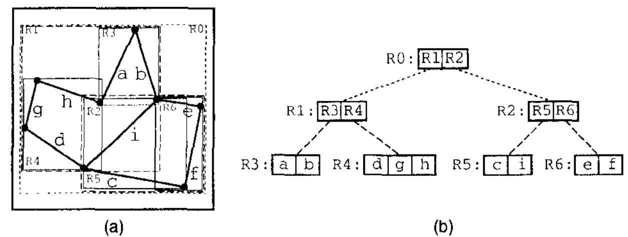

Figure 2: An R-tree for a set of 9 line segments. (a) Spatial rendering of the line segments and bounding rectangles, and (b) a tree access structure for (a). Bounding rectangles for individual line segments are omitted from (a) in the interest of clarity.

We make use of an R-tree variant called the R*-tree [5]. It differs from the conventional R-tree in employing a more sophisticated insertion and node-splitting algorithms that attempt to minimize a combination of overlap and area increase between minimum bounding rectangles.

2.2 Computing Distance Join

Our incremental distance join algorithm may be viewed as simultaneously applying an incremental nearest neighbor algorithm [18] (see [17] for the application of a similar approach to the LSD tree) to the two spatial data structures corresponding to the spatial attributes of the joined relations. The algorithm works for any spatial data structure based on a hierarchical decomposition. In our description, we assume a spatial data structure that forms a tree structure, where each tree node represents some region of space and where objects (or pointers to them in external storage) are stored in the leaf nodes whose region intersects the objects. Further, we assume that each object is stored in only one leaf. We handle both

the case that the objects are stored directly in the leaf as well as the case that the leaf nodes contain the minimum bounding rectangles of objects along with a pointer to the actual object representation. This set of assumptions was chosen as it holds for the R-tree. However, the algorithm can be easily adapted to handle most spatial data structures that do not satisfy these assumptions, such as the hB-tree [24] (which forms a directed acyclic graph), and quadtrees [27, 28] (where non-point objects may be stored in more than one leaf node). In the remainder of this section, we do not make a distinction between a node and the region that it represents; the meaning should be clear from the context.

The input to the incremental distance join algorithm is two spatial indexes, R_1 and R_2. The algorithm maintains a set of pairs P, with one item from each of R_1 and R_2, each item being either a node or an object. Initially, P contains just one pair corresponding to the root nodes of R_1 and R_2. We obtain the set of all pairs, i.e., the Cartesian product of the sets of objects in R_1 and R_2, as follows. As long as P contains a pair p with at least one item being a node, replace p in P by all the pairs resulting from replacing the node by its entries (child nodes for non-leaf nodes, objects for leaf nodes). It is intuitively obvious that this process will result in P containing the set of all pairs. The algorithm essentially computes P in this way, but processes the pairs in P in order of their distance, thereby attempting to report object pairs as soon as possible.

The algorithm works for data objects of arbitrary type and dimension (although our experiments use two-dimensional points), provided that consistent distance functions are used. Four distance functions are needed: one between objects of each collection, two between objects of one collection and nodes of the spatial index of the other collection, and one between nodes of each spatial index. More accurately, the functions we need are $d_{oo}(o_1, o_2)$, $d_{on}(o_1, n_2)$, $d_{no}(n_1, o_2)$, and $d_{nn}(n_1, n_2)$, where o_1 and n_1 are an object and a node from R_1, respectively, and o_2 and n_2 are an object and a node from R_2, respectively. If the leaf nodes store minimum bounding rectangle for objects, then the functions d_{on} and d_{no} are not required. Instead, we need the functions $d_{bn}(b_1, n_2)$, $d_{nb}(n_1, b_2)$, in addition to $d_{bb}(b_1, b_2)$, where b_1 and b_2 denote a minimum bounding rectangle for objects in R_1 and R_2, respectively. If node regions are rectangles, then d_{nn} can serve the purpose of all three functions.

Usually, the distance functions are all based on a distance metric for points, $d(p_1, p_2)$, such as the Chessboard, Manhattan or Euler metrics. However, this need not be the case. As long as the distance functions are "consistent", the algorithm will function correctly. Informally, by consistent, we mean that no pair can have a smaller distance than a pair that gives rise to it during the processing of the algorithm. For example, if o_1 and o_2 are objects in R_1 and R_2, respectively, and n_1 is a leaf node that contains R_1, then we must have $d_{oo}(o_1, o_2) \geq d_{no}(n_1, o_2)$. If the distance functions are all based on the same metric, this condition will hold due to the triangle inequality property. In what follows, we usually refer to the distance functions collectively with the symbol d, as the particular distance function to be used can be inferred from the context.

2.2.1 Basic Algorithm

We first describe the basic version of the algorithm, and then introduce extensions to it as well as ways to improve its performance. The heart of the algorithm is a priority queue, where each element contains a pair of items, one from each of the input spatial indexes R_1 and R_2. An item can be either a data object or a node, so there are four kinds of possible pairs, node/node, node/object, object/node, and object/object. If object bounding rectangles (abbreviated by *obr*) are stored in leaves, then these will become the third type of pair items, resulting in nine possible kinds of pairs, of which we use five: node/node, node/obr, obr/node, obr/obr, and object/object[1]. The key used to order the queue elements is the distance between each pair. We later discuss how to handle ties, i.e., how to order pairs with equal distance.

At each step in the algorithm, the element at the head of the priority queue is retrieved, i.e., the element with the smallest distance key. If the element stores a pair of data objects, then the pair is reported as the next closest pair. No pair that is subsequently reported will have a smaller distance due to this pair having the smallest key in the queue. Furthermore, the consistency constraints on the distance functions guarantee that no pair on the queue will result in generating a pair of data objects with a smaller distance[2]. If one of the items in the dequeued element is a node, then the algorithm pairs up the entries of the node (objects for leaf nodes, sub-nodes for non-leaf nodes) with the other item.

The basic algorithm is presented in Figure 3 for the case that the leaf nodes of the spatial indexes contain object bounding rectangles. In the figure, item 1 in a queue element is from R_1, while item 2 is from R_2. The INCDISTJOIN procedure contains the high level control structure for the algorithm, while procedures PROCESSNODE1 and PROCESSNODE2 enqueue new pairs for each entry in a node from R_1 and R_2, respectively. In lines 6 and 11 of INCDISTJOIN, the next closest pair of objects is reported. The entire state of the algorithm is represented by the priority queue. Thus, at this point, control can be passed to the process that invoked the incremental distance join algorithm, which may or may not decide to retrieve more pairs. If one of the items in the dequeued element is a node, then one of the procedures PROCESSNODE1 and PROCESSNODE2 is called. This version of the algorithm arbitrarily chooses to call PROCESSNODE1 if both items are nodes.

In line 4 of PROCESSNODE1, $[O]$ denotes the bounding rectangle of O (note that in practice the object reference must be enqueued along with the bounding rectangle). If the object geometry is represented directly in the leaf nodes, then the actual objects would be used here instead of the bounding rectangles. Also, in this case, the **if** statement in line 7 of INCDISTJOIN would not be needed.

The connection of the incremental distance join to our incremental nearest neighbor algorithm [18] is easy to see from Figure 3, as PROCESSNODE1 and PROCESSNODE2 are essentially the same as the basic loop of the nearest neighbor algorithm. In particular, in PROCESSNODE1, item 2 serves the role of the query object.

2.2.2 Priority Queue Ordering and Tree Traversal

The key for ordering the priority queue of pairs is the distance between the items. An important question is how to break ties for pairs with the same distance. Different choices will lead to vastly different traversal patterns. Since our goal is to produce result pairs as soon as possible, it is obvious that we want to order pairs containing objects or object bounding rectangles ahead of (i.e., with greater priority than) pairs of nodes. Furthermore, given two pairs with nodes, the pair containing nodes at a deeper level is given a higher priority. This leads to a depth-first-like traversal pattern of the tree hierarchy of the spatial indexes for pairs having the same distance (a version using this approach is termed "DepthFirst" in Section 4.1.1). Alternatively, if nodes at a higher level are given priority, a breadth-first-

[1] Note that objects only appear in one of the combinations that we allow in order to reduce the number of accesses to the object storage. With our scheme, each object must be accessed at most once for each object/object pair.

