BFRJ: Global Optimization of Spatial Joins Using R-trees *

Yun-Wu Huang
University of Michigan
ywh@eecs.umich.edu

Ning Jing†
Changsha Institute of Technology
jning@eecs.umich.edu

Elke A. Rundensteiner‡
Worcestor Polytechnic Institute
rundenst@cs.wpi.edu

Spatial joins are important operations in applications such as Geographic Information Systems, Cartography, and CAD/CAM. Spatial join using existing R-trees is a very useful and popular technique because of both its superior performance and the wide spread implementation of R-trees as spatial index structures. This paper describes a new spatial join method called **BFRJ** (Breadth-First R-tree Join). **BFRJ** synchronously traverses both R-trees in breadth-first order processing the join computation one level at a time. This way an intermediate join index can be created at each level to guide the join process at the next lower level. Unlike the limitation of the state-of-the-art depth-first R-tree join method which can only optimize I/O within local sub-trees, the breadth-first ordering allows **BFRJ** to deploy global optimization strategies among *all* nodes at the next lower level. In particular, **BFRJ** optimization strategies include index ordering, memory management, and buffer management of the intermediate join indices. This paper also presents an experimental evaluation of the effect of the proposed optimizations as well as a performance comparison between *BFRJ* and the state-of-the-art approach. Our experimental results indicate that **BFRJ** with global optimizations can outperform the competitor by a significant margin (up to 50%).

Dept. of Computer Science, Worcester Polytechnic Institute, Tech. Report WPI-CS-TR-97-5, Jan. 1997.

* This work was supported in part by the University of Michigan ITS Research Center of Excellence grant (DTFH61-93-X-00017-Sub) sponsored by the U.S. Dept. of Transportation and by the Michigan Dept. of Transportation.

† This work was performed while the author was a visitor at the University of Michigan.

‡ This work was performed while the author was a faculty member at the University of Michigan.

1 Introduction

The ability to manage spatial data has become more and more crucial in a wide range of applications such as geographic information systems, image processing, VLSI, and CAD/CAM. To effectively manage spatial data, a spatial database must efficiently process queries on spatial data. Spatial joins are one such query function that combines objects from two data sets based on a spatial predicate such as *intersect* or *contain*.

Queries **Q1**: "*Find all parks which are in a city.*" and **Q2**: "*Find all trails that go through some forest.*" are examples of spatial joins. In **Q1**, the two data sets are parks and cities, the spatial predicate is *contain*, whereas in **Q2**, the two data sets are trails and forests, and the spatial predicate is *intersect*.

Spatial joins are very expensive in terms of both CPU and I/O because (1) spatial join operations require multiple scans of the data sets, (2) spatial objects are typically represented by structures that require extensive storage, and (3) resolving spatial predicates between two objects requires super-linear time complexity. The first two factors contribute to high I/O costs whereas the third one results in high CPU costs. As a result, spatial join queries over large data sets usually incur a long response time.

This paper now presents a new method in spatial joins whose performance is a substantial improvement (up to 50%) over the state-of-the-art approach. Like other spatial join techniques [3, 9], our method is based on the existence of R-tree indexes created for the two target data sets. Using R-tree indexes for spatial join processing is very useful because many spatial data sets are large, therefore they require spatial indexes for query optimization. Furthermore, in recent years, the R-tree and its variants have become one of the most popular spatial access methods. Examples of spatial database systems which use R-trees are the Illustra database [20], Intergraph's GIS databases [8], and Postgres [19].

An important advantage of R-tree based spatial joins is that the join process traverses both R-tree hierarchies such that subtrees from both R-trees are only explored further if the Minimum Bounding Rectangles (MBRs) of their root nodes satisfy the spatial predicate. Thus, spatial join efficiency is improved because many of the potential false hits can be discarded early in the join process. To optimize spatial joins without using existing indexes, one typically requires special-purpose data structures in order to detect and reduce false hits. For example, [9] constructs seeded trees; [13] performs spatial partitioning; and [10] uses hash tables. While these techniques are important when no spatial index exists for the target data sets, they are not the method of choice when such indexes exist. This is because spatial joins based on existing indexes require no extra data access structures and typically have a superior performance [13].

While previous techniques on R-tree spatial joins [3, 9] follow a depth-first order for traversing the two input R-trees, we demonstrate in this paper that a spatial join technique based on a breadth-first ordering approach offers new unique opportunities for optimization and thus results

in significant performance improvements beyond previous solutions. This new technique, which we call **Breadth First R-tree Join** (**BFRJ**), traverses both R-trees synchronously and processes join computation level by level. The **BFRJ** then exploits the intermediate join results created at a given level, called the *intermediate join index* (IJI), in order to make informed decisions as to which two nodes from the two R-trees respectively are to be joined at the next lower level. This is in contrast to the state-of-the-art R-tree spatial join technique [3] which using depth-first ordering has the inherent limitation that optimization can only be achieved *locally* because the access pattern for nodes beyond the current scope (i.e., local sub-trees) is not captured. The IJIs generated by the **BFRJ** instead capture more *global* information such as the order of accesses for *all* nodes at a certain level and the number of times each of these nodes will be accessed. This enables the **BFRJ** to apply global optimization strategies for effectively managing these IJIs. In particular, in this paper, we propose three such global optimization dimensions that include the IJI ordering optimization, IJI memory management optimization, and the buffer management optimization.

The IJI ordering optimization dimension incorporates strategies for ordering each IJI such that page faults are minimized during join computation at the next level. The IJI memory management optimization determines the proper means of storage (main memory buffer or secondary storage) for IJIs based on various buffer sizes. The buffer management optimization adjusts the buffer paging behavior exploiting knowledge available in the IJI about which pages are more likely to be accessed in the (near) future. Although managing IJIs incurs overhead such as computation and storage costs, we will demonstrate in our experimental studies that these costs are small compared to the performance gain achieved by the global optimizations.

Because an analytical investigation of R-tree based spatial join is very difficult [3], it remains an open issue to date. Therefore, our performance studies of **BFRJ**, like other spatial join research in the literature, are based on an experimental evaluation. We experiment with contrasting the respective impacts of alternative solutions for each of the three global optimization dimensions for **BFRJ**, as well as comparing the performance of **BFRJ** with the state-of-the-art techniques in R-tree joins [3]. Our experimental evaluation shows that, with the proper selection of options in global optimizations, **BFRJ** consistently outperforms the competitor. Its performance gain over the competitor is particularly significant (50%) when a medium or large buffer space is available for the spatial join task. This is important because modern databases tend to have a large system buffer so that compute-intensive tasks such as spatial joins are likely to have access to at least a medium-sized buffer space. **BFRJ** therefore fits modern databases better because it improves spatial join performance by deploying global optimizations that take advantage of a larger buffer allocation. Although the extreme case where a very small buffer ($<$ 400 KBytes) is used for a spatial join task is not practical, **BFRJ** still moderately outperforms the competitor in this case.

The rest of the paper is organized as follows. Section 2 provides the background on spatial joins. Section 3 introduces the framework of **BFRJ**, followed by Section 4 where **BFRJ** global optimizations are proposed. We present our experimental results in Section 5 and conclude the

paper in Section 6.

2 Background on Spatial Joins

2.1 Related Work

There are many recent research efforts reported in the literature that focus on spatial join processing. In [12], the z-ordering technique is used to transform multi-dimensional data into the 1-dimensional domain. Spatial join is then conducted on the B^+-tree structures that store z-ordering values of the spatial data. In [15], spatial join indexes are computed using Grid files [11] to index the spatial data. In [6], a model of the generalization tree is proposed to compare the tree-based spatial joins with the alternative approaches using cost estimation. Spatial joins based on depth-first traversal of R-trees were proposed in [3]. Their techniques exploit the R-tree hierarchy by synchronously traversing subtrees from both R-trees only if the MBRs of the subtrees' root nodes overlap. A variety of CPU and I/O optimizations are also presented in [3]. To this date, this R-tree join [3] has become the state-of-the-art approach for spatial joins when R-tree indexes exist for both spatial data sets. Its performance has also become the yardstick used by other researchers to measure the performance of their proposed non-index based spatial join methods [9, 10, 13].

More recently, spatial join research has focused on joining spatial data when the associated spatial indexes do not exist for both data sets. In [9], a seeded tree is constructed for the data set without index in order to join it with the R-tree of the other data set. When indexes do not exist for both data sets, a spatial hash join is proposed in [10] that uses spatial partitioning as the hash function. A similar partition-based spatial-merge join is proposed in [13].

