Generating Seeded Trees from Data Sets*

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. In this paper we study the problem of how to perform spatial joins between two data sets with no pre-computed spatial indices. No techniques appear to exist to date that specifically target this problem. Our solution is also useful in the context of query optimization for complex spatial queries. In addition, we demonstrate that simple sampling techniques can be effective in reducing spatial join costs.
We extend the work in [LR94, LR95] and introduce the bootstrap-seeding technique, which allows seeded trees to be constructed directly from input data sets. We can thus dynamically construct two seeded trees for two data sets and perform a spatial join between them. The task of bootstrap-seeding comprises the subtasks of determining the number and the contents of the slots, and constructing the tree. Simple sampling techniques are used to determine the slot contents efficiently.
Our experiments show that spatial joins using our methods are very comparable in performance to that of joins between the same data sets with pre-computed R-trees, and confirm the viability of our method. When joining two data sets with different sizes, our studies suggest that it would be beneficial to bootstrap an initial seeded tree for the smaller data set, and then to construct a seeded tree for the larger data set using copy-seeding and the seed level filtering technique.

1 Introduction

Spatial join operations are important and expensive operations in spatial databases. However, relatively little work has been done on them. The large sizes of the data sets involved is only one reason for the high cost of spatial joins. In addition, since the values of spatial attributes can not be totally ordered, join methods developed for traditional databases cannot be used directly in spatial joins. Existing spatial join algorithms can generally be divided into those based on tree-like indices [LR94, LR95, BKS93, Gut84, BKSS90, FSR87, SRF87] and those based on other methods such as spatial join indices [Rot91, LH92, Val87] and z-ordering [Ore89, Ore90, Ore91].

Most of these methods assume that some spatial indices have been constructed for both operand data sets of the spatial join [BKS93, Gut84, BKSS90,

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

FSR87, SRF87], or that some pre-computation has been done before a spatial join is invoked [Rot91, LH92, Val87]. If an operand data set does not have a pre-computed spatial index, the cost of constructing one at join time can be prohibitive. Such requirements place a operational burden on the spatial database system, since it may not be cost-effective to maintain index structures for all data sets regardless of their patterns and frequencies of usage. Furthermore, such requirements can be impossible to satisfy in many situations. For example, the inputs to a spatial join may be the results dynamically generated by other database operations, in which case no spatial indices would exist for such input data sets. Since most spatial indices are designed for incremental updates rather than all-at-once construction, building spatial indices dynamically at join time can be very expensive.

Our work in [LR95] first demonstrated the feasibility of dynamically constructing spatial index structures, called *seeded trees*, to perform spatial joins. In that work, one operand data set was assumed to have a pre-computed tree-like spatial index. A *seeded tree* index was then constructed for the the data set without a spatial index, using information extracted from the index existing for the other operand data set. We have also shown that the cost of this approach is much lower than that of building an R-tree index for the second data set [LR94, LR95]. There are two reasons for the low costs of the seeded tree join method. First, constructing a seeded tree dynamically is much cheaper than constructing R-tree dynamically. Second, joining a R-tree and a seeded tree constructed with information from it is cheaper than joining two independently constructed R-trees.

However, that work represents only a partial solution to the problem. The approach to seeded tree construction in [LR94, LR95] required copying and then possibly transforming the upper levels of an existing R-tree index. If neither operand data set had a pre-computed spatial index, we would be forced to construct at least one R-tree dynamically, the cost of which could be exorbitant. The relative magnitudes of the costs of constructing R trees and seeded trees are illustrated in Fig. 1.

In this paper, we study methods to construct seeded trees directly from their associated data sets, with no pre-existing index structures, while retaining construction and join cost advantages. We show how to generate seeding information directly from the data sets, and without relying on existing index structures.

1.1 Bootstrapping Seeded Trees

This approach to seeded tree construction involves three steps: determining the structure of the seed levels, sampling the input data set to extract information, and building the seed levels of the index using this information. We call the method of generating seeding information directly from the data set *bootstrap seeding* to distinguish it from that of [LR94, LR95]. That earlier method, based on copying information from an existing index, will be called *copy seeding*.

To join two spatial data sets without existing spatial indices, we first build a seeded tree by bootstrapping from one operand data set. Then we build a

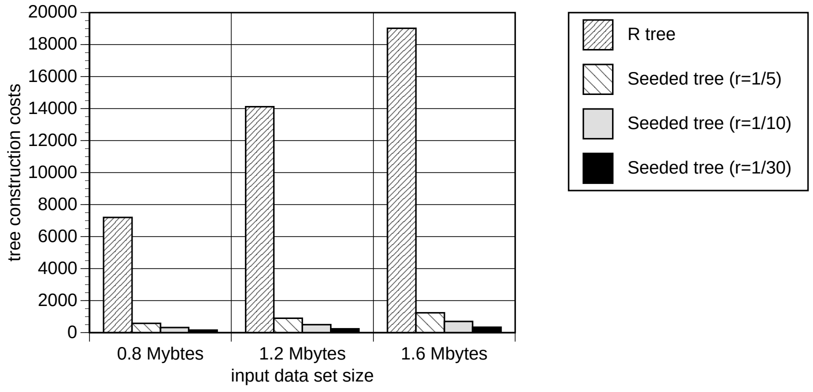

Fig. 1. Costs of constructing R trees and seeded trees. r is the ratio of the cost of accessing a disk page sequentially to that of accessing it randomly.

copy-seeded index tree for the other data set by copying the upper levels of the bootstrap-seeded tree. Thus, the bootstrap-seeded index plays the role of the pre-computed R-tree when one exists. After both seeded trees are built, they are joined to produce the final result. The tree matching algorithm is the same as that between two R-trees, or between a seeded tree and an R-tree [LR94, LR95, BKS93].