[2] A pair $\langle i_1, i_2 \rangle$ is said to be generated from a pair $\langle i'_1, i'_2 \rangle$ if the pair $\langle i_1, i_2 \rangle$ results from a sequence of algorithm operations starting with $\langle i'_1, i'_2 \rangle$. As an example, all object/object pairs are ultimately generated from the initial pair of root nodes.

```
INCDISTJOIN(R1, R2)
 1 Q ← NEWPRIORITYQUEUE()
 2 ENQUEUE(Q, 0, ⟨ROOTNODE(R1), ROOTNODE(R2)⟩)
 3 while not ISEMPTY(Q) do
 4   Elem ← DEQUEUE(Q)
 5   if both items in Elem are data objects then
 6     Report Elem
 7   else if both items are object bounding rectangles then
 8     let O1 and O2 be the corresponding object references
 9     D ← DIST(O1, O2)
10     if ISEMPTY(Q) or D ≤ FRONT(Q).DIST then
11       Report ⟨O1, O2⟩
12     else
13       ENQUEUE(Q, D, ⟨O1, O2⟩)
14     endif
15   else if item 1 in Elem is a node then
16     PROCESSNODE1(Q, Elem)
17   else
18     PROCESSNODE2(Q, Elem)
19   endif
20 enddo

PROCESSNODE1(Q, Elem)
 1 Node ← item 1 of Elem
 2 Item2 ← item 2 of Elem
 3 if Node is a leaf node then
 4   for each entry [O] in Node do
 5     ENQUEUE(Q, DIST([O], Item2), ⟨[O], Item2⟩)
 6   enddo
 7 else
 8   for each Child node of Node do
 9     ENQUEUE(Q, DIST(Child, Item2), ⟨Child, Item2⟩)
10   enddo
11 endif

PROCESSNODE2(Q, Elem)
 1 Same as PROCESSNODE1, with items 1 and 2 exchanged
```

Figure 3: Basic version of incremental distance join algorithm where leaf nodes contain bounding rectangles.

like traversal would result (termed "BreadthFirst" in Section 4.1.1). This could be of advantage if we wanted to compute a large portion of the distance join operation (i.e., generate a very large number of pairs), as it would in certain cases enable the algorithm to better schedule node and object accesses [21]. However, given our usage assumptions, much of the work may be wasted, as a breadth-first traversal would require processing all pairs at one level before any pairs at the next level are considered.

In the version of the INCDISTJOIN procedure that we presented in Figure 3, when the dequeued pair contains two nodes, $\langle n_1, n_2\rangle$, node n_1 is arbitrarily chosen to be processed (i.e., its entries examined) rather than n_2. This is not a good strategy, as it will cause R_1 to be traversed down to the leaf level before the root of R_2 is processed. A better strategy would attempt to traverse the two indexes more evenly so that the level of the nodes in node/node pairs does not differ by much. This is done by choosing to process the node that is at a shallower depth. If both nodes are at the same level in their respective trees, then the algorithm chooses to process the node whose region has a larger area. Although the strategy that we have outlined is not always the best one, our experiments have shown it to perform well overall.

An alternative to processing only one of the nodes for node/node pairs is to process both simultaneously (termed "Simultaneous" in Section 4.1.1). This is more in line with traditional spatial join algorithms [8, 21]. In fact, if this is done, then many of the optimization techniques developed for spatial join can be applied [8], such as the usage of plane sweep and the restriction of the search space. The idea is that when processing pair $\langle n_1, n_2\rangle$, we first mark the entries in n_1 that are within the specified distance range (see Section 2.2.3) from the space spanned by n_2, and similarly for the entries in n_2 we mark the ones that are within the specified distance range from the space spanned by n_1. This serves to eliminate entries that cannot possibly become members of any of the new pairs. Next, a plane sweep along one of the axis is used to pair up the entries in the two nodes (which have previously been sorted along that axis). Figure 4 illustrates the plane-sweep process, where r1 and r2 are entries in n_1, and s1, s2, s3 and s4 are entries in n_2. Without plane sweep, r1 would have to be checked for intersection with all the entries in n_2, but with plane sweep we only have to check intersection of r1 with s1 and s2. The plane-sweep algorithm given in [8] has to be modified to work for a non-zero maximum distance (recall that [8] focuses on spatial join with the intersection predicate). For example, if the rectangle currently being used has the coordinate range (x_1, x_2) along the sweep axis, then the algorithm must sweep along the entries in the other node up to the coordinate value $x_2 + D_{\max}$, where $D_{\max}$ is the maximum distance. As an example, in Figure 4, we would have to check whether s3 is within the proper distance of r1, in addition to s1 and s2.

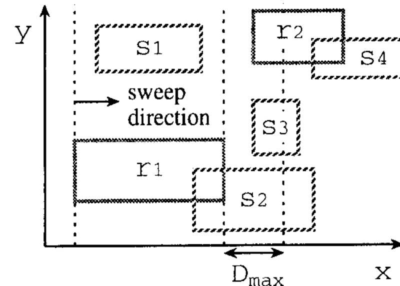

Figure 4: Plane sweep along x-axis over the entries in two nodes.

Processing both nodes simultaneously for node/node pairs is not always better than processing only one node as in our original formulation. Intuitively, it seems likely that the optimizations that it affords will only yield significant benefits if the distance range is rather narrow. As an extreme case, if the minimum is 0 and the maximum is unbounded, then all possible pairs of entries from the two nodes will have to be generated, a total of $|n_1| \cdot |n_2|$ ($|n|$ denotes the number of entries in node n). In contrast, if only one of the nodes is processed, say n_1, then only $|n_1|$ pairs will result. All of these pairs may have a greater distance than the next closest pair. Thus, in best case, only $|n_1|$ pairs are generated from $\langle n_1, n_2\rangle$ with our original formulation of the incremental distance join algorithm before the next object pair is reported. The downside, of course, is that processing only one node at a time may lead to each node being accessed more times from disk when the algorithm has to compute many object pairs.

If both nodes in node/node pairs are processed simultaneously, then the incremental distance join algorithm resembles somewhat the spatial join algorithm introduced in [21]. The difference is that [21] is breadth-first and is limited to finding intersecting object pairs, although it would be straightforward to generalize it to compute a spatial join with a within predicate (however, unlike with our algorithm, if the pairs are desired in order of distance, then the en-

tire result would have to be computed and sorted before the first pair can be reported).

Some widely used spatial data structures form unbalanced tree hierarchies (e.g., quadtrees [28] and the buddy-tree [29]). Bounding rectangles are not always present in the leaf nodes of these structures, even when objects are not represented directly in the leaves (i.e., the leaves only contain pointers to the objects). If this is the case, then it is better to defer processing leaf nodes until both items in node/node pairs are leaf nodes, at which time both leaf nodes are processed simultaneously. This strategy will tend to reduce the number of times each object needs to be accessed from disk.

2.2.3 *Distance Range*

A shortcoming of the algorithm as stated in Section 2.2.1 is that a very large number of pairs will be inserted into the priority queue, even when computing a modest number of object pairs for relatively small object relations. Most of the pairs inserted in the priority queue will have a large distance, and will most likely never be retrieved from the queue unless a very large number of object pairs is requested. However, for object relations of non-trivial size, the number of pairs in the Cartesian product of the two relations (recall that a full distance join operation computes the Cartesian product) is immense. For example, for two relations with 50,000 objects each, the Cartesian product contains 2.5 billion pairs. Typical queries will only require computing a very small fraction of this high number. Thus, it is unlikely that pairs with a large distance are ever retrieved from the queue. The large number of pairs put on the queue and never requested occupies a great deal of memory space and slows down queue operations[3]. Thus, we need a way of limiting the number of pairs inserted into the queue. One way of doing so is to impose a maximum distance on object pairs. Any pair that has a distance larger than the maximum can be rejected, as no object pair with less distance can be derived from it (this is guaranteed by the consistency of the distance functions).

