Spatial Joins Using Seeded Trees*

Ming-Ling Lo and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
University of Michigan–Ann Arbor
1301 Beal Avenue, Ann Arbor, MI 48109
mingling, ravi@eecs.umich.edu

Abstract

Existing methods for spatial joins assume the existence of indices for the participating data sets. This assumption is not realistic for applications involving multiple map layer overlays or for queries involving non-spatial selections. In this paper, we explore a spatial join method that dynamically constructs index trees called *seeded trees* at join time. This method uses knowledge of the data sets involved in the join to speed up the join process.

Seeded trees are R-tree-like structures, and are divided into the *seed levels* and the *grown levels*. The nodes in the seed levels are used to guide tree growth during tree construction. The seed levels can also be used to filter out some input data during construction, thereby reducing tree size. We develop a technique that uses intermediate linked lists during tree construction and significantly speeds up the tree construction process. The technique allows a large number of random disk accesses during tree construction to be replaced by smaller numbers of sequential accesses.

Our performance studies show that spatial joins using seeded trees outperform those using other methods significantly in terms of disk I/O. The CPU penalties incurred are also lower except when seed-level filtering is used.

1 Introduction

Spatial databases and GIS systems have received increasing attention in recent years. Most research on query processing in such system has focused on the spatial selection, or spatial search operation. Examples of spatial selection operations are window queries, which find all spatial objects contained within or intersecting a predefined window area, and point queries, which find all objects overlapping a particular point in space. Many spatial access methods have been devised to facilitate such operations [Sam90].

*This work was supported in part by the Consortium for International Earth Science Information Networking.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

However, relatively little work has been done on the spatial join, one of the most important operations for spatial databases and GIS applications. Spatial joins are expensive but necessary in such applications as map overlay. Existing join algorithms can generally be divided into those based on join indices, those based on z-ordering, and those based on tree-like indices. The spatial join index [Rot91] method builds a spatial version of a join index [Val87] for two data sets if spatial joins between these data sets are known to be frequent. This method trades the overhead of pre-computation at index building time for accelerated processing at join invocation time. It assumes that grid-files [NHS84] are used as the main access method, which must exist for the relevant data sets before spatial join indices can be built for them. A similar method using distance-associated join indices [LH92] has also been proposed to speed up spatial range queries. Orenstein [Ore89, Ore90, Ore91] proposed z-order-based algorithms to perform both spatial selection and join. In these algorithms, the space under study is first decomposed into *elements*, which can then be sequenced by their z-order and organized using one-dimensional indices such as the B^+ tree. Joining two spatial data sets amounts to merging two z-value streams.

1.1 Tree-like Index-Based Join Algorithms

Several spatial join algorithms using tree-like indices have also been proposed. Gunther [Gun93] analyzed the applicability to spatial joins of the join techniques used in relational database systems, and proposed a general join algorithm using the concept of *generalization trees*, which are abstractions of tree-like spatial indices. The algorithm can be applied to join two data sets as long as some tree-like indices exist for both data sets. Since breadth-first order is used in tree traversal, the pairs of matching tree-nodes at tree level n must be recorded before the algorithm can descend to level $n + 1$. In practice, the amount of memory required to hold such information could be large for indices with high fan-out, such as R-trees. Analytical models were to used to study the performance of various techniques, but

memory constraints were not considered in depth.

The R-tree and its variations [Gut84, BKSS90, FSR87, SRF87] have been gaining popularity due to their relatively simple structure and their efficient handling of spatial objects with extent, such as region objects. An R-tree is a B^+-tree like access method that stores multidimensional spatial objects. A non-leaf R-tree node contains entries of the form `(mbr, cp)`, where `cp` is a pointer to a child node and `mbr` is the minimum bounding rectangle of all objects described by entries in the child node. A leaf-node contains entries of the form `(mbr, oid)`, where `oid` refers to a spatial object, and `mbr` is its minimum bounding rectangle. R-trees reference their stored spatial objects in whole units, without clipping or transforming them into higher-dimensional points.

Brinkhoff *et al.* [BKS93] proposed a join algorithm based on the R-tree or variations. This join method requires each participating data set to have a pre-computed R-tree index. The join algorithm consists of a R-tree matching algorithm and a collection of techniques to reduce CPU and disk I/O costs.

The tree matching algorithm is straightforward. It starts by matching the children of the root nodes of the two R-trees for overlaps, and then recursively traverses down the matched children, resulting in a depth-first tree traversing order. Results are reported when leaf nodes are reached in both trees. We will denote this tree matching component of the join algorithm by *TM* in subsequent discussions.

Improvement techniques described in the paper included those aimed at reducing CPU costs and those aimed at reducing the amount of disk I/O. When the bounding boxes representing two R-tree nodes R_1 and R_2 were found to overlap, their intersection area was used to eliminate some children from further consideration. If the bounding box of a child of R_1 did not overlap the intersection area, no answer would result from matching this child and any child of R_2. Also, when looking for overlapping child pairs, the children could be sorted on one axis, and a *plane-sweeping* technique used to further reduce the number of comparisons needed. As a result, the number of overlapping tests in the join process was significantly reduced. To reduce disk I/O, plane-sweeping order was also used to decide the order in which the children of a node were traversed. A page pinning technique based on *degrees* was also used. The results also showed decreases in I/O costs, though less substantial than those in CPU costs. Realistic buffer sizes were used in performance experiments.

1.2 Motivation

All these previous methods require some index structures to have been constructed for the input data sets and/or some pre-computation to have been done before a spatial join is invoked. Such requirements can be inconvenient or even impossible to satisfy in many situations. First, it may not be cost-effective to maintain index structures for all data sets regardless of their usage patterns and frequencies. In addition, joins may be required on the results of a series of previous selections on non-spatial attributes, the results of previous spatial joins, or the results of some other spatial operations. Clearly, in such cases, no spatial indices exist for the newly derived data sets. Any pre-computed spatial index for the original data sets may not be easily or efficiently applied to the join operation.

As an example, consider two data sets D_B and D_P that cover some common area. D_B contains all buildings in the area, and D_P contains all parks in the area. Assuming that R-tree indices have been constructed for both D_B and D_P, consider the following join operations:

Q1: Find all buildings that overlap with a park.

Q2: Find all government-owned buildings that overlap with a park.

For the first query, existing algorithms, such as that of Brinkhoff [BKS93], can be applied using the pre-computed R-trees. However, for the second query, if the total number of buildings is large, and only a small fraction of the buildings is government-owned, it may be more efficient to perform a non-spatial selection first to find the set of all government-owned buildings and then perform the spatial join. This approach would involve an intermediate data set for which no spatial index exists.

