# Multi-way Spatial Joins Using R-Trees: Methodology and Performance Evaluation

Ho-Hyun Park[1], Guang-Ho Cha[2⋆], and Chin-Wan Chung[1]

[1] Department of Computer Science, KAIST, Taejon 305-701, Korea
{hhpark,chungcw}@islab.kaist.ac.kr
[2] IBM Almaden Research Center, San Hose, CA 95120, USA
ghcha@almaden.ibm.com

**Abstract.** We propose a new multi-way spatial join algorithm called *M-way R-tree join* which synchronously traverses M R-trees. The M-way R-tree join can be considered as a generalization of the *2-way R-tree join*. Although a generalization of the 2-way R-tree join has recently been studied, it did not properly take into account the optimization techniques of the original algorithm. Here, we extend these optimization techniques for M-way joins. Since the join ordering was considered to be important in the M-way join literature (e.g., relational join), we especially consider the ordering of the search space restriction and the plane sweep. Additionally, we introduce *indirect predicates* in the M-way join and propose a further optimization technique to improve the performance of the M-way R-tree join. Through experiments using real data, we show that our optimization techniques significantly improve the performance of the M-way spatial join.

## 1 Introduction

The spatial join is a common spatial query type which requires a high processing cost due to the high complexity and large volume of spatial data. Therefore, the spatial join is processed in two steps (the *filter step* and the *refinement step*) to reduce the overall processing cost [14,5]. Many 2-way spatial join methods have been published in the literature: the join using Z-order elements [14], the join using R-trees (called *R-tree join*) [3], the seeded tree join (STJ) [10], the spatial hash join (SHJ) [11], the partition based spatial merge join (PBSM) [20], the size separation spatial join ($S^3J$) [9], the scalable sweeping-based spatial join (SSSJ) [1] and the slot index spatial join (SISJ) [12]. However, there has been little research on the multi-way spatial join [16]. The M-way (M>2) spatial join combines M spatial relations using M-1 or more spatial predicates[1]. An example of a 3-way spatial join is "Find all buildings which are adjacent to roads that

⋆ The work reported here was performed while Guang-Ho Cha was at Tongmyong University of Information Technology, Korea

[1] If the number of spatial predicates is less than M-1, the join necessarily includes cartesian products, in which case we regard the join not as one spatial join but as several spatial joins.

R.H. Güting, D. Papadias, F. Lochovsky (Eds.): SSD'99, LNCS 1651, pp. 229–250, 1999.

intersect with boundaries of districts." An M-way spatial join can be modeled by a *query graph* whose nodes represent relations and edges represent spatial predicates.

One way to process M-way spatial joins is as a sequence of 2-way joins [12]. Another possible way, when all join attributes have spatial indexes and each join attribute is shared among the associated join predicates[2], is to combine the filter and refinement steps respectively as follows:

(1) Scan the relevant indexes synchronously for all join attributes to obtain a set of spatial object identifier tuples.
(2) Read objects for object identifier (oid) tuples obtained from Step (1), and perform an M-way spatial join using geometric computation algorithms.

Step (1) is called *combined filtering* and Step (2) *combined refinement* in [17]. Especially when the R-trees are used in Step (1), the combined filtering is called *M-way R-tree join* which is the scope of this paper. The M-way R-tree join is also called *synchronous traversal* (ST) in [16]. An advantage of the combined filtering is that it removes unnecessary refinement operations for some object pairs. For example, let Figure 1 be an MBR (Minimum Bounding Rectangle) combination of spatial objects for the above query. Let $a$, $b$ and $c$ be instances of the relations *buildings*, *roads* and *boundaries*, respectively. If it is processed by a sequence of 2-way joins and the evaluation order is determined to be $\langle a, b, c\rangle$ by a query optimizer, the refinement operation between $a$ and $b$ will be performed unnecessarily. However, the combined filtering can avoid this situation.