This paper makes three significant contributions by demonstrating the feasibility of bootstrap seeding. First, it solves a problem that was hitherto prohibitively expensive, namely the problem of spatial joins when no spatial indices exist. Second, it makes a significant contribution to spatial query optimization by giving the spatial query optimizer new alternatives while planning for the execution of queries. For example, consider a complex spatial join between three spatial data sets of equal size A, B and C (see Fig. 2). Assume A has a pre-computed R-tree, while B and C do not. Furthermore, assume the selectivity between B and C is much lower than that between A and B, or that between A and C. Without bootstrap seeding, we would be forced to join A–B and A–C first. However, with bootstrap seeding, we can perform the B–C join first and exploit its low selectivity by constructing seeded trees for B and C. Third, it demonstrates the viability of using sampling techniques in helping with spatial joins.

This paper is organized as follows. Section 2 first describes the structure and the basic algorithm for constructing seeded trees. Section 3 discusses how to build a seeded tree directly from an input data set. This task involves the subtasks of determining the topology of seed levels, extracting information from the data set, and constructing the seed levels using this information. The tasks are elaborated in sections 4, 5, and 6, respectively. Section 7 discusses issues in joining two seeded trees. Section 8 presents experimental results, and section 9 concludes this paper.

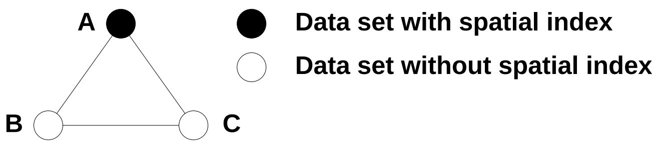

Fig. 2. Join graph for a complex spatial join. Data set A has a pre-computed R-tree, data sets B and C have not.

2 Seeded Tree Basics

This section outlines the basic algorithms of constructing seeded trees and using them in spatial joins. For details please refer to [LR94, LR95]. Structurally, a seeded tree consists of the *seed levels* and *grown levels* (see Fig. 3). 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.

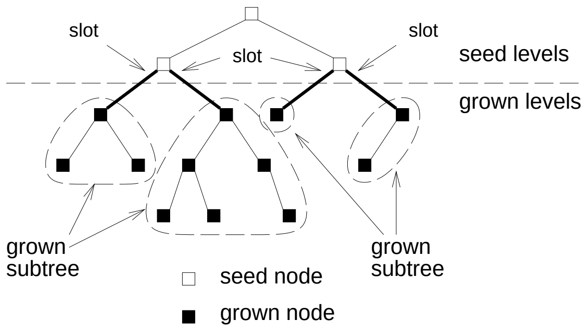

Fig. 3. Example of a seeded tree.

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).

2.1 Seeding and Growing Phases

In the seeding phase the seed levels of a seeded tree is determined. These upper levels will be used to guide the growth of the tree when the data object of the underlying data set are inserted into it in the growing phase, and will help determining the shape the tree eventually grows into. With copy seeding, this is done by extracting information from another tree-like spatial index, calling a *seeding tree* [LR94, LR95]. More precisely, we copy the first k levels from the seeding trees into the seeded tree under construction. These k may undergo some transformation before they are used in subsequent tree construction. With bootstrap seeding, this is done by extracting information from the underlying data set, the details of which will be discussed in the following sections. We call each `(mbr, cp)` pair at level $k-1$ a *slot*, and level $k-1$ of the seeded tree, the *slot level*.

During the growing phase, data objects of the underlying data set are inserted into the seeded tree, and the tree grows accordingly. 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 Fig. 3).

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, 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 *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.

2.2 Using Linked Lists in Growing Phase

To avoid random disk accesses due to buffer overflow at the growing phase, we use intermediate linked lists to assist in tree construction. 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, attached to the slot. 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 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. For each slot, an R-tree is built using the data objects stored in the linked lists grown under that slot. The slot pointer is modified to point to the root of this R-tree.

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.

3 Bootstrapping Seeding

Now we consider how a seeded tree may be built directly from an input data set. Our objective is to generate a seeded tree which can be constructed efficiently and result in efficient join with other seeded trees.

From [LR94, LR95], we know how to construct a seeded tree given its seed levels. Thus, our first task is to determine the seed levels for bootstrap seeding. That is, we must determine the topology of the seed levels and the contents of the various seed nodes. We have found that the structure and contents of the slots are most important to the success of the method. The contents of the slots decide in large measure how the lower levels of the index (the grown levels) develop. The number of slots used in the seed levels also affects tree construction costs and join performance. The key to bootstrap seeding thus lies in determining the number and contents of the slots.