2.2 The R-tree Structure

R-trees [7] are an extension of B-trees [1] that store multi-dimensional data. Like B-trees, R-trees are balanced and dynamically adjustable. Unlike B-trees, a non-leaf node in an R-tree contains entries of the form $< addr, mbr >$ where $addr$ is the address of a child node and mbr is the MBR that encloses MBRs of all entries in that child node. A leaf node contains entries of the form $< oid, mbr >$ where oid refers to a spatial object stored in the database and mbr is the MBR of that spatial object.

R-trees are dynamically balanced by queries such as *insert* or *delete*. Therefore, no periodic reorganization is necessary. In most R-tree variants, however, entry MBRs are allowed to overlap one another [2, 7, 5]. This means that there may not be only one search path as in the case of B-trees. To improve this weakness, recently proposed R-tree variants tried to minimize the overlap between the entry MBRs. Among them, R*-tree [2] introduces heuristics that yields a better query performance. In [5], R-trees are constructed in a bottom-up approach called the packed R-tree based on the Hilbert curve transformation. As a result, the node occupancy rate is maximized whereas the overlap between entry MBRs is minimized. We exploit these previous results in this

paper by basing our performance studies on spatial joins using packed R-trees.

2.3 The Notations

For brevity, we denote the two R-trees used for spatial joins as R and S. Below, we present the notations that describe R. Applying the notations to S is straightforward.

- $|R|$ is the number of spatial objects indexed by R
- hR is the height (number of levels) of R.
- lR^i is the number of nodes at level i of R, where $0 \leq i < hR$. Note that $lR^0 = 1$.
- nR_i^l is the i-th tree node at level l of R, where $0 \leq l < hR$ and $0 \leq i < lR^l$. Since there is only one root node at level 0, we use nR^0 to denote the root node of R.
- eR_i^l is the number of entries in the tree node nR_i^l.
- $< oidR_j^l, mbrR_j^l >_i$ is the i-th entry in the tree node nR_j^l, where $0 \leq l < hR$, $0 \leq j < lR^l$, $0 \leq i < eR_j^l$, $oidR_j^l$ is the *addr* (for non-leaf nodes) or *oid* (for leaf nodes) and $mbrR_j^l$ is the *mbr* of this entry.

In this paper, the spatial join process pertains to the MBR-spatial joins and the spatial predicate is *overlap*[1]. The result of the MBR-spatial join, called the **candidate set**, is a set of 2-tuples $< oidR^{hR-1}, oidS^{hS-1} >$ where $oidR^{hR-1}$ and $oidS^{hS-1}$ are the spatial object IDs from the leaf nodes of R and S respectively such that their associated MBRs overlap each other. The MBR-spatial join process is called the filter step. To complete the spatial join process, a spatial intersect algorithm [14, 18] is then applied to each item in the **candidate set** to determine if the two objects really overlap. This process is called the refinement step.

2.4 Local Optimizations for R-tree Based Spatial Joins

In R-tree based spatial joins, such as the techniques proposed in this paper and in [3], an important atomic operation is to retrieve two nodes, one from each R-tree, and join the entry MBRs between the two nodes. We call this process node-pair join computation. Let nR_i^r and nS_j^s ($0 \leq r < hR$ and $0 \leq s < hS$) be the two nodes retrieved from R and S respectively. The node-pair join computation between nR_i^r and nS_j^s computes a set of ID pairs $< oidR^r, oidS^s >$ such that their associated MBRs, $mbrR^r$ and $mbrS^s$, overlap. The naive approach in node-pair join computation is to check all entry MBRs in one node for each entry MBR in the other node. Such a nested-loop approach demands a high CPU computation complexity $O(n \times m)$ where $n = eR_i^r$ and $m = eS_j^s$. Local optimizations pertain to the techniques that improve the CPU cost for node-pair join computation. In the following, we describe two local optimization techniques presented in [3], namely *restricting the search space* and *plane sweep*, that are incorporated by **BFRJ**.

[1] We use *overlap* and *intersect* interchangeably.

2.4.1 Restricting the Search Space

During join computation for a node-pair nR_i^r and nS_j^s, the intersecting area between the MBRs of the two nodes can be easily computed[2]. The intersected area itself is an MBR, called intersect-MBR. If an entry $< oidR_i^r, mbrR_i^r >_x$ in nR_i^r overlaps an entry $< oidS_j^s, mbrS_j^s >_y$ in nS_j^s, it must be true that both $mbrR_i^r$ and $mbrS_j^s$ intersect the intersect-MBR between nR_i^r and nS_j^s. Based on this observation, we can scan all entries in nR_i^r and nS_j^s once to discard the entries whose MBRs do not overlap the intersect-MBR between the two nodes. This is called *restricting the search space* [3]. The CPU time complexity for *restricting the search space* is $O(n+m)$. The actual join computation takes only the selected entries as input. Let n' and m' be the numbers of entry respectively in nR_i^r and nS_j^s that overlap the intersect-MBR between the two nodes. The CPU time complexity for the nested-loop join computation with *restricting the search space* becomes $O(n' \times m')$. The CPU complexity for the entire node-pair join operation therefore is $O(n+m)+O(n' \times m')$. With $n' < n$ and $m' < m$ being very likely, CPU computation time improvement is expected.

2.4.2 Plane Sweep

Plane sweep optimization is similar to the sort-merge join technique used to join two simple data sets. Sort-merge join first sorts the two data sets, then conducts join computation by sequentially scanning both data sets simultaneously. Because the two data sets are sorted, only a single scan of both data sets is required for join computation. Therefore, sort-merge join is an improvement over the nested-loop join.

Similarly, during the *plane sweep* optimization of a node-pair join computation between nR_i^r and nS_j^s, we first sort the MBR entries in the two nodes respectively. To sort multi-dimensional data, we use the low x-coordinate value of each MBR as the key. In the merge process, we scan the two sorted entries of MBRs sequentially based on their ordered key values. For each MBR (say mbr_i) evaluated in the merge process, we only conduct intersect tests against the MBRs from the opposite entry which overlap mbr_i based on their x-coordinate values. The term *plane sweep* pertains to the merge process in which a vertical line can be imagined to sweep from mbr_i's low x-coordinate value to its high x-coordinate value in order to find the potential intersecting MBRs from the other entry.

The *restricting the search space* and *plane sweep* techniques incorporated by **BFRJ** are referred to as local optimizations because they improve the computation efficiency within each node-pair join process. In Section 4, we introduce three novel techniques that can further optimize **BFRJ** by exploiting the inter-relation between the node-pair join computations. We call them global optimizations.

[2]The enclosing MBR for a R-tree node k can be passed from k's parent node or computed by one scan of k's entries.

3 Spatial Joins Based on Breadth-First Traversal of R-Trees

In this section we present the framework of the proposed Breadth-First R-tree Join (**BFRJ**). We start by assuming that R and S are of the same height, and relax this restriction in Section 3.3.

3.1 Search Pruning by Traversing R-Trees

An R-tree can be viewed as multiple levels of MBRs such that MBRs at each level partition[3] the entire data space. The MBRs of the higher level nodes in an R-tree form more coarse-grained partitions whereas those of the lower level nodes form more fine-grained partitions. The spatial join between R and S corresponds to joining entries in R's leaf nodes with those in S's leaf nodes.

