The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins

Ming-Ling Lo and Chinya V. Ravishankar, *Member*, *IEEE*

Abstract—Existing methods for spatial joins require pre-existing spatial indices or other precomputation, but such approaches are inefficient and limited in generality. Operand data sets of spatial joins may not all have precomputed indices, particularly when they are dynamically generated by other selection or join operations. Also, existing spatial indices are mostly designed for spatial selections, and are not always efficient for joins. This paper explores the design and implementation of *seeded trees* [1], which are effective for spatial joins and efficient to construct at join time. Seeded trees are R-tree-like structures, but divided into *seed levels* and *grown levels*. This structure facilitates using information regarding the join to accelerate the join process, and allows efficient buffer management. In addition to the basic structure and behavior of seeded trees, we present techniques for efficient seeded tree construction, a new buffer management strategy to lower I/O costs, and theoretical analysis for choosing algorithmic parameters. We also present methods for reducing space requirements and improving the stability of seeded tree performance with no additional I/O costs. Our performance studies show that the seeded tree method outperforms other tree-based methods by far both in terms of the number disk pages accessed and weighted I/O costs. Further, its performance gain is stable across different input data, and its incurred CPU penalties are also lower.

Index Terms—Spatial databases, query processing, join processing, database index, spatial index, buffer management.

1 Introduction

Spatial databases and GIS systems have received increasing attention in recent years. Most research on query processing in such systems has focused on spatial selection, or the 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 indices have been designed to facilitate such operations [2].

Relatively little work has been done on the spatial join. Spatial joins are expensive but indispensable in such applications as map overlay. Existing join algorithms can generally be divided into those based on tree-like indices and those which are not. The latter group includes methods based on join indices, on z-ordering or on separational representations. The spatial join index [3] method builds a spatial version of a join index [4] for two data sets if these data sets are frequently joined spatially. This method trades the overhead of precomputation at index building time for accelerated processing at join invocation time. It assumes the grid-file [5] as the underlying spatial access method, and requires grid-files to exist for relevant data sets before spatial join indices can be built for them. A similar method using distance-associated join indices [6] has also been proposed to speed up spatial range queries. Orenstein [7], [8], [9] 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. Güting and Schilling [10] proposed a divide-and-conquer algorithm for finding intersecting pairs of rectangles given a set of rectangles. The method transforms rectangles into *separational representation*, which represents rectangles by their left and right edges. The edges are sorted and a divide-and-conquer algorithm is applied. This method does not require indices to be built, but it does require external sorting for large data sets, which may involve a lot of random I/O operations. Unfortunately, [10] provides only asymptotic analysis of the costs, and explictly omits the costs of external sorting.

1.1 Tree-Based Join Algorithms

Several spatial join algorithms have used tree-like indices. Gunther [11] analyzed the applicability of relational join techniques to spatial joins, and proposed a general join algorithm using *generalization trees*, which are abstractions of tree-like spatial indices. This algorithm can be used to join any two data sets with precomputed tree-like indices. Since breadth-first tree traversal is used in this method, the matching pairs of 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 [12], [13], [14], [15] have been gaining popularity due to their relatively simple

• *M.-L. Lo is with the IBM Thomas J. Watson Research Center, 30 Saw Mill River Rd., Hawthorne, NY 10532. E-mail: mingling@watson.ibm.com.*
• *C.V. Ravishankar is with the Electrical Engineering and Computer Science Department, University of Michigan–Ann Arbor, 1301 Beal Ave., Ann Arbor, MI 48109. E-mail: ravi@eecs.umich.edu.*

Manuscript received 1 Dec. 1994; revised 8 Jan. 1996.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104430.

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. An internal 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. [16] proposed a join algorithm based on the R-tree or variations. This join method requires each participating data set to have a precomputed R-tree index. The join algorithm consists of an 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 the matched children, resulting in a depth-first tree traversal. 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.

1.2 The Need for Dynamically Constructed Index Structures

All these methods require index structures to have been constructed for the input data sets and/or some precomputation or external sorting to have been done before the 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. Second, the operand data sets of a spatial join may be fresh output from other spatial or nonspatial operations, and therefore not have spatial indices already constructed for them.

For spatial queries that allow multiple execution paths, it may be beneficial to execute other operations before spatial joins. For example, suppose we have two data sets covering the same area, one containing buildings, the other containing parcels of land. Consider the following query:

Query: Find all buildings built on government-owned land.

If the total number of buildings is large, but only a small fraction of them are built on government-owned land, it may be more efficient to perform a nonspatial selection first to identify government-owned land before performing the spatial join. This approach introduces an intermediate data set without a spatial index.

It is also common for a spatial query to involve multiple spatial joins. If an input to a spatial join is the output of another spatial join, such input might only be remotely related to the original data set. In such cases, using precomputed indices from the original data set could be very inefficient or even infeasible. Other spatial operations, such as *reclassification*, *aggregation*, and *buffering* [17], may also occur together with spatial joins, 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 nonspatial 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 [12], [13], [14], [15] were originally designed for a different context. In particular, such indices are assumed to be built up incrementally, and are not optimized for all-at-once construction. Most such indices are also designed to minimize the cost of spatial selection, not that of spatial join. They try to reduce the average number of disk accesses per spatial selection and the worst-case selection costs, and to minimize disk space consumption. Such characteristics are not always the most crucial 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.

1.2.1 The Seeded-Tree Approach

In this paper, we address this problem with a new spatial join method using index structures called *seeded trees*, first introduced in [1]. We assume a system in which the R-tree is the main type of spatial index. However, our method does not require R-tree-like indices to pre-exist for both participating data sets, thus handling situations where using precomputed indices is impractical. The seeded tree construction algorithm uses information about the join and the input data sets, and constructs the seeded tree dynamically. Since the seeded tree is constructed at join time, low tree construction costs are crucial to its performance. We present a tree construction method using intermediate linked lists that performs I/O mostly in sequential accesses. This method avoids most buffer thrashing and constructs seeded trees at very low cost. Upon encountering a join operation, our join algorithm first constructs a seeded tree for one of the participating data sets, and then proceeds to use some standard tree matching algorithm to compute join results. Our method works with any tree matching algorithm, but we will use algorithm TM proposed in [16] as the tree matching component of our method, given its simplicity and reasonable performance.

In addition to discussing the basic structure and behavior of seeded trees [1], we present a new buffer management strategy for tree construction. This method does not follow the conventional rationale of delaying writing buffer pages as long as possible, but reflects buffer contents to disk in large chunks well before space reclamation is required. This method lowers I/O costs by an additional 40 percent. We also discuss a space utilization problem intrinsic to the seeded tree, namely, that caused by the grown subtree roots, and provide a simple but efficient solution. Our solution exploits the fact that the grown subtrees are constructed in sequence, and packs each subtree root immediately after its grown subtree is built. This packing algorithm contributes to the stability of the seeded tree method, while incurring no additional I/O overhead and requiring no separate packing pass. We also examine the theoretical aspects of the seeded tree method, and provide guidelines for choosing algorithmic parameters.

We perform experiments on the seeded tree method with input data of various characteristics, including data sets designed to stress the seeded tree method, and with real-life data. Our experiments show that the seeded tree method uniformly outperforms existing tree-based methods by many times, both in terms of the number disk pages accessed and the translated I/O costs. Further, the performance gains of the seeded tree method over other methods are very stable across different input data characteristics.

This paper is organized as follows. Section 2 describes the structure and basic behavior of the seeded trees. Section 3 discusses various issues in seeded tree construction and presents tree construction techniques. Section 4 analyzes theoretical aspects of our method. Performance studies on the seeded tree method with various input data sets are reported in Section 5. Section 6 discusses related issues, and Section 7 concludes this paper.

2 Seeded Trees Basics

Let us assume that we want to join a derived data set D_S, for which no precomputed 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 use 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 also on the order in which data objects were inserted into it [12], [13]. 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 reinserting a fraction of the data objects in an R-tree improves its performance [13]. 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 Fig. 1a. Say we have an R-tree T_R and 14 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. Fig. 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 nonminimal but are organized as in Fig. 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 in Fig. 1c, both I/O and CPU cost can be reduced during the join process.