In real-life applications, it is also common for a spatial query to involve joins between multiple data sets. If an input to a spatial join is the output of previous spatial joins, it might be only remotely related to the original data set. Utilizing pre-computed access methods for the original data sets for such queries could be very inefficient or infeasible. Other spatial operations such as *re-classification*, *aggregation* and *buffering* [SE90] may also occur in queries, requiring transformation of the original data sets and further complicating the situation. For convenience, we call a data set that is the output of an earlier spatial or non-spatial operation a *derived* data set.

Our approach to supporting such queries is to dynamically build access methods for the derived data sets as necessary to support spatial join. However, most spatial access methods [Gut84, BKSS90, FSR87, SRF87] were originally designed for a different context. In particular, such indices are assumed to be built up incrementally, so the construction algorithms for these indices are not optimized for all-at-once construction. Most of these indices are designed to minimize the cost of spatial selection, not that of spatial join. Thus, they tend to reduce the average number of disk accesses per spatial selec-

tion and the worst-case selection costs, and to minimize disk space consumption. Such characteristics are not always the most important ones in the context of spatial joins. In addition, there is useful information available at join time that existing spatial index construction algorithms do not exploit. For example, the sizes of the input data sets are known at join time. So is the spatial distribution characteristics of these data sets. We need a spatial data structure that is inexpensive to create at join time, and takes advantage of information available at join time to support efficient spatial join processing.

In this paper, we address this problem with a new spatial join method using index structures called *seeded trees*. We assume a system in which the R-tree is the main type of spatial index, but our method does not require R-tree-like indices to pre-exist for both participating data sets. Thus, it handles situations where using pre-computed indices is not practical. The seeded trees are constructed dynamically, its construction algorithm taking advantage of the information about the join and the input data sets available at join time. Upon encountering a join operation, a seeded tree is first constructed for one of the participating data sets. The join then proceeds, using some standard tree matching algorithm to compute results. Our method works with any tree matching algorithm, but we will use algorithm TM proposed in [BKS93] as the tree matching component of our method, given its simplicity and reasonable performance. Our experiments show that our method outperforms existing methods, both during tree construction and actual join phases.

This paper is organized as follows. Section 2 describes the structure and behavior of the seeded trees in detail. Section 3 presents techniques to reduce tree construction costs. Section 4 reports on our performance studies. Section 5 discusses related issues, and Section 6 concludes this paper.

2 Seeded Trees

Let us assume that we want to join a derived data set D_S, for which no pre-computed spatial index exists, with an ordinary data set D_R , for which we are given an R-tree index. Our algorithm constructs a seeded tree for the derived data set, and matches the seeded tree with the existing R-tree index. Let T_S denote the seeded tree for D_S, and let T_R denote the R-tree for D_R.

The central idea behind the seeded tree method is to use available information to reduce join costs. When a seeded tree T_S is to be constructed for D_S, we know that D_R and its R-tree T_R will be used in the join process. We can make use of the characteristics of D_R and T_R to expedite the join process. It has been noted that the performance of an R-tree-like index depends not only on its constituent data objects but

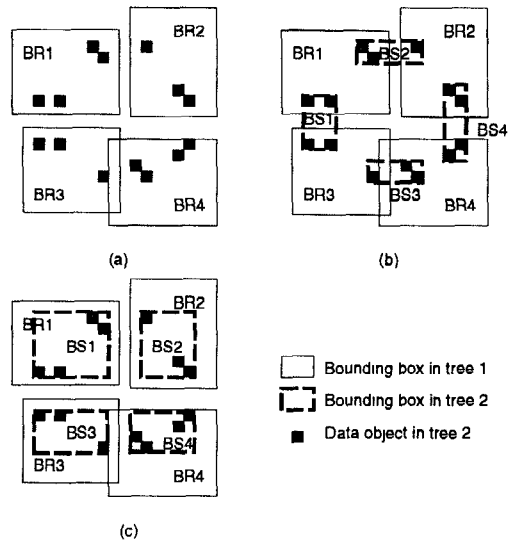

Figure 1: Beneficial and non-beneficial formation of bounding boxes in a seeded tree.

also on the order in which data objects were inserted into it [Gut84, BKSS90]. The data inserted earlier will decide the initial organization of the tree and hence the position in the tree of the data inserted subsequently. It has also been observed that deleting and re-inserting a fraction of the data objects in an R-tree improves its performance [BKSS90]. Such phenomena suggest that when constructing a seeded tree for a spatial join, we can start with a small tree to guide tree growth, instead of starting from a single root node. Furthermore, since we know the seeded tree will be matched with T_R, the characteristics of T_R can be used to determine this small initial tree. By choosing the information used in the initial tree well, we may expect to have a seeded tree that is shaped more suitably for joining with D_R.

The importance of tree organization in spatial joins is illustrated by the example in Figure 1a. Say we have an R-tree T_R and fourteen data objects to be inserted into a seeded tree T_S, and that T_R will be joined with T_S. Assume tree fan-outs of four. Figure 1b shows the bounding boxes of the children of the root of T_S when the bounding boxes are organized to achieve smallest area. If the data objects are inserted into T_S in such an order that these bounding boxes are actually achieved, the join process will match each of the seeded tree bounding boxes BS_1, BS_2, BS_3 and BS_4 against two bounding boxes in T_R. However, if the bounding boxes in T_S are allowed to be non-minimal but are organized as in Figure 1c, each seeded tree bounding box will be matched against only one bounding box in T_R. Thus, the criteria for organizing tree indices are different when the tree is optimized for spatial selection and when the tree is optimized for spatial join. If we can create an initial tree so that the data objects will be inserted as

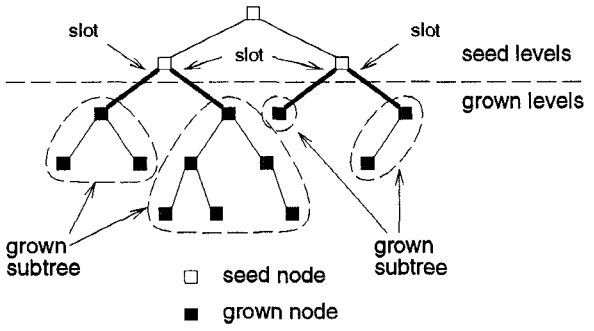

Figure 2: Example of a seeded tree.

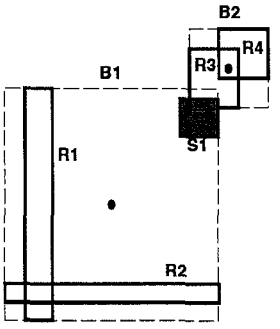

Figure 3: Efficient and inefficient bounding boxes

in Figure 1c, both I/O and CPU cost can be reduced during the join process.