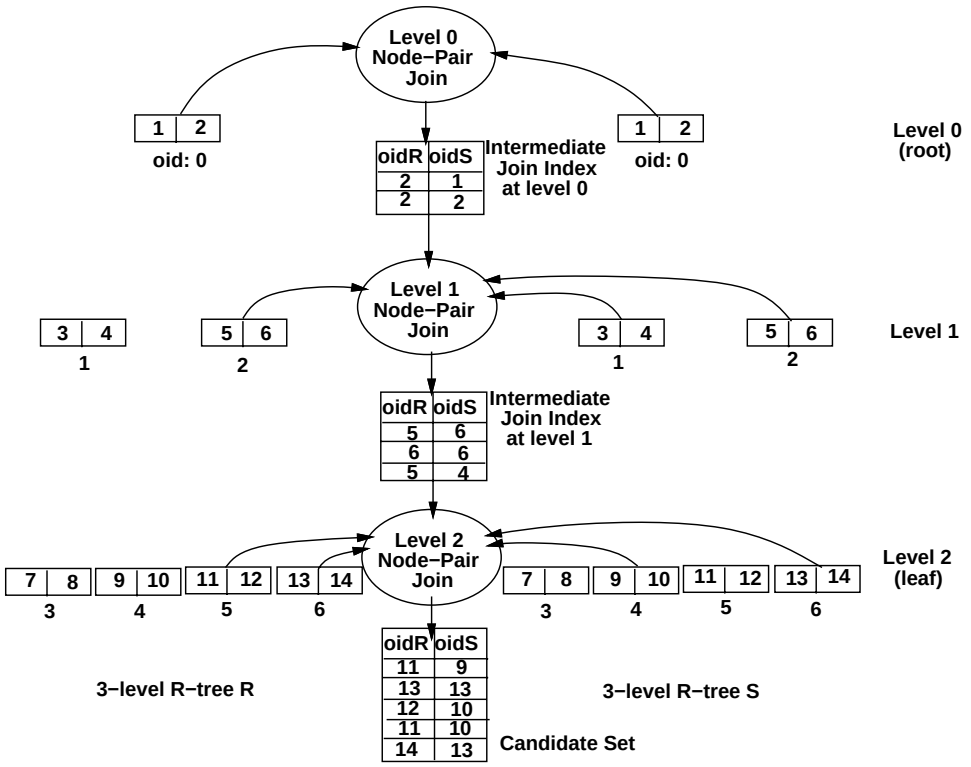

Figure 1: Breadth-First R-tree Join (BFRJ) on R-trees of the Same Height.

One important information captured in an R-tree is that its hierarchy manifests the *enclose* relation, i.e., the MBR of a tree node always encloses the MBRs of its descendant nodes. To take advantage of this property, pair-wise join computation between two nodes, nR_i^r and nS_j^s, is only needed when the MBR of nR_i^r's parent node overlaps that of nS_j^s's parent node. We call this *search pruning*. Simple top-down graph-traversal algorithms can be used to achieve *search pruning* at all levels. In [3], *search pruning* is done by synchronously traversing the two input R-trees depth-first

[3] In most R-trees variances [2, 7, 5], partitions at each level may overlap. An exception is R$^+$-tree [17], for which partitions at each level do not overlap. **BFRJ** is independent of this variance.

whereas in **BFRJ** it is achieved by synchronized breadth-first traversal of both R-trees. The effect of *search pruning* at all R-tree levels is that, starting from the top level, the two nodes, one from each R-tree, are only traversed for join computation if the MBRs of their parent nodes overlap. Thus, the number of node-pair traversals is reduced by *search pruning* comparing with the nested-loop approach.

The BFRJ Framework. The **BFRJ** first joins the entries in R's root node (nR^0) with those in S's root node (nS^0) (see Figure 1). The join results are a set of 2-tuples $< oidR^0, oidS^0 >$ called *intermediate join index* at level 0, or IJI_0. Because we focus on spatial *overlap* join in this paper, each tuple $< oidR^0, oidS^0 >$ specifies that the MBRs of the two elements overlap. Next, for each tuple in IJI_0, **BFRJ** retrieves the two nodes referenced by the tuple items from R and S respectively. It then conducts spatial node-pair join between the entries from the two nodes. While **BFRJ** reads tuples of IJI_0 for join computation, it stores the join results, also in the form of 2-tuple $< oidR^1, oidS^1 >$, to the current *intermediate join index* at level 1, or IJI_1. When it completes join computation for all tuples in IJI_0, it discards IJI_0 and proceeds to process the tuples in IJI_1 for join computation. This process continues as **BFRJ** traverses down the two R-trees synchronously level by level. It terminates when the *intermediate join index* is created by joining the leaf entries in R with the leaf entries in S. At this point, the **filter** step of the spatial join process is completed and the current (leaf-level) *intermediate join index* is the output of the spatial join process.

Figure 1 depicts the join process of **BFRJ** between two R-trees with the same height. Note that nodes 1, 3, 4 from R and nodes 3, 5 from S are never read from disk because the *search pruning optimization* determines that these nodes are not needed for join computation.

3.2 BFRJ on R-Trees with the Same Height

We now give the algorithm (Figure 2) that conducts an R-tree spatial join based on the **BFRJ** framework described previously. We call it *BFRJ_Same_Height* because we assume that the heights of the two input R-trees are the same ($hR = hS$).

The $Node\_Pair\_Join\_Same\_Level()$ procedure (lines 1 and 5 in Figure 2) takes two nodes from the two input R-trees respectively and conducts a spatial join between the entries in the two nodes. We assume the node-pair join process in Figure 2 deploys the local optimizations described in Section 2.4.

3.3 BFRJ on R-Trees with Different Heights

The *BFRJ* algorithm that assumes the two input R-trees are of different heights is illustrated in Figure 3. Let $hR < hS$, the *BFRJ* algorithm behaves exactly the same as the *BFRJ_Same_Height* algorithm before it reaches level $hR - 1$ (R's leaf level). After it reaches level $hR - 1$ while traversing R, the *BFRJ* algorithm stays at R's leaf level but proceeds to traverse S downwards level by level until S's leaf level is reached.

The $Node\_Pair\_Join()$ procedure (lines 1 and 5 in Figure 3) behaves slightly differently from the

```
PROCEDURE BFRJ_Same_Height (R, S)
// R, S are two R-trees, hR = hS
DATA STRUCTURES: set IJI[hR] := ∅;
// IJI[i] is the intermediate join indexes created at level i
01  IJI[0] := Node_Pair_Join_Same_Level(nR^0, nS^0); // join the two root nodes
02  integer i := 0;
03  while i < hR − 1 do
04      ∀ < oidR^i, oidS^i > ∈ IJI[i] do
05          IJI[i + 1] = IJI[i + 1] ∪ Node_Pair_Join_Same_Level(oidR^i, oidS^i);
06      end do
07      i := i + 1; //down one level
08  end while
09  output IJI[i]; // IJI[i] is the output
```

Figure 2: The *BFRJ* Algorithm with Input R-trees of the Same Height.

$Node\_Pair\_Join\_Same\_Height()$ procedure in the $BFRJ\_Same\_Height$ algorithm. While joining two nodes nR_i^r and nR_j^s, where $0 \leq r < hR$ and $0 \leq s < hS$, the $Node\_Pair\_Join()$ procedure checks to see if either node is a leaf node. Suppose nR_i^r is a leaf node ($r = hR - 1$) and nR_j^s is not ($s < hS - 1$), the $Node\_Pair\_Join()$ procedure uses the enclosing MBR of node nR_i^r and scans through all nS_j^s's entry MBRs to conduct the MBR overlap test. The result is a list of 2-tuples $< oidR^r, oidS^s >$ where $oidR^r$ is the ID of node nR_i^r and $oidS^s$ stands for various entry IDs in nS_j^s whose MBRs overlap nR_i^r's MBR. When the two input nodes nR_i^r and nS_j^s are both leaf nodes or both non-leaf nodes, the $Node\_Pair\_Join()$ procedure behaves exactly the same as the $Node\_Pair\_Join\_Same\_Height()$ procedure, which is to conduct overlap join between all entries in nR_i^r and all entries in nS_j^s.

4 Global Optimizations of BFRJ

This section investigates how spatial join based on the **BFRJ** framework offers unique opportunities for global optimizations. Note that in the **BFRJ** framework, an *intermediate join index* at level i (IJI_i) is created after all R nodes at level i are joined with all S nodes at level i. The selection of an R node and an S node for node-pair join computation at level i can now be based on IJI_{i-1} which was generated at the previous higher level (level $i-1$). We thus have global information at our avail about all anticipated accesses of nodes at a given level (including their possible order of access as well as the number of times each node gets re-accessed) before processing joins at that level. This naturally lends itself to the application of alternative techniques for the effective management of the *intermediate join indexes*. In this section, we investigate alternative design decisions on three different IJI optimization dimensions: IJI ordering, IJI memory management, and IJI-related buffer management. We assume $hR = hS$ in the following sections. Applying the global optimization to cases when $hR \neq hS$ is straightforward.

```
PROCEDURE BFRJ (R, S)
DATA STRUCTURES: set IJI[max(hR, hS)] := ∅;
01  IJI[0] := Node_Pair_Join(nR^0.nS^0); // join the two root nodes
02  integer i := r := s := 0;
03  while r < hR − 1 or s < hS − 1 do
04      ∀ < oidR^r, oidS^s > ∈ IJI[i] do
05          IJI[i + 1] = IJI[i + 1] ∪ Node_Pair_Join(oidR^r, oidS^s);
06      end do
07      if r :≠ hR − 1 then
08          r := r + 1; //down one level if not yet leaf-level
09      end if
10      if s :≠ hS − 1 then
11          s := s + 1; //down one level if not yet leaf-level
12      end if
13      i := i + 1;
14  end while
15  output IJI[i]; // IJI[i] is the output
```

Figure 3: The *BFRJ* Algorithm.