Finding the optimal set of slots for bootstrap seeding requires involved analysis of the characteristics of the data sets participating in the join, and of the system resources available for join processing. Since we intend to construct seeded trees on the fly, we prefer an efficient but sub-optimal method. We divide the task of determining the slots into the two subtasks of determining their number and of determining their contents. The system resources and data set sizes determine the number of slots. The data set contents determine the contents of the slots.

Spatial data sets being generally quite large, it is infeasible to examine them in entirety in making such determinations. Instead, we sample the data sets and determine the slot contents based on the samples. Our experiments have shown that an appropriately-chosen sample preserves enough information about the data set to serve our purposes.

The task of finding a set of slots for bootstrap seeding thus consists of the following steps:

1. Determining the number of slots S.
2. Taking a sample of size k efficiently from the input data set.
3. Placing the S slots on the map area using information in the sample.

Once the set of slots is determined, we can build the seed levels by simply constructing an R-tree using this set of slots as input. In section 4, we show that the number of slots is so determined that the size of seed levels is always smaller than the available buffer size. Therefore, the cost of constructing this R-tree requires limited CPU costs and *no* I/O cost whatsoever. Once the seed levels are built, the tree growing process for both bootstrap- and copy-seeding is exactly as described in [LR94, LR95].

4 Determining the Number of Slots

For copy seeding, determining the number of slots is not an issue, since the seed levels are copied from another tree, thereby determining the number slots. However, it is clearly an issue for bootstrap seeding.

parameter	definition	parameter	definition
D	input dataset size (# blocks)	d	number of data item
B	buffer size (# blocks)	S	number of slots
l	number of seed levels	h	tree height
N_s	number of nodes in seed levels	n_i	number of nodes at level i
n_l	number of nodes at slot level	f_{max}	maximum fanout
f_{min}	minimum fanout	f_{ave}	average fanout of whole tree
f_i	average fanout at level i		

Table 1. Parameters in seeded tree construction

This section discusses the relationship between the desirable number of slots, the system buffer size, and input data set sizes. We will show that for given buffer and data set sizes, there is a preferred range of choices for the number of slots. Table 1 lists the parameters used in our discussion. Without loss of generality, we assume that a tree node, a buffer page, and a disk page are of the same size. There can be many ways of organizing spatial data. We assume that a data set contains, for each spatial object, a 16-byte minimum bounding box and a 4-byte pointer to the detailed description of the object. We also assume that the seeded tree has large node fanout, so that the size of a tree is approximately the size of its leaf level.

4.1 Algorithm Requirements

For feasibility and efficiency of the seeded tree algorithm, we require the following constraints on the structure of seeded trees:

R1 The size of the seed levels should
- **R1.1** be smaller than the buffer size.
- **R1.2** occupy a small fraction of the buffer so as to obtain good tree construction performance.

R2 The average grown subtree size should
- **R2.1** be smaller than the the available buffer space to avoid random disk access during grown subtree construction.
- **R2.2** occupy a small fraction of the available buffer for good performance during actual join.

Requirement R1 Requirement R1.1 is expressed as $N_s < B$. For large fanout, we have $n_i \ll n_{i+1}$, for $1 \leq i < h$, and $N_s = \sum_{i=1}^{l} n_i \approx n_l$. Therefore, we require $n_l < B$.

To obtain better performance, we use R1.2 and require seed level nodes to occupy only a fraction of the buffer, which translates into formula $n_l < B/E$, or

$$S < \frac{B \cdot f_l}{E}, \tag{1}$$

where $E > 1$ is a tunable constant.

Requirement R2 Requirement R2.2 states the average grown subtree size should be smaller than a fraction of the available buffer space. Since the input data set has D blocks, each grown subtree must hold D/S blocks of input data on average. A leaf node of a grown subtree is f_{ave}/f_{max} full on average, hence a grown subtree needs $(\frac{D}{S})/(\frac{f_{ave}}{f_{max}})$ leaf nodes to hold D/S blocks of input data. With large fanout, the number of leaf nodes is approximately the number of nodes of the tree, and the average size of a grown subtree is thus $(\frac{D}{S})/(\frac{f_{ave}}{f_{max}})$ nodes.