In the seeded tree method, this goal is achieved by copying the first k levels of the R-tree T_R to the seeded tree. Structurally, a seeded tree consists of the *seed levels* and *grown levels* (see Figure 2). The tree nodes at the seed levels are called *seed nodes*, and those at the grown levels are called *grown nodes*. The seed levels start from the root and continue consecutively for a small number of levels. The grown levels span from the children of the last seed level to the leaf level. As with R-tree nodes, a non-leaf node in the seeded tree contains entries of the form `(mbr, cp)`, where `cp` points to a child node, and `mbr` is the minimum bounding rectangle of all objects contained in the child node. A leaf node contains entries of the form `(mbr, oid)`, where `oid` refers to a spatial object in the database, and `mbr` is the bounding box of that object. The entries in a seed node can have an additional `shadow` field, if *seed level filtering* will be performed during data insertion (see Section 3.2). In the following discussion we assume `shadow` fields do not exist.

2.1 Seeding Phase

The construction of a seeded tree consists of a *seeding phase*, a *growing phase* and a simple *clean-up phase*. The seed levels are numbered from 0 (the root level) through $k-1$, and the grown levels span from level k to level l (the leaf level).

In the seeding phase, the seed levels of the seeded tree T_S are set up by copying over the top k levels of the R-tree T_R. The R-tree T_R, from which the copied information is derived, is called the *seeding tree*. The bounding box fields of the T_R nodes may undergo some simple transformations before being copied into corresponding T_S nodes. The pointer fields of the seed nodes at levels 0 to $k-2$ are set to point to their child nodes. The pointer fields of level $k-1$ seed nodes are set to `NULL`. We call each `(mbr, cp)` pair at level $k-1$ a *slot*, and level $k-1$ of the seeded tree, the *slot level*. The information copied into the seed nodes will guide data insertion in the growing phase, thus deciding the shape into which the tree will eventually grow.

The seed levels of the seeded tree have the following properties:

(1) The seed nodes at the slot level have null pointers but non-null bounding boxes.

(2) During tree construction, the bounding boxes in the seed nodes are used only to guide data insertion. Thus, in a seed node, the value of `mbr` in a bounding box and pointer pair `(mbr, cp)` need not reflect the true minimal bounding box of all data reachable through `cp`. The bounding box fields must be modified into the true minimal bounding boxes before tree matching begins.

(3) Although the bounding box fields of seed nodes and the pointer fields of the slot level nodes may change as needed during data insertion (the growing phase), the *structure* of the seed levels never changes. *In particular, node splitting at the grown levels never propagates upwards into the seed levels.* The behavior of the grown nodes will be described in detail in Section 2.2.

Since the seed levels guide the growth of the tree, the values in the bounding box fields of seed nodes are crucial to the performance of the seeded tree. Simply copying over the bounding boxes from the seeding tree T_R to the seeded tree T_S may not always be the best strategy. Copying badly formed minimum bounding boxes from the seeding tree will penalize the performance of the seeded tree. As an example, consider Figure 3, where minimal bounding box B_1 contains two long rectangles R_1 and R_2, and minimal bounding box B_2 contains two squares R_3 and R_4. As a result, bounding box B_1 has a large dead area and only badly describes its children, whereas B_2 is a more compact and better description of its children. If the bounding boxes B_1 and B_2 were to be copied unchanged into the seeded tree, and we use minimal area increase as the criterion for insertion [Gut84], object S_1 would be inserted into B_1 instead of B_2, which could result in unnecessary disk accesses during the join process.

Other information can be copied into the bounding box field of seed nodes. In the previous example, if we had copied the center points of bounding boxes from the seeding tree, S_1 would have been inserted properly into B_2. In this study we investigate three different strategies for copying information from the bounding box fields of seeding tree nodes:

C_1: copy the minimal bounding boxes.

C_2: copy the center points of the minimal bounding boxes.

C_3: At the slot level, copy the center points of the minimal bounding boxes. At other levels, the bounding box field contains the true minimum bounding box of its children.

Our results show that copy strategies C_2 and C_3 almost always out-perform strategy C_1.

2.2 Growing Phase

During the growing phase, data objects in D_S are inserted into the seeded tree. Figure 4 shows an example of seeded tree growth. To insert a data object, we traverse the tree from the root to the slot level, at each level choosing a suitable node to traverse from the next level. Eventually the slot level is reached and a slot chosen for inserting the data. If this is the first insertion through this slot, the child pointer of the slot will be `NULL`. In this case, a new grown node is allocated, the child pointer is set to point to the new node, and the data object inserted into it. Otherwise the data are inserted into the grown node found through the slot pointer. This grown node behaves like the root of an ordinary R-tree. When it overflows due to insertions, it will be split into two grown nodes, and a third grown node allocated to become the parent of the two nodes. The slot pointer is modified to point to the new root. Subsequent insertions through this slot behave like ordinary R-tree insertions, the root of the R-tree being the node pointed to by the slot pointer.

Recall that node splitting does not propagate up to the seed levels, and that the structure of the seed levels remains unchanged during the whole growing phase. Thus, a seeded tree can be visualized as consisting of a small tree of seed nodes, with an R-tree forest of grown nodes attached to the slots. The R-tree pointed to by the each slot pointer is called a *grown subtree* (see Figure 2).

At each seed level we must choose a child from the next level to traverse, until a slot is found. We make this choice based on the information stored in the bounding box fields of each node. The exact criterion for child selection depends on whether the value stored is a central point or an area. If central points are stored, we choose a child whose central point is close to the central point of the data being inserted. If areas are stored,

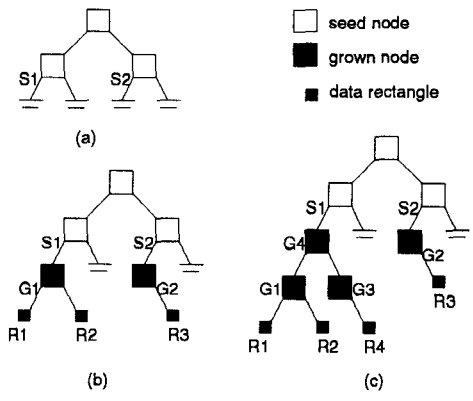

Figure 4: Example of seeded tree growth. The fan-out of tree nodes is two. In (a) the seed levels have just been set up. In (b) rectangles R_1, R_2 and R_3 are inserted through slot S_1 and S_2. Grown nodes G_1 and G_2 are allocated as a result. In (c), rectangle R4 is inserted through slot S_1, and overflows G_1. A new root G_4 of the sub-tree is allocated and S_1 is made to point to it.

we choose a child that yields the smallest bounding box area after insertion, subtracting from it the sum of the areas of the old bounding box and the input rectangle. This criterion is the same as that used in R-tree construction.