4.1 Ordering of Intermediate Join Indexes

Suppose the MBR of an R node nR_i^l intersects the MBRs of k different l-level S nodes, where $k > 1$. Then the ID of nR_i^l will appear k times in IJI_{l-1}. This means that, during the join computation at level l, nR_i^l will participate in the node-pair join computation exactly k times. With a fixed-sized LRU system buffer, node nR_i^l may be read from a disk multiple (up to k) times if the k appearances of its ID are widely scattered in IJI_{l-1}. This is because the initial and subsequent retrievals of nR_i^l may be too far apart, and nR_i^l may already be paged out by the time it is needed again. Therefore, we propose that the IJIs be kept in an order so that no multiple appearances of the same node ID are spread too widely in the *intermediate join indexes*.

However, each tuple $< oidR, oidS >$ in IJIs has two items that need to be fetched from the secondary storage. Clustering by one obviously does not assure a good clustering for the other. Consequently, an effective ordering may need to take into account both items of the index tuples in order to achieve better global optimization. We investigate the following ordering options[4]:

Option 1: No particular ordering (OrdNon). OrdNon does not perform global ordering for the *intermediate join indexes*, therefore it incurs no ordering cost. The *intermediate join index* created at each level however is not truly randomly ordered because the *plane sweep* local optimization partially orders the entries within each node. Therefore, there may exist many regional orderings in each IJI. However, because an MBR from one R-tree may overlap more than one MBR from the other R-tree, its corresponding entry ID may exist in several locally ordered regions in the IJI. Therefore, OrdNon is not expected to contribute to a good global ordering.

[4]We now ignore specifying the levels since ordering optimization applies to IJIs at all levels.

Option 2: Ordering by items from one tree (OrdOne). OrdOne sorts the *intermediate join indexes* by the lx's of items from one tree. Because each IJI tuple is composed of two items $< oidR, oidS >$, one from each R-tree, ordering based on items from one tree, say $oidR$, creates a perfect clustering for $oidR$ while ignoring the clustering of $oidS$.

Option 3: Ordering by the sum of the centers (OrdSum). For each tuple $< oidR, oidS >$ in IJI, OrdSum first calculates the center x coordinate values of the MBRs for $oidR$ and $oidS$, namely:

$$CX_{oidR} = (lx_{oidR} + hx_{oidR})/2.$$
$$CX_{oidS} = (lx_{oidS} + hx_{oidS})/2.$$

OrdSum then sorts the IJI based on the sum of CX_{oidR} and CX_{oidS}. Therefore,

$$sortkey = (lx_{oidR} + hx_{oidR})/2 + (lx_{oidS} + hx_{oidS})/2.$$

Option 4: Ordering by center point (OrdCen). OrdCen creates an enclosing MBR by combining $oidR$'s MBR with $oidS$'s MBR. It then sorts the IJI based on the x coordinate values of the center point of the enclosing MBRs. Namely,

$$sortkey = (lx_{min} + hx_{max})/2,$$

where lx_{min} is the smaller lx and hx_{max} is the larger hx between $oidR$'s MBR and $oidS$'s MBR.

Option 5: Ordering by Hilbert curve value of the center (OrdHil). OrdHil is similar to OrdCen in that it sorts the IJI based on the x-coordinate values of the center point of the enclosing MBRs. Instead of using the x coordinate values, OrdHil calculates a Hilbert curve value for each center point, and sorts the IJI by the Hilbert curve values.

4.2 Memory Management of Intermediate Join Indexes

The most efficient way of ordering the *intermediate join indexes* is to sort them in main memory. This is only possible if the largest IJI fits into the main memory buffer allocated to the spatial join task. Because **BFRJ** is based on tree-structured indexes, the largest IJI is the one created at the lowest level before the final join output is computed. Let IJI^{max} be the largest IJI created by **BFRJ** in spatial join computation on R and S. IJI^{max} is then computed by joining the entries in nR^{hR-2} with those in nS^{hS-2}. Let k be the average number of nodes at level $hS-2$ in S whose MBRs overlap that of a node at level $hR-2$ in R, o the average node occupancy rate of R, and m the maximum number of entries an R node can hold. The size of IJI^{max} can be approximately estimated as:

$$|\text{IJI}^{max}| = (|R| \times k)/(o \times m).$$

In modern databases, typically $50\% \leq o \leq 100\%$ and $50 \leq m \leq 800$. For example, our test data[5], $|R| = 131,461$, $|S| = 128,971$, $m = 203$, o is 100% for the packed R-tree, and k is

[5]The test data are derived from the TIGER/Line files [4] that represent the streets, rivers, and rails of an area in California.

approximately 12 for the packed R-tree[6]. With each join index tuple being 12 bytes long, the size of IJI^{max} in our test data is less than 100 KBytes, and therefore fits into a buffer of moderate size. The sizes of the smaller IJIs at the higher levels are significantly smaller than $|IJI^{max}|$, therefore are negligible. If $|IJI^{max}|$ is larger than the available buffer size, IJI^{max} must be stored on disk, and a disk-based sort such as merge-sort can be used to reorder the index. In addition to the more costly sorting process, the disk-based approach has an overhead of moving the IJIs between the main memory buffer and disk. If $|IJI^{max}|$ is smaller than the buffer size, it may reside on disk or in main memory. While the main memory solution is more efficient in sorting IJIs, it requires buffer space to store the IJIs, hence has less buffer pages for join computation.

StorDisk: Storing indexes on disks. In the StorDisk approach, the *intermediate join indexes* are stored on disk. During the join computation, only one buffer page needs to be reserved for them as they can be written out sequentially. All other buffer pages can be dedicated to join computation. Sorting the indexes happens after the indexes at one level are completely written and before join computation starts at the next level. This means the entire buffer space can be dedicated to the sorting process. During sorting, the *intermediate join indexes* only need to be read once if they fit into the buffer, or more than once[7] if merge-sort is required for a smaller buffer. After sorting, the join computation at the next level can then start based on the ordered indexes. Note that these sorted indexes need not be removed from main memory on purpose during join computation because the LRU paging mechanism will automatically expel the least recently used pages.

StorMem: Storing indexes in main memory. StorMem keeps the *intermediate join indexes* at the current level in main memory ($|IJI^{max}|$ must be smaller than the buffer size). This way, join computation has less buffer pages available, but the indexes do not need to be shuffled between disk and memory. During join computation, a special purge technique can be used to remove a index page from the active buffer to the free page list if all index tuples in this page have been processed for join computation. This technique creates more room for join computation as more index tuples are being processed.

4.3 Buffer Management of Intermediate Join Indexes

The ordering optimization (Section 4.1) attempts to keep the *join indexes* in an order such that no two appearances of the same ID are spread out too widely. However, since a perfect clustering is not possible for both tuple items, multiple disk reads for a tree node may still happen during join computation. Such multiple reads can be further minimized if the buffer manager can predict which nodes have completed their join computation and which ones are to be fetched again in the future. This way, the buffer manager may retain the node pages to be accessed in the future in main memory and purge node pages that have completed their join computation from main memory.

[6] k can be approximated by sampling on both R-trees.

[7] In our experiments, the merge-sort process reads and writes the *intermediate join indexes* once for partial sorting, and reads the partially sorted indexes once for merging.

To accomplish such an optimization, we assume the buffer manager supports three buffer operations: *pin*, *unpin* and *purge*. The *pin* marks a page in the LRU buffer so that this page is retained in memory until it is *unpinned*. The *unpin* simply removes the *pin* marker. The *purge* removes a page from the LRU and inserts it into the free list of pages that are available for use during page faults.