Above, we have established the need to be able to impose a maximum on the distance of object pairs. In addition, it may be useful for some queries to impose a minimum on the distance of object pairs. The incremental distance join algorithm is easily modified so that it limits the distance of the pairs that are returned to a range of values. In order to effectively prune pairs based on a minimum distance, we need functions that compute an upper bound on the distance of any object pair that can be generated from any pair $\langle i_1, i_2 \rangle$ (such a function is clearly not needed for object/object pairs). In other words, for any object pair $\langle o_1, o_2 \rangle$ generated from $\langle i_1, i_2 \rangle$, we have $d(o_1, o_2) \leq d_{\max}(i_1, i_2)$, where $d_{\max}$ is the upper bound function appropriate for $\langle i_1, i_2 \rangle$. This means that if $d_{\max}(i_1, i_2)$ is smaller than the minimum, then we can discard the pair $\langle i_1, i_2 \rangle$, as no object pair with a distance larger than the minimum will be generated from $\langle i_1, i_2 \rangle$.

Now, the question is how to compute $d_{\max}$ for the various types of pairs. For pairs of nodes, $\langle n_1, n_2 \rangle$, we have $d_{\max}(n_1, n_2) = \max_{p_1 \in n_1, p_2 \in n_2} d(p_1, p_2)$. For a node/object pair $\langle n_1, o_2 \rangle$, we have $d_{\max}(n_1, o_2) = \max_{p_1 \in n_1} d(p_1, o_2)$, and similar for object/node pairs. The functions for node/obr, obr/node, and obr/obr pairs can be defined in a similar manner as for node/node pairs. However, a closer approximation to the upper bound is possible for these types of pairs through the use of a distance metric that has been termed MINMAXDIST [26]. The object bounding rectangles are required to minimally bound the objects. The key idea behind the MINMAXDIST metric is that if b is the d-dimensional minimum bounding rectangle of object o, then each of the $d-1$ dimensional faces[4] of b must touch o at some point. Thus, given a point p, we have $d(p, o) \leq \max_{p_f \in f} d(p, p_f)$, for each $f \in F(b)$, where $F(b)$ denotes the set of faces of b. The face f causing the right hand side of the inequality to reach its minimum is the best approximation of $d(p, o)$ given the bounding rectangle b, so the function computing the MINMAXDIST for a point and a bounding rectangle is $d_{\mathrm{mm}}(p, b) = \min_{f \in F(b)}(\max_{p_f \in f} d(p, p_f))$. A practical way of computing the value of $d_{\mathrm{mm}}(p, b)$ is to first compute the maximum distance from p to a vertex of b, say $v_{\max}$, and then to determine the vertex adjacent to $v_{\max}$ (i.e., along an edge) that is closest to p [26]. Now, we can define $d_{\max}(n_1, b_2) = \max_{p \in n_1} d_{\mathrm{mm}}(p, b_2)$, and similarly for obr/node pairs. The MINMAXDIST definition of $d_{\max}$ for two object bounding rectangles is more complicated: $d_{\max}(b_1, b_2) = \min_{f_1 \in F(b_1), f_2 \in F(b_2)}(\max_{p_1 \in f_1, p_2 \in f_2} d(p_1, p_2))$. The price of basing the $d_{\max}$ functions for pairs with at least one bounding rectangle on the MINMAXDIST metric is that they are more expensive to compute than the simpler $d_{\max}$ function for node/node pairs.

Figure 5 presents a version of PROCESSNODE1 that restricts distances to a range of values. The arguments *Min* and *Max* specify the minimum and maximum desired distance. MINDIST denotes the regular distance functions (i.e., DIST in Figure 3) while MAXDIST denotes the $d_{\max}$ functions. Again, this version of PROCESSNODE1 assumes that the leaf nodes of the spatial indexes store bounding rectangles. We must also modify the **if** statement in line 7 of the INCDISTJOIN procedure in Figure 3 to check that the distance D falls in the desired range. In case the object geometry were represented directly in the leaf nodes, then the actual objects would be used in line 4 of Figure 5. Also, if item 2 is an object, then MAXDIST is equivalent to MINDIST.

```
PROCESSNODE1(Q, Elem, Min, Max)
 1 Node ← item 1 of Elem
 2 Item₂ ← item 2 of Elem
 3 if Node is a leaf node then
 4   for each [O] in Node do
 5     if MAXDIST([O], Item₂) ≥ Min and
          MINDIST([O], Item₂) ≤ Max then
 6       ENQUEUE(Q, MINDIST([O], Item₂), ⟨[O], Item₂⟩)
 7     endif
 8   enddo
 9 else
10   for each Child node of Node do
11     if MAXDIST(Child, Item₂) ≥ Min and
          MINDIST(Child, Item₂) ≤ Max then
12       ENQUEUE(Q, MINDIST(Child, Item₂), ⟨Child, Item₂⟩)
13     endif
14   enddo
15 endif
```

Figure 5: Portion of incremental distance join algorithm with distance range restriction.

2.2.4 *Estimating Maximum Distance*

As we pointed out in Section 2.2.3, a reasonably narrow distance range (i.e., small interval between minimum and maximum distance) is crucial for the incremental distance algorithm to perform

[3] But see Section 3.2 for a description of a priority queue implementation that puts part of the queue on disk if its size is too large to fit in memory.

[4] In two dimensions, the faces are line segments.

well. However, it is often not practical to require the user to set a maximum distance. Furthermore, the maximum distance is likely to be greatly overestimated. It is therefore important to have another way of estimating the maximum distance, given some other information. One way of doing so is to set an upper bound on the number of pairs that the algorithm must compute. In many applications, especially involving interactive queries, a fairly low number of pairs are known to be needed. This is aided by query language extensions that enable limiting the number of tuples in the result of queries (e.g., the "STOP AFTER" clause proposed for the "SELECT" statement of SQL [10]).

Given that the algorithm must compute a maximum of K pairs, the algorithm can estimate the maximum distance based on the pairs that have been seen so far. Obviously, if K object/object pairs have been seen, then the pair with the largest distance among those K pairs will provide a lower bound on the maximum distance necessary to compute the K closest pairs. However, we can do better than this by also making use of other types of pairs (e.g., node/node pairs). In general, more than one object/object pair may be generated from a pair $\langle i_1, i_2 \rangle$. This means that much fewer than K pairs are sufficient for estimating the maximum distance of K object/object pairs.

In the following, $D_{\max}$ and $D_{\max}$ denote the minimum and maximum distance imposed on the pairs to be computed by the algorithm, d denotes a regular distance function (i.e., computing minimum distance between two items) and $d_{\max}$ denotes the functions computing the upper bound on the distance of any object pairs generated from a pair. If the query specifies no maximum on the distance, then $D_{\max}$ is initially ∞. Our goal is to reduce $D_{\max}$ as much as possible, given K, the maximum number of pairs requested. Whenever a pair $\langle i_1, i_2 \rangle$ is inserted into the priority queue, we show below how to use the pair for the purpose of estimating a lower value for $D_{\max}$. Doing this adds overhead to the algorithm, but unless K is very large, it reduces considerably the number of pairs inserted into the priority queue, and thereby improves the overall running time of the algorithm.