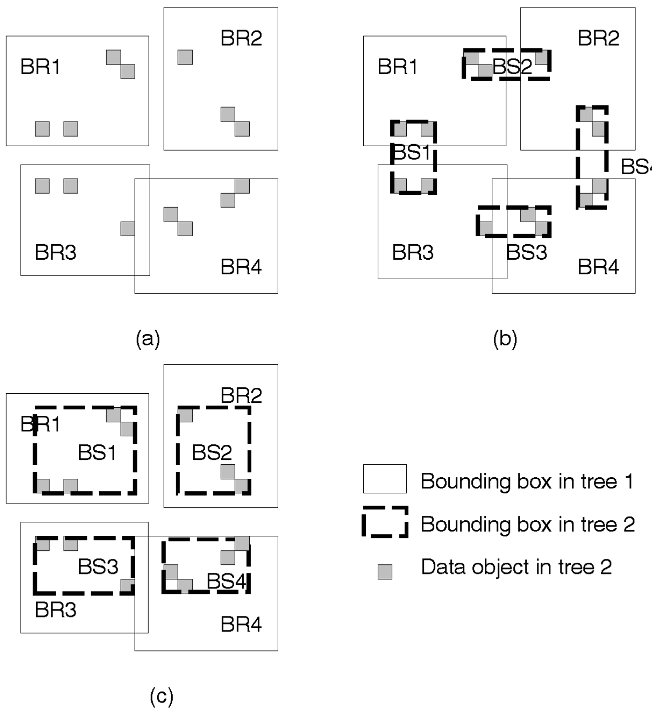


Fig. 1. Beneficial and nonbeneficial formation of bounding boxes in a seeded tree.

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 (see Figs. 2 and 3). Structurally, a seeded tree consists of the *seed levels* and *grown levels* (see Fig. 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.

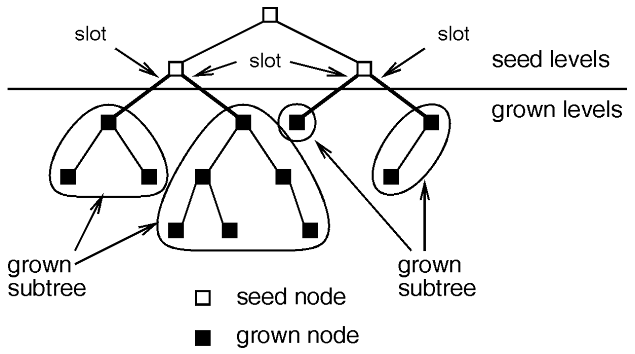


Fig. 2. Example of a seeded tree.

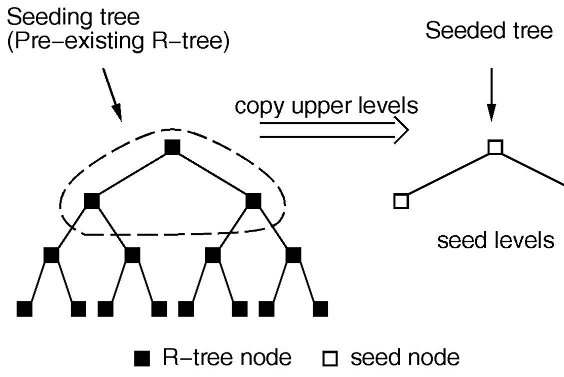


Fig. 3. Seeding phase.

The grown levels span from the children of the last seed level to the leaf level. As with R-tree nodes, a nonleaf 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.

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 (see Fig. 3). 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 is never allowed to change. *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 Fig. 4, 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 [12], object S_1 would be inserted into B_1 instead of B_2, which could result in unnecessary disk accesses during the join process.

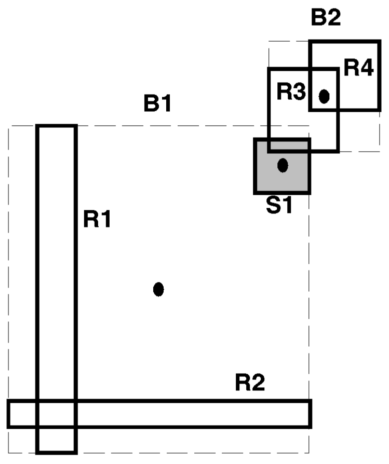


Fig. 4. Efficient and inefficient bounding boxes.

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 outperform strategy C_1.

2.2 Growing Phase

During the growing phase, data objects in D_S are inserted into the seeded tree. 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. 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, 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 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 [16] 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 prerequisite.

3 Tree Construction Issues

A seeded tree is constructed dynamically at join time, so low tree-construction costs are crucial to its performance. In this section, we discuss the construction of seeded tree in detail. First we present a tree construction technique that lowers the tree construction costs by using linked lists in an intermediate step. Then we discuss a space underutilization problem caused by grown subtree roots, and show how to overcome it by packing the roots immediately after a grown subtree is built. We also present a new buffer management strategy for tree construction that does not follow the convention rationale of delaying disk writes as long as possible, but reflects dirty buffer pages to disk in large chunks well before space reclamation. This new strategy results I/O costs lower by as much as 40 percent over that implemented in [1]

3.1 Tree Construction Using Linked Lists

Traditional tree-like indices have been optimized for incremental updates rather than for all-at-once construction. As a result, their total construction costs can be high.

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 with an 800 Kbyte data set (40,000 data objects), using a 512 Kbyte buffer with 1 Kbyte page size, can result in 7,206 disk accesses during construction (see Table 5, second row in Section 5.1). This number is nine times the number of disk pages read to access 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. 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 (see Fig. 5). 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 predefined constant to disk, 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 (see Fig. 6).

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 accesses 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 from a few tens to hundreds, assuming a fan-out of at least 50. This means that the algorithm could work for seeded trees of size at least 10 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.

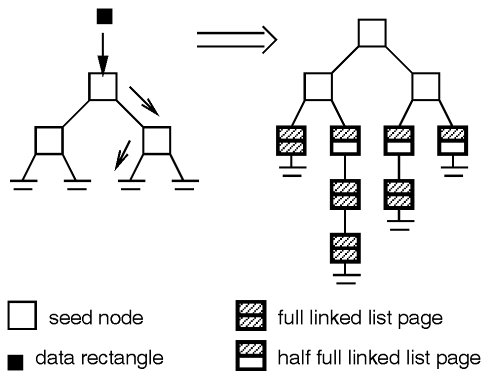


Fig. 5. Organizing inserted objects into linked lists.

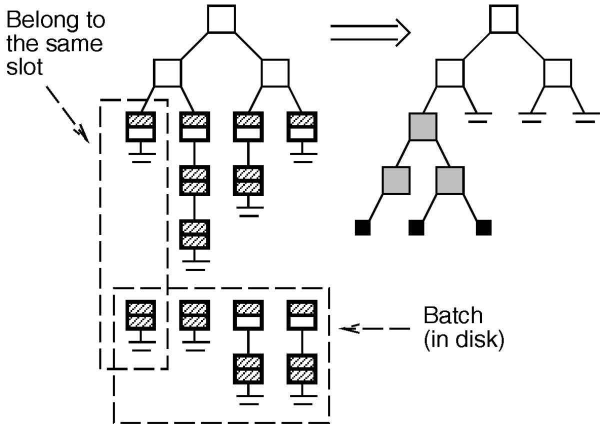


Fig. 6. Converting linked lists into grown subtrees.

3.2 Packing Grown Subtree Roots

Let f denote the fanout of a tree node, and f_{min} and f_{max} denote the lower and upper bounds on f. All R tree nodes have between f_{min} and f_{max} children, except for the root, which has between two and f_{max} children. A root node thus has a smaller number of children than a nonroot node on average, and utilizes space less efficiently. A seeded tree may be larger than an R tree constructed using the same data set, since every grown subtree is a small R-tree and introduces a root node. The size difference can become significant when the number of slots, and hence the number of grown subtrees, in the seeded tree is large. The larger size of the seeded tree not only consumes more disk space, but more seriously, also incurs more random disk I/O to page in the tree during the matching phase. If no measures are taken, this phenomenon will either limit the range for the feasible number of slots, and/or compromise the performance of the seeded tree method.