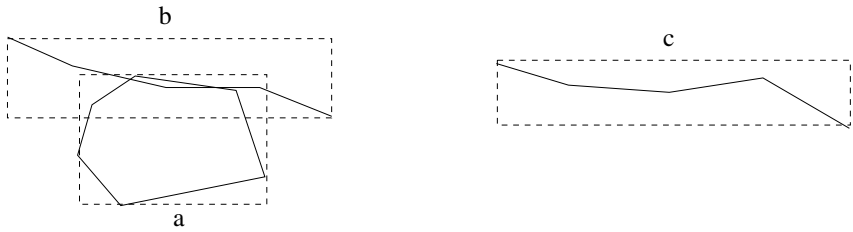



**Fig. 1.** An MBR combination in a 3-way join

The M-way R-tree join can be considered as a generalization of the *2-way R-tree join* of [3,7] and does not create intermediate results. Although a generalization of the 2-way R-tree join called *multi-level forward checking* (MFC) has recently been studied [15,16], it did not properly take into account the optimization techniques of the original 2-way R-tree join.

The main contributions of this paper are as follows: First, we generalize the 2-way R-tree join to consider the order of search space restrictions and plane sweeps because the join ordering was considered to be important in the M-way join

[2] In this case, only one spatial predicate per relation participates in the join.

literature (e.g., relational join) [8]. Second, we introduce *indirect predicates* in the M-way spatial join and propose a further optimization technique to improve the performance of the M-way R-tree join. Through experiments, we show that our optimization techniques significantly improve the performance of the M-way spatial join (especially the filter step) against MFC. Additionally, we find that the M-way R-tree join becomes CPU-bound as M increases.

The remainder of this paper is organized as follows: Section 2 provides some background by briefly explaining the 2-way R-tree join and the state-of-the-art M-way spatial joins using R-trees. In Section 3, we propose an algorithm of the M-way R-tree join, which considers the ordering of search space restrictions and plane sweeps, as a new generalization of the 2-way R-tree join, and further improve the performance of the M-way R-tree join using the concept of indirect predicates. In Section 4, we present some experiments for the performance analysis of our algorithms using the TIGER data [22]. Finally in Section 5, we conclude this paper and suggest some future studies.

## 2 Background

### 2.1 2-Way Spatial Joins Using R-Trees

Assuming that R-trees [4,2] exist for both join inputs, a join algorithm which synchronously traverses both R-trees using depth-first search was proposed [3]. The basic idea of the algorithm is as follows: First, it reads the root nodes of the R-trees and checks if the rectangles of entries of both nodes mutually intersect. Next, only for intersected entry pairs, it traverses the child node pairs by depth-first search and continuously checks the intersection between the rectangles of entries of both child nodes. In this way, if the algorithm reaches the leaf nodes, it outputs the intersected entry pairs and backtracks to the parent nodes. Two optimization techniques, called *search space restriction* and *plane sweep*, are used to reduce the CPU time. The search space restriction heuristic picks out the entries whose rectangles do not intersect with the rectangle enclosing the other node, before the intersection is actually checked between the rectangles of entries of both nodes. The plane sweep first sorts the rectangles of entries of both nodes for one axis, and then goes forward along the sweep line and checks the intersection for the other axis. The algorithm using the above techniques is shown below: (We skip the detailed algorithm for *SortedIntersectionTest* in Step (6) due to space limitation. Refer to [3] for details.)

```
   RtreeJoin (Rtree_Node R, S)

(1)  FOR all E_i ∈ R DO
(2)       IF E_i.rect ∩ S.rect == ∅ THEN R = R-{E_i}; /* space restriction
     on R */
(3)  FOR all E_j ∈ S DO
(4)       IF E_j.rect ∩ R.rect == ∅ THEN S = S-{E_j}; /* space restriction
     on S */
(5)  Sort(R); Sort(S);
```

```
 (6)  SortedIntersectionTest (R, S, Seq); /* plane sweep */
 (7)  FOR i = 1 TO ‖Seq‖ DO
 (8)      (E_R, E_S) = Seq[i];
 (9)      IF R is a leaf page THEN /* S is also a leaf page */
(10)          output (E_R, E_S);
(11)      ELSE
(12)          ReadPage(E_R.ref); ReadPage(E_S.ref);
(13)          RtreeJoin (E_R.ref, E_S.ref);

   END RtreeJoin
```