The bounding box fields of the traversed seed nodes are not always updated after each data insertion. We can choose whether to update these bounding boxes, and how how to update them. If we choose not to update these bounding boxes, subsequent insertions will continue using the original bounding boxes in trying to find a slot, and will be guided only by the characteristics of the seeding tree. Updating bounding boxes right after each insertion causes the bounding boxes to reflect the data inserted through their associated pointer at all times, so that subsequent insertions will be guided not only by the information derived from the seeding tree but also by the part of data set D_S inserted so far. In this study, we investigate the following bounding box update policies:

U_1: No updates after insertions.

U_2: Update traversed bounding boxes after each insertion to enclose the inserted data objects and the original seed bounding box.

U_3: Same as U_2, but the updated bounding box encloses only inserted data, but not the seed bounding box.

U_4: Update bounding box at the slot level as in U_2. Bounding boxes at other seed levels are not updated.

U_5: Update bounding box at the slot level as in U_3. Bounding boxes at other seed levels are not updated.

The bounding boxes at the grown level are updated as in ordinary R-trees.

The *clean-up phase* begins after all data object in D_S are inserted into the seeded tree. The bounding box fields of seed node are adjusted to be the true minimum bounding boxes of their children. Slots containing no data objects are deleted and relevant data structures made consistent.

The tree matching process begins after the seeded tree is built. Note that with the above insert algorithm, more data objects may have been inserted into some slots than into others. As a result, grown subtrees may have different heights. However, since the tree matching procedure TM [BKS93] does not require the participating trees to be balanced, it can be applied directly without any difficulty. Furthermore, any optimization technique developed for matching R-trees can be applied to matching seeded trees as long as tree balance is not a pre-requisite.

3 Tree Construction Improvements

Since traditional tree-like indices have been optimized for incremental updates rather than for being constructed all at once, their total construction costs can be high. As we construct seeded trees dynamically at join time, it is important that we reduce such costs. In this section, we examine a method that uses intermediate linked lists to eliminate most of the random disk accesses during tree construction, hence substantially reducing total join costs. We also present a technique that exploits the structure of the seed levels to reduce the sizes of seeded trees.

3.1 Tuning Construction Costs

We have found that the costs of constructing tree-like indices all at once arise mainly from buffer misses as the trees grow and overflow the memory buffer space. The actual construction costs depend on the relative sizes of the tree and the buffer. For both R-trees and seeded trees using straightforward construction algorithms, such costs can be very high. In the particularly bad cases, buffer misses have resulted in disk I/O costs several times higher than that of the actual join process. For example, we have observed that constructing an R-tree for a 800K-byte data set with 40K data objects, using a 512 page buffer with 1K-byte page size can result in 7,234 disk accesses during construction (see Table 2, second row). This number is nine times the number of disk accesses needed to read the input data set, disregarding the difference between random and sequential disk accesses, and three times the number of disk access needed for a match with an R-tree having 100K entries. Another factor affecting the construction cost is the degree of clustering in the input data stream. If data objects close to each other in space are also close in their input order, the chances of buffer misses will be lower. However, such clustering is hard to guarantee in general.

For seeded trees, we have been able to avoid most random disk accesses due to buffer misses by forming intermediate linked lists under the slots (see Figure 5). During the growing phase, if we estimate that the tree size will be larger than the buffer size, the data inserted through a slot will not be built into a grown subtree immediately, but first organized into a linked list of data pages. A data page in the linked lists contains an array of entries, each with a bounding box and a data pointer field. The linked lists grow as data objects are inserted. Eventually all data pages in the buffer will be allocated. If we now want to insert an additional data object into a linked list in which all data pages are full, we write all linked lists longer than a small pre-defined constant to disks, freeing up most of the buffer space. The corresponding slot pointers are reset to `NULL`. The set of linked lists so written is called a *batch*. The insertion process then proceeds as before. When all data objects in D_S are inserted, we can start constructing grown subtrees from the linked lists. An R-tree is built for each group of linked lists that have been grown under the same slot, using the data objects recorded in the lists. The slot pointer is then modified to point to the root the R-tree.

By using such intermediate linked lists, we can construct the grown subtrees one by one instead of all together. Since there are many slots in the seeded tree, and hence many grown subtrees, the average size of a grown subtree is much smaller than the size an R-tree built with the same input data. The chances of a grown subtree overflowing the buffer are therefore much smaller, and the number of random disk access is significantly reduced. The price this method must pay is an increase in the number of sequential accesses for writing and reading the linked lists. However, since sequential access is much faster than random access in disk I/O, this results in much faster construction times.

The number of slots in a seeded tree is determined by the number of seed levels. For as few as two seed levels, the number of slots varies be from a few tens to hundreds, assuming at least a fan-out of at least 50. This means that the algorithm could work for seeded trees of size at least tens of times larger than the buffer size. Note that even if some grown subtrees do overflow the buffer, they are likely to be much smaller than R-trees built using the same input, and the penalty incurred likely to be much smaller. In practice, however, buffers are unlikely to overflow even with some data skew since the average subtree size can be made much smaller than the buffer size. In our experiments, we have constructed seeded trees with more than 2,500 nodes using a 512-page buffer size. The worst buffer overflow we have experienced during seeded tree construction is two buffer misses.

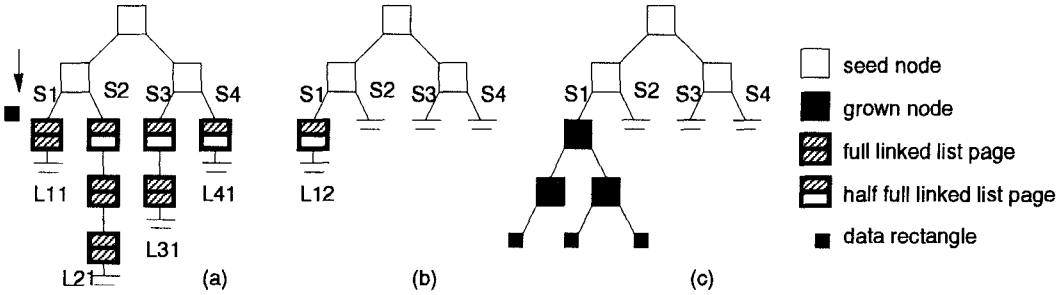

Figure 5: Tree construction using linked lists. Assume each data page has a capacity of two data objects, and the buffer has a capacity of 10 pages. (a) shows four linked lists formed under the four slots. All 10 pages in the buffer have been allocated, and one additional data object is now inserted into linked list under slot S_1, resulting in buffer overflow. In (b), the batch of linked lists formed in (a) has been written to disk due to buffer overflow. A new linked list is formed under S_1 for the data object inserted in the last step. Grown subtrees are constructed using the data in the linked lists when all the data objects are inserted. (c) shows one grown subtree constructed using linked lists L_{11} and L_{12}, the linked lists constructed under slot S_1.