We avoid this problem by packing multiple grown subtree roots into one physical node. Our objective here is not to optimize space utilization, but to find a simple yet effective solution to the problem. Because subtrees are built in sequence, we can readily pack subtree root as each is constructed. No separate packing phase is necessary.

For convenience, we assume that children of a tree node are numbered from 1 to f, and are always accessed in ascending order. Conceptually, each grown subtree has its own root node. Physically, we pack several grown subtree roots into one physical node, and place markers between fragments of the node that belong to different subtree roots. A marker is simply an ordinary pointer in a physical node set to some reserved value.

When a grown subtree is first built, its root occupies a full grown node. We invoke the algorithm in Fig. 7 to pack the root immediately after each subtree is built.

With this packing method, slots in the same leaf seed node may have the same pointer values, but slots in different leaf seed nodes will always have different pointer values. Assume a packed grown subtree root is attached to a leaf seed node L through a slot S. To access the contents of the subtree root, we scan L to find out the number of slots before S that have the same pointer value as s does. If there are j such slots, the subtree root is packed in the $j + 1$ fragment of L. Since the nodes packed by this algorithm often have higher space utilization than average nodes, our experiments show that the seeded trees built with the packing algorithm almost never exceed the size of an R-tree built using the same data set, and provide stable performance regardless of the number of slots used.

3.3 Buffer Management During Seeded Tree Construction

The seeded tree construction algorithm using intermediate linked lists is outlined in Fig. 8. For clarity of discussion, we use the term *write-purge* to mean the action of writing the contents of a buffer page to disk, and freeing the buffer page by marking it unused. We use the term *write-reflect* to mean the action of writing the contents of a buffer page to disk and marking it clean. A buffer page after write-reflect remains a used page.

The tree-construction time buffer management strategy implemented in [1] is outlined in Fig. 9. Steps A3 and A4 follow the conventional rationale of delaying disk writes as much as possible. However, a closer examination reveals that this method reduces the total number of disk pages accessed, but not necessarily the total I/O costs. In step A3, when a page is required to accommodate tree growth but the buffer is full, the oldest page in the buffer is written to disk, incurring a random disk write. Similarly in step A4, buffer cache misses may cause dirty grown subtree nodes to be written to disk. Since tree nodes are visited one at a time during tree matching, buffer reclamation is done one page at a time, incurring random accesses.

3.3.1 Early Bulk I/O

If, instead of delaying writing dirty pages to disk until the pages are to be reclaimed, we write-reflect all dirty pages in linked lists and grown subtrees at an early time, no random writes will be incurred during space reclamation. Further, if the write-reflects are done in large chunks, most pages will be written to disk in sequential I/O. This approach may increase the total number of pages written to disk, but the benefits of performing I/O sequentially well outweigh the penalties.

PR1. Locate a grown node: If the subtree root is the first child of its parent seed node, allocate a new grown node for use in the next step.

Otherwise, the subtree is attached to the ith slot in its parent seed node for some $i > 1$. Locate the grown node G pointed to by the $i - 1$th slot. If the number of unused (pointer, bounding box) pairs in G is greater than the number of children in the subtree root, use G in the next step. Otherwise, allocate a new grown node.

PR2. Copy: Denote the located physical grown node as G, the number of children of the subtree root as k. Locate the position of the last marker in G. Let it be the jth pointer. If there is no marker in G, set $j = -1$.

Copy the contents of the subtree root to pointers $j + 1, j + 2, \dots, j + k$ of G. Change the slot pointer that points to the original subtree root to point to G. The original subtree root is not used anymore and can be freed.

PR3. Mark: If $j + k + 1 \le f_{max}$, put a marker at pointer $j + k + 1$ of G

Fig. 7. Algorithm for packing grown subtree roots.

A1. Copy seed levels: Copy seed levels from the seeding tree.

A2. Insert data and build linked lists: Insert data and build linked lists under the slots.

> *A2.1. Build linked list batches*: When buffer cache overflows, write linked lists longer than θ nodes to disk. If there are no linked lists with more than θ nodes, reduce θ by one and repeat until some linked lists are written. Continue until all data items are inserted.
>
> *A2.2. Build last batch*: Write the remaining linked lists with more than θ nodes to disk. If there are no linked lists with more than θ nodes, reduce θ by one and repeat until some linked lists are written.

A3. Build grown subtrees: For each slot, read in the linked lists built under it one by one, and build an R-tree from the data entries in the lists. Attach the R-tree to the corresponding slot.

A4. Enter tree matching phase: After all grown subtree are built, enter the tree matching phase.

Fig. 8. Outline of seeded tree construction algorithm.

A1. Copy seed levels: *Cost:* sequential or random read of seeding tree nodes.

A2. Insert data and build linked lists:

> A2.1. *Build linked list batches*: Use write-purge method for batch writes. *Cost:* one random write and a number of sequential writes for each batch write.
>
> *A2.2 Build last batch*: Same as *A2.1*.

A3. Build grown subtrees: Read each linked list in one sequential access. Use ordinary LRU buffer replacement policy during tree construction. *Cost:* one random read and a number of sequential reads for initial reading of each linked list. Each buffer cache overflow incurred by subtree growth causes one random write if the replaced page is marked dirty. Each buffer cache miss upon accessing a subtree node or a linked list node causes one random read and possibly one random write if the replaced page is marked dirty.

A4. Enter tree matching phase: Enter the tree matching phase with a warm buffer cache. There are dirty grown subtree nodes at this point. *Cost:* A dirty subtree node will cause one random write when its space is reclaimed in the tree matching phase.

Fig. 9. Old tree construction buffer management strategy.

We use individual linked lists and grown subtrees as units for such early write-reflects. This improved buffer management strategy and other fine-tuning are shown in Fig. 10. Step A3 in this algorithm requires the buffer space to be large enough to hold one whole linked list, but this is guaranteed to be the case. Since we reclaim linked lists pages as we finish processing them, no linked list pages remain in the buffer at the end of this step.

A1. Copy seed levels: Same as old algorithm.

A2. Insert data and build linked lists.

A2.1. Build linked list batches: Same as old algorithm.

A2.2 Build last batch: Write-reflect all remaining linked lists, instead of write-purging linked lists longer than *k* nodes to disk. *Cost:* similar to old algorithm, but with more sequential writes.

A3. Build grown subtrees:

- Build each grown subtree using its linked lists in the last batch first, since they may still be in the buffer.
- When accessing the first page of a linked list, pin all linked list pages in the buffer to avoid being paged out.
- When a linked list page is fully processed, unpin and free the page.
- When a grown subtree is built, write-reflect it to disk. Its pages remain in memory and subject to ordinary LRU replacement.
- All other aspects are the same as ordinary LRU buffer replacement policy.

Cost:

- Reading each linked list incurs one random read and a number of sequential reads. If a linked list accessed is left in memory by *A2.2*, the cost of reading it is saved.
- Writing a grown subtree to disk immediately after its construction incurs one random write and a number of sequential writes.

A4. Enter tree matching phase: Enter the matching phase with a warm buffer cache. *Cost:* Since all subtree pages are reflected to disks in *A3*, no writes occur in the matching phase.

Fig. 10. New tree construction buffer manager strategy.

Table 1 compares the performance of the seeded tree method using the old and the new buffer management strategies, using a pre-existing R tree constructed from an input file of 100,000 data items, and a seeded tree dynamically constructed from an input file of 40,000 data items. Columns "Matching" and "Construction" in Table 1 indicate tree matching costs and tree construction costs, respectively. Various "read" and "write" subcolumns show the numbers of disk pages accessed at different stages through different types of disk accesses.

Table 1 shows that the new buffer manager strategy eliminates all random writes from the tree matching phase. It also reduces the number of random writes substantially for the tree construction phase. The price paid is an increased number of sequential writes during tree construction, caused by write-reflects of newly constructed grown subtrees. Due to other fine tuning techniques, the total number of pages accessed is actually smaller for the new strategy, as shown in the column "total access." Since the new strategy incurs mostly sequentially accesses, its improvement over the old strategy is even greater when we compare the I/O costs. The columns labeled "Total cost" show the costs calculated under different assumptions for the ratio of sequential and random page access costs. Depending on the ratio assumed, the new buffer management strategy represents as much as 40 percent improvement in I/O costs over the old strategy.