Let B' be the buffer space available after holding the seed levels in the buffer. Since the size of the seed levels is approximately n_l, $B' = B - n_l$. Requirement R2.2 is expressed as

$$\frac{D \cdot f_{max}}{S \cdot f_{ave}} < \frac{B'}{C}$$

for some tunable constant $C > 1$. Using $s = n_l \cdot f_l$ and $B' = B - n_l$, this formula can be rearranged into $n_l > \frac{CDf_{max}}{(f_{ave}f_l)} \cdot \frac{1}{B - n_l}$. Using K to denote the factor $CDf_{max}/(f_{ave}f_l)$ and solving for n_l, we get a lower bound for n_l:

$$\frac{B - \sqrt{B^2 - 4K}}{2} < n_l. \tag{2}$$

Bounds on Number of Seed Levels and Number of Slots Using (1) and (2) and $S = n_l \cdot f_l$, we can derive bounds for the number of slots as follows:

$$\frac{(B - \sqrt{B^2 - 4K})f_l}{2} < S < \frac{Bf_l}{E}. \tag{3}$$

4.2 Requirements Imposed by Intermediate Linked Lists

The previous requirements guarantee the feasibility of the algorithm and good performance during the join phase. For the intermediate linked list mechanism to provide substantial benefit during the tree construction phase, we further require

R3 The total number of slot should be
- **R3.1** smaller than the number of buffer pages to avoid buffer thrashing, and thus random disk accesses, during linked list construction.
- **R3.2** smaller than a fraction of buffer pages to obtain good performance during grown subtree construction.

There are two reasons for R3.2. When we write a batch of linked lists to disk, the pages at the end of the linked lists are, on average, half-full. When the number of slots is large and the linked lists are short on average, the fragmentation problem becomes serious. For example, if the number of slots is equals the number of buffer pages, the average linked list length will be be 1 when a batch write occurs. In this case, 50% of of the information moved by I/O effort is useless. There is also a second and more serious problem. The total number of random reads incurred when reading linked lists back from disk to build grown subtrees is proportional to the number of slots. Let N_{ba} be the average number of batches a slot participated in the batch writes[2] The number of random reads required to read the linked lists back to memory is $S \cdot N_{ba}$. Therefore, it is beneficial to make the number of slots as small as possible.

Requirement 3 can be formally expressed as $S < \frac{B'}{Q}$, where $Q > 1$ is a tunable parameter. The above formula can be solve for S:

$$S < \frac{B}{Q + \frac{1}{f_l}}.$$

[2] When a linked list is shorter than a tunable threshold, it does not participate in the batch write.

Since $Qf_l + 1 > E$, and hence $\frac{B}{Q+\frac{1}{f_l}} < \frac{B \cdot f_l}{E}$, for reasonable choices for parameters, when considering all three requirements, the bounds on n_l becomes

$$\frac{(B - \sqrt{B^2 - 4K}) \cdot f_l}{2} < S < \frac{B}{Q + \frac{1}{f_l}}. \tag{4}$$

Note that the upper bound depends on buffer size and average leaf node fanout only. The lower bound is a monotonically increasing function of the data size set. When we plan to construct two seeded trees with the same set of slots, we must ensure that the number of slots satisfies the bounds imposed by both data sets simultaneously. This means we need only to apply inequality 4 to the larger data set size to derive the bounds for the number of slots.

5 Taking Samples

After determining k, the number of slots, we must sample from the input data set. The number of samples n would be a function of k. If the seeded tree is being constructed for data that is the output of some other operations in the same query, we can sample as the output is produced. Sampling would incur no additional I/O costs.

In many cases, we expect the data set to reside in a data file. If the data in the data file are known to be randomly distributed, that is, the placement of data in the file has no relation to the closeness in space of the objects they represent, the sampling task would be easy. We can simply read the first n items sequentially from disk and use them as sample. The sampling cost is very small in this case. In practice, we cannot rely on the data set having this property. Often spatial closeness of data does effect in the closeness of data in the file. In these cases, we have to sample randomly from the file.

If the number of samples is small in comparison to the size of the file, we can read the necessary disk blocks through random access. If the number of samples is large relative to the size of the file, it would be cheaper to sequentially access all disk blocks and perform the sampling. Algorithms such as those in [Vit85, Vit84, Ahr85] can be used to sample this sequential data stream.

The expected number of disk blocked accessed as a function of the size of the random sample has been presented in [Yao77]. Let m be the total number of disk blocks in the data file, n be the total number of data items and k be the size of the random sample. The expected number of accessed disk block X_D is

$$X_D = m \cdot \left[\prod_{i=1}^{k} \frac{nd - i + 1}{n - i + 1} \right] \quad \text{where } d = 1 - 1/m. \tag{5}$$

A simpler expression for a tight lower bound of this formula is [Yao77, Car75]

$$X_D = m(1 - (1 - 1/m)^k). \tag{6}$$

Let ρ be the ratio of the cost of a sequential disk block access to that of a random block access. If $X_D > m \cdot \rho$, it will be cheaper to perform random sampling through sequential reads. Approximating X_D with 6, this condition can be expressed as

$$k > \frac{\ln(1-\rho)}{\ln(1-1/m)}. \tag{7}$$

Whether it is cheaper to sample by sequentially accessing the data file depends on the ratio ρ and the number of blocks in the file. As an example, assume there are $100,000$ data items in a data file, each occupying 20 bytes. The file will have 2000 blocks. If the ratio ρ is $1/5$, it would be cheaper to sample by sequentially reading all blocks for sample sizes over 447 items. If the ratio is $1/30$, the threshold becomes 68 items.