Additionally, the algorithm applied the *page pinning* technique for I/O optimization. The algorithm used only a local optimization policy to fetch the child node pairs. Later, a global optimization algorithm by breadth-first search was proposed [7]. In this paper, we call both of the join algorithms *2-way R-tree join* or simply *R-tree join*. When R-trees exist for both join inputs, it has been shown that the R-tree join is most efficient [10,20,12].

### 2.2 State-of-the-Art M-Way Spatial Joins Using R-Trees

In a recent study, two methods called *multi-level forward checking* (MFC) and *window reduction* (WR) were proposed to process structural queries for image similarity retrieval [15]. Later, they were applied to the multi-way spatial join [16]. MFC and WR were motivated by a close correspondence between multi-way spatial joins and constraint satisfaction problems (CSPs). A multi-way spatial join can be represented in terms of a binary CSP [16]:

- A set of $n$ variables, $v_1, v_2, \ldots, v_n$, each corresponding to a dataset.
- For each variable $v_i$, a domain $D_i$ which consists of the data in tree R.
- For each pair of variables $(v_i, v_j)$, a binary constraint $Q_{ij}$ corresponding to a spatial predicate.

If $Q_{ij}(E_{i,x}, E_{j,y}) =$ TRUE, then the assignment $\{v_i = E_{i,x}, v_j = E_{j,y}\}$ is *consistent*. A solution is an $n$-tuple $\tau = \langle E_{1,w}, \ldots, E_{i,x}, \ldots, E_{j,y}, \ldots, E_{n,z} \rangle$ such that $\forall i, j, \{v_i = E_{i,x}, v_j = E_{j,y}\}$ is consistent. In the sequel, we use the terms variable/dataset/relation and constraint/predicate/join condition interchangeably.

**1) Multi-level Forward Checking** MFC is a kind of ST algorithms which synchronously traverses $n$ R-trees as follows: It starts from the root nodes of $n$ R-trees and checks all predicates for each $n$-combination (called *entry-tuple*) from the entries of the nodes. If an entry-tuple satisfies all the predicates, one of the following occurrs: If the *node-tuple* (an $n$-combination of the R-tree nodes) is in the intermediate level, the algorithm is recursively called for the child node-tuple pointed by the entry-tuple. Otherwise, i.e., if the node-tuple consists of leaf nodes, the algorithm outputs the entry-tuple and processes the next entry-tuple. If an entry-tuple does not satisfy at least one predicate, the entry-tuple is pruned. MFC was considered as a generalization of the 2-way R-tree join.

At each R-tree level, MFC applies *forward checking* (FC), which is known to be one of the most effective algorithms for solving CSP, to find the entry-tuples satisfying the predicates. FC maintains an $n * n * C$ *domain table* ($n$: number of variables, $C$: the maximum number of entries of an R-tree node) in main memory. $domain[i][j] (0 \leq i, j < n)$ is a subset of an R-tree node $N_j$. FC works as follows [15]: First, $domain[0][j]$ is initialized to an R-tree node $N_j$ for all $j$. When a variable $v_0$ is assigned a value $u_k$, $domain[1][j]$ is computed for each remaining $v_j$, by including only values $u_l \in domain[0][j]$ such that $Q_{0j}(u_k, u_l) =$ TRUE. In general, if $u_k$ is the current value of $v_i$, $domain[i+1][j]$ is the subset of $domain[i][j]$ which is valid w.r.t. $Q_{ij}$ and $u_k$. In this way, at each instantiation the domain of