The trends and the performance gain of the new over the old buffer management strategy presented in Table 1 are typical of all our experiments. In the following discussions, we will present only the results using the new buffer management strategy.

4 Theoretical Analysis

A parameter crucial to the performance of the seeded tree algorithm is the number of seed levels copied from the seeding tree. If only one seed level is copied, very little information is carried from the seeding tree to the seeded tree, and because the number of slots is small, the seeded tree will consist of a small number of big grown subtrees. In this case, the seeded tree may behave much like ordinary R-trees, and cannot take advantages of its features. On the other hand, if the number of seed levels is too large, the seed levels will occupy too much buffer space, and because the number of slots is very large, there can be too many grown subtrees, with each subtree degenerating into a very small tree, possibly a single node. A balance in the choice of the seed levels is important for good performance of the algorithm.

If the data set to be constructed into a seeded tree is too large, individual grown subtrees can still overflow the buffer and cause performance degradation, even with a choice of a large number of slots.

This section relates these issues and discusses the relationships between buffer size, data set size, and the number of seed levels. We start with a simple consideration of the requirements for the seeded tree algorithm to achieve good performance, then identify the feasible ranges for the input data set size and the number of slots. Based on these results, we give a simple algorithm to provide guidelines for choosing the number of seed levels.

TABLE 1
Comparison of New and Old Buffer Manager Policies

	Matching		*Construction*					*Total cost (ρ = sequential/random)*		
Algorithm	*ran. read*	*ran. write*	*ran. write*	*ran. write*	*seq. read*	*seq. write*	*Total access*	ρ = 1/5	ρ = 1/10	ρ = 1/30
STJ-OLD	1598	510	159	831	697	854	4649	3408	3253	3150
STJ-NEW	1599	0	87	116	415	2119	4336	2309	2055	1886

ρ is the cost ratio of accessing a disk page sequentially to that of accessing one randomly.

Table 2 lists the parameters used in the following discussion. Without loss of generality, we assume that tree nodes, buffer pages, and disk pages are all of the same size. Since there can be numerous 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 seeding tree and the seeded tree both have large node fanout, so that the size of a tree is approximately the size of its leaf level.

TABLE 2
SEEDED TREE CONSTRUCTION PARAMETERS

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

4.1 Seeded Tree Algorithm Requirements

For feasibility and efficiency, the seeded tree algorithm requires the following:

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 available buffer space to avoidrandom disk access during grown subtree construction.

R2.2 occupy a small fraction of the available buffer for good performance during actual join.

4.1.1 Requirement R1

Requirement R1.1 may be expressed as $N_s < B$. Since the nodes have large fanout, the number of nodes in seed levels is approximately the number of node in the slot levels, i.e., $N_s \approx n_l$. Therefore, we require $n_l < B$.

To obtain better performance, we apply R1.2 and require seed level nodes to occupy only a fraction of the buffer, which translates into the formula

$$n_l < B/E, \tag{1}$$

where $E > 1$ is a tunable constant.

4.1.2 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 $\left(\frac{D}{S}\right) / \left(\frac{f_{ave}}{f_{max}}\right)$ 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 $\left(\frac{D}{S}\right) / \left(\frac{f_{ave}}{f_{max}}\right)$ 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 may be expressed as

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

for some tunable constant $C > 1$.

Recalling $S = n_l \cdot f_l$, we can rearrange this formula into

$$n_l > \frac{CDf_{max}}{(B - n_l)(f_{ave} f_l)}. \tag{2}$$

Defining $K = CDf_{max}/(f_{ave}\ f_l)$ and solving for n_l, we get a lower bound for n_l :

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

4.2 Choosing the Number of Seed Levels and Number of Slots

Combining (1) and (2), we derive bounds for the numbers of seed nodes and slots as follows:

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

$$\frac{(B - \sqrt{B^2 - 4K})}{2} < S < \frac{Bf_l}{E}. \tag{5}$$

Thus, the seeded tree algorithm will perform well if the number of seed nodes/slots falls within the range prescribed by (4) and (5). However, our algorithm is designed not to break down even if these numbers fall outside the prescribed ranges (see Section 5.2). It simply runs less efficiently.

Determining the number of seed levels is straightforward. We descend from the root level of the seeding tree, and find the first level whose n_l and f_l satisfy (4). The number of levels descended in the seeding tree is the number of seed levels in the seeded tree. If no level satisfies both conditions in (4), we repeat the process but check only the upper bound of (4) the second time, since Requirement R1, which induces the upper bound, is more fundamental. The bounds on the number of slots prescribed by (4), and thus the number of seed levels chosen by this algorithm, are advisory, and may be considered as default values. When more information is available, such as data clustering and correlations between the input data sets, other choices may emerge as more reasonable.

In practice, the range between the upper and lower bounds on the number of seed nodes/slots given by (4)/(5) is usually very wide, so that the seeding tree almost always contains a level with a number of seed nodes that satisfies both bounds (see Section 5.1).

The following theorem sheds light on the implication of the parameter C and on the stability of seeded trees.

THEOREM 1. *Let the average grown subtree size be smaller than 1/C of the available buffer space. The number of subtrees with size greater than the available buffer space is at most 1/C of the total number of grown subtrees. In other words, if* $\frac{D}{S} < \frac{B'}{C}$*, then the number of grown subtrees with size greater than* B' *is at most* S/C.

PROOF. Assume there are a grown subtrees with size greater than B'. Denote the average size of these subtrees B_a, and the average size of the rest of the subtrees B_b. Then, we have,

$$a \cdot B_a + (S - a) \cdot B_b = D.$$

This can be transformed into

$$a = \frac{D - (S-a) \cdot B_b}{B_a} < \frac{D}{B_a}.$$

Because $B_a > B'$, we have $a < \frac{D}{B'}$. From $\frac{D}{S} < \frac{B'}{C}$, $a < \frac{S}{C}$. □

4.3 Feasible Input Data Set Sizes

Inequality (4) can also be used to determine what input data sizes can be handled by our algorithm.

If the number of seed levels is 1, then $l = 0$, and n_0 must satisfy (4). The upper bound in (4) is easily satisfied since the seed levels now occupy only one page, and any reasonable system will have a much larger buffer. In this case, we can apply the lower bound from (4) on n_0, and obtain the requirement

$$D < \frac{Bf_{ave}f_0}{Cf_{max}}. \tag{6}$$

If the number of seed levels is more than 1, i.e., $l \geq 1$, let i be the lowest level satisfying the lower bound in (4). Now, if

$$f_{i-1} < \left(\frac{B}{E}\right) / \left(\frac{B - \sqrt{B^2 - 4K}}{2}\right) \tag{7}$$

then $n_i = n_{i-1} \cdot f_{i-1} < \frac{B - \sqrt{B^2 - 4K}}{2} \cdot f_{i-1} < \frac{B}{E}$. Thus, n_i will also satisfy the upper bound in (4).

Simplifying (7) and solving for D, we have

$$D < \frac{B^2}{CE} \cdot \frac{f_{ave}}{f_{max}} \cdot \frac{f_l}{f_{l-1}} \cdot \left(1 - \frac{1}{Ef_{l-1}}\right) \tag{8}$$

Thus, for an input data set of size D, if there exists a f_l in the seeding tree satisfying Inequality (6) or (8), we are guaranteed to find a number of seed levels for the seeded tree that satisfies requirements R1 and R2 (or equivalently, Inequality (4).

The above upper bound on the feasible data set size D depends on f_l, and therefore on the structure of the particular seeding tree in the join. To obtain a sense of the feasible data set sizes D, we prefer a bound independent of f_l. Appendix A proves that the following condition implies (8):

$$D < \frac{B^2}{CE}\left(\frac{f_{ave}}{f_{max}}\right)^2 \cdot \frac{Ef_{min} - 1}{Ef_{min}} \cdot \frac{Ef_{max} - 1}{Ef_{max}} \tag{9}$$

Condition (9) provides a good first-cut estimate for the upper bound on the feasible size of the input data set. Note that the upper bound is proportional to the square of the buffer size, and therefore is quite large even for moderate buffer sizes.