There are other ways to reduce sampling costs. For instance, when a sample of size k is needed, we can first read in a some small number b of blocks from the data file and then obtain our k data samples from these b blocks. However, the feasibility of this approach depends on how the data are clustered in the file. As our purpose is to demonstrate the viability of building seeded trees from data sets, and the sampling cost does not dominate the overall cost of join, we do not pursue this approach further.

6 Slot Placement

In this section, we discuss how to generate a set of S slots from a sample set of size k. We have the following guidelines for choosing slots for a bootstrap-seeded tree.

- Place slots at or near the centers of clusters of data. This strategy inserts data objects that are close to each other into the same grown sub-tree, reduces dead area in bounding boxes of sub-trees, and improves performance.
- Avoid letting one slot cover too many data objects. Otherwise, we may have a grown sub-tree that overflows the buffer. Thus, we might want to place more than one slot in large clusters.
- Use point slots, i.e., slots whose bounding box value is a point. In [LR94], we have observed that setting the contents of slots to their center points at the beginning of the growing phase results in seeded trees that are more efficient during tree matching. Using point slots also simplifies the task of determining slots for bootstrap-seeded trees, as we need to worry about setting fewer parameters.

The study of cluster analysis in statistics [Eve93, KR90] addresses problem similar to ours. However, techniques developed there do not apply directly to our context for several reasons. First, most cluster analysis is assumed to be performed off-line. Therefore, expensive, multiple-pass algorithms can be used to analyze the clustering structure of data sets. For bootstrap seeding, we cannot afford expensive off-line computation. Second, our purpose is not to determine the precise clustering structure of data sets, but rather, to choose slots that support efficient spatial operations. Therefore, we are interested in identifying fast but formally less rigorous methods.

6.1 Direct Sample Placement

The simplest method is to directly use the data objects in sample set as the slots. In the method, the number of sampled objects is set to be the number of slots, that is, $k = S$. After taking a size S sample from the data file, we calculate the center point of the sample data objects, which are then used as slots. The method is simple and straightforward, and is used as basis of comparison in our study.

6.2 Nearest Group Center Placement

As noted above, it would be advantageous to place the slots inside data clusters. Moreover, it is useful to place more slots inside clusters that have more data objects. We adopt the following somewhat stricter criterion:

> Given a spatial data set and the number of slots S, place the S slots in the map area so that the sum of distances from the center of each data object to its nearest slot is minimized.

In this section, we discuss a heuristic to approach this goal. This method forms S groups of data, each representing a slot. Data objects from the sample set are placed one by one into the group whose center of gravity is nearest to the center of the object. The slot contents are set to the centers of gravity of the groups at the end of this processing.

The method is described in detail as follows. For each slot, we form a *data group*. Each group has a center of gravity and a weight. That is, each group is represented by triple (u, v, w), where (u, v) is a vector describing the position of the center of gravity, and w is the weight of the group.

At first, we randomly choose S data objects from the sample set, each to initialize one data group. For each such object with center at (x, y), its associated group is initialized to $(x, y, 1)$.

For each remaining data object with center at (u, v), we then identify the group (x, y, w) whose center of gravity is nearest to (u, v). The object is added into this group and the group triple is updated to

$$\left(\frac{wx + u}{w + 1}, \frac{wy + v}{w + 1}, w + 1\right) \tag{8}$$

When all objects in the sample set are processed, the centers of gravity can be output as slots.

This method is sensitive to how we initialize the group centers. To avoid instability, we can go through the sample data set more than once. At the end of each pass, we reset the group weights to smaller values, and re-insert all sample objects into their nearest group.

The parameters of this method are the number of passes and the value to which we reset the group weights at the beginning of a new pass. The ratio of the sample set size and slot number is another parameter of this method. This method requires the sample size k to be several times greater than the slot number S. Again, we are interested in an efficient but suboptimal method, not in fine-tuning these parameters. We set the number of passes to two and reset the weights to one third its original value at the beginning of the second pass. The ratio of sample set size and slot number is set to 20. Note that when the ratio and the number of passes are one, the method degenerates into the direct sample placement method.

7 Joining Two Seeded Trees

Consider a spatial join between two data set D_R and D_S, neither of which has any spatial index. Suppose we choose data set D_R and bootstrap a seeded tree T_R directly from it. Next we need to build a seeded tree T_S for D_S. There are three ways to obtain the seed levels for T_S:

1. Copy the seed levels of T_R as when it is built.
2. Use the same seed levels originally constructed for T_R.
3. Construct the seed levels of T_S by bootstrapping from D_S.

The first two alternatives fall into the category of copying seeding. The third is bootstrap seeding. Our experiments show the first alternative outperforms the second alternative slightly. The third alternative incurs additional sampling and slot placement costs in tree construction without providing better performance in tree matching. In the following discussion, all seeded tree joins use the first alternative unless indicated otherwise.

After both seeded trees are built, we join these trees using the same tree matching algorithm used to join two R-trees, or one R-tree and one seeded tree [BKS93, LR94, LR95]. Since the tree matching algorithm does not restrict the types of the participating trees, it proceed exactly as before.

