Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations [*]

Yun-Wu Huang
IBM T. J. Watson Research Center
ywh@watson.ibm.com

Ning Jing
Changsha Institute of Technology
ningjing@pdns.nudt.edu.cn

Elke A. Rundensteiner
Worcester Polytechnic Institute
rundenst@cs.wpi.edu

Abstract

R-tree based spatial join is useful because of both its superior performance and the wide spread implementation of R-trees. We present a new R-tree join method called BFRJ (Breadth-First R-tree Join). BFRJ synchronously traverses both R-trees in breadth-first order while processing join computation one level at a time. At each level, BFRJ creates an intermediate join index and deploys global optimization strategies (ordering, memory management, buffer management) to improve the join computation at the next level. We also present an experimental evaluation 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%).

0.05in

[*]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 authors were at the University of Michigan.

**Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997**

1 Introduction

Spatial data management has become more and more crucial in a wide range of applications such as geographic information systems, image processing, VLSI, and CAD/CAM. A successful spatial database management system must provide efficient query functions such as spatial joins that combine multiple data sets based on a spatial predicate (e.g., *intersect* or *contain*) [9]. Examples of spatial join queries are **Q1**: "*Find all parks which are in a city.*" and **Q2**: "*Find all trails that go through some forest.*" In **Q1**, the two data sets are parks and cities, the spatial predicate is *contains*, whereas in **Q2**, the two data sets are trails and forests, and the spatial predicate is *intersects*.

It is well known that processing such spatial join queries is an extremely I/O and CPU expensive process. This paper presents a new method for spatial joins with a significant performance improvement (up to 50%) over the state-of-the-art approach [3]. Like other spatial join techniques [3], our method is based on existing R-tree indexes for the two input data sets. Using R-trees for spatial join processing is effective because R-tree (and its variants) has become a very popular spatial index structure, e.g., Illustra, Intergraph's GIS databases, Postgres, and MapInfo all offer R-tree support. Furthermore, join processing based on R-trees has been shown to result in superior performance as compared to alternate index structures [15].

While the state-of-the-art R-tree join approach [3] follows a depth-first order for traversing the two input R-trees, we demonstrate in this paper that a spatial join technique based on the 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. Based on the intermediate join results created at a given level, called the *intermediate join index* (IJI), BFRJ can 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 depth-first R-tree join approach [3] which 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 BFRJ instead capture more information enabling BFRJ to apply global optimization strategies for effectively managing these IJIs. In particular, we propose three such global optimization dimensions that include IJI ordering, IJI memory management, and buffer management optimizations.

The IJI ordering optimization 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.

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. We also compare 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 by up to 50%.

The rest of the paper is organized as follows. Section 2 discuses related work, while Section 3 provides background material on spatial join processing. Section 4 introduces the framework of BFRJ, followed by Section 5 where BFRJ global optimizations are proposed. We present our experimental results in Section 6 and conclude the paper in Section 7.

2 Related Work on Spatial Joins

In [14], 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 [16], spatial join indexes are computed using Grid files [13] 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. 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 even has become the yardstick used by other researchers to measure the performance of their proposed non-index based spatial join methods [12, 15].

Besides R-tree joins, recent spatial join research has also focused on joining spatial data when the associated spatial indexes do not exist for the data sets. When no index exists for the two input data sets, then the spatial hash join proposed in [12] uses spatial partitioning as the hash function. A similar partition-based spatial-merge join is also proposed in [15].

Like most other recent work on R-tree based joins, the BFRJ technique we propose in this paper optimizes the filter step of spatial join processing (i.e., it works at the Minimum Bounding Rectangle (MBR) level). In [10], on the other hand, we present an optimization of the refinement step of spatial join processing, called SID (Symbolic Intersect Detection), for which we demonstrate unparalleled performance improvements. Given that SID is complimentary to BRFJ, our combined solution of BRFJ + SID thus offers a radical improvement of the overall R-tree join processing task making it practical for even very large spatial data sets as commonly found in advanced GIS applications. Finally, newly developed estimation techniques for spatial join costs [11] will allow for spatial database engines to trade off between using these versus other join techniques for processing spatial queries.

3 Background on R-Tree Spatial Joins

3.1 The R-tree Structure

R-trees [7] are an extension of B-trees [1] that store multi-dimensional data. 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.

