Incremental Distance Join Algorithms for Spatial Databases

Gíslí 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 operation 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]. A generalization of this is within, where the objects are required to lie within some distance of each other [24, 29]. 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, 24].

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) AS 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) AS 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 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 [32] 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 [32]. However, that algorithm is not well suited for spatial data that resides in d-dimensional Euclidean space. The reason is that [32] 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 a query engine 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, 33]. 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 [30], closest pair [6], and all nearest neighbors [2, 11, 31] 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.](image)

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 [23] (which forms a directed acyclic graph), and quadtrees [26, 27] (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 should be 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 \( d_{\text{obj}}(o_1, o_2) \), \( d_{\text{obj}}(n_1, n_2) \), \( d_{\text{obj}}(n_1, o_2) \), and \( d_{\text{obj}}(n_1, n_2) \), where \( o_1 \) and \( n_1 \) are an object and a node from \( R_1 \), respectively, and \( n_2 \) and \( o_2 \) are an object and a node from \( R_2 \), respectively. If the leaf nodes store a minimum bounding rectangle for objects, then the functions \( d_{\text{rect}} \) need not be used. Instead, we need the functions \( d_{\text{rect}}(n_1, n_2) \), \( d_{\text{rect}}(n_1, o_2) \), in addition to \( d_{\text{rect}}(n_1, n_2) \), where \( n_1 \) and \( n_2 \) denote a minimum bounding rectangle for objects in \( R_1 \) and \( R_2 \), respectively. If node regions are rectangles, then \( d_{\text{rect}} \) 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 Euclidean 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 \( o_1 \), then we must have \( d_{\text{obj}}(o_1, o_2) \geq d_{\text{obj}}(n_1, o_2) \). 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. 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, child 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 \textsc{incDistJoin} procedure contains the high level control structure for the algorithm, while procedures \textsc{ProcessNode1} and \textsc{ProcessNode2} enqueue new pairs for each entry in a node from \( R_1 \) and \( R_2 \), respectively. In lines 6 and 11 of \textsc{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 \textsc{ProcessNode1} and \textsc{ProcessNode2} is called. This version of the algorithm arbitrarily chooses to call \textsc{ProcessNode1} if both items are nodes.

In line 4 of \textsc{ProcessNode1}, \{\textit{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 \textsc{if} statement in line 7 of \textsc{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 \textsc{ProcessNode1} and \textsc{ProcessNode2} are essentially the same as the basic loop of the nearest neighbor algorithm. In particular, in \textsc{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

---

\(^1\)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 \( (i_1, i_2) \) is said to be generated from a pair \( (i'_1, i'_2) \) if the pair \( (i_1, i_2) \) results from a sequence of algorithm operations starting with \( (i'_1, i'_2) \). As an example, all object/object pairs are ultimately generated from the initial pair of root nodes.
In Figure 3, when the dequeued pair contains two nodes, \( \mathbf{R} \) to be traversed down to the leaf level before the root of
ined) rather than
ined) 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 \((n_1, n_2)\), 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 \(r_1\) and \(r_2\) are entries in \(n_1\), and \(s_1, s_2, s_3, s_4\) are entries in \(n_2\). Without plane sweep, \(r_1\) 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 \(r_1\) with \(s_1\) and \(s_2\). 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 \(s_3\) is within the proper distance of \(r_1\), in addition to \(s_1\) and \(s_2\).

\[
\text{INC_DISTJOIN}(R_1, R_2)
\]

1. \(Q \leftarrow \text{NEW_PRIORITY_QUEUE}()\)
2. \(\text{ENQUEUE}(Q, 0, (R_1, \text{ROOT_NODE}, R_2, \text{ROOT_NODE}))\)
3. \(\text{while } \neg \text{IS_EMPTY}(Q) \text{ do}\)
4. \(\text{Elem} \leftarrow \text{DEQUEUE}(Q)\)
5. \(\text{if both items in } \text{Elem} \text{ are data objects then}\)
6. \(\text{Report } \text{Elem}\)
7. \(\text{else if both items are object bounding rectangles then}\)
8. \(\text{let } O_1 \text{ and } O_2 \text{ be the corresponding object references}\)
9. \(D \leftarrow \text{DIST}(O_1, O_2)\)
10. \(\text{if } \text{IS_EMPTY}(Q) \text{ or } D \leq \text{FRONT}(Q), \text{DIST then}\)
11. \(\text{Report } (O_1, O_2)\)
12. \(\text{else}\)
13. \(\text{ENQUEUE}(Q, D, (O_1, O_2))\)
14. \(\text{endif}\)
15. \(\text{else if item } 1 \text{ in } \text{Elem} \text{ is a node then}\)
16. \(\text{PROCESS_NODE1}(Q, \text{Elem})\)
17. \(\text{else}\)
18. \(\text{PROCESS_NODE2}(Q, \text{Elem})\)
19. \(\text{endif}\)
20. \(\text{enddo}\)

\[
\text{PROCESS_NODE1}(Q, \text{Elem})
\]

1. \(\text{Node} \leftarrow \text{item } 1 \text{ of } \text{Elem}\)
2. \(\text{Item}_2 \leftarrow \text{item } 2 \text{ of } \text{Elem}\)
3. \(\text{If } \text{Node} \text{ is a leaf node then}\)
4. \(\text{for each entry } [O] \text{ in } \text{Node do}\)
5. \(\text{ENQUEUE}(Q, \text{DIST}([O], \text{Item}_2), ([O], \text{Item}_2))\)
6. \(\text{enddo}\)
7. \(\text{else}\)
8. \(\text{for each Child node of } \text{Node do}\)
9. \(\text{ENQUEUE}(Q, \text{DIST}(\text{Child, Item}_2), ([\text{Child, Item}_2]))\)
10. \(\text{enddo}\)
11. \(\text{endif}\)

\[
\text{PROCESS_NODE2}(Q, \text{Elem})
\]

1. \(\text{Same as } \text{PROCESS_NODE1}, \text{with items } 1 \text{ and } 2 \text{ exchanged}\)

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

Figure 4: Plane sweep along \(z\)-axis over the entries in two nodes.
pute a spatial join with a within predicate (however, unlike with our algorithm, if the pairs are desired in order of distance, then the entire 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 [27] and the buddy-tree [28]). 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 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. 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 can be derived from it (this is guaranteed by the consistency of the distance functions).

That has been termed MINMAXDIST [25]. The object bounding rectangles are required to minimally bound the objects. The key idea behind the MINMAXDIST metric is that if a point is a distance equal to the minimum distance bound, then we can discard the pairs. Thus, given a point p, we have d(p, o) ≤ max p∈F d(p, f), for each f ∈ 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_{min}(p, b) = min f∈F(b) max p∈F(d(p, f)).

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. 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, it is required to minimally bound the objects. The key idea behind the MINMAXDIST metric is that if a point is a distance equal to the minimum distance bound, then we can discard the pairs. Thus, given a point p, we have d(p, o) ≤ max p∈F d(p, f), for each f ∈ 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_{min}(p, b) = min f∈F(b) max p∈F(d(p, f)).

Thus, given a point p, we have d(p, o) ≤ max p∈F d(p, f), for each f ∈ 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_{min}(p, b) = min f∈F(b) max p∈F(d(p, f)). A practical way of computing the value of d_{min}(p, b) is to first compute the maximum distance from p to a vertex of b, say r_{max}, and then to determine the vertex adjacent to r_{max} (i.e., along an edge) that is closest to p [25]. Now, we can define d_{max}(n_1, b_2) = max e∈n_1 d_{max}(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_j∈F(b_1), f_k∈F(b_2) max m_{p,n_1}∈f_j min m_{p,n_2}∈f_k d(p, f), where 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_{min} function for node/node pairs.

Figure 5 presents a version of PROCESSNODE1 that restricts distances to a range of values. 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. The arguments Min and Max in Figure 5 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. If the object geometry is represented directly in the leaf nodes, then the actual objects would be used in line 4 of Figure 5. Also, in that case, if item 2 is an object, then MAXDIST is equivalent to MINDIST in line 5.

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

\[ \text{PROCESSNODE1}(Q, \text{Elem}, \text{Min}, \text{Max}) \]

1 Node — item 1 of Elem
2 Item_{m2} — item 2 of Elem
3 if Node is a leaf node then
4 for each \{O\} in Node do
5 if MAXDIST([O], Item_{m2}) ≥ Min and
MINDIST([O], Item_{m2}) ≤ Max then
6 ENQUEUE(Q, MINDIST([O], Item_{m2}), \{[O], Item_{m2}\})
7 endif
8 enddo
9 else
10 for each Child node of Node do
11 if MAXDIST(Child, Item_{m2}) ≥ Min and
MINDIST(Child, Item_{m2}) ≤ Max then
12 ENQUEUE(Q, MINDIST(Child, Item_{m2}), \{Child, Item_{m2}\})
13 endif
14 enddo
15 endif

\[ \text{Min} \]

\[ \text{Max} \]

In two dimensions, the faces are line segments.
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 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 $(i_1, i_2)$. This means that much fewer than $K$ pairs are sufficient for estimating the maximum distance of $K$ object/object pairs.

In the following, $D_{\min}$ 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 $\infty$. Our goal is to reduce $D_{\max}$ as much as possible, given $K$, the maximum number of pairs requested. Whenever a pair $(i_1, i_2)$ 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 $(i_1, i_2)$ 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 $(i_1, i_2)$ 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 $(i_1, i_2)$, 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 $(i_1, i_2)$ based on the average node occupancy. However, if the number of pairs generated from $(i_1, i_2)$ is over-estimated, then this may lead to a value of $D_{\max}$ that is too small (i.e., smaller than the $K^\text{th}$ 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 is highly geared towards producing pairs early.

The second extension is to impose a secondary ordering on 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 $\infty$ for nonintersecting pairs, but for intersecting pairs, 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 object pairs that will be requested. This is instead of estimating the maximum distance as 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 \((o_1, o_2)\) 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 \((o_1, o_2)\) 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 \((o_1, o_2)\) has been seen before (i.e., if it is present in the set \(S_o\)).

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 \((i_1, i_2)\) 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 \((i_1, i_2)\) is bounded by the number of objects in the subtree of \(i_1\), assuming \(i_1\) is a node. When an item \((i_1, i_2)\) is about to be inserted into \(M\), we must first check if another item \((i_1, i_2)\) exists in \(M\). If so, then we replace \((i_1, i_2)\) by \((i_1, i_2)\) if the latter has a smaller \(d_{\text{max}}\) value and ignore \((i_1, i_2)\), otherwise. There are two additional subtle differences. When \((o_1, o_2)\) is reported, any pair \((o_1, i_2)\) in \(M\) must be removed. Also, a pair \((n_1, i_2)\) may only be inserted into \(M\) if \(n_1\) has never been processed for any pair \((n_1, i'_2)\). Otherwise, some of the objects in the subtree of \(n_1\) would be counted more than once (since processing \(n_1\) in the pair \((n_1, i'_2)\) may lead to some pairs \((e_1, i'_2)\) to be inserted into \(M\) where \(e_1\) is an entry in \(n_1\)).

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 with the second definition since it corresponds to reporting for each object \(o_1\) the first pair \((o_1, o_2)\) that occurs in a reverse distance join. The first definition would mean reporting for each object \(o_1\) the last pair \((o_1, o_2)\) 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

\(^{5}\)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.
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.

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 \( D_1 \) are stored in the memory-based heap, pairs with a distance less than \( D_2 \) are stored in an unorganized list in memory, while pairs with a distance of \( D_2 \) 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 \( D_1 \) is changed to \( D_2 \), a new value is chosen for \( D_2 \), and pairs on disk with distance between the new values of \( D_1 \) and \( D_2 \) 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 \( D_1 \) and \( D_2 \), 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, is subjects for further investigation.

In Section 2.3, a set \( S_n \) 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 \( S_n \). 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 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 the experiments, we joined Water with Roads, except where noted.

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 producing up to 100,000 result pairs of the distance join. The algorithm version used in these experiments was such that 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. 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 “Even/DepthFirst”) against three other versions: (1) “Even/BreadthFirst” orders pairs with the same distance such that it leads to breadth-first traversal; (2) “Basic/DepthFirst” is the basic algorithm of Figure 3, where we always process the first node in node/node pairs; and (3) “Simultaneous/DepthFirst” 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 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.

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.
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.

<table>
<thead>
<tr>
<th>Pairs</th>
<th>Time</th>
<th>Dist. Calc.</th>
<th>Queue Size</th>
<th>Node I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.9</td>
<td>307994</td>
<td>1002536</td>
<td>3019</td>
</tr>
<tr>
<td>10</td>
<td>9.0</td>
<td>393758</td>
<td>1333856</td>
<td>4087</td>
</tr>
<tr>
<td>100</td>
<td>9.4</td>
<td>395780</td>
<td>1356985</td>
<td>4652</td>
</tr>
<tr>
<td>1000</td>
<td>9.8</td>
<td>403281</td>
<td>1434160</td>
<td>6487</td>
</tr>
<tr>
<td>10000</td>
<td>12.6</td>
<td>422392</td>
<td>1632895</td>
<td>11502</td>
</tr>
<tr>
<td>100000</td>
<td>23.8</td>
<td>479262</td>
<td>2229874</td>
<td>28356</td>
</tr>
</tbody>
</table>

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 Roads is larger, many more pairs are generated (this case, Roads is traversed first). In fact, for producing result 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.

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 100 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 ±2.4%).

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. 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

---

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.
higher $D_T$ value (i.e., “Hybrid2”) was better, most likely because 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 a higher number of unnecessary priority queue elements were kept out of the memory based part of the queue. 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.](image)

### 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 re-reading 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. Unfortunately, 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. However, a large disk space would also be required to generate and sort 100 million pairs using the nested loop approach.

### 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 $(i_1, i_2)$ where $i_1$ is an object or an object bounding rectangle and $i_3$ 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 and thus is not shown beyond 10,000 pairs. 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 $S_x$ 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.](image)

#### 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. In the figure, “MaxDist All” is the result of setting the maximum distance to be the largest possible distance between two objects in the result of the dis-
distance semi-join and for “MaxPair All”, the upper bound on the number of pairs is set to the number of points in Water. The figure confirms the benefit of restricting the maximum distance. Notice that setting the maximum number of pairs to 1000 does indeed 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”. These numbers are about 14% lower and 13% higher, respectively, than when maximum distance is not set. Thus, we can see that imposing a maximum distance or setting an upper bound on the number of pairs to be generated only yields significant savings in execution time when the maximum is low (up to 50% savings or more), while high values on the maximum yields little if any savings.

![Figure 10: Execution time for different maximum distance and maximum pairs for distance semi-join.](image)

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 Roads 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 \(d_{\text{max}}\) 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 deep-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 \((o_1, o_2)\) 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 engine 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 a 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 Bjoern P. Jonsson and Dr. Robert E. Webber for their critical comments.

References