As an example, suppose f_{min}, f_{ave}, and f_{max} are 16, 33, and 50, respectively. Let parameters C and E both be 3, so that the average grown subtree size and the maximum size of the seed levels are both smaller than 1/3 of buffer size. If buffer size is 500 pages, by (9) we have $D < 11{,}769$ pages. That is, for data set up to approximately 11.8 Mbytes in size, we are guaranteed to find a number of seed levels satisfying both bounds in (4). Note that (9) is pessimistic. In practice, data set sizes larger than its upper bound may still satisfy (4).

5 PERFORMANCE STUDIES

We conducted experiments to examine the behavior of seeded trees with both simulated and real-life data. 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. [16]. 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 [16]. For generality, the original R-tree structure was used, and not any of its variations.

For simplicity, we assume that the disk page size and the memory page size are both 1 Kbyte, 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 reused. Both methods enter the tree matching phase with a warm buffer. That is, the pages of the newly constructed trees are left in the buffer for use in the tree matching phase. For **STJ**, the new buffer management strategy described in Section 3.3 write-reflects grown subtrees page to disk right after they are constructed. Therefore, no disk writes are involved when these pages are replaced. For **RTJ**, we do not write-reflect the pages of the newly constructed R-tree, and so there are disk writes at tree matching time when these pages are replaced by LRU policy. The reasons are as follows. First, due to the way R-trees are constructed, the dirty pages left in the buffer will be assigned to nonconsecutive locations in disk, and cannot be written to disk in a few large sequential writes. The costs are approximately the same whether we write-reflect or not. Second, most of the new R tree pages would have been written to disk during tree construction anyway, and the disk writes during tree matching constitute a very small fraction of **RTJ** costs.

We studied data of different degrees of spatial clustering, which was controlled by a simple scheme in our experiments. 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 cover quotient of the clustering rectangles (total area of the clustering rectangles divided by the map area) is denoted as CCQ. The smaller the value of CCQ, the more clustered the data set.

The length and 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.

5.1 Effects of Size and Data Clustering

We conducted two series of basic experiments. In the first series, we fixed the cardinality of D_R at 100,000 and varied the cardinality of D_S from 20,000 to 80,000. We will use $\|D\|$ to mean the cardinality of a set D. The upper bound on the side length of clustering rectangles was set to 0.04. The resulting CCQ 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 percent 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 CCQ 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 tested the same set of seeded tree variants 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 the combination (C_3, U_3).

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

Table 3 shows the number of slots suggested by our guidelines (5) for various sizes of D_S. Parameters B and E in the equation are both chosen to be 3. The upper bound is the same for all data set sizes, since it depends only on the buffer size. The ranges between the upper and lower bounds are actually very wide with our test cases (more than two orders of magnitude). Therefore, we have great flexibility in choosing a suitable number of seed levels. With a D_R of 100,000 objects, the seeding R-tree we constructed has four levels, with 1, 3, 94, and 3,099 nodes, respectively, at each of its four levels. Using the algorithm in Section 4.2, we choose the number of slots for our seeded trees to be 2. This means there will be 94 slots in each seeded tree.

TABLE 3
LOWER AND UPPER BOUNDS ON NUMBER OF SLOTS FOR VARIOUS DATA SET SIZES PRESCRIBED BY (5)

bound	$\|D_R\| = 20K$	$\|D_R\| = 40K$	$\|D_R\| = 60K$	$\|D_R\| = 80K$
lower	3.60	7.20	10.80	14.41
upper	5555.56	5555.56	5555.56	5555.56

Table 4 shows the CPU costs for one experiment in the first series, where $\|D_R\|$ = 100,000, $\|D_S\|$ = 40,000, and CCQ of D_R was fixed at 0.2. Column *"mbr-overlap"* lists the numbers of bounding box overlap tests performed during tree construction. Column *"X-overlap"* and *"Y-overlap"* are the numbers of operations that test whether two bounding boxes overlap along the X and Y axis, respectively. These two operations are used during tree matching [16], [1]. The CPU costs of both the **RTJ** and **STJ** methods are an order of magnitude better than that of **BFJ**, with **STJ** somewhat better but comparable with **RTJ**. This was the case throughout all our experiments. We will focus on the disk I/O costs in the following discussions.

TABLE 4
JOIN CPU COSTS
$\|D_R\| = 100K$, $\|D_S\| = 40K$, CCQ of D_R = 0.2

Algorithm	*mbr-overlap*	*X-overlap*	*Y-overlap*
BFJ	4648868	0	0
RTJ	295433	203776	168293
STJ	169112	189851	159554

Tables 5 and 6 show the I/O costs from the experiments in detail. The column *"Parameter"* indicates the parameter that was varied across the experiments. The numbers of random and sequential disk block accesses for tree construction and matching stages are under columns *"Matching"* and *"Construction,"* respectively. For some of the algorithms, we may need to flush memory pages containing T_S nodes to disk during tree matching time if they

TABLE 5
JOIN PERFORMANCE FOR EXPERIMENT SERIES 1, VARYING D_S SIZES

Parameter =		*Matching*		*Construction*					*Total cost (ρ = sequential/random)*		
D_S sizes	*Alg.*	*ran. read*	*ran. write*	*ran read*	*ran. write*	*seq. read*	*seq. write*	*total access*	ρ = 1/5	ρ = 1/10	ρ = 1/30
$\|\|D_S\|\|$ = 20*K*	BFJ	438	0	0	0	0	0	438	438.0	438.0	438.0
	RTJ	1141	386	130	243	0	0	1900	1900.0	1900.0	1900.0
	STJ	688	1	0	121	0	1014	1824	1012.8	911.4	843.8
$\|\|D_S\|\|$ = 40*K*	BFJ	8864	0	0	0	0	0	8864	8864.0	8864.0	8864.0
	RTJ	2438	58	5987	1219	0	0	9702	9702.0	9702.0	9702.0
	STJ	1599	0	87	116	415	2119	4336	2308.8	2055.4	1886.5
$\|\|D_S\|\|$ = 60*K*	BFJ	13650	0	0	0	0	0	13650	13650.0	13650.0	13650.0
	RTJ	2563	46	12232	1887	0	0	16728	16728.0	16728.0	16728.0
	STJ	2374	0	185	109	828	3193	6689	3472.2	3070.1	2802.0
$\|\|D_S\|\|$ = 80*K*	BFJ	17151	0	0	0	0	0	17151	17151.0	17151.0	17151.0
	RTJ	3274	48	16499	2525	0	0	22346	22346.0	22346.0	22346.0
	STJ	3004	0	276	111	1242	4264	8897	4492.2	3941.6	3574.5

TABLE 6
JOIN PERFORMANCE OF EXPERIMENT SERIES 2, VARYING *CCQ* OF D_R

Parameter =		*Matching*		*Construction*					*Total cost (ρ = sequential/random)*		
CCQ of D_R	*Alg.*	*ran. read*	*ran. write*	*ran read*	*ran. write*	*seq. read*	*seq. write*	*total access*	ρ = 1/5	ρ = 1/10	ρ = 1/30
CCQ = 0.2	BFJ	8864	0	0	0	0	0	8864	8864.0	8864.0	8864.0
	RTJ	2438	58	5987	1219	0	0	9702	9702.0	9702.0	9702.0
	STJ	1599	0	87	116	415	2119	4336	2308.8	2055.4	1886.5
CCQ = 0.4	BFJ	14803	0	0	0	0	0	14803	14803.0	14803.0	14803.0
	RTJ	2874	57	6881	1217	0	0	11029	11029.0	11029.0	11029.0
	STJ	2261	0	92	124	414	2102	4993	2980.2	2728.6	2560.9
CCQ = 0.6	BFJ	23177	0	0	0	0	0	23177	23177.0	23177.0	23177.0
	RTJ	3448	62	6342	1202	0	0	11054	11054.0	11054.0	11054.0
	STJ	3163	0	96	135	408	2121	5923	3899.8	3646.9	3478.3
CCQ = 0.8	BFJ	25167	0	0	0	0	0	25167	25167.0	25167.0	25167.0
	RTJ	3303	66	6259	1195	0	0	10823	10823.0	10823.0	10823.0
	STJ	3100	0	93	136	410	2124	5863	3835.8	3582.4	3413.5
CCQ = 1.0	BFJ	31831	0	0	0	0	0	31831	31831.0	31831.0	31831.0
	RTJ	3704	71	5948	1207	0	0	10930	10930.0	10930.0	10930.0
	STJ	3366	1	90	132	410	2108	6107	4092.6	3840.8	3672.9

have not been reflected to disk and space is required for loading other tree nodes. The write costs under *"Matching"* should thus be charged to the tree construction part of each algorithm. In addition, we list the total number of disk pages accessed and the I/O costs calculated for different ratios of sequential and random access costs. We define the *sequential/random ratio*, denoted ρ, to be the ratio of the average cost of accessing one disk block sequentially to that of accessing it through a random access. For each experiment, we show the weighted I/O costs for ρ = 1/5, 1/10, and 1/30, respectively. The column *"total accesses"* shows the count of all disk blocks accessed sequentially or randomly. This number equals the total cost for ρ = 1. The total costs and cost break-down of the first series of experiments are shown in Figs. 11, 12, and 13, where "STJ-1/5," "STJ-1/10," and "STJ-1/30" are the costs for the seeded tree join calculated with ρ = 1/5, 1/10, and 1/30, respectively. The total costs of the second series of experiments are shown in Fig. 14.