To predict which tree nodes are to be accessed, a counter for each node in both R-trees is kept. During the generation of each *intermediate join index*, each appearance of a tree node nR_i^r increases its counter by 1. Therefore a counter corresponds to the number of appearances of its tree node in IJI. After the join computation between a node-pair, say nR_i^r and nS_j^s, is complete, their counters are both decremented by 1. If a counter reaches 0, it means that the tree node associated with this counter no longer appears in the remainder of the current IJI and will no longer be needed in the spatial join processing. Therefore, such a tree node can be *unpinned* if it has been *pinned*, and *purged* so that its page can be used immediately. If a counter remains above 0, its tree node will be accessed again in the future. The buffer manager can then keep the page of this tree node *pinned* until its counter reaches 0 later on.

Note that if the size of the buffer is small or the *intermediate join index* is not in an adequate order, it is possible that all buffer pages are *pinned* during join computation. In this case, we assume that the buffer manager handles this situation by *unpinning* the least recently used page in order to free up the buffer space.

4.4 BFRJ versus Spatial Join Based on Depth-First Traversal of R-Trees

Because the depth-first approach [3] goes not keep any global information (such as the *intermediate join indexes*), it requires no additional data structures to store the IJIs. However, the depth-first approach does not have the ability to achieve global optimization by doing global ordering or global paging prediction. This is because the order by which each node-pair is to be joined is determined by the recursive depth-first sequence that cannot be globally changed. Another major difference between **BFRJ** and the depth-first approach is that **BFRJ** never traverses upwards in an R-tree while the depth-first approach traverses upwards as part of function returns of the recursive routines. Therefore, redundant disk access of the same page may happen to **BFRJ** while processing joins at the same level if the ordering of the *intermediate join indexes* is not optimized, whereas it may happen to the depth-first approach during backtracking.

5 Experimental Results

Our performance studies are based on the experiments conducted on a testbed implemented in C++ on a SUN Sparc-20 workstation running the UNIX operating system. The testbed includes the **BFRJ** with all optimizations introduced in this paper, the spatial join techniques proposed in [3], an I/O buffer manager, and many other supporting data structures and procedures. We

use real-world test data that consists of a data set of streets (131,461 objects) and a data set of rivers and railway tracts (128,971 objects) from an area in California. The data is derived from the TIGER/Line files distributed by the US Census Bureau [4]. We created two Hilbert curve packed R-trees [5], each for a data set, with the page size set to 4 KBytes.

5.1 Intermediate Join Index Ordering Optimizations

The first set of experiments determine which index ordering optimization has the best performance. These schemes include OrdNon, OrdOne, OrdSum, OrdCen, and OrdHil (Section 4.1). We conduct **BFRJ** spatial join on the two packed R-trees for each index ordering optimization by varying the buffer sizes from 100 KBytes to 1,200 KBytes. We fix the other global optimization options to StorDisk and PinNo, meaning we store the *intermediate join indexes* on disk and we do not deploy the *pinning* optimization (Section 4.3).

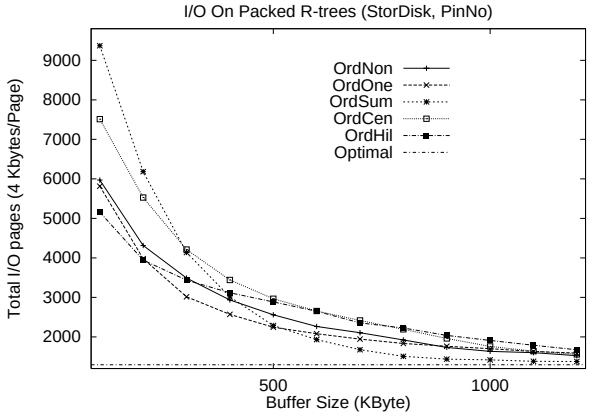

Figure 4: Ordering Optimization I/O Cost.

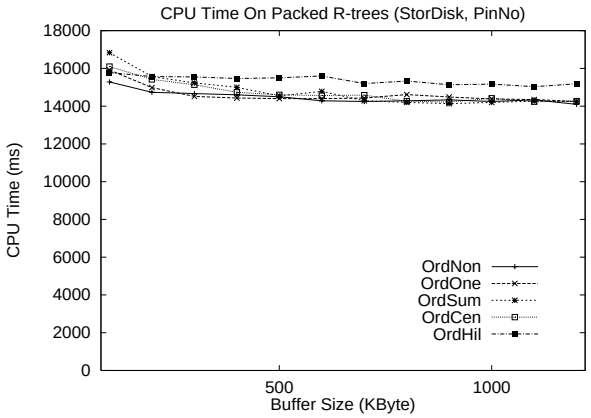

Figure 5: Ordering Optimization CPU Cost.

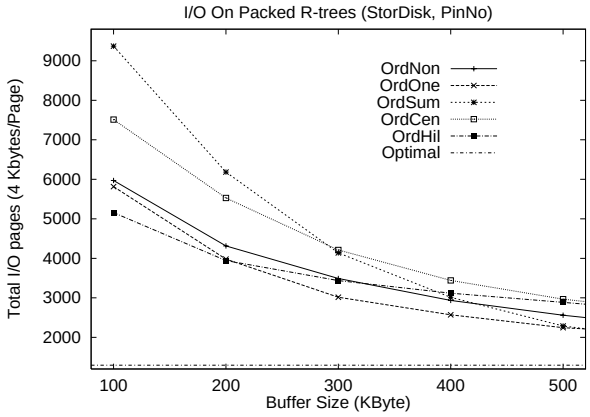

Figure 6: I/O Cost on Ordering Optimization (Small Buffers).

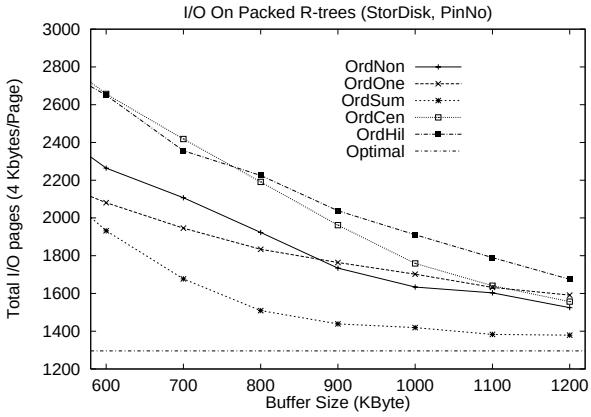

Figure 7: I/O Cost on Ordering Optimization (Large Buffers).

Figure 4 shows the I/O results and Figure 5 the CPU results of all IJI ordering optimizations.

Note that the horizontal line that is marked optimal in Figure 4 and other figures represents the theoretical lower bound of page I/O based on the two packed R-trees[8] used for testing. Because the I/O costs with smaller buffers in Figure 4 are very high, the performance difference with larger buffers cannot be clearly seen. For clarity, we separate the I/O results into two charts, namely Figure 6 representing the results for small buffer sizes and Figure 7 the results for larger buffers.

Figure 5 shows that the differences in CPU cost among OrdNon, OrdOne, OrdSum, and OrdCen are negligible, whereas the CPU cost of OrdHil is consistently higher than others. This is because computing the Hilbert curve values requires additional CPU time, making OrdHil the most CPU expensive option. The results in Figure 6 show that OrdOne outperforms all other alternatives in I/O for all buffer sizes (100 KBytes - 500 KBytes) except for the case of buffer size 100 KBytes where OrdOne is second to OrdHil. We believe processing spatial joins with only an available buffer size of 100 KBytes is an extreme case, given that modern databases tend to have a large system buffer. Therefore, when the buffer size is moderate ($\leq$ 500 KBytes), OrdOne is the best choice in IJI ordering optimization for processing **BFRJ**. Although the I/O cost for OrdSum is very high when the buffer size is small, it decreases dramatically as the buffer size grows larger. From Figure 7, we can see that OrdSum is the clear winner when a more generously-sized buffer ($\geq$ 600 KBytes) is available. Without the ordering overhead, OrdNon performs better than OrdCen and OrdHil in I/O, but worst than OrdOne for smaller buffers and OrdSum for larger buffers. In conclusion, we believe OrdOne is a good choice when the buffer size is moderate, and OrdSum is the best choice when the buffer size is larger.