A pair $\langle i_1, i_2 \rangle$ is eligible to be used for estimating $D_{\max}$ if $d(i_1, i_2) \geq D_{\min}$ and $d_{\max}(i_1, i_2) \leq D_{\max}$. This guarantees that all object/object pairs generated from $\langle i_1, i_2 \rangle$ will have a distance in the range $[D_{\min}, D_{\max}]$. Since we cannot know in advance how many object pairs are generated from a pair $\langle i_1, i_2 \rangle$, we must instead determine a lower bound on this number. This can be derived from the minimum number of objects in the subtree of i_1 and i_2, assuming they are nodes (if they are objects or object rectangles, this number is one). The minimum number of objects in the subtree of a node can, in turn, be derived from the minimum fan-out and the height of the corresponding tree. For the R-tree, for example, the minimum fan-out of nodes is typically 40% of the maximum fan-out (except for the root node). A more aggressive strategy would result from using the expected number of pairs generated from $\langle i_1, i_2 \rangle$ based on the average node occupancy. However, if the number of pairs generated from $\langle i_1, i_2 \rangle$ is over-estimated, then this may lead to a value of $D_{\max}$ that is too small (i.e., smaller than the Kth object/object pair), thereby causing us to find less than K pairs which will force us to restart the query. The reason we need to restart is that the priority queue does not provide us any useful information as we will have pruned too many entries by our maximum distance heuristic.

The process for estimating K maintains a set of pairs M, each of which has been inserted in the priority queue but not retrieved from it. When an eligible pair (i.e., with the distance function values as specified above) is inserted into the priority queue, it is also inserted into M. If this causes the sum of the number of object/object pairs (actually, the lower bound of object/object pairs as described above) that can be generated for the pairs in M to be larger than K, then we remove pairs from M until this is not the case, setting $D_{\max}$ to the $d_{\max}$ value of the pair removed last. The pair to remove next is chosen based on the largest $d_{\max}$ value. When a pair is retrieved from the priority queue, we must also remove the pair from M if it is present. However, when reporting the next object/object pair, we can reduce the value of K by one.

The question is how to organize the set M. The operations that are performed on M, in addition to insertion, are to remove the pair with largest $d_{\max}$ as well as to remove a pair given the particular items in the pair. There is no single data structure that supports efficient execution of both of these operations. In our implementation, we chose to use a priority queue Q_M organized on the $d_{\max}$ values to support finding the largest value, and a hash table to support locating a particular pair. The hash table entries contain a pointer to the corresponding priority queue entry, thereby enabling deleting the entry from Q_M for a pair that must be removed. It is important not to confuse Q_M with the main priority queue of the algorithm (i.e., Q in Figure 3). Q_M will not be discussed further in the remainder of this paper.

2.2.5 *Other Extensions*

A number of other extensions of the incremental distance join are possible. The first is to add some spatial criterion to one or both of the relations involved in the join. As an example, the objects may be required to fall inside a given rectangle, or they may be required to have some minimum area. Such an extension can actually be applied equally to other spatial join algorithms, and does not necessarily involve modifying the algorithm. Instead, the distance functions (which may be parametrized) can check the additional spatial criteria, and return some special value if the pair should be discarded. Of course, if the spatial criterion has a high selectivity (i.e., such that few objects in each relation participating in the join satisfy the criterion), then it may be better to first restrict the number of objects by using the spatial criterion before computing the join. However, the cost of that alternative will include building a spatial index on the resulting restricted relations, or it will require using some algorithm other than the incremental distance join. In either case, it may take longer to produce the first few pairs with the alternative than with the incremental distance join, since it it highly geared towards producing pairs early.

The second extension is to impose a secondary ordering on the pairs produced by the algorithm, besides the distance between the objects. This is probably most useful if the resulting pairs are required to intersect, i.e., the maximum distance is 0. For example, we may wish to find the intersections of roads and rivers in order of distance from a given house. In the general case, this extension requires modifying the algorithm. However, for the special case of finding intersections, the distance functions could return ∞ for pairs that don't intersect, but for pairs that intersect, the functions would return some ordering value (such as the distance from the house in our example).

Another possible extension is to find the pairs in reverse order of distance, i.e., the farthest pair first, etc. This is relatively simple to achieve. Instead of ordering the elements on the priority queue in ascending order of distance, we would order them in descending order of distance (for example, this can be done by simply using the negative of the distance as a key). In addition, instead of using the regular distance functions as a key to order the pairs on the priority queue, the $d_{\max}$ functions must be used for all types of pairs except object/object pairs (recall that the $d_{\max}$ functions compute an upper bound on the distance of object/object pairs generated from pairs). As before, the algorithm will perform better if the distance range is rather narrow. However, in this case, we can estimate the minimum distance in the presence of an upper bound K on the number of ob-

ject pairs that will be requested. This is instead of estimating the maximum distance as was described in Section 2.2.4.

2.3 Computing Distance Semi-Join

Recall that distance semi-join is a subset of a distance join, where an object pair $\langle o_1, o_2 \rangle$ appears in the result only if none of the prior pairs contain o_1 as the first item. Thus, we must keep track of the set S_o of objects o_1 whose pairs $\langle o_1, o_2 \rangle$ have been reported. The easiest way to extend the incremental distance join algorithm to compute a distance semi-join is to use the algorithm unchanged and check outside of the algorithm if object o_1 in output pairs $\langle o_1, o_2 \rangle$ has been seen before (i.e., if it is present in the set S_o). However, that approach (termed "Outside" in Section 4.2.1) does not take advantage of the special structure of the distance semi-join to reduce the amount of work expended by the algorithm. In this section we identify several possible ways to modify the incremental distance join algorithm such that it computes the distance semi-join operation more efficiently. Also, we discuss how the extensions and optimizations described in section 2.2 for the incremental distance join algorithm apply for computing the distance semi-join operation.

First, we must bring into the algorithm the knowledge of the set S_o, the set of objects from the first collection that has already been seen. This is straightforward to do, and requires minor modifications to the INCDISTJOIN procedure of Figure 3 (termed "Inside1" in Section 4.2.1) as well as to procedure PROCESSNODE1 (termed "Inside2" in Section 4.2.1). Specifically, in line 4 of INCDISTJOIN, if in the dequeued element $\langle i_1, i_2 \rangle$, i_1 is an object or an object bounding rectangle, then we check if i_1 is present in S_o. If so, then we discard the pair. In PROCESSNODE1, if the node is a leaf node, then in line 5 we must ignore entries that correspond to objects that are present in S_o.

Bringing the knowledge of the set S_o into the algorithm is a definite improvement, but we can do better still. The next improvement is based on the fact that for each pair $\langle o_1, o_2 \rangle$ in the output of the distance semi-join of A and B, o_2 is the object in B nearest to o_1. This can be exploited locally in the PROCESSNODE2 procedure (termed "Local" in Section 4.2.1). To see how, note that for a pair $\langle o_1, n_2 \rangle$, the object in the subtree of n_2 closest to o_1 is most likely in some of the entries of n_2 whose region is near o_1. Specifically, we compute $d_{\max}(o_1, e_2)$ for each entry e_2 in n_2, and determine the minimum value, D_{mm}. Any entry in n_2 that is farther away from o_1 than D_{mm} can be discarded as it is guaranteed not to contain the object in the subtree of n_2 nearest to o_1. This principle can be applied in PROCESSNODE2 even for pairs $\langle i_1, n_2 \rangle$ where i_1 is an object bounding rectangle or a node, since $d_{\max}(i_1, n_2)$ is an upper bound on the distance of any object pair derived from $\langle i_1, n_2 \rangle$. Observe that this approach is analogous to the downward pruning strategy of the nearest neighbor algorithm of [26].