In the first series of experiments, we found as expected that I/O costs go up as the size of D_S increases. **STJ** outperforms **RTJ** in all experiments. 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 mainly from reading back the linked lists when constructing the grown subtrees. The number of pages read at creation time remains very small for **STJ** even for large D_S sizes, while for **RTJ** this number is at least an order of magnitude greater. Our earlier experiments showed that **STJ** incurred similar numbers of creation time reads as **RTJ** when intermediate linked lists were not used. Using intermediate linked lists in tree construction successfully eliminated the I/O cost caused by the LRU buffer

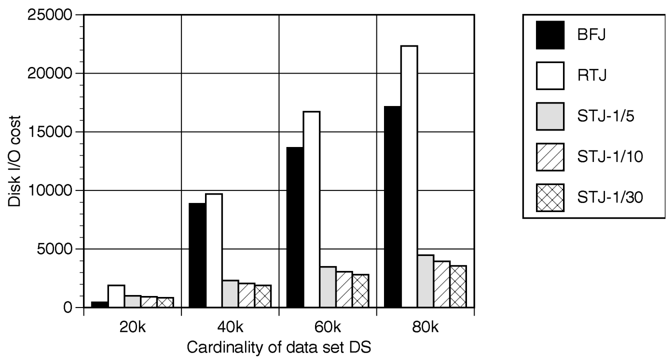


Fig. 11. Total disk I/O costs. Experiment Series 1.

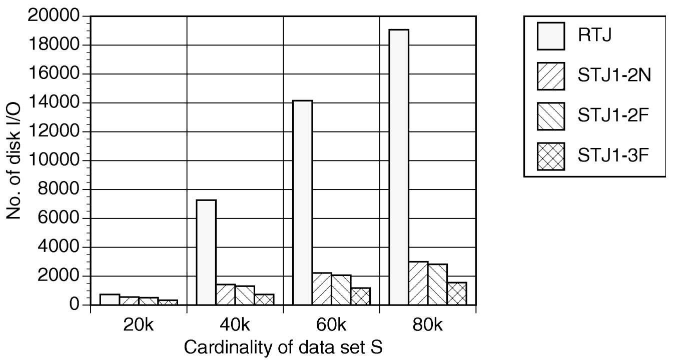


Fig. 12. I/O costs for tree construction, Experiment Series 1.

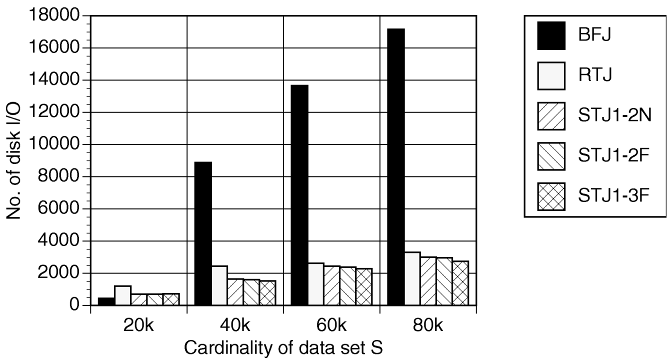


Fig. 13. I/O costs for tree matching, Experiment Series 1.

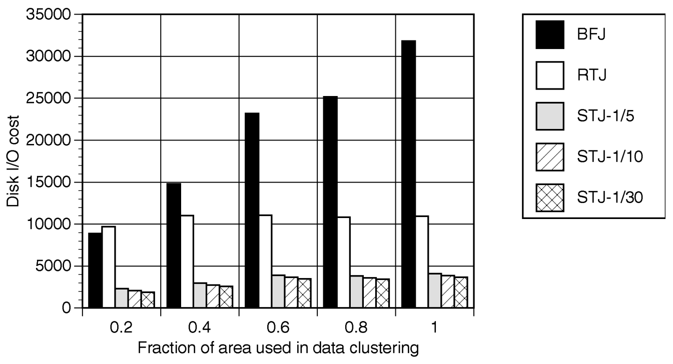


Fig. 14. Total disk I/O costs, Experiment Series 2.

misses. It is worth noting that **STJ** did not incur any disk read at all for $\|D_S\| = 20K$ in Table 5.

STJ outperforms **BFJ** in all cases expect when D_S size is the smallest (see experiment for $\|D_S\| = 20K$ in Table 5). We found that **BFJ** accessed only 483 $_{\text{different}}$ T_R nodes in this case. Since this number was smaller than the buffer size, no buffer overflow occurred. Overflow occurred with input data set of the same sizes for **STJ** and **RTJ**. This is because the buffer must hold nodes from both T_S and T_S during tree matching, and the number of nodes from both trees was greater than the number of buffer pages. This suggests that if the number of pages accessed in the index tree is smaller than the buffer size, **BFJ** should be used as the join method. However, with larger data sets, **BFJ** accessed at least twice the number of disk pages than did **STJ**. The differences in terms of translated I/O cost is about four to five times, for $\rho = 1/5$ and $\rho = 1/30$, respectively. To our surprise, **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 any savings.

It is worth noting that in general **STJ** outperformed other methods even in terms of total number of disk accesses. When the disk accesses numbers were translated into weighted I/O costs, the differences were even more pronounced.

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 40 percent 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.

Again, **STJ** outperforms other methods significantly even in terms of the number of disk I/O accesses.

5.2 Effects of Buffer Size

We now compare the stability of the seeded-tree algorithm with other join methods across a range of buffer sizes. We are also interested in exploring the behavior of our method when there does not exist a choice for the number of seed levels that satisfies both bounds in Inequality (5). We ran experiments with $\|D_R\| = 100{,}000$, $\|D_S\| = 40{,}000$, and buffer sizes varying from 16 to 512 pages. Figs. 15 and 16 show the tree construction costs and the total costs of **RTJ** and **STJ** under different buffer sizes.

As these figures show, **STJ** outperforms **RTJ** substantially throughout the whole buffer size range, both in terms of tree construction costs and total costs. Further, the performance of **STJ** degrades more slowly than that of **RTJ** as buffer size decreases for most of the buffer size range. This shows that **STJ** is much less sensitive to buffer sizes than **RTJ**.

We would like to validate the assertion in Section 4 that the seeded tree algorithm runs with good performance if the number of slots is within the range prescribed by Inequality (5), and will not break down when the number of slots falls outside this range. The bounds in the Inequality depend on the

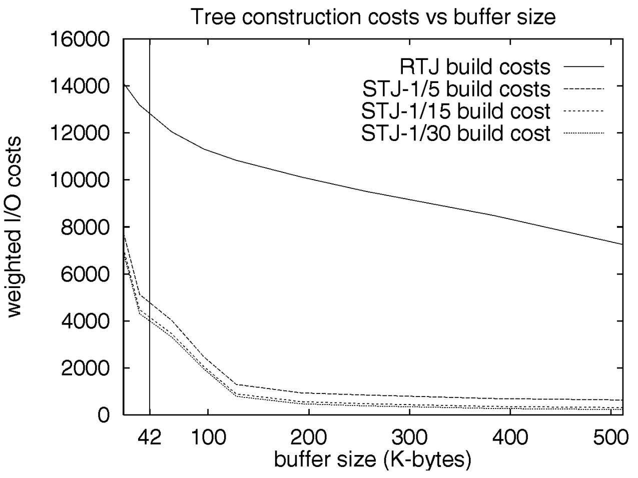


Fig. 15. I/O costs for tree construction, Experiment Series 2.