7.1 Seed-Level Filtering

Seed level filtering [LR94] is a technique that we can use to enhance join performance when one participant is a seeded tree. It works as follows. Say we are building a seeded tree by copy-seeding from an existing R-tree. When data objects are inserted into the new seeded tree, we can check to see if the object overlaps at least one bounding box in the nodes at the slot level. If not, the object will not overlap any object in the R-tree and need not be inserted at all. Using this technique in the construction of seeded trees results in smaller seeded trees, saving costs in subsequent processing. Seed level filtering is useful if the data set used to construct the seeded tree is large and only a portion of the data objects in the data set overlap data objects in the R-tree.

This technique is particularly useful with joins between two seeded trees. In this context, we can first bootstrap a seeded tree from the smaller of the two data sets, and then construct a seeded tree for the larger data set by copy-seeding. Seed level filtering can be applied to the larger data set in the during construction of the second tree.

One argument against bootstrapping from the smaller data set could be that the common seed levels are built using less information, and may hence result in inferior tree join performance. However, our experiments show that even without seed level filtering, the penalty for using the smaller data set for bootstrapping (and thus to determine the common seed levels) is marginal. Using seed level filtering, bootstrapping from the smaller data set allows us to filter objects from the larger data set, potentially screening more data objects from participating in the subsequent processing. The benefits of filtering objects from the larger data set far outweigh any penalties for bootstrapping with the more limited information in the smaller data set.

8 Experiments

Assume we need to perform a spatial join between data sets D_R and D_S, both without pre-computed spatial indices. We will call the seeded tree constructed for D_R, T_R, and that constructed for D_S, T_S.

For simplicity, we assume that the disk page size and the memory page size are both 1K bytes, as are the node sizes for both seeded trees. 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.

We studied data of different degrees of spatial clustering. The degree of clustering was controlled by a simple scheme, similar to the one in [LR94]. 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 total 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.

The degree of clustering of the data sets used in our experiments were categorized as high, medium, or low, meaning that the side lengths of the clustering rectangles were no larger than 0.03, 0.04, and 0.09, respectively. We used four join dataset pairs A, B, C, and D, respectively. A, B, C contained data sets of cardinalities 100k and 40k, respectively. The data sets in D had cardinalities 100k and 80k. The degree of clustering in A, B, and C was low, medium, and high, respectively. The degree of clustering in D was medium.

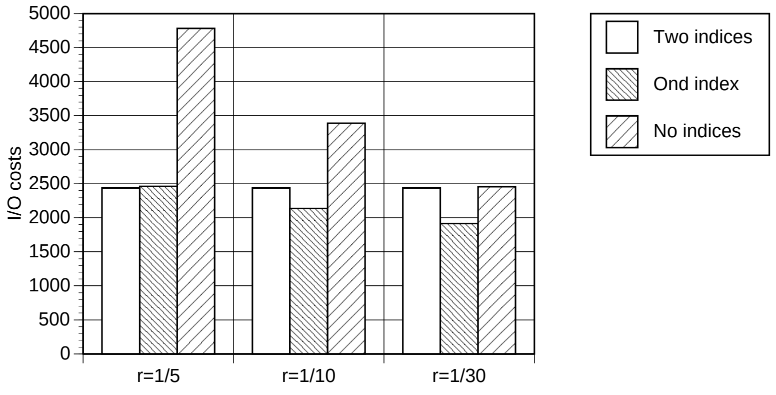

Fig. 4. Costs of spatial joins under various conditions.

A preview of the performance of our method may be appropriate at this point. Although we assume no pre-existing indices, and are addressing an inherently more difficult problem, the performance of our method compares very favorably with the easier cases where indices pre-exist. Figure 4 illustrates the total join costs of for three scenarios, with different numbers of pre-existing indices. Data set pair B was used as input data set. Column "Two indices" is the cost of joining two pre-existing R-trees. No construction costs are incurred in this case. Column "Once index" is the combined cost of constructing a seeded tree for the smaller data set and that of joining the seeded tree with a pre-existing R-tree for the larger data set. Column "No indices" is cost of constructing two seeded trees plus that of joining them[3]. The costs are shown under different assumptions for ρ, the ratio of the costs of sequential and random I/O. In the first group of columns, labeled "$r = 1/5$", the value of $\rho = 1/5$. In the second and the third groups, $\rho = 1/10$, and $\rho = 1/30$, respectively. As Fig. 4 shows, even in the case in which there is not any pre-computed spatial indices, the costs of dynamically constructing two seeded trees and then joining them is quite comparable to the cost of joining two R-trees pre-computed from the same input data sets.

Figure 5 shows the break up of the total cost of spatial joins on data set pair B, with the number of slots set to 100. The columns labeled "input" show the cost

[3] The number of slots used is 100.

of Initially reading in the input data sets during tree construction. The columns labeled "sampling bound" are the maximum possible costs of sampling from the larger of the two data sets. This value is the upper bound on sampling cost in all cases. The columns labeled "processing" shows all other costs, including the costs incurred during the growing phases of the two seeded trees and those during tree matching. As seen in the figure, the upper bound on the sampling cost is very small under all assumptions, which makes it less rewarding to optimize on the sampling costs. The input costs are fixed and are also small portions of the total costs. Our experiments on other data set showed similar cost break-downs. The numbers lists in the following discussion are the processing costs only only. Also, different ρ ratios in general demonstrated the same trends in various experiments. For clarity of presentation, we will show only the performance numbers obtained under the assumption of $\rho = 1/30$.