The reason why OrdSum outperforms others when the buffer size is large is because it sorts the join indexes by taking the spatial locations (on x-axis) of both index tuple items into account. However, its storage locality spreads wider in order to cover *both* items. Therefore OrdSum has the best performance if a larger buffer that can cover OrdSum's storage locality is available. If the buffer is too small to cover the locality, OrdSum's performance deteriorates dramatically. For smaller buffers, sorting by one item (OrdOne) performs better because its storage locality does not spread as widely as in OrdSum. Given that both OrdCen and OrdHil do not improve over no ordering (OrdNon), the center points of the MBR combined from the MBRs of both items is not relevant in controlling the storage locality for either item of the join index tuples. In the following sections, we continue to investigate the performance of other global optimization options, such as memory and buffer management, with OrdOne and OrdSum as the chosen ordering optimizations.

5.2 Intermediate Join Index Memory Management Optimizations

We conduct experiments to test the performance of the alternatives in memory management of IJIs, namely StorDisk and StorMem. For the StorDisk option, IJIs are stored on disk and only loaded into main memory when necessary, such as during sorting or join computation. The StorMem

[8]Because our two test data are evenly distributed in the same area, the optimal lower bound is equal to the sum of the number of tree nodes in both R-trees.

option, on the other hand, keeps the current IJI in main memory at all time while it remains current. Because the estimated size of the largest *intermediate join index* is about 100 KBytes for our test data (See Section 4.2), we test the memory management options by ranging buffer sizes starting from 200 KBytes to 1,200 KBytes. Based on the previous experimental results, we select OrdOne and OrdSum as the ordering optimizations.

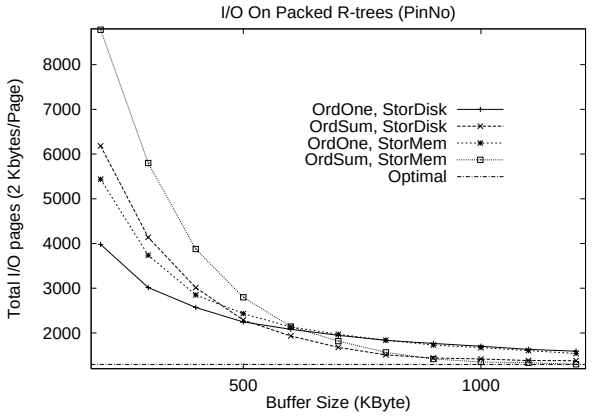

Figure 8: Memory Management I/O Cost.

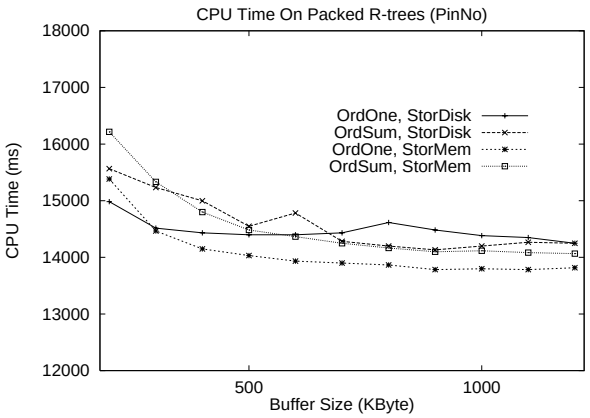

Figure 9: Memory Management CPU Cost.

The results in Figure 8 show that storing the join indexes in main memory (StorMem) has a worse I/O performance than storing them on disk (StorDisk) when the available buffer is small or medium in size ($<$ 900 KBytes). The CPU results in Figure 8 indicate StorMem can improve the CPU usage time over StorDisk, but not very significantly. Figures 10 and 11 provide close-ups of Figure 8 differentiated by buffer sizes. The results in Figure 11 show that StorMem starts to outperform StorDisk in I/O when the buffer size is larger than 800 KBytes. The reason StorMem performs so poorly with smaller buffers is that it needs additional main memory space to store the join indexes. Thus, join computation in StorMem has less buffer pages to work with, thereby creating a buffer contention over a limited number of buffer pages.

We conclude that StorMem does improve the CPU time over StorDisk, but not to a significant degree. Although StorMem outperforms StorDisk in I/O when the buffer sizes are large ($>$ 800 KBytes in Figure 11), its performance on a smaller buffer is much worse than StorDisk. Besides, when the buffer size is smaller than the size of the largest join index ($<$ 100 KBytes), StorMem is not applicable. Therefore, StorDisk is a more viable option with small- or moderate-sized buffers, whereas StorMem become advantageous when a large buffer is available.

5.3 Intermediate Join Index Buffer Management Optimizations

In Section 4.3, we described a buffer management technique (*pinning* optimization) that could further improve the I/O performance for **BFRJ**. We use PinYes to denote that the *pinning* optimization is applied, and PinNo to denote otherwise. Because so far we have identified that OrdOne works the best for smaller buffers and OrdSum has the best performance for larger buffers, we con-

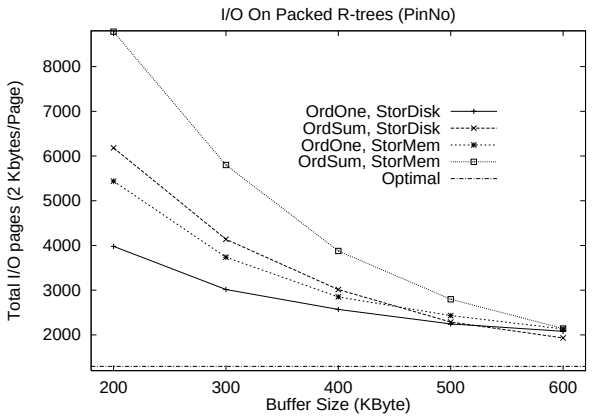

Figure 10: I/O Cost on Memory Management (Small Buffers).

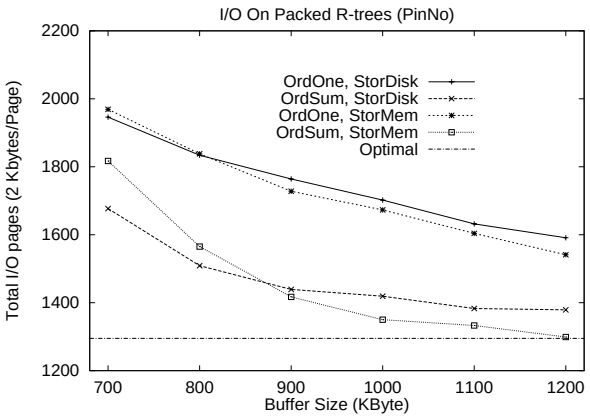

Figure 11: I/O Cost on Memory Management (Large Buffers).

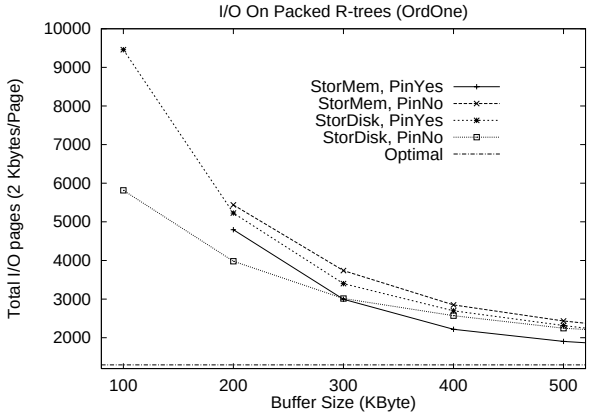

Figure 12: I/O Cost on Buffer Management (Small Buffers).

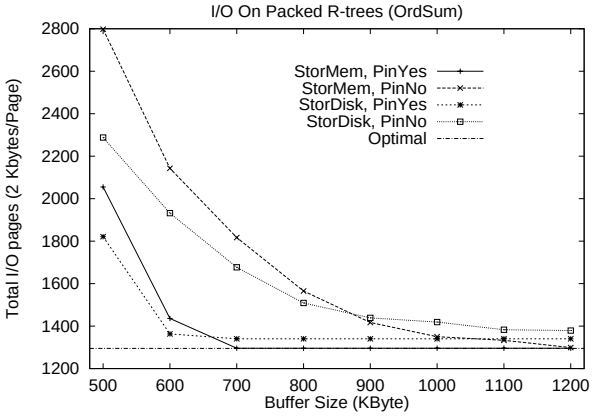

Figure 13: I/O Cost on Buffer Management (Large Buffers).

duct experiments on the *pinning* optimization based on OrdOne with smaller buffers and OrdSum on larger buffers separately. For the first set of experiments (Figure 12), we run **BFRJ** based on OrdOne with both *pinning* optimizations by varying buffer sizes from 100 KBytes to 500 KBytes for StorDisk option, and from 200 KBytes to 500 KBytes for StorMem option. We do not test StorMem with a 100 KBytes buffer because we need about 100 KBytes just to store the *intermediate join indexes* in the main memory buffer. For the second set (Figure 13), we run **BFRJ** based on OrdSum with both *pinning* optimizations and vary the buffer sizes from 600 KBytes to 1,200 KBytes.