3.2 Reducing Tree Sizes with Seed Level Filtering

The seed levels of the seeded tree serve one additional function: they can be used to reduce the seeded tree size. Smaller tree sizes incur less disk I/O during both tree construction and actual join time.

It is easy to see that a rectangle overlaps some data objects indexed by an R-tree if and only if it overlaps at least one bounding box at *each level* of that R-tree. Since the seed levels of T_S resemble the first k levels of the R-tree T_R, they can be used to test whether a data object overlaps T_R. Data objects found to be non-overlapping with T_R need not be inserted into T_S at all. We call this test *seed level filtering*.

When seed level filtering is used, the entries in the seed nodes carry one additional bounding box field during tree construction time, called `shadow`. When seeding phase information is copied from the T_R to the seed levels of T_S, the `mbr` fields of the T_R nodes are copied into the `shadow` fields of the corresponding T_S nodes without change. Once copied, these fields are not changed during the entire tree construction phase. When a data object is to be inserted during the growing phase, we first check if the data object overlaps at least one `shadow` field at each of the k seed levels. If there is no overlap, the data object does not overlap any data object in D_R and need not be inserted into T_S at all. The `shadow` fields are not used after tree construction, and so the space occupied by them becomes free.

4 Experiments

Assume that a spatial join is to be performed on a derived data set D_S and a data set D_R, for which an R-tree T_R exists. We conducted experiments with seeded-tree joins using three spatial join algorithms: ***STJ***, ***RTJ***, and ***BFJ***. Algorithm ***STJ*** (*Seeded Tree Join*) is our algorithm, as described so far. It constructs a seeded tree T_S for the data set D_S, and then matches the tree indices T_S and T_R. Algorithm ***RTJ*** (*R-Tree Join*) is a simple variation of the algorithm proposed by Brinkhoff *et al.* [BKS93]. It first constructs an R-tree T_S for D_S, and then matches T_S with T_R. Algorithm ***BFJ*** (*Brute Force Join*) simply performs a series of window queries on the R-tree T_R, using the data rectangles in D_S as query windows. The aggregation of answers to these window queries is equivalent to a spatial join between D_R and D_S. ***RTJ*** and ***STJ*** both use the CPU and disk I/O tuning techniques described in [BKS93]. For generality, the original R tree structure was used, and not any of its variations.

For simplicity, we assume that both the disk page size and the memory page size are 1K bytes, as are the sizes of both the seeded tree nodes and the R-tree nodes. The data files are assumed to contain entries consisting of a 16-byte bounding box and a 4-byte object identifier. We also assume a dedicated buffer of 512 pages. For algorithms ***STJ*** and ***RTJ***, the buffer is used during both tree construction and tree matching. During construction of T_S, the buffer pages containing newly created tree nodes are marked as dirty and must be written to disks if the pages are to be re-used. We do not purge the dirty buffer pages after tree construction. Hence tree matching starts with a warm buffer cache. Note that there could be disk writes during the tree matching process if dirty buffer pages containing T_S nodes are re-allocated by the buffer manager.

We studied data of different degrees of spatial clustering. The degree of clustering was controlled by a simple scheme. When generating a data set of $x \times y$ objects, we first generated x *cluster rectangles*, whose centers were randomly distributed in the map area. We then randomly distributed the centers of y data rectangles within each clustering rectangle. By controlling the total area of the clustering rectangles, we could control the degree of clustering of the data set. The smaller the to-

tal area of the clustering rectangles, the more clustered the data set. The length and the width of each clustering rectangle was chosen randomly and independently to lie between 0 and a predefined upper bound. This upper bound controlled the total area of the clustering rectangles. The size and shape of data rectangles were similarly chosen using a smaller upper bound. When clustering rectangles or data rectangles extended over the boundary of the map area, they were clipped to fit into the map area. When a data rectangle extended over the boundary of its clustering rectangle, it was not clipped. In the experiments, the number of data objects per cluster was set to be 200, and the number of clustering rectangles was set according to the total number of data objects. Without loss of generality, the map area under study was assumed to range from 0 to 1 along both X and Y axes.

4.1 Experimental Results

We conducted two series of experiments. In the first series, we fixed the cardinality of D_R at 100,000, resulting in an R-tree of 4 levels, and varied the cardinality of D_S from 20,000 to 80,000. The degree of spatial clustering of data was set by setting the upper bound on the side length of clustering rectangles to 0.04. The resulting cover quotient of the clustering rectangles in D_R was 0.2, meaning that the centers of all the data objects in D_R were restricted to 20% of the map area. For each data configuration, we tested ***RTJ*** and ***BFJ*** and conducted an extensive study of ***STJ*** variations by applying combinations of different seed node copy and update policies, as described in Section 2. For each combination of policies, we studied the effect of the number of seed levels, and the effect of seed level filtering on performance.

In the second series of experiments, we fixed the cardinality of D_R and D_S at 100,000 and 40,000, respectively, and varied the degree of clustering of the data sets. We adjusted the upper bound on side length of the clustering rectangles so that the cover quotient of the clustering rectangles of D_R equaled 0.2, 0.4, 0.6, 0.8 and 1.0, respectively. The upper bound on side length of the clustering rectangles of D_S was set to be same as that of D_R in each experiment. We ran the same set of seeded tree variations as the in first series of experiments for each data configuration.

Among the various combinations of seed node copy and update policies, we found that copy strategies C_2 and C_3 (see Section 2.1) and update policies U_3, U_4 and U_5 (see Section 2.2) always gave better performance. The differences between the three best update policies were marginal. Due to space limitations, we list only the results from seeded trees built using combinations (C_3, U_3), and (C_3, U_4), denoted as STJ1 and STJ2, respectively.