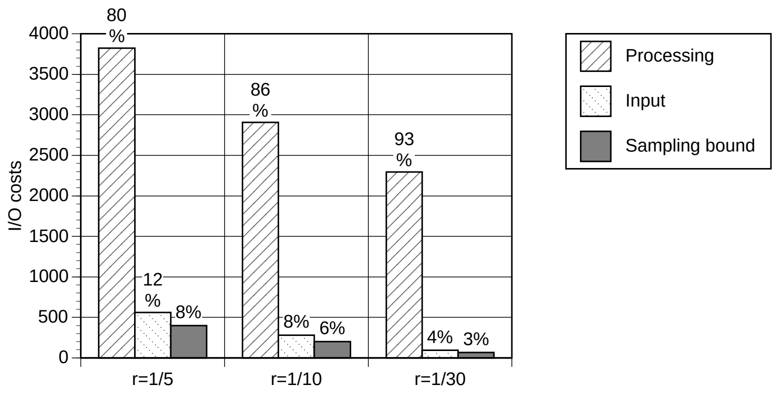

Fig. 5. Break-ups of seeded tree join costs.

8.1 Choice of Number of Slots

We studied the effect of using different numbers of slots on the performance of seeded-tree joins. We were particularly interested in studying whether the lower and upper bounds derived in Sect. 4 were useful guides to choosing slot numbers. The group-center sampling method was used to determine the values of slots after their number was decided.

We chose the values of parameters C and Q to be 3 in inequality 4, which means that the average grown subtree size and the number of slots will both be about 1/3 of the number of pages in the buffer. For the buffer and data set sizes used, this formula yields 16 and 165 as the lower and upper bounds respectively, for the number of slots.

Figure 6 shows the effects of slot numbers, using data sets A, B and D. Two observations are of interest. First, there is in general a range of slot numbers

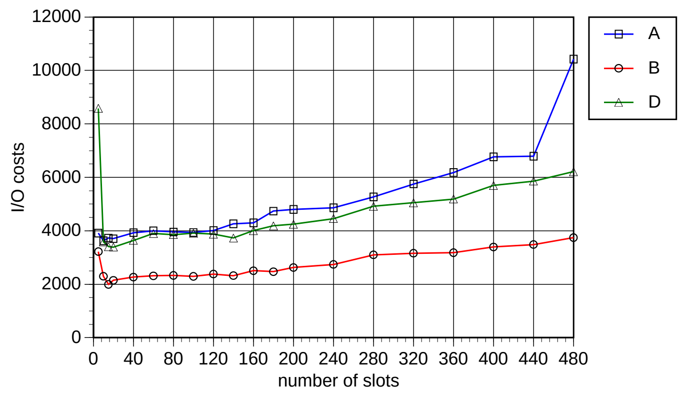

Fig. 6. Effect of slot number on seeded tree join performance.

yielding low cost, instead just a single point of lowest cost. This means when performing a spatial join, we do have a range of slot numbers to choose from. This flexibility can be useful when optimizing the other parameters. Second, the ranges defined by the lower and upper bounds of inequality 4 generally do coincide with the actual ranges of good performance, but are slightly stricter. This means it is safe to use formula 4 to guide the choice of number of slots.

8.2 Slot Placement Methods

Figure 7 shows the results of joins using seeded trees constructed with two different slot placement methods: direct placement and the group-center methods. The direct placement method samples the same number of data objects as the number of slots. The number of data objects sampled by the group center methods is a parameter of the method. Here we set that number to be 20 times the number of slots. The group center method used here makes two passes, resetting the group weights at the start of the second pass to 1/3 of their values at the end of the first pass.

As Fig. 7 shows, the performance of the direct placement method is surprisingly close to that of the group center method. This demonstrates the adaptability of the seeded tree construction algorithm and robustness of the performance of seeded trees. Even if the slots are not placed in the best positions, the performance of the seeded tree join shows very little degradation. The closeness of the two methods precludes the need for testing group center method with smaller sample size-slot number ratios.

8.3 Seed Level Filtering and Data Set Sizes

As discussed earlier, given two data sets with no indices, we can bootstrap the initial seeded tree for either the larger or the smaller data set, and use that index to copy-seed the second. Figure 8 shows the performance differences between

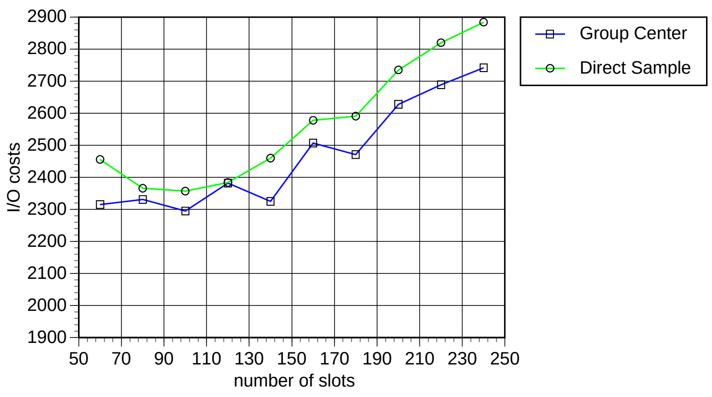

Fig. 7. Performance of boot-strap seeded trees constructed with different slot placement methods.