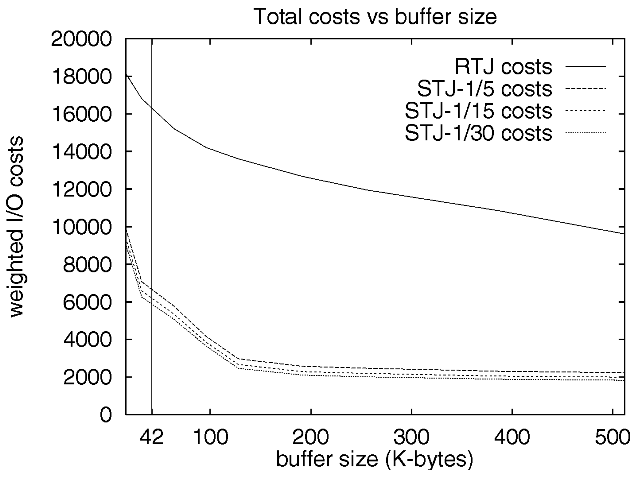


Fig. 16. I/O costs for tree matching, Experiment Series 2.

buffer size. In these experiments, the input data set of cardinality was 40,000, parameters C and E were both 3, and the seeding tree had 1, 9, 94, and 3,009 nodes, respectively, at each of its levels. In this case, Inequality (5) is satisfied for buffer sizes between 42 and 512 pages if we pick the number of seed levels to be 2 (i.e., 94 slots). For buffer sizes smaller than 42 pages, no choice of the number of seed levels will satisfy both bounds in (5) simultaneously.

Fig. 15 and Fig. 16 show that the performance **STJ** remains low throughout the recommended operational range. Even for buffer sizes smaller than 42 pages, the performance degraded gracefully. These experiments validate our claims in Section 4. We emphasize that even for buffer sizes below the theoretical threshold, **STJ** still incurs only approximately half the amount of I/O incurred by **RTJ**.

5.3 Stress Tests and Test with Real-Life Data

To test the stability of the **STJ** method, we ran a series of of experiments with data designed to induce degraded performance in **STJ**. We also conducted tests with real-life data. Except for the experiment with real-life data, all these experiments were performed on methods **STJ** and **RTJ**, with $\|D_R\| = 100{,}000$ and $\|D_S\| = 40{,}000$. Table 7 shows the results of these experiments.

As a control reference, we obtained the performance of **STJ** and **RTJ** on independently generated clustered D_R and D_S, both with $CCQ = 0.2$. This data set configuration appeared in both experiments series on data set size and degree of clustering, and shows the typical performance gain of **STJ** over **RTJ**. This experiment is labeled "CONTROL."

In experiment "CLU/UNI," D_R is a clustered data set with $CCQ = 0.2$, while D_S is a uniform data set for which $CCQ = 1$. This results in the seeded tree inserting evenly distributed data into unevenly distributed slots. Experiment "UNI/CLU" works the other way around, and results in the seeded tree inserting unevenly distributed data into evenly distributed slots. Both experiments are designed to induce uneven grown subtree sizes and oversized subtrees in the seeded tree, thus stressing **STJ**.

Experiment "EXCLU" tested two spatially clustered but negatively correlated spatial data sets, both having $CCQ = 0.2$. In this experiment, D_R was a independently generated clustered data set, while D_S was generated by randomly placing the centers of its cluster rectangles in the area outside the cluster rectangles of D_R. For such data sets, the number of matched pairs is much smaller than those in other experiments. For the same input data sets sizes, this experiment has an output of 40K matched pairs of objects, while other experiments all have 70K matched pair of objects.

We also tested **STJ** on real-life data. Experiment "REAL" is a join between two data sets extracted from the TIGER/line files of the U.S. Bureau of Census [18]. In this experiment D_R is a map of the streets and has 131,461 objects, while D_S is a map of rivers and railway tracks, and has 128,971 objects. This experiment shows a performance gain of **STJ** over **RTJ** similar to that of "CONTROL."[1]

Table 7 shows the various cost components of join in these experiments, and Table 8 lists the performance gain of **STJ** over **RTJ** with $\rho = 10$. In all cases, the performance of **STJ** is many times better than **RTJ**. As expected, the performance gains of **STJ** over **RTJ** for both "CLU/UNI" and "UNI/CLU" were lower than that of "CONTROL," due to the over-sized grown subtrees we introduced. In experiment "EXCLU," the gain of **STJ** over **RTJ** is higher than even in "CONTROL." This shows that the seeded tree was more successful in taking advantage of the mutually exclusive nature of the data sets, and in avoiding accessing unnecessary tree nodes during the tree matching phase.

Over all, these experiments shows that **STJ** steadily outperforms **RTJ** by a large margin even under various boundary cases and real-life data sets.

6 Discussion

Our experiments demonstrate that the seeded tree method is not just efficient, but also stable. It outperforms other methods by large margins across different input data sizes, degrees of clustering, and other data characteristics. For experiments designed to induce degraded performance in the seeded tree method, it still runs three times faster than

1. In [16] a tree matching experiment was performed on two R^*-trees built from the same data sets and resulted in 9,385 random accesses. Despite its better tree matching performance, we do not use the R^*-tree in our studies because its construction cost is much higher than that of the R-tree.

TABLE 7
STRESS TESTS

Data sets	Alg.	*Matching* ran. read	*Matching* ran. write	*Construction* ran read	*Construction* ran. write	*Construction* seq. read	*Construction* seq. write	total access	*Total cost (ρ = sequential/random)* ρ = 1/5	ρ = 1/10	ρ = 1/30
CONTROL	RTJ	2304	59	6028	1223	0	0	9614	9614.0	9614.0	9614.0
	STJ	1601	1	37	120	324	2084	4167	2240.6	1999.8	1839.3
CLU/UNI	RTJ	3103	65	5895	1204	0	0	10267	10267.0	10267.0	10267.0
	STJ	2736	1	41	128	368	2087	5361	3397.0	3151.5	2987.8
UNI/CLU	RTJ	3056	59	6028	1223	0	0	10366	10366.0	10366.0	10366.0
	STJ	2751	0	54	142	432	2080	5459	3449.4	3198.2	3030.7
EXCLU.	RTJ	2056	61	5813	1215	0	0	9145	9145.0	9145.0	9145.0
	STJ	1368	0	34	123	322	2082	3929	2005.8	1765.4	1606.1
REAL	RTJ	15835	64	30243	4000	0	0	50142	50142.0	50142.0	50142.0
	STJ	10309	1	313	212	2028	6753	19616	12591.2	11713.1	11127.7

RTJ. An experiment with real-life data mirrors the results from experiments with generated data.

The seeded tree join method presented in this paper uses a seeded tree and a precomputed R-tree. However, there can still be situations where there is no precomputed index for either input data set. For instance, both input data sets can be outputs from other database operations. In the one-seeded-tree scenario, the seed levels of the seeded tree are derived from the seeding tree, which is the precomputed R-tree. In the two-seeded-tree scenario, there is no the seeding tree, and no obvious source to derive the seed levels for either seeded tree. To solve this problem, we extract information from the input data sets using sampling techniques [19], and use such information as a basis for building the seed levels. This approach is elaborated in [20].

A closely related problem is to find 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 valuable 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 scope 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. 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.

TABLE 8
PERFORMANCE GAINS OF **STJ** OVER **RTJ** IN STRESS TESTS WITH ρ SET TO 10

Experiments	NORMAL	CLU/UNI	UNI/CLU	EXCLU.	REAL
gain	4.80	3.25	3.24	5.18	4.28

7 CONCLUSIONS

This paper presents a spatial join method that dynamically constructs a new kind of index tree, called the seeded tree, at join time. This method addresses the situations where existing R-trees cannot help with join processing, or where no R-trees exist for the input data sets.