In most R-tree variants, entry MBRs are allowed to overlap one another [2, 7, 5]. This means that there may be more than one search path. Recently proposed R-tree variants tried to minimize the overlap between the entry MBRs. Among them, R*-tree [2] introduces heuristics that yield 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. The experimental results presented in this paper are based on packed R-trees.

3.2 Notations Related to R-tree Joins

For brevity, we denote the two R-trees used for spatial joins as R and S. Below, we present the notations that describe R. The notations for S are 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 $0 \le i < hR$. Note that $lR^0 = 1$.
- nR^l_i is the i-th tree node at level l of R, where $0 \le l < hR$ and $0 \le i < lR^l$. Since there is only one root node at level 0, we use nR^0 to denote the root node of R.
- eR^l_i is the number of entries in the tree node nR^l_i.
- $< oidR^l_j, mbrR^l_j >_i$ is the i-th entry in tree node nR^l_j, where $0 \le l < hR$, $0 \le j < lR^l$, $0 \le i < eR^l_j$, $oidR^l_j$ is the *addr* (for non-leaf nodes) or *oid* (for leaf nodes) and $mbrR^l_j$ is *mbr* of this entry.

In this paper, spatial join pertains to 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 [10] is then applied to each item in the candidate set to determine if the two objects really overlap. This latter process is called the refinement step.

3.3 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 a node-pair join. Let nR^r_i and nS^s_j ($0 \le r < hR$ and $0 \le s < hS$) be the two nodes retrieved from R and S respectively. The node-pair join between nR^r_i and nS^s_j computes a set of ID pairs $< oidR^r, oidS^s >$ such that their associated MBRs, $mbrR^r$ and $mbrS^s$, overlap. Local optimizations pertain to the techniques that improve the CPU cost for node-pair joins. We now describe two local optimization techniques [3], namely *search-space-restriction* and *plane-sweep*, that are incorporated by BFRJ.

Search Space Restriction. During a node-pair join between nR^r_i and nS^s_j, the intersecting area between the MBRs of the two nodes is itself an MBR (called intersect-MBR). If an entry $< oidR^r_i, mbrR^r_i >_x$ in nR^r_i overlaps an entry $< oidS^s_j, mbrS^s_j >_y$ in nS^s_j, both $mbrR^r_i$ and $mbrS^s_j$ must intersect their intersect-MBR. Therefore, we can scan all entries in nR^r_i and nS^s_j once to discard the entries whose MBRs do not overlap the intersect-MBR between the two nodes. The actual node-pair join takes only the selected entries as input.

Plane Sweep. This optimization is similar to the sort-merge join technique used to join two simple data sets. During the *plane sweep* optimization of a node-pair join computation between nR^r_i and nS^s_j, 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), we conduct intersect tests against the MBRs from the opposite entry which overlap mbr_i based on their x-coordinate values.

Local Versus Global Optimizations. The *search-space-restriction* 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 [3]. In Section 5, 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.

4 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).

4.1 Search Pruning by Traversing R-Trees

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, a node-pair join between nR^r_i and nS^s_j, is only needed when the MBR of nR^r_i's parent node over-

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

laps 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 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* compared to the nested-loop approach.

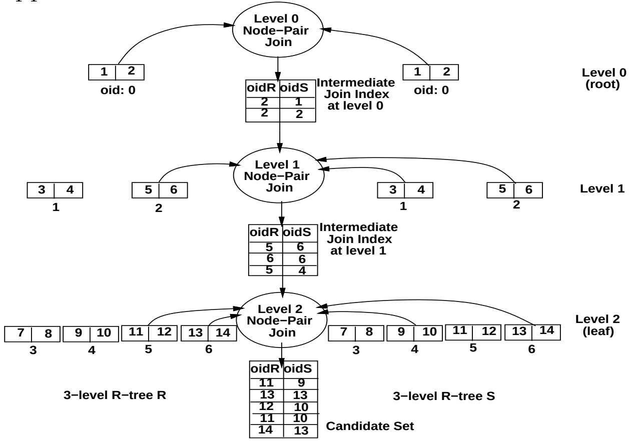

Figure 1: Breadth-First R-tree Join (BFRJ) on R-trees of the Same Height.