such choices. Consider first the lines labeled “big” and “small”, which show join performance using the larger and smaller data set as the basis for bootstrapping, respectively. The data sets used here were of medium clustering and the larger of the two data sets was always of cardinality 100,000. The X-axis is the ratio of the size of the smaller data set to that of the larger data set. The performance of the two alternatives is quite close, with the “big” variation being slightly better.

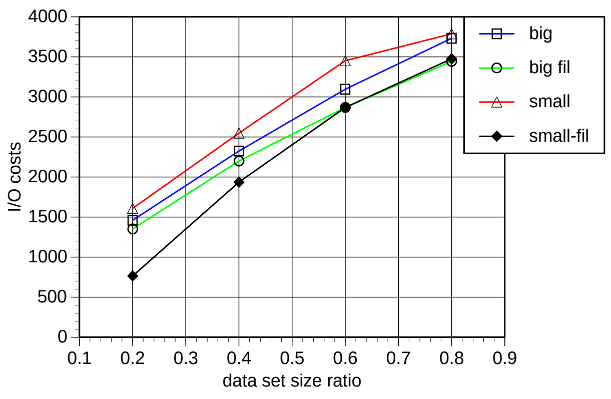

Fig. 8. Effect of seed level filtering.

When we turned on the seed level filtering technique [LR94], the situation reversed. When the larger data set was used for bootstrapping and filtering was applied the smaller data set (“big-fil”), the improvements were marginal. On the other hand, when the roles were reversed and the larger data set was filtered, many more data objects became exempt from insertion into the tree. Thus the size of the copy-seeded tree was reduced significantly, resulting in significant cost savings. The improvement is particularly significant when the size difference between the two data sets is high. In the case where the size ratio was 0.2, the difference between "big" and "sml-fil" was close to 50%. When the size of the two data sets was close (e.g. when ratio = 0.8), the performance of the two filtering options was very close. Generally, filtered variations outperformed the non-filtered variations of the method.

This result suggests that when joining two data sets of different sizes, it is beneficial to bootstrap from the smaller data set, then construct a copy-seeded tree for the larger data set, with seed-level filtering turned on.

9 Conclusions

In this paper we studied the problem of how to perform spatial joins between two data sets with no pre-computed spatial indices. To the best of our knowledge, no solution has hitherto existed for this problem in the context of spatial joins. Solving this problem also contributes to query optimization for complex spatial queries. In addition, we have demonstrated the feasibility of using sampling techniques to advantage in spatial joins.

We have extended the work in [LR94, LR95] and introduced the bootstrap-seeding technique, which allows seeded trees to be constructed directly from underlying data sets. The task of bootstrap-seeding is divided into that of determining the number of slots, assigning values to slots, and constructing the index tree. We have derived upper and lower bounds on the range of the acceptable number of slots for a seeded tree. We have also developed a technique for determining the contents of slots. This technique first samples the underlying data set to extract information, then derives slot contents from this information using simple, but effective data clustering heuristics.

Our experiments have confirmed that the upper and lower bounds we have derived do accurately describe the range of acceptable slot numbers. They also indicate that the slot contents need not be optimally determined for good join performance, and further, that complicated data clustering algorithms are unnecessary. When joining two data sets with different sizes, our results suggest that it would be beneficial to bootstrap an initial seeded tree for the smaller data set, and then to construct a copy-seeded tree with the larger data set using the seed level filtering technique.

References

[Ahr85] J. H. Ahrens. Sequential random sampling. *ACM Transactions on Mathematical Software*, 11(2):157–169, June 1985.

[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.

[Car75] A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of ACM*, 18(5):253–263, May 1975.

[Eve93] Brian Everitt. *Cluster Analysis*. Edward Arnold, London, third edition edition, 1993.

[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.

[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.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data, An Introduction to Cluster Analysis*. John Wiley & Sons, Inc., New York, 1990.

[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.

[LR94] Ming-Ling Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 209–220, Minneapolis, MN, May 1994.

[LR95] Ming-Ling Lo and C. V. Ravishankar. Seeded trees for spatial joins: Structure and implementation. Technical report, Department of EECS, University of Michigan, Ann Arbor, Michigan, 1995.

[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.

[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.

[Vit84] Jeffery Scott Vitter. Faster methods for random sampling. *Communications of the ACM*, 27(7):703–718, July 1984.

[Vit85] J. S. Vitter. Random sampling with reservoir. *ACM Transactions on Mathematical Software*, 11:37–57, March 1985.

[Yao77] S. B. Yao. Approximating block access in database organizations. *Comm. of ACM*, 20:260–261, Apr. 1977.