A more aggressive strategy can be obtained by using the same insight in a global fashion. In other words, for each object and node in R_1 (the spatial data structure representing the objects in A), maintain the smallest $d_{\max}$ distance that has been seen so far (termed "GlobalAll" in Section 4.2.1). Any time we consider enqueuing a pair $\langle i_1, i_2 \rangle$, we would first make sure that the distance of the pair is smaller than the smallest $d_{\max}$ distance for i_1. Employing this strategy requires a considerable amount of memory space if R_1 contains many objects. Nevertheless, it is useful as a comparison with the other strategies. Moreover, we can compromise by only maintaining the globally smallest $d_{\max}$ distance for the nodes of R_1, which requires an order of magnitude less space than doing so also for the objects in R_1 (termed "GlobalNode" in Section 4.2.1).

As in the case of computing the distance join incrementally, we can estimate the maximum distance needed to produce a maximum of K pairs for the distance semi-join. This is done in much the same way as described in Section 2.2.4. The difference, here, is that in the set M of the pairs being used in the estimation process, the first item in each pair is unique. In other words, if $\langle i_1, i_2 \rangle$ is a pair in M, then no other pair in M has i_1 as the first item[5]. Also, the number of pairs generated from a pair $\langle i_1, i_2 \rangle$ is bounded by the number of objects in the subtree of i_1, assuming i_1 is a node. When an item $\langle i_1, i_2 \rangle$ is about to be inserted into M, we must first check if another item $\langle i_1, i_2' \rangle$ exists in M. If so, then we replace $\langle i_1, i_2' \rangle$ by $\langle i_1, i_2 \rangle$ if the latter has a smaller $d_{\max}$ value and ignore $\langle i_1, i_2 \rangle$, otherwise. There are two additional subtle differences. When $\langle o_1, o_2 \rangle$ is reported, any pair $\langle o_1, i_2 \rangle$ in M must be removed. Also, a pair $\langle n_1, i_2 \rangle$ may only be inserted into M if n_1 has never been processed for any pair $\langle n_1, i_2' \rangle$. Otherwise, some of the objects in the subtree of n_1 would be counted more than once (since processing n_1 in the pair $\langle n_1, i_2' \rangle$ may lead to some pairs $\langle e_1, i_2' \rangle$ to be inserted into M where e_1 is an entry in n_1). This may lead to an estimate of $D_{\max}$ that is too low thereby causing us to find less than K pairs which will force us to restart the query. The reason we need to restart is that the priority queue does not provide us any useful information as we will have pruned too many entries by our maximum distance heuristic.

The extensions discussed in Section 2.2.5 also apply for the distance semi-join version of the our incremental algorithm. However, modifying the algorithm to find pairs in reverse order of distance leads to what may seem an unintuitive, and perhaps not very useful, result. There are two possible ways of defining a reverse distance semi-join operation on relations A and B. The first is to report in reverse order of distance the object in B closest to each object in A. The second is to report in reverse order of distance the object in B *farthest* from each object in A. The straightforward way of applying the incremental distance join to the reverse distance semi-join will be in accordance to the second definition since it corresponds to reporting for each object o_1 the first pair $\langle o_1, o_2 \rangle$ that occurs in a reverse distance join. The first definition would mean reporting for each object o_1 the last pair $\langle o_1, o_2 \rangle$ that occurs in a reverse distance join, which would be extremely inefficient.

3 Experimental Environment

3.1 System and Data

All of our experiments were run on a Sun Ultra 1 Model 170E machine, rated at 6.17 SPECint95 and 11.80 SPECfp95, with 64MB in main memory and a 2.1GB internal disk drive. The spatial data structure that we used is an R*-tree [5]. The size of the nodes was 1K, for a maximum fan-out of 50, with 256K of memory used for buffers. The spatial objects were represented directly in the leaves of the R*-trees. We chose that approach in order to simplify the analysis of the execution time results. Also, the organization of the external object storage has a large effect on the performance, and thus introduces an additional variable. The software was compiled with a GNU C++ compiler set for maximum optimization (–O3). The distance functions were based on the Euclidean metric.

As in other evaluations of spatial algorithms (e.g., [8, 21]), we derived our test data from the TIGER/Line File [9]. We used two sets of points from the coverage of the Washington, DC area: *Water* contains the centroids of water features (37,495 points), and *Roads* contains the centroids of road features (200,482 points). It should be clear that dealing with line data is much more complex than points. Making experiments with line data and more complex spatial features is a subject for future study.

[5] As far as M is concerned, an object bounding rectangle is treated in the same way as the corresponding object; both are represented by the object identifier.

3.2 Implementation Details

An important issue is the implementation of the priority queue. It should be clear that the number of object pairs in the result of a full distance join operation is extremely large, or the same as in the Cartesian product of the relations (in the absence of distance range restrictions). Even when computing the entire distance join (this is not likely to be very useful in practice, however), the size of the priority queue in the incremental distance join algorithm remains much smaller than the size of the result. Nevertheless, a small fraction of a very large number is still a large number. Thus, the size of the priority queue may be too large to fit in memory. However, an exclusively disk-based scheme for representing the priority queue is not desirable, due to poor performance.

In our experiments, we use a simple hybrid memory/disk scheme that stores parts of the priority queue in a memory-based heap structure (we chose the pairing heap structure [13]), while the rest is offloaded to disk. If a relatively small number of object pairs is requested, then the vast majority of pairs put on the priority queue will never be needed. Thus, our goal in developing the scheme was that the contents of the priority queue that were put on disk would only be needed when a large number of object pairs were requested. Another reason for limiting the contents of the memory-based heap to pairs that are likely to be needed is that the algorithmic complexity of heap operations is directly related to the size of the heap. We chose to use a three-tiered scheme for representing the priority queue, based on the distance of the pairs. Pairs with a distance less than `D1` are stored in the memory-based heap, pairs with a distance less than `D2` are stored in an unorganized list in memory, while pairs with a distance of `D2` or greater are stored on disk. If the heap becomes empty, then the contents of the unorganized list is put into the heap, the value of `D1` is changed to `D2`, a new value is chosen for `D2`, and pairs on disk with distance between the new values of `D1` and `D2` are put into the unorganized list (actually, we avoid accessing the pairs on disk unless they need to be inserted into the priority queue). In our implementation, a fixed distance increment D_T is used to update `D1` and `D2`, with their initial values being D_T and $2D_T$, respectively. The part of the queue stored on disk is organized in linked lists of pages with the pairs in each list having distances in the range $[kD_T, (k+1)D_T)$.

The drawback of our priority queue scheme is that it depends on a fixed constant D_T rather than responding dynamically to the distribution of the queue contents. In the experiments, we chose a value for D_T that worked well for the input relations. Developing a way of choosing D_T based on the input relations, or finding some other dynamic method of deciding what part of the priority queue is stored on disk, are subjects for further investigation.

In Section 2.3, a set S_o is maintained of objects from A for whom a pair has been reported by the incremental algorithm for the distance semi-join. In our experiments, we use a bit string representation for S_o. The reason is that a bit string representation is extremely efficient, both for membership tests and insertions. There is certainly a space/time tradeoff involved, since a bit string representation of a set occupies a fixed amount of space, regardless of the size of the set. For sets of only a few elements, it would be much more space efficient to use some other approach. Nevertheless, given the memory capacity of modern computers, the size of the bit strings is modest even for large data sets. For example, a bit string representation of a subset of 1 million elements would occupy 122K.

4 Performance Results

In this section, we evaluate the effectiveness of the strategies that we presented for enhancing the efficiency of the incremental distance join algorithm, as well as compare its performance to competing approaches for computing the distance join and distance semi-join. In our experiments, we always joined Water with Roads, except where noted[6].