4.2 The BFRJ Solution

The BFRJ first conducts a node-pair join between R's root node (nR^0) and S's root node (nS^0) (see Figure 1). The results are a set of 2-tuples $< oidR^0, oidS^0 >$ called *intermediate join index* at level 0 (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 each tuple item from R and S respectively. It then conducts node-pair join between the entries of the two retrieved nodes. While BFRJ reads tuples of IJI_0 for join computation, it stores the join results, also in the form of 2-tuples $< oidR^1, oidS^1 >$, in the current *intermediate join index* at level 1 (IJI_1). When it completes join computation for all tuples in IJI_0, it discards IJI_0 and proceeds to now 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. If the two trees are not of the same height, then after reaching the leaf level of one of the two trees, say R, then the algorithm will stay at this leaf level of R while proceeding to traverse S downwards level by level until S's leaf level is reached as well. At this point, the filter step of the spatial join process is completed and the current (leaf-level) IJI is the output of the spatial join process.

Figure 1 depicts an example of the processing 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.

4.3 The BFRJ Algorithm

Finally, the algorithm that conducts an R-tree spatial join based on the above described BFRJ framework is given in Figure 2. It assumes that the heights of the two input R-trees are the same ($hR = hS$), however a generalization for this *BFRJ* algorithm for trees of unequal height is straightforward and can be found in our technical report [8].

```
PROCEDURE BFRJ ($R$, $S$)
// $R$, $S$ are two R-trees, $hR = hS$
// $IJI[i]$ intermediate join index at level $i$
DATA STRUCTURES: set $IJI[hR] := \emptyset$;
01  $IJI[0] := Node\_Pair\_Join(nR^0, nS^0)$;
02      // join two root nodes
03  integer $i := 0$;
04  while $i < hR - 1$ do
05      $\forall < oidR^i, oidS^i > \in IJI[i]$ do
06          $IJI[i+1] = IJI[i+1]$
07          $\cup\, Node\_Pair\_Join(oidR^i, oidS^i)$;
08      end do
09      $i := i + 1$; //down one level
10  end while
11  output $IJI[i]$;
```

Figure 2: The *BFRJ* Algorithm (Shown for Input R-trees of the Same Height).

The $Node\_Pair\_Join()$ 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 this node-pair join process deploys the local optimizations described in Section 3.3.

5 Global Optimizations of BFRJ

In the BFRJ framework, an *intermediate join index* (IJI) 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 (IJI)*. 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 a straightforward extension of the proposed strategies [8].

5.1 Ordering of Intermediate Join Indexes

Let the MBR of nR_i^l intersect 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}. During the join computation at level l, nR_i^l will participate in the node-pair join 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 of the buffer 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 *IJIs*.

However, each tuple $< oidR, oidS >$ in IJIs has two items by which it may need to be fetched from 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[2]:

Option 1: No particular ordering (OrdNon). OrdNon does not perform global ordering for the *IJIs*. The *IJI* 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 may not contribute to a good global ordering.

Option 2: Ordering by items from one tree (OrdOne). OrdOne sorts the *IJIs* 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 $oidS$.

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

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.

5.2 Memory Management of Intermediate Join Indexes

The *IJIs* can be stored in main memory or disk. The former improves IJI ordering efficiency and eliminates additional I/Os for accessing IJIs while the latter gives more main memory space for join computation.

StorDisk: Storing indexes on disks. In the StorDisk approach, the *IJIs* 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. The indexes are sorted only 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 *IJIs* only need to be read once if they fit into the buffer, or more than once[3] if

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

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.

StorMem: Storing indexes in main memory. StorMem keeps the *IJIs* at the current level in main memory ($|\text{IJI}^{max}|$ 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.

5.3 Buffer Management of Intermediate Join Indexes

The ordering optimization (Section 5.1) 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.

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 *IJI*, 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 for the rest of the spatial join processing of the current IJI. 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*[4] until its counter reaches 0 later on.

5.4 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 as accomplished by our proposed BFRJ technique. This is because the order by which each node-pair is to be joined is determined by the recursive depth-first sequence that consequently makes it difficult to globally modify any ordering of traversal. 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 only during backtracking (hence leaving not much of an opportunity for optimization).

6 Experimental Results

Our performance studies are based on 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 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.