Our experiments showed that the ***STJ*** versions

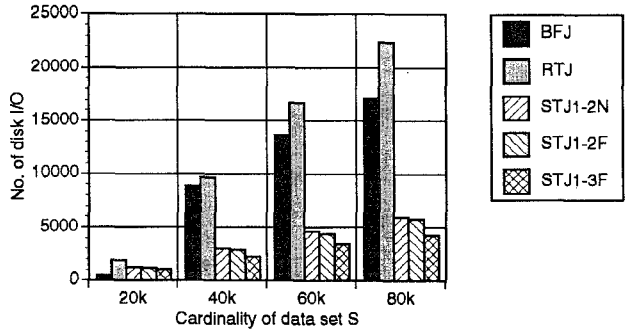

Figure 6: Total disk I/O costs, Experiment Series 1.

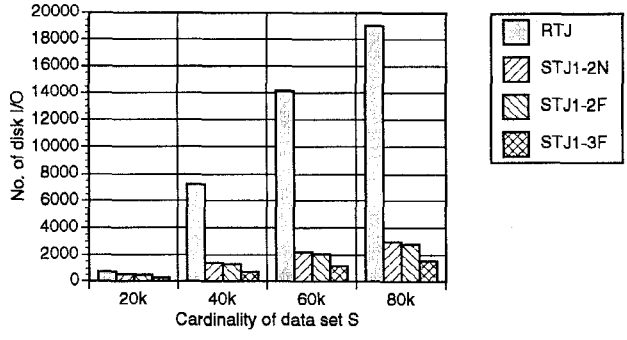

Figure 7: I/O costs for tree construction, Experiment Series 1.

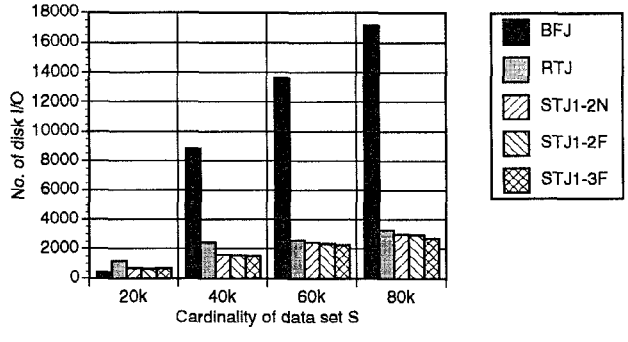

Figure 8: I/O costs for tree matching, Experiment Series 1.

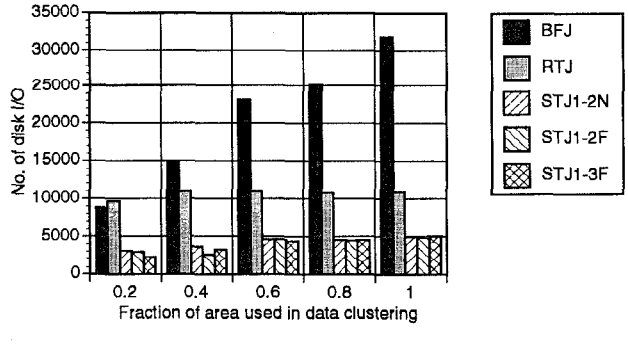

Figure 9: Total disk I/O costs, Experiment Series 2.

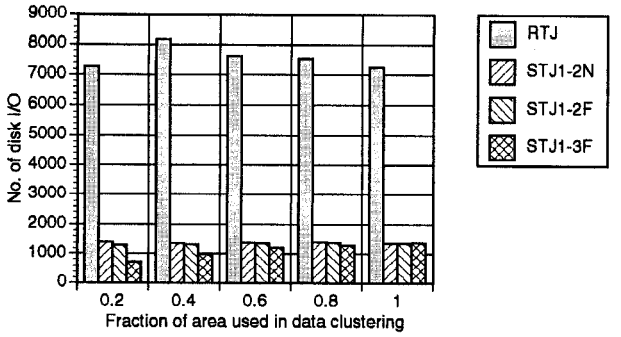

Figure 10: I/O costs for tree construction, Experiment Series 2.

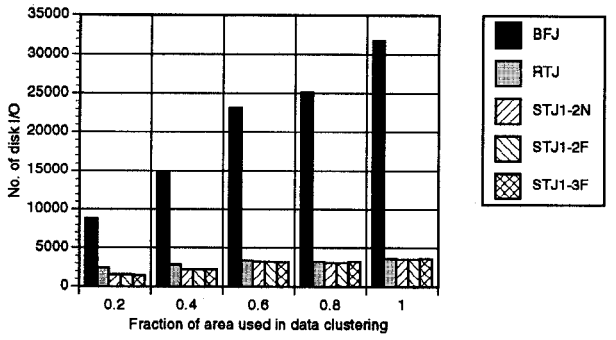

Figure 11: I/O costs for tree matching, Experiment Series 2.

substantially outperformed ***BFJ*** and ***RTJ*** in all cases in terms of disk I/O costs. The ***STJ*** versions also incurred the lowest CPU costs among all algorithms when seed level filtering was not activated. With seed level filtering, the CPU costs of ***STJ*** were between the costs of ***BFJ*** and ***RTJ***, and were about an order of magnitude greater than the costs without seed level filtering. Figures 6 and 9 summarize the disk I/O costs observed in the two series of experiments. Figures 7, 8, 10, and 11 break down the I/O costs into those of tree creation and those of tree matching. The ***BFJ*** method creates no tree structures on the fly, and so incurs no tree creation costs.

Tables 1 through 4 show the disk I/O and CPU costs of the first series of experiments in detail. The number of seed levels of the seeded trees used in ***STJ*** variants appears after the hyphen following the algorithm name. A letter "F" following this number indicates that seed level filtering was activated, and a letter "N" indicates that no filtering was performed. Thus, **STJ2-2F** stands for variation 2 of ***STJ*** with two seed levels and no filtering. Disk I/O is shown as the number of random disk accesses. A sequential disk access counts as 1/30 of a random disk access. Under I/O costs, the columns "match" and "construct" give the costs incurred during tree matching and construction, and "rd" and "wr" show read and write costs, respectively. Note that since tree matching starts with a warm buffer left over from tree construction time, dirty pages containing T_S nodes may need be written to disks if the pages are re-allocated during tree matching. The write column under "match" should thus be charged to the tree construction part of the algorithms.

The CPU costs given are the numbers overlap tests performed by the algorithms in units of 1000. The column "bbox" lists the numbers of bounding box overlap tests performed during tree construction. Column "XY" is the numbers of operations that test whether two bounding boxes overlap along the X or Y axis, used during tree matching [BKS93].