4.1 Distance Join

4.1.1 Priority Queue Ordering and Tree Traversal

Section 2.2.2 discussed the effect of choosing a different priority queue ordering (i.e., how ties are resolved for pairs with the same distance) as well as how to process pairs of two nodes. Table 1 lists the values of some performance measures (the number of object distance calculations, the maximum queue size, and the number of node I/O operations) for retrieving up to 100,000 pairs of the distance join for a version of the incremental distance join algorithm where pairs with the same distance are ordered so that the algorithm performs a depth-first traversal (i.e., nodes at a deeper level are given priority); only one node is processed at a time in node/node pairs; and the two spatial indexes are traversed evenly[7]. In all experiments below, except where otherwise noted, this type of queue order and traversal is used. In Figure 6, we plot the execution times of this version (labeled "DepthFirst") against three other versions: (1) "BreadthFirst" orders pairs with the same distance such that it leads to breadth-first traversal; (2) "Basic" is the basic algorithm of Figure 3, where we always process the first node in node/node pairs; and (3) "Simultaneous" where both nodes of node/node pairs are processed simultaneously.

Overall, the shape of the graphs is similar. For the versions using the priority queue order leading to depth-first traversal ("DepthFirst", "Basic" and "Simultaneous"), obtaining the first pair is relatively inexpensive, while the cost does not rise much for between 10 and 10,000 pairs. However, for computing a larger number of pairs, the cost rises dramatically.

The difference in execution times for the four versions is due to differences in the values of all performance measures in Table 1. However, the dominant factor, although not shown here, is the number of distance calculations and the size of the priority queue, which are much larger for "Basic" and "Simultaneous". Since a maximum distance is not specified for these experiments, the "Simultaneous" version is not able to benefit from its filtering and plane-sweep techniques. The reason for "DepthFirst" being somewhat faster than "BreadthFirst" for retrieving one pair is that there is one object pair with a distance of 0. This pair is reported as soon as it is found by "DepthFirst", but in "BreadthFirst" it is only reported after all intersecting nodes have been processed. After the first pair, the difference between these methods is negligible.

An interesting question is what the reason is for the sharply higher cost for computing 100,000 pairs compared to computing 10,000 pairs. Table 1 reveals that there is a relatively larger increase in node I/O between computing 10,000 and 100,000 pairs (node I/O counts the number of times a requested node is not in the node buffer). The number of node accesses (not shown in the table) increases by about 43%, and almost all of the additional accesses

[6] Since the distance join is symmetric, the result of joining Roads with Water is the same. However, the incremental distance join algorithm is not necessarily symmetric in its execution pattern, so that the execution time may be different based on the order of the joined relations. The distance semi-join operation is not symmetric, so that result of a distance semi-join of Roads with Water is different from a distance semi-join of Water with Roads.

[7] Recall that by traversing *evenly* we mean that if the nodes in a node/node pair are at a different level in their respective trees, then we choose to process the node at a shallower level.

Pairs	Time (sec.)	Dist. Calc.	Queue Size	Node I/O
1	6.9	307994	1002536	3019
10	9.0	393758	1333856	4087
100	9.4	395780	1356985	4652
1,000	9.8	403281	1434160	6487
10,000	12.6	422392	1632895	11502
100,000	23.8	479262	2229874	28356

Table 1: Values of performance measures for incremental distance join algorithm using depth-first traversal, processing one node at a time, and using even traversal.

are for nodes that are not present in the node buffer. A larger node buffer, or a better buffer strategy, will most likely improve the performance for computing 100,000 pairs. Another factor in the higher cost of computing 100,000 pairs is that for that many pairs, parts of the priority queue contents that were written to disk must be read back into memory.

The values of the performance measures when joining Roads with Water, instead of Water with Roads, is virtually the same for these versions of the algorithm, except for "Basic". Since Water is larger, many more pairs are generated (in this case, Water is traversed first). In fact, for retrieving 100,000 pairs, too many pairs were generated for the priority queue to fit on disk. Thus, the treatment of node/node pairs in "Basic" is clearly too simplistic.

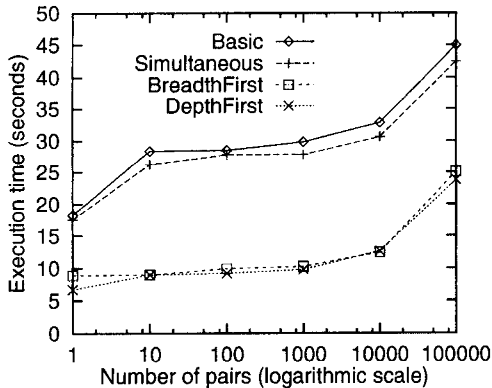

Figure 6: Execution time for different queue order and node processing.

4.1.2 Maximum Distance and Maximum Pairs

In Section 2.2.3 we discussed the importance of imposing a maximum distance, and in Section 2.2.4 we described how the maximum distance can be estimated based on an upper bound on the number of object pairs that will be requested. Figure 7 compares the execution time of the regular algorithm (i.e., "DepthFirst" from the preceding section) to two versions of the algorithm applied to distance join: (1) "MaxDist" is the regular algorithm with maximum distance set to the distance of pair number 1000, 10,000, and 100,000 (for "MaxDist 1000", we only compute up to 1000 pairs, etc.); (2) "MaxPair" uses the maximum distance estimation for an upper bound of 1000 and 10,000 pairs (setting the maximum to 100,000 was slower than the "Regular" version). The purpose of showing the "MaxDist" plots is to demonstrate the effect of setting the maximum distance, and it also provides a useful benchmark of the effectiveness of the maximum distance estimation of "MaxPair". Of course, in practice we will not know in advance the distance of pair number 1000, etc.

Figure 7 confirms the benefit of setting the maximum distance. The performance was very similar for the three values for the maximum distance. Setting the maximum number of pairs is seen to be only beneficial for a relatively small number of pairs. For a maximum of 1000 pairs, we get a similar performance as for setting the maximum distance. When the maximum is set to 10,000 pairs, there is less benefit, as the maximum distance estimate is not as tight and the overhead of the estimation process is greater.

In Section 4.1.1 we confirmed that processing both nodes simultaneously for node/node pairs is worse than processing only one at a time if no maximum distance is specified. We performed the same experiments as shown in Figure 7 using the "Simultaneous" version of the incremental distance join algorithm. Although we do not explicitly present these results here, as expected, the performance of "Simultaneous" was better than that of "DepthFirst" when a relatively small maximum distance was specified, or up to 20% for "MaxDist 1000". However, the improvement was most pronounced for retrieving only a few pairs, and was much smaller for retrieving 10 or more pairs, or usually about 3-5%. Specifying a maximum on the number of pairs was also a little faster using the "Simultaneous" version for a very small number of pairs. For 10 pairs or more, however, it proved better to process only one node at a time in node/node pairs, although the improvement was not great (typically about 2-4%).

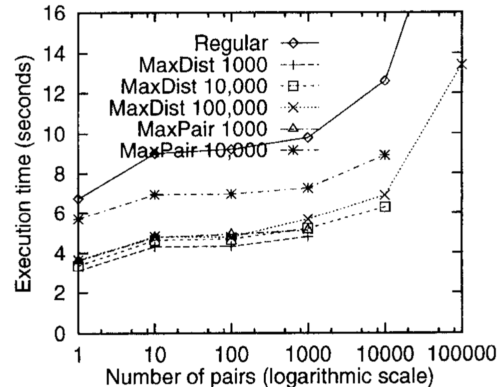

Figure 7: Execution time for different maximum distance and maximum pairs for distance join.

4.1.3 Priority Queue Implementation

In Section 3.2 we discussed a hybrid implementation of the priority queue that offloads parts of the queue to disk. Figure 8 gives the execution time for a purely memory-based queue implementation as well as the hybrid one, where two different values of D_T are used for the hybrid approach[8]. The memory-based queue is only a little slower for up to 10,000 pairs. However, for 100,000 pairs, it is almost an order of magnitude slower, due to excessive virtual memory thrashing, taking over 180 seconds to compute. The hybrid approach performed almost equally well for the different values of D_T, except when retrieving 100,000 pairs. In that case, the higher D_T value (i.e., "Hybrid2") was better, most likely because

[8]The values of D_T, chosen somewhat arbitrarily, correspond to the distances of pairs number 7,663 and 34,906. The latter value was used for all the other experiments.

it required fewer reads from the disk portion of the priority queue. For fewer than 100,000 pairs, the lower D_T value (i.e., "Hybrid1") gave slightly better performance, as fewer priority queue elements had to be written to disk. The best value for D_T depends both on the nature of the data sets and the amount of available memory.