The results in Figure 12 show that when the buffer is very small (< 300 KBytes), the combined option of StorDisk and PinNo works the best with OrdOne, although combining StorMem and PinYes outperforms StorDisk+PinNo for OrdOne for a more moderate buffer size (400 KBytes — 500 KBytes). When the buffer sizes are larger, the results in Figure 13 indicate that StorMem+PinYes with OrdSum achieves the optimal performance when the buffer size is greater than 700 KBytes,

and StorDisk+PinYes with OrdSum perform very close to the optimal when the buffer size is greater than 600 KBytes. The reason that the performance of StorDisk+PinYes with OrdSum can only be very close to the optimal is that StorDisk has an overhead of transferring the join indexes between disks and main memory. We did not show the comparison in CPU time because our test results do not show any noticeable difference between PinNo and PinYes options. Therefore, I/O is the dominant factor in determining the performance between PinYes and PinNo .

We conclude that when the buffer size is relatively small, OrdOne+StorDisk+PinNo is the most attractive combination. With a moderate buffer size, the combination of OrdOne+StorMem+PinYes starts to outperform OrdOne+StorDisk+PinNo. When the buffer sizes are larger, the *pinning* optimization is effective for OrdSum as both OrdSum+StorMem+PinYes and OrdSum+StorDisk+PinYes have excellent performance, with OrdSum+StorMem+PinYes slightly better because it does not require any overhead in transferring the IJIs between disk and main memory.

5.4 Comparing BFRJ with the State-of-the-Art R-Tree Join

We believe the state-of-the-art in spatial join methods using existing indexes is the depth-first R-tree join technique with various CPU and I/O optimizations proposed in [3]. Our assumption is based on the evidence that it is the most recently proposed spatial join technique based on existing indexes; it uses R-trees which are deployed by many spatial database products [8, 19, 20]; and most importantly, its performance has become the yardstick used by other researchers to measure the performance of their recently proposed non-index based spatial join methods [9, 10, 13]. We have implemented this method with proper optimizations, and will call it **DFRJ**, which stands for Depth-First R-tree Join.

To compare with **DFRJ**, we choose two combinations of global optimizations in **BFRJ**, namely OrdOne+StorDisk+PinNo (denoted as **Combo1**) and OrdSum+StorMem+PinYes (denoted as **Combo2**). The choice of the two combinations is based on the results of the previous experiments where we concluded that **Combo1** is among the best options for small buffers, and **Combo2** is the best for large buffers. We ran experiments varying buffer sizes from 100 KBytes to 1,200 KBytes for both **Combo1** and **DFRJ**, and from 200 KBytes to 1,200 KBytes for **Combo2**. The **Combo2** option stores IJIs in the main memory buffer and therefore is not applicable when the available buffer is very small. We collected both the I/O and CPU results.

Figure 14 shows the I/O results for all buffer sizes. For clarity, we plot the I/O results in two close-up charts, namely Figures 16 and 17 (discussed later) differentiated by the buffer sizes. The CPU usage results in Figure 15 show that the differences between the three alternatives (**Combo1**, **Combo2**, **DFRJ**) are insignificant, with **DFRJ** having a very slight edge. This is because **DFRJ** does not need to manage the *intermediate join indexes* as required by the **BFRJ** approaches.

As for the I/O cost, **Combo1** consistently outperforms **DFRJ** and **Combo2** when the buffer size is relatively small (Figure 16). Although **Combo2** incurs a higher I/O cost with small buffers ($<$ 400 KBytes), its performance improves dramatically as the buffer size increases. In fact, in

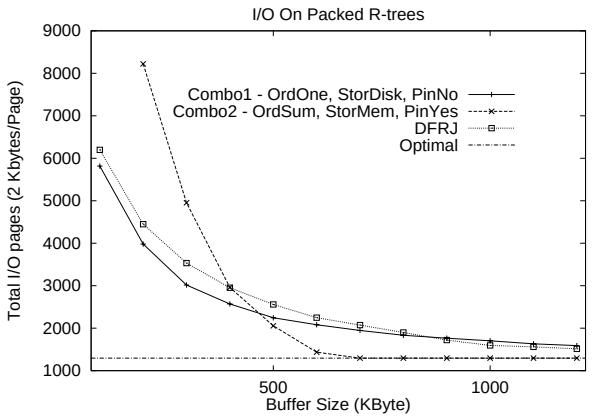

Figure 14: I/O Cost: BFRJ Vs. DFRJ.

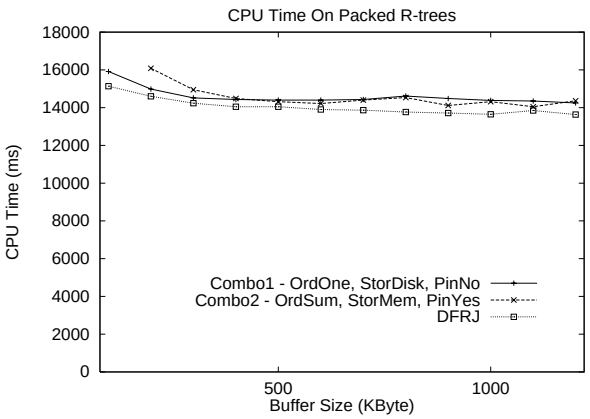

Figure 15: CPU Cost: BFRJ Vs. DFRJ.

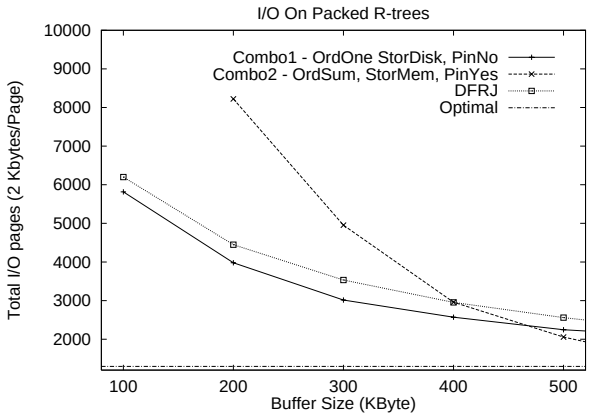

Figure 16: I/O Cost: BFRJ Vs. DFRJ (Small Buffers).

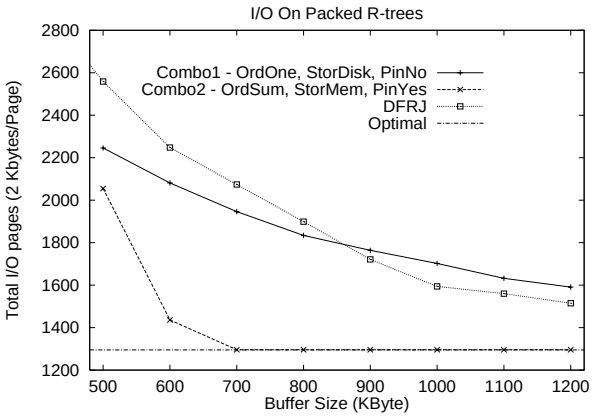

Figure 17: I/O Cost: BFRJ Vs. DFRJ (Large Buffers).

Figure 17, the performance of **Combo2** achieves the optimal performance when the buffer size is greater than 700 KBytes.

From this set of experiments, we conclude that for smaller buffers, **BFRJ**'s **Combo1** has the best I/O performance. For large buffer sizes, **BFRJ**'s **Combo2** performs the best in I/O. Because **Combo1** and **Combo2** do not outperform **DFRJ** in CPU usage time, we in the next section combine the I/O cost and CPU usage cost in order to evaluate the overall performance between **BFRJ** and **DFRJ**.

5.5 Combining CPU Usage Cost and I/O Cost

Our testbed is built on the UNIX operating system which caches file blocks and conducts CPU scheduling independent of our testbed database operations. As a result, the elapse time recorded by UNIX does not serve as a good measurement of query performance. Instead, we use the combination of the CPU usage time and the total I/O access time to measure the overall query performance.

We believe such a measurement is more accurate as it takes both the CPU cost and I/O cost into consideration.