6.1 Experiments on Intermediate Join Index Ordering Optimizations

We implemented IJI ordering optimization options OrdNon, OrdOne, OrdSum, OrdCen, and OrdHil, and conducted BFRJ spatial join on the two packed R-trees for each option by varying buffer sizes from 100 KBytes to 1,200 KBytes. We fix the other global optimization options to StorDisk and PinNo, meaning we

[4]We assume if all pages are *pinned*, the buffer manager *unpins* and *purges* the least recently used page.

store the *intermediate join indexes* on disk and we do not deploy the *pinning* optimization.

Because the I/O costs with smaller buffers are very high, the performance difference with larger buffers cannot be clearly seen. For clarity, we separate the I/O results into two charts (Figures 3 and 4) based on buffer sizes. Note that the horizontal line marked optimal in both figures (and subsequent figures) represents the theoretical lower bound of page I/O based on the two packed R-trees[5] used for testing.

The results in Figure 3 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. From Figure 4, we can see that OrdSum is the clear winner when a more generously-sized buffer ($\geq$ 600 KBytes) is available.

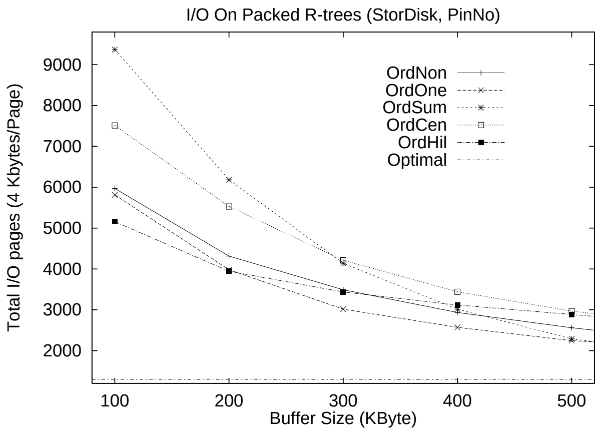

Figure 3: I/O Cost on Ordering Optimization (Small Buffers).

The OrdSum option outperforms others with larger buffer sizes because it orders the IJIs by taking the spatial locations (on x-axis) of both 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.

Our separate CPU results (not shown due to space limitation) reveal that the differences in CPU cost

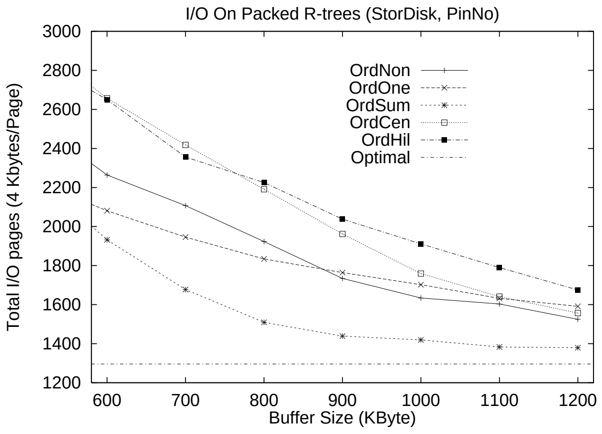

Figure 4: I/O Cost on Ordering Optimization (Large Buffers).

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.

6.2 Experiments on Intermediate Join Index Memory Management Optimizations

To evaluate IJI memory management optimization, we experimented with StorDisk and StorMem options. Because the estimated size of the largest *intermediate join index* is already about 100 KBytes for our test data, 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.

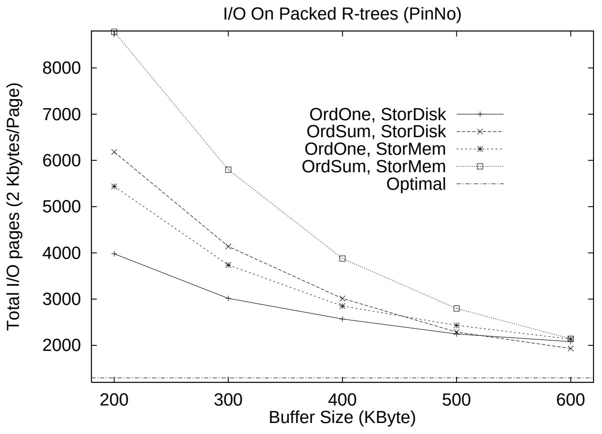

Figure 5: I/O Cost on Memory Management (Small Buffers).

For this set of experiments, we also separate the I/O results into two charts based on buffer sizes. The results in Figure 6 show that StorMem starts to outperform StorDisk in I/O when the buffer size is larger than

[5] 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.