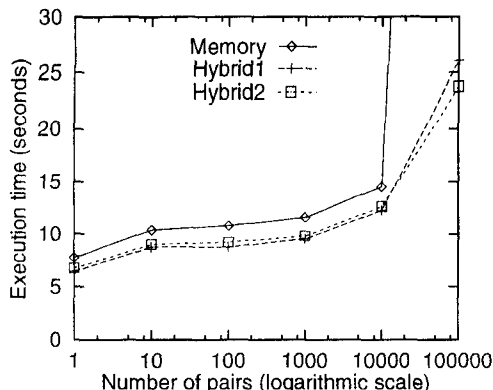

Figure 8: Execution time for storing the priority queue entirely in memory vs. offloading parts on disk.

4.1.4 Alternative Implementations

The distance join operation can be computed in other ways than with the incremental distance join algorithm. If a maximum distance is imposed, then a spatial join with a within predicate can be executed, with the output being sorted once it is done. If no maximum distance is imposed, then some distance must be guessed at if an algorithm for the spatial join with within predicate is to be used. If the distance is too small and not enough pairs result, then the spatial join must be executed again with a larger distance. Due to this problem, we do not use a spatial join algorithm for comparison.

Another way of computing a distance join is to use a nested loop approach and compute the distance between all possible pairs of objects. However, this will not compare favorably with using the incremental distance join algorithm unless a very large number of pairs is needed, which is unlikely to arise in practice (for example, the full join for our data sets contains about 7.5 billion pairs). Nevertheless, we did an experiment with this approach using the Water and Roads data sets. For simplicity sake, we only computed the distance values but didn't store them nor did we sort at the end, which would be necessary for a real implementation. The data set of the inner loop was read completely into memory in order to avoid rereading it. The time to execute the experiment was over 3 1/2 hours. In that amount of time, the incremental distance join is able to compute at least 100 million pairs. However, for that many pairs, the priority queue becomes so large that the incremental distance join is not practical unless a very large disk space is available.

4.2 Distance Semi-Join

In this section we discuss some results of our experiments for computing the distance semi-join with variants of the incremental distance join algorithm. Since we are joining Water with Roads, this results in finding the nearest neighbors of points in Water.

4.2.1 Pair Filtering and Smallest $d_{\max}$ Distance

In Section 2.3 we enumerated several ways of filtering out pairs $\langle i_1, i_2 \rangle$ where i_1 is an object or an object bounding rectangle and i_1 has already been reported. Also, we presented ways of limiting the number of pairs generated based on the $d_{\max}$ distance of pairs. Figure 9 gives the execution time for these various filtering methods: (1) "Outside" executes the regular incremental distance join algorithm and filters out resulting pairs that contain objects that have already been reported; (2) "Inside1" filters only in the INCDISTJOIN procedure of Figure 3; and (3) "Inside2" filters also in the PROCESSNODE1 procedure. There are three schemes that exploit the $d_{\max}$ distance, all of which use the filtering of "Inside2": (1) "Local" only works locally in the PROCESSNODE1 procedure; (2) "GlobalNodes" uses the local strategy, as well as globally maintaining the smallest $d_{\max}$ distance of nodes; and (3) "GlobalAll" globally maintains the smallest $d_{\max}$ distance of both nodes and objects.

Filtering pairs outside the INCDISTJOIN procedure appears to be slightly better for up to 1000 pairs. However, the priority queue became too large to find the neighbors of all points in Water. Filtering inside INCDISTJOIN and/or PROCESSNODE1 saves some distance calculations and node accesses for retrieving 1000 or more pairs, but this was outweighed by more member checks against the S_o set, at least for up to 1000 pairs. For more pairs the benefit of more filtering becomes greater, and for finding the neighbors of all points in Water "Inside1" is about 47% slower than "Inside2" (530 vs. 362 seconds; this is not shown in Figure 9 in order not to obscure the time difference for smaller numbers of pairs).

The three schemes for exploiting $d_{\max}$ distances also are very similar for up to 10,000 pairs. However, for much larger number of pairs, the benefit of maintaining the $d_{\max}$ distance of all objects and nodes ("GlobalAll") becomes more pronounced. Doing it only for nodes ("GlobalNode") did not seem to result in appreciable improvement compared to "Local".

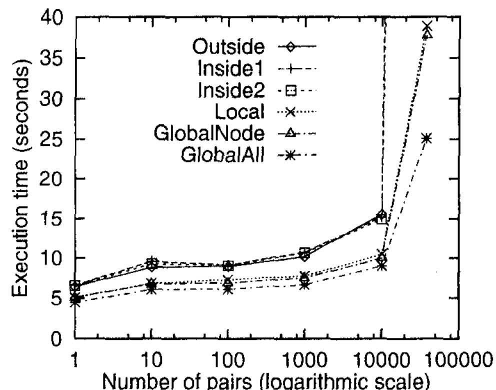

Figure 9: Execution time for storing priority queue entirely in memory vs. offloading parts on disk.

4.2.2 Maximum Distance and Maximum Pairs

As in Section 4.1.2, we now report on experiments testing the effect of setting a maximum distance or an upper bound on the numbers of pairs for computing the distance semi-join operation with the incremental distance join algorithm. Figure 10 shows the result of doing this using the "Local" version of Section 4.2.1. The figure confirms the benefit of restricting the maximum distance. However, setting the maximum distance to be the largest possible distance between two objects in the result of the distance semi-join ("MaxDist All") does not appear to speed up the process. Notice that setting the maximum number of pairs to 1000 does improve the execution time, making it virtually identical to setting the maximum distance to the distance of the 1000th pair. However, choosing 10,000 or more as

the maximum number of pairs makes the algorithm slower, as such a large limit does not give a tight estimate for the maximum distance, and the overhead cost incurred in estimating the maximum distance exceeds its benefit. The cost of computing the neighbor of all points in River (not shown in the figure) is about 35 seconds for "MaxDist All" and 44 seconds for "MaxPair All". The former is about 14% less than if maximum distance is not set. Thus, we can see that setting a low limit on the distance or the number of pairs in the output gives significant savings (up to 50% or more), while less so for higher limits.

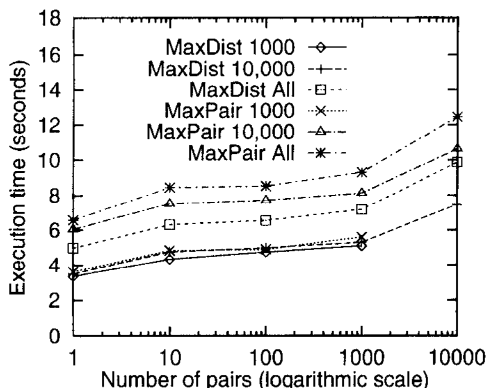

Figure 10: Execution time for different maximum distance and maximum pairs for distance semi-join.

4.2.3 *Alternative Implementations*

The distance semi-join can also be implemented using a nearest neighbor algorithm. For each object in relation A, we perform a nearest neighbor computation in relation B, and sort the resulting array of distances once all neighbors have been computed. For the data sets in question, the execution time for doing this is about 27 seconds. The incremental distance join methods reported in Figure 9 compare favorably with this method for computing the entire distance semi-join, especially "GlobalAll" (which took around 25 seconds). An even better result is obtained if we switch the order of the relations (i.e., compute the distance semi-join of Rivers and Water), in which case "GlobalAll" takes about 102 seconds while the nearest neighbor implementation takes 141 seconds.