Let t be the total cost, c be the CPU usage time in ms, p be total number of page I/Os incurred during spatial join query computation, and m be the average page access time, then our overall cost formula is as follows:

$$t = c + (m \times p).$$

To compute t, we need to estimate the m value since our experimental results have already yielded the c and p values. In this paper, we assume $m = 10$ ms for each 4-KByte page. In theory, the total page access time is the sum of the seek time, latency time, and transfer time. The 10 ms page access time that we use here is derived from the performance specifications of one class of modern disk drives, namely the Seagate Barracuda 2LP family disk drives [16]. This family of hard drives have an average seek time between 8 and 9 ms, an average latency time of 4.17 ms, and an average transfer time for a 4-KByte page between 0.4 and 0.6 ms. Although the total access time per 4-KByte page exceeds 10 ms, our test data are not very large (both packed R-trees are around 2.7 MBytes), which can be properly clustered to reduce disk seek time. Therefore, we conservatively use 10 ms as our estimated access time per 4-KByte page to compute the overall cost.

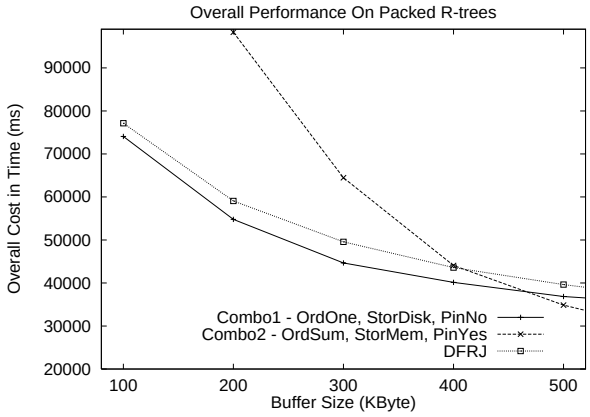

Figure 18: Overall Cost: BFRJ Vs. DFRJ (Small Buffers).

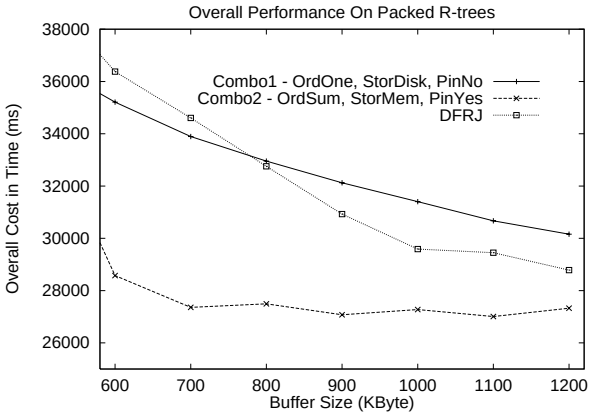

Figure 19: Overall Cost: BFRJ Vs. DFRJ (Large Buffers).

The results in Figure 18 show that **Combo1** has a better overall performance than **DFRJ** and **Combo2** when the buffer sizes are small. When a larger buffer is available, the results in Figure 19 indicate that **Combo2** outperforms both **Combo1** and **DFRJ** by a significant margin (up to 50%). Note that the curves in Figures 18 and 19 look very similar to those in Figures 16 and 17 respectively. This is because the CPU cost difference among the three options is very small, therefore the I/O cost becomes the dominant factor in overall performance.

6 Conclusions

Efficient processing of spatial joins in spatial databases is crucial for many applications such as GIS, CAD/CAM, etc. In this paper, we present a new spatial join method that is based on breadth-first traversal of R-trees. We call it Breadth-First R-tree Join (**BFRJ**). Spatial join using R-trees is very important because it is one of the most efficient spatial join methods when R-tree indexes exist for both data sets. Whereas the state-of-the-art technique in R-tree spatial joins relies on local optimizations for performance improvement, our proposed **BFRJ** is capable of both local and global optimizations. As a result, our experimental evaluation shows that, with the proper selection of options in global optimizations, **BFRJ** consistently outperforms the competitor.

The contributions of this paper are:

1 A new join method, **BFRJ**, is developed for spatial joins based on breadth-first traversal of R-trees that offers unique opportunities for performance optimization.

2 Three dimensions for global optimization are proposed for **BFRJ**, namely, the ordering, memory management, and buffer management optimizations of the *intermediate join indexes*. Alternative solution techniques are identified for each of these three dimensions.

3 Extensive experimental evaluation of the performance of **BFRJ** (using real GIS data sets from the US Census Bureau) is conducted to show the effectiveness of alternative options of the three global optimization techniques based on various buffer sizes. The experimental results give insights into selecting the best combinations of global optimization strategies based on the available system resources such as the buffer space.

4 Comparative performance evaluation between **BFRJ** with effective global optimizations and the state-of-the-art competitor is conducted. Our experimental results show that while the **BFRJ** with one combination of global optimizations (**Comb1**) outperforms the competitor by some margin when the buffer space is small, another combination (**Comb2**) outperforms the competitor by an even more significant margin (up to 50%) when a medium or large buffer space is available. The significant performance gain by **Combo2** is particularly important because modern databases tend to have a very large system buffer so that each task is likely to have access to at least a medium-sized buffer space. **BFRJ** therefore is well-suited for modern databases because it improves spatial join performance by deploying global optimizations that take advantage of a larger buffer allocation.

References

[1] Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica,* Vol. 1, No. 3, 1972, pp. 173 – 189.

[2] Bechmann, N., Kriegel, H., Schneider, R., and Seeger, B., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data,* May 1990, pp. 322 – 332.

[3] Brinkhoff, T., Kriegel, H., and Seeger, B., "Efficient Processing of Spatial Joins Using R-trees", *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data,* 1993, pp. 237 – 246.

[4] Bureau of Census., "Tiger/Lines Precensus Files: 1990 Technical Documentation", Technical report, Bureau of Census, Washington, D.C., 1989.

[5] Faloutsos, C. and Kamel, I. "On Packing R-tree," *Proc. of the CIKM*, 1993, pp. 490 – 499.

[6] Gunther, O., "Efficient Computation of Spatial Joins," *Proc. of the 9th Int. Conf. on Data Eng.,* 1993, pp. 50 – 59.

[7] Guttman, A., "R-tree: a dynamic index structure for spatial searching", *Proc. of the 1984 ACM SIGMOD Int. Conf. on Management of Data,* 1984, pp. 45 – 57.

[8] Intergraph Corporation, "GIS/AM/FM Information," *http://www:intergraph.com/utilmap.shtml*, 1995.

[9] Lo, M.L. and Ravishankar, C.V., "Spatial Join Using Seeded Trees," *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data,* May 1994, pp. 209 – 220.

[10] Lo, M.L. and Ravishankar, C.V., "Spatial Hash-Joins," *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data,* June 1996, pp. 247 – 258.

[11] Nievergelt, J. and Hinterberger, H., "The Grid File: A Adaptable, Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol. 9, No. 1, Mar. 1984, pp. 39 – 71.

[12] Orenstein, J.A., "Spatial Query Processing in an Object-Oriented Database System", *Proc. of the 1986 ACM SIGMOD Int. Conf. on Management of Data,* 1986.

[13] Patel, J.M. and DeWitt, D.J., "Partition Based Spatial-Merge Join," *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data,* June 1996, pp. 259 – 270.

[14] Preparata, F.P. and Shamos, M.I., "Computational Geometry," Springer, 1985.

[15] Rotem, D., "Spatial Join Indices," *IEEE 7th Int. Conf. on Data Engineering,* 1991, pp. 500 – 509.

[16] Seagate Corporation, "Performance Specifications for Barracuda 2LP Family Disk Drives," *http://www:seagate.com/disc/cuda/cuda2lp.shtml*, 1996.

[17] Sellis, T., Roussopoulos, N. and Faloutsos, C., "The R^+-Tree: A Dynamic Index for Multi-dimensional Objects," *Proc. of the VLDB Conf.*, Brighton, England, 1987, pp. 3 – 17.

[18] Shamos, M.I. and Hoey, D.J., "Geometric Intersection Problems," *Proc. 17th Annual Conf. on Foundations of Computer Science*, 1976, pp. 208 – 215.

[19] Stonebraker, M., Rowe, L., and Hirohama, M., "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990, pp. 125-142.

[20] Ubell, M., "The Montage Extensible DataBlade Architecture," *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, May, 1994.