800 KBytes. The reason StorMem performs so poorly with smaller buffers is that it needs additional main memory to store the join indexes (Figure 5). Thus, join computation in StorMem has less buffer pages to work with, thereby creating a buffer contention over a limited number of buffer pages.

Although StorMem outperforms StorDisk in I/O when the buffer sizes are large ($>$ 800 KBytes in Figure 6), 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.

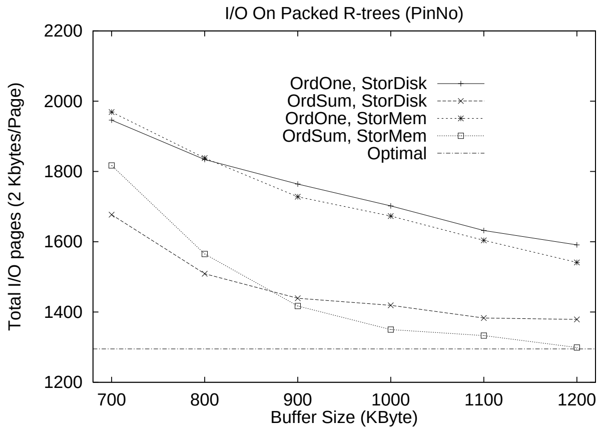

Figure 6: I/O Cost on Memory Management (Large Buffers).

Our separate CPU results (not shown due to space limitation) reveal that StorMem can improve the CPU usage time over StorDisk, but not very significantly.

6.3 Experiments on Intermediate Join Index Buffer Management Optimizations

To test the effect of the (*pinning* optimization) described in Section 5.3, we implemented this buffering strategy and 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 conduct experiments on the *pinning* optimization based on OrdOne with smaller buffers and OrdSum on larger buffers separately. For the first set of experiments (Figure 7), 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 8), we run BFRJ based on OrdSum with both *pinning* optimizations and vary the buffer sizes from 600 KBytes to 1,200 KBytes.

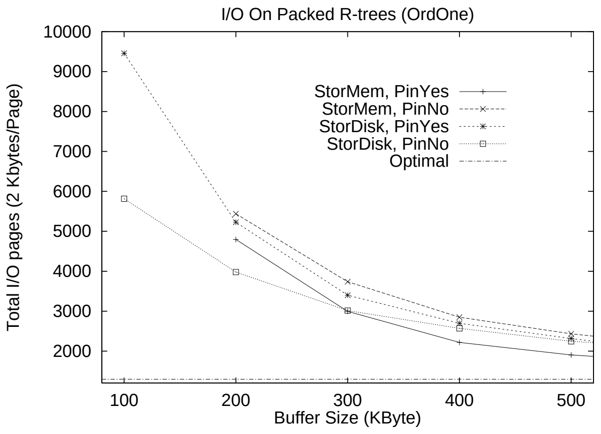

Figure 7: I/O Cost on Buffer Management (Small Buffers).

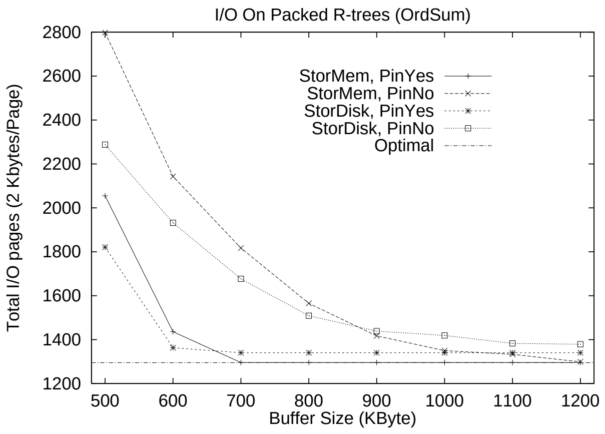

Figure 8: I/O Cost on Buffer Management (Large Buffers).

The results in Figure 7 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 8 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.