Observe that the "GlobalAll" strategy must keep track of the $d_{\max}$ distance for all objects and nodes in the R-tree for relation A, which can occupy considerable storage. However, an implementation that uses a nearest neighbor algorithm must also store distance values for all objects.

5 Concluding Remarks and Directions for Future Research

Two new spatial join operations have been defined where the join output is ordered by the distance between the spatial attribute values of the joined tuples, and a number of different incremental strategies for computing them have been examined. The rationale behind our solutions is that frequently only a small part of the join result will actually be needed. Our experiments revealed that for distance join, the variant of the incremental distance join algorithms that performed best overall was the one that processed only one node in node/node pairs at a time, attempted to traverse the two trees evenly (i.e., so as not to descend much farther into one than the other), and ordered pairs with the same distance to result in a depth-first traversal. Setting a limit on the distance of pairs was shown to improve performance considerably, even if the maximum distance limit is relatively large. However, imposing an upper bound on the number of pairs is only worthwhile if the upper bound is not very large (e.g., in our experiments, an upper bound of 100,000 pairs did not improve performance). Nevertheless, in many of the applications that we envision for our algorithm—most notably for interactive query interfaces, which quickly present the user with the most relevant part of the query result—a small upper bound can be established.

For the distance semi-join, the strategies for improving the performance of the incremental distance join were shown to yield significant improvements, especially for computing a large part of the result. The strategies use different means for eliminating from consideration pairs that are sure not to be needed to compute the output of the algorithm. The best overall strategy used every possible opportunity for eliminating pairs containing object o_1 if a pair $\langle o_1, o_2 \rangle$ has been reported earlier, and uses global knowledge of distance bounds to further eliminate pairs when processing nodes ("GlobalAll" in Figure 9). This version was found to be better than a non-incremental approach that computes the distance semi-join using a nearest neighbor algorithm. However, maintaining the global knowledge of distance bounds requires a somewhat large amount of storage. A reasonable compromise is to exploit the distance bounds only locally within a node as it is being processed ("Local"). The effect of restricting the maximum distance or the maximum number of pairs was found to yield similar benefits as when computing the distance join.

Our algorithm finds use for processing queries such as "find the city nearest to any river, such that the city has a population of more than 5 million". There are at least two options for a query optimizer to use the incremental distance join algorithm to answer this query:

1. Execute the algorithm on the city and river relations and filter out the result pairs where the city has too small a population, and
2. First find the cities with a population greater than 5 million and use that in the incremental distance join algorithm.

For the second option, a spatial index must be built on the result of finding cities with a population of more than 5 million for the algorithm to be applicable. Hence, this option is most appropriate if the population criteria has a high selectivity. However, if the population criteria has a low selectivity, then the first option would be superior. More query plans may even exist, employing some other algorithm. To enable the query optimizer to choose between these options requires a cost model for the relevant algorithms (e.g., as developed in [20] for the traditional R-tree spatial join). Developing such cost models for the incremental distance join algorithms presented in this paper is a subject for further study.

Other issues for further investigation include developing techniques to dynamically partition the priority queue between a memory-based structure and a disk-based one. Our experiments were limited to using two-dimensional points. Further work is needed to determine how appropriate our approach is for more complex spatial objects (i.e., with extent, such as lines and polygons), as well as for higher dimensions.

6 Acknowledgements

We wish to thank Björn Þ. Jónsson and Dr. Robert E. Webber for their critical comments.

References

[1] W. G. Aref and H. Samet. The spatial filter revisited. *Proc. of 6th International Symposium on Spatial Data Handling*, pp. 190–208, Edinburgh, Scotland, September 1994.

[2] F. Bartling and K. Hinrichs. Probabilistic analysis of an algorithm for solving the k-dimensional all-nearest-neighbors problem by projection. *BIT*, 31(4):558–565, 1991.

[3] R. J. Bayardo and D. P. Miranker. Processing queries for first few answers. In *Proc. of 5th CIKM*, pp. 45–52, Rockville, MD, November 1996.

[4] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. *Proc. of 9th IEEE Int. Conf. on Data Engineering*, pp. 190–197, Vienna, Austria, April 1993.

[5] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. *Proc. of ACM SIGMOD*, pp. 322–331, Atlantic City, NJ, June 1990.

[6] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. *Proc. of 11th Symp. on Computational Geometry*, pp. 152–161, Vancouver, British Columbia, June 1995.

[7] T. Brinkhoff, H. P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. *Proc. of ACM SIGMOD*, pp. 197–208, Minneapolis, MN, June 1994.

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

[9] Bureau of the Census. *Tiger/Line precensus files*. Washington, DC, 1989.

[10] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. *Proc. of ACM SIGMOD*, pp. 219–230, Tucson, AZ, May 1997.

[11] K. L. Clarkson. Fast algorithm for the all nearest neighbors problem. *Proc. of 24th IEEE Symp. on the Foundations of Computer Science*, pp. 226–232, Tucson, November 1983.

[12] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[13] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[14] O. Günther. Efficient computation of spatial joins. *Proc. of 9th IEEE Int. Conf. on Data Engineering*, pp. 50–59, Vienna, Austria, April 1993.

[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of ACM SIGMOD*, pp. 47–57, Boston, MA, June 1984.

[16] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. *Proc. of ACM SIGMOD*, pp. 171–182, Tucson, AZ, May 1997.

[17] A. Henrich. A distance-scan algorithm for spatial access structures. *Proc. of 2nd ACM Workshop on GIS*, pp. 136–143, Gaithersburg, MD, December 1994.

[18] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Advances in Spatial Databases — 4th Int. Symp., SSD'95*, pp. 83–95, Portland, ME, August 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).

[19] E. Hoel and H. Samet. Data-parallel spatial join algorithms. *Proc. of 23rd Int. Conf. on Parallel Processing*, pp. 227–234, St. Charles, IL, August 1994.

[20] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using r-trees. *Proc. of 9th Int. Conf. on Scientific and Statistical Database Management*, pp. 30–38, Olympia, WA, August 1997.

[21] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: breadth-first traversal with global optimizations. *Proc. of 23rd VLDB Conf.*, pp. 396–405, Athens, Greece, August 1997.

[22] M. Kitsuregawa, L. Harada, and M. Takagi. Join strategies on k-d-tree indexed relations. *Proc. of 5th IEEE Int. Conf. on Data Engineering*, pp. 85–93, Los Angeles, February 1989.

[23] M. L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. *Proc. of ACM SIGMOD*, pp. 209–220, Minneapolis, MN, June 1994.

[24] D. Lomet and B. Salzberg. A robust multi-attribute search structure. *Proc. of the 5th IEEE Int. Conf. on Data Engineering*, pp. 296–304, Los Angeles, February 1989.

[25] D. Rotem. Spatial join indices. *Proc. of 7th Int. Conf. on Data Engineering*, pp. 500–509, Kobe, Japan, April 1991.

[26] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proc. of ACM SIGMOD*, pp. 71–79, San Jose, CA, May 1995.

[27] H. Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[28] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, Reading, MA, 1990.

[29] B. Seeger and H. P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. *Proc. of 16th VLDB Conf.*, pages 590–601, Brisbane, Australia, August 1990.

[30] J. C. Shafer and R. Agrawal. Parallel algorithms for high-dimensional proximity joins. *Proc. of 23rd VLDB Conf.*, pp. 176–185, Athens, Greece, August 1997.

[31] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers*, 31(3):181–187, March 1982.

[32] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4(2):101–115, 1989.

[33] T. L. Wang and D. Shasha. Query processing for distance metrics. *Proc. of 16th VLDB Conf.*, pages 602–613, Brisbane, Australia, August 1990.

[34] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Proc. of 1st Int. Conf. on Parallel and Distributed Information Systems*, pp. 68–77, Miami, FL, December 1991.