In the the first series of experiments, we found as expected that the costs go up as the size of D_S increases. ***STJ*** outperforms ***RTJ*** in all experiments in terms of disk I/O. The number of disk reads during tree construction is particularly interesting. For ***RTJ*** this cost arises from buffer misses during tree construction. For ***STJ*** the cost arises both from buffer misses and from I/O for reading in the linked lists during tree construction. The numbers of creation time reads remain very small for ***STJ*** even for large D_S sizes, while for ***RTJ*** they vary from 1.5 times (Table 1) to 30 times (Table 2, 3 and 4) larger than those of ***STJ***. Our earlier experiments showed that ***STJ*** incurred similar numbers of creation time reads as ***RTJ*** when intermediate linked list was not used. Using intermediate linked lists in tree construction successfully eliminated most of the buffer misses in these cases.

STJ outperforms ***BFJ*** in all cases expect when D_S size is the smallest (see Table 1). We found that ***BFJ*** accessed only 483 different T_R nodes in this case. Since this number was smaller than the buffer size, no buffer overflow occurred. Overflow occurred for ***STJ*** with the same data sizes, since the buffer held nodes from both T_S and T_S during tree matching. However, with larger data sets, ***BFJ*** incurred two to three times the numbers of disk accesses incurred by ***STJ*** as soon as the number of accessed R-tree nodes exceeded the buffer size. To our surprise, we found that ***RTJ*** performed worse than ***BFJ*** in all cases. A closer look showed that though tree matching costs are lower for ***RTJ*** the high tree creation I/O cost due to buffer misses outweighed the savings.

Seed level filtering provides a consistent gain for the ***STJ*** variations in terms of disk I/O. Filtering using three seed levels performs better than that using only two. This trend is especially clear for tree creation costs. The reduction in I/O cost is paid for by an increase in CPU costs during tree creation. When seed level filtering is not used, ***STJ*** also incurs the lowest CPU cost among all algorithms.

Tables 2 and Table 5 through Table 8 list the results of the second series of experiments. In general, we have found that the processing costs rise as the degree of clustering decreases, especially for tree matching. This

Alg	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	438	0	0	0	438	2381	0
RTJ	1182	359	144	243	1914	130	170
STJ1-2N	694	319	94	137	1244	79	168
STJ2-2N	849	358	94	150	1451	84	170
STJ1-2F	685	314	94	85	1178	896	168
STJ2-2F	823	349	94	99	1365	898	170
STJ1-3F	712	226	94	5	1037	1945	160
STJ2-3F	746	223	94	5	1068	2001	167

Table 1: Join performance with $||D_R|| = 100K$, $||D_S|| = 20K$. Cover quotient of D_R clustering rectangles equals 0.2.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	8864	0	0	0	8864	4648	0
RTJ	2439	50	6015	1219	9695	295	372
STJ1-2N	1623	364	236	817	3040	169	349
STJ2-2N	1648	360	236	820	3064	174	355
STJ1-2F	1588	357	236	715	2896	1735	349
STJ2-2F	1606	359	236	719	2920	1739	356
STJ1-3F	1519	342	236	140	2237	3767	330
STJ2-3F	1537	353	236	120	2246	3843	344

Table 2: Join performance with $||D_R|| = 100K$, $||D_S|| = 40K$. Cover quotient of D_R clustering rectangles equals 0.2.

is because with less clustered data, the possibility that a data rectangle in D_S overlaps data rectangles in D_R is higher, and thus we must access more T_S and T_R leaf nodes during tree matching. This suggests that degree of spatial clustering should be an important factor in estimating the results of spatial joins.

Also, as the degree of clustering decreases, the number of disk accesses by ***STJ*** at tree matching time becomes close to that of ***RTJ***. This is because for low degrees of spatial clustering, most leaf tree nodes must be accessed, leaving little room for optimization. In this case, tree creation costs become the deciding factor for performance. For ***STJ***, the tree creation costs remain consistently low, and the total costs are always less than half of those of ***RTJ***. For ***BFJ***, the number of disk accesses grows rapidly and become the worst of all methods as the degree of clustering decreases, because the number of touched T_R nodes becomes much larger than the buffer size.

We have also found that the effectiveness of seed level filtering is high when the degree of spatial clustering of data is high, and diminishes as the degree of clustering decreases. Again, this is because most data objects in D_S overlap some objects in D_R, and cannot be eliminated by the filtering process.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	13650	0	0	0	13650	6984	0
RTJ	2608	27	12274	1887	16754	315	560
STJ1-2N	2422	370	366	1483	4641	263	538
STJ2-2N	2439	369	367	1477	4652	267	538
STJ1-2F	2362	358	366	1343	4429	2603	535
STJ2-2F	2429	367	366	1357	4519	2610	536
STJ1-3F	2274	349	366	451	3440	5613	498
STJ2-3F	2244	368	366	426	3404	5709	520

Table 3: Join performance with $||D_R|| = 100K$, $||D_S|| = 60K$. Cover quotient of D_R clustering rectangles equals 0.2.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	17151	0	0	0	17151	9085	0
RTJ	3292	38	16555	2525	22354	415	741
STJ1-2N	2996	361	506	2126	5989	334	685
STJ2-2N	3063	362	505	2154	6084	353	691
STJ1-2F	2956	353	507	1952	5768	3418	686
STJ2-2F	3068	363	507	1947	5885	3431	690
STJ1-3F	2739	344	505	698	4286	7328	638
STJ2-3F	2745	354	505	672	4276	7435	666

Table 4: Join performance with $||D_R|| = 100K$, $||D_S|| = 80K$. Cover quotient of D_R clustering rectangles equals 0.2.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	14803	0	0	0	14803	6628	0
RTJ	2881	57	6909	1217	11036	405	443
STJ1-2N	2265	329	236	794	3624	268	437
STJ2-2N	2347	374	236	795	3752	284	445
STJ1-2F	2242	330	236	770	3578	2688	436
STJ2-2F	2328	374	236	752	3690	2702	445
STJ1-3F	2265	337	236	430	3268	5268	411
STJ2-3F	2342	358	236	430	3366	5364	429

Table 5: Join performance with $||D_R|| = 100K$, $||D_S|| = 40K$. Cover quotient of D_R clustering rectangles equals 0.4.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct				
	rd	wr	rd	wr	total	bbox	XY
BFJ	23177	0	0	0	23177	7773	0
RTJ	3451	62	6370	1202	11057	564	534
STJ1-2N	3263	350	236	813	4662	419	514
STJ2-2N	3280	366	236	802	4684	410	524
STJ1-2F	3251	352	236	782	4621	2707	514
STJ2-2F	3268	366	236	763	4633	2701	529
STJ1-3F	3212	346	236	637	4431	5788	481
STJ2-3F	3385	354	236	583	4558	5879	509

Table 6: Join performance with $||D_R|| = 100K$, $||D_S|| = 40K$. Cover quotient of D_R clustering rectangles equals 0.6.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct		total		
	rd	wr	rd	wr		bbox	XY
BFJ	25167	0	0	0	25167	7228	0
RTJ	3304	62	6287	1195	10820	587	556
STJ1-2N	3141	358	236	814	4549	450	550
STJ2-2N	3206	366	236	820	4628	457	557
STJ1-2F	3142	358	236	790	4526	2242	550
STJ2-2F	3217	366	236	805	4624	2248	552
STJ1-3F	3268	335	236	736	4575	5104	497
STJ2-3F	3487	344	236	677	4744	5205	526

Table 7: Join performance with $||D_R|| = 100K$, $||D_S|| = 40K$. Cover quotient of D_R clustering rectangles equals 0.8.