A 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. Since tree construction cost is essential to the performance of the seeded tree method, we presented a tree construction technique that drastically reduces construction time I/O costs by using intermediate linked lists at an intermediate step. We also presented a new buffer management strategy for tree construction that reflects dirty buffer pages in large chunks to disk long before buffer space reclamation. The new buffer management strategy eliminates all random writes at the join phase and provides up to 40 percent improvement over that implemented in [1] in disk I/O costs, depending on the assumed ratio of sequential to random disk access costs. We also provided theoretical analysis of the seeded tree method and choice of algorithmic parameters.

We have tested the seeded tree technique against other methods with input data sets of various characteristics, as well as with real-life data. 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, except in one boundary case. In general, the seeded tree method accessed less than 50 percent of the number of disk pages accessed by the faster of the other two methods. The total weighted I/O costs of the seeded tree method are always lower than the faster of the other two methods by a factor of three to five, depending on the assumed ratio of sequential to random disk access costs. Tree construction using intermediate linked lists is shown to be very effective in eliminating tree construction time buffer misses. The new buffer management strategy has successfully eliminated all tree matching phase random writes. CPU costs for the seeded tree method are always the lowest among all methods.

APPENDIX A

Assume level l of an index tree has m nodes in all, and let the fanout of each node be denoted by $f_l^1, f_l^2, \ldots,$ and f_l^m. The average fanout of level l is $\frac{f_l^1 + f_l^2 + \ldots + f_l^m}{m}$. When m is large, the Central Limit Theorem [21] can be applied, and we have $f_l \approx f_{ave}$.

Assuming the Central Limit Theorem holds when the number of nodes at a level is greater than f_{min}, then we have the following theorem.

THEOREM 2. *If the input data set size satisfies*

$$D < \frac{B^2}{CE}\left(\frac{f_{ave}}{f_{max}}\right)^2\left(1 - \frac{1}{Ef_{min}}\right)\left(1 - \frac{1}{Ef_{max}}\right),$$

then it will satisfy

$$D < \frac{B^2}{CE} \cdot \frac{f_{ave}}{f_{max}} \cdot \frac{f_l}{f_{l-1}} \cdot \left(1 - \frac{1}{Ef_{l-1}}\right).$$

We first note that for $2 \le f_{l-1} < x$, we have algebraically

$$\frac{1}{f_{l-1}}\left(1 - \frac{1}{Ef_{l-1}}\right) \ge \frac{1}{x}\left(1 - \frac{1}{E \cdot x}\right). \qquad (10)$$

PROOF. To prove the theorem, it suffices to prove

$$\frac{f_l}{f_{l-1}}\left(1 - \frac{1}{Ef_{l-1}}\right) > \frac{f_{ave}}{f_{max}}\left(1 - \frac{1}{Ef_{min}}\right)\left(1 - \frac{1}{Ef_{max}}\right).$$

We will call the left-hand side of this inequality LHS, and the right-hand side RHS.

For $f_{l-1} > f_{min}$, since $n_l = n_{l-1} \cdot f_{l-1} \ge f_{l-1} > f_{min}$, the Central Limit Theorem applies to level l and $f_l \approx f_{ave}$. By using (10) with f_{max} substituted for x, we have

$$\begin{aligned} LHS &\ge \frac{f_{ave}}{f_{max}}\left(1 - \frac{1}{E \cdot f_{max}}\right) \\ &> \frac{f_{ave}}{f_{max}}\left(1 - \frac{1}{E \cdot f_{max}}\right) \cdot \left(1 - \frac{1}{E \cdot f_{min}}\right) \\ &= \text{RHS}. \end{aligned}$$

For $f_{l-1} \le f_{min}$, by noting $f_l < f_{min}$ by definition, and substituting f_{min} for x in (10), we have

$$\begin{aligned} \text{LHS} &\ge \left(1 - \frac{1}{E \cdot f_{min}}\right) \\ &> \left(1 - \frac{1}{E \cdot f_{min}}\right) \cdot \frac{f_{ave}}{f_{max}}\left(1 - \frac{1}{E \cdot f_{max}}\right) \\ &= \text{RHS} \end{aligned}$$

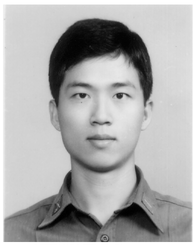
□

ACKNOWLEDGMENTS

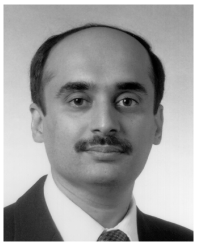
The authors would like to thank Dr. Thomas Brinkhoff, Prof. Hans-Peter Kriegel, and Prof. Bernhard Seeger for generously providing the data sets used in our experiment with real-life data. This work was supported, in part, by the Consortium for International Earth Science Information Networking.

REFERENCES

[1] M.-L. Lo and C.V. Ravishankar, "Spatial Joins Using Seeded Trees," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 209–220, Minneapolis, Minn., May 1994.
[2] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
[3] D. Rotem, "Spatial Join Indices," *Proc. Int'l Conf. Data Eng.*, pp. 500–509, Kobe, Japan, 1991.
[4] P. Valduriez, "Join Indices," *ACM Trans. Database Systems*, vol. 12, no. 2, 1987.
[5] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Systems*, vol. 9, no. 1, 1984.
[6] W. Lu and J. Han, "Distance-Associated Join Indices for Spatial Range Search," *Proc. Int'l Conf. Data Eng.*, pp. 284–292, 1992.
[7] J.A. Orenstein, "Redundancy in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, Portland, Ore., 1989.
[8] J. Orenstein, "A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 343–352, 1990.
[9] J. Orenstein, "An Algorithm for Computing the Overlay of k-Dimensional Spaces," *Advances in Spatial Databases (SSD '91)*, O. Gunther and H.-J. Schek, eds., pp. 381–400, Zurich, Switzerland. Springer-Verlag, Aug. 1991.
[10] R.H. Güting and W. Schilling, "A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem," *Information Sciences*, vol. 42, no. 2, pp. 95–112, July 1987.
[11] O. Gunther, "Efficient Computation of Spatial Joins," *Proc. Int'l Conf. Data Engineering*, pp. 50–59, 1993.
[12] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 47–57, Aug. 1984.
[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The *R**-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 322–332, May 1990.
[14] C. Faloutsos, T. Sellis, and N. Roussopoulos, "Analysis of Object Oriented Spatial Access Methods," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 427–439, 1987.
[15] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. Very Large Data Bases*, pp. 3–11, Brighton, England, 1987.
[16] T. Brinkhoff, H.P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 237–246, May 1993.
[17] J. Star and J. Estes, *Geographic Information Systems*. Englewood Cliffs, N.J.: Prentice Hall, 1990.
[18] U.S. Bureau of Census, "Tiger/Lines Precensus Files: 1990 Technical Documentation," technical report, U.S. Bureau of Census, Washington, D.C., 1989.
[19] F. Olken and D. Rotem, "Sampling from Spatial Databases," *Proc. Int'l Conf. Data Eng.*, pp. 199–208, 1993.
[20] M.-L. Lo and C.V. Ravishankar, "Generating Seeded Trees from Data Sets," *Proc. Fourth Int'l Symp. Large Spatial Databases (Advances in Spatial Databases: SSD '95)*, Portland, Maine, Springer-Verlag, Aug. 1995.
[21] W.J. Dixon, *Introduction to Statistical Analysis*, fourth ed. New York: McGraw-Hill, 1983.



Ming-Ling Lo received his bachelor's degree in electrical engineering from National Taiwan University, and his MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1992 and 1996, respectively. Dr. Lo has been a research staff member at the IBM Thomas J. Watson Research Center, Hawthorne, New York, since 1996. His research interests include parallel databases, multidimensional data and query processing, and database systems and network computing in general.



Chinya V. Ravishankar received the BTech degree in chemical engineering from the Indian Institute of Technology, Bombay, and the MS and PhD degrees in computer sciences from the University of Wisconsin, Madison, in 1986 and 1987, respectively. He has been on the faculty of the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor, since 1986. His teaching and research at the University of Michigan have been in the areas of databases, distributed systems, networks, and programming languages. Dr. Ravishankar founded the Software Systems Research Laboratory at the University of Michigan, where he is also a member of the Real-Time Computing Laboratory. Dr. Ravishankar is a member of the IEEE, the IEEE Computer Society, and the Association for Computing Machinery.