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

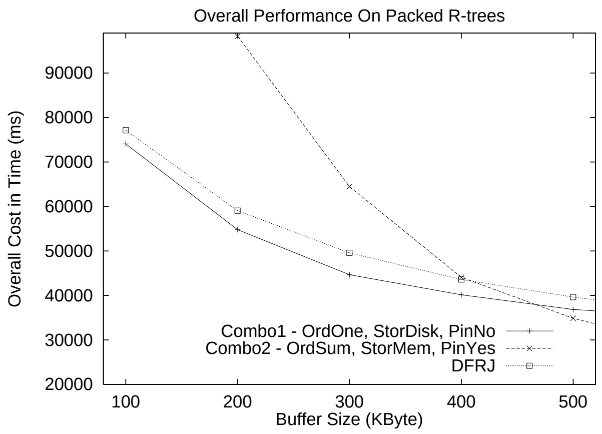

Figure 9: Overall Cost: BFRJ Vs. DFRJ (Small Buffers).

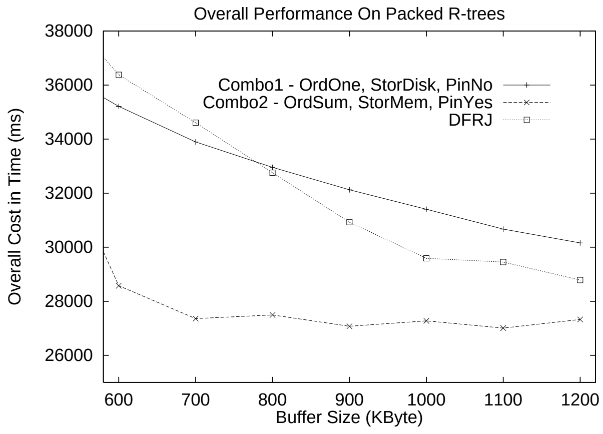

Figure 10: Overall Cost: BFRJ Vs. DFRJ (Large Buffers).

We have implemented the state-of-the-art R-tree join technique with proper optimizations presented in [3]. We call it Depth-First R-tree Join, or DFRJ. 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 and combined them using the following formula:

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

where t is 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.

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 [17] [6].

The results in Figure 9 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 10 indicate that **Combo2** outperforms both **Combo1** and DFRJ by a significant margin (up to 50%). Our separate I/O and CPU results (not shown due to space limitation) indicate that the CPU difference among the three options is very small while the I/O cost becomes the dominant factor in overall performance.

7 Conclusions

In this paper, we present a new spatial join method that is based on breadth-first traversal of R-trees, called Breadth-First R-tree Join (BFRJ). 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. In particular, three dimensions for global optimization are proposed for BFRJ, i.e., the ordering, memory management, and buffer management optimizations of the *intermediate join indexes*,

[6] 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.

with alternative solution techniques identified for each of these three dimensions.

We conducted an extensive experimental evaluation of the performance of BFRJ using real GIS data sets from the US Census Bureau. Our experimental results give insights into selecting the best combination of global optimization strategies based on the available system resources such as the buffer space. Our results also show that with the proper selection of options in global optimizations, BFRJ consistently outperforms the competitor by up to 50%.

Acknowledgements. We would like to thank all students in the University of Michigan Database Group (UMDG) for their help and suggestions for this work and its presentation. In particular, we thank Matthew Jones, Nauman Chaudhry, Stacie Hibino, Anisoara Nica, Viviane Crestana, Lei Zhou, and Amy Lee.

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] Huang, Y.W., Jing, N., and Rundensteiner, E.A., "BFRJ: Global Optimization of Spatial Joins Using R-trees," Dept. of Computer Science, Worcester Polytechnic Institute, Tech. Report WPI-CS-TR-97-5, Jan. 1997.

[9] Huang, Y.W., Jing N. and Rundensteiner, E. A., "Integrated Query Processing Strategies for Spatial Path Queries," *IEEE Int. Conf. on Data Engineering,* Birmingham, UK, April 1997.

[10] Huang, Y.W., Jones, M. and Rundensteiner, E. A., "Improving Spatial Intersect Joins Using Sympolic Intersect Detection," *5th Int. Symp. on Large Spatial Databases*, Berlin, Germany, 1997.

[11] Huang, Y.W., Jing N. and Rundensteiner, E. A., "A Cost Model for Estimating the Performance of Spatial Joins Using R-trees," *9th Int. Conf. on Scientific and Statistical Database Management*, Olympia, Washington, Aug. 1997.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] Sellis, T., Roussopoulos, N. & Faloutsos, C,

"The R^+-Tree: A Dynamic Index for Multidimensional Objects," *Proc. of the VLDB Conf.*, Brighton, England, 1987, pp. 3 – 17.