Alg.	I/O costs					CPU costs (K tests)	
	Match		Construct		total		
	rd	wr	rd	wr		bbox	XY
BFJ	31831	0	0	0	31831	8300	0
RTJ	3710	69	5976	1207	10934	763	623
STJ1-2N	3582	338	236	800	4956	551	587
STJ2-2N	3611	340	236	808	4995	566	613
STJ1-2F	3579	333	236	793	4941	2353	588
STJ2-2F	3600	330	236	799	4965	2367	615
STJ1-3F	3689	297	236	849	5071	5772	553
STJ2-3F	4125	371	236	769	5501	5872	581

Table 8: Join performance with $||D_R|| = 100K$, $||D_S|| = 40K$. Cover quotient of D_R clustering rectangles equals 1.0.

5 Discussion

We have studied the seeded tree join method using a seeded tree and a pre-computed R-tree. However, it may be necessary to perform a spatial join using two seeded trees when the following situations occur: (1) the input data sets are the results of complicated spatial operations including other spatial joins, and no pre-computed R-trees are related closely enough to the input data sets to be used in the join, or (2) one or both input data sets are the results of non-spatial selections, and it is determined that dynamically constructing two seeded trees is more efficient for the join. This can be the case if the selectivity of the non-spatial selection is high. In the one-seeded-tree scenario, the seed levels of the seeded tree are derived from the seeding tree. For the two-seeded-tree scenario, there is no obvious choice for the seeding tree. We must still somehow determine the seed levels in both seeded trees.

Suppose the join $join_A(op_B(D_B), op_C(D_C))$ is to be processed by constructing two seeded trees. If at least one of op_B or op_C, say op_B, is a non-spatial selection, we can use the pre-computed R-tree for D_B as the seeding tree for both seeded trees. If no pre-computed R-trees can be applied, we can use a common set of seed levels for both seeded trees that is artificially constructed rather than being copied from any pre-computed R-tree. One approach may be to have a set of seed levels whose slots uniformly divide the map area. Another solution may be to extract information from the data sets using techniques such as spatial data sampling [OR93], and use such information as a basis for building the common seed levels. More research is necessary to find the most suitable way to construct seeded trees in these situations.

A closely related issue is finding quantitative measures to predict the characteristics, such as the sizes, of the outcomes of spatial operations based on the characteristics of their input data sets. Such techniques are necessary in choosing the best way to realize a spatial query. Relatively little work has been done for spatial databases in this area. Addressing these issues will be within the focus of our future work.

It is worth noting that if necessary, a seeded tree can be retained after join and used as an ordinary spatial access method for spatial selections[1]. The height of a seeded tree is no greater than the height of the R-tree constructed with the same input data plus the number of seed levels. However, most paths from the root to leaf nodes will be shorter than this upper bound.

6 Conclusions

In this paper, we have presented and studied a spatial join method that dynamically constructs index trees, called seeded trees, at join time. This method addresses the situations where existing R-trees cannot be used to help with join processing, or where no R-trees exist for the input data sets.

The seeded tree is divided into the seed levels and the grown levels. The characteristics of the input data sets are utilized to build the seed levels. Tree nodes in the seed levels are used to guide tree growth during tree construction, resulting in a tree better shaped for the join. The seed levels are also the basis of a tree construction algorithm that uses intermediate linked lists to drastically reduce construction time I/O costs. We also presented a technique called seed level filtering that can be used to further reduce the size of the tree.

We have tested the seeded tree technique against other methods with input data sets of varying sizes and degrees of spatial clustering. Our results show that the total I/O costs of the seeded tree method are always lower than the faster of the other two methods by a factor of two to three, except in one boundary case. Tree construction using intermediate linked lists is shown to be very effective in eliminating tree construction time buffer misses. When the input data set is spatially clustered, the tree matching costs of the seeded tree method are also lower due to better tree organization.

[1] If a seeded tree is to be used in spatial selections after the original spatial join terminates, seed level filtering should not be used, as the set of data indexed by the tree will be a subset of the original data.

CPU costs for the seeded tree method are always the lowest among all methods when seed level filtering is not activated.

Seed level filtering is effective in reducing tree sizes when the degree of data clustering is high, and can reduce the total I/O costs by almost 30%. However, it could incur CPU costs without gain when the degree of spatial clustering of data is low. If the characteristics of the input are not known before a join, this technique should be used only when CPU capacity is not a special concern.

References

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 237–246, May 1993.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 322–332, May 1990.

[FSR87] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 427–439, 1987.

[Gun93] Oliver Gunther. Efficient computation of spatial joins. *Proceedings of International Conference on Data Engineering*, pages 50–59, 1993.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, Aug. 1984.

[LH92] Wei Lu and Jiawei Han. Distance-associated join indices for spatial range search. In *Proceedings of International Conference on Data Engineering*, pages 284–292, 1992.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1), 1984.

[OR93] Frank Olken and Doron Rotem. Sampling from spatial databases. In *Proceedings of International Conference on Data Engineering*, pages 199–208, 1993.

[Ore89] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Portland, OR, 1989.

[Ore90] Jack Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 343–352, 1990.

[Ore91] Jack Orenstein. An algorithm for computing the overlay of k-dimensional spaces. In O. Gunther and H.-J Schek, editors, *Advances in Spatial Databases (SSD '91)*, pages 381–400, Zurich, Switzerland, August 28-30 1991. Springer-Verlag.

[Rot91] D Rotem. Spatial join indices. In *Proceedings of International Conference on Data Engineering*, pages 500–509, Kobe, Japan 1991.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[SE90] Jeffrey Star and John Estes. *Geographic Information Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of Very Large Data Bases*, pages 3–11, Brighton, England, 1987.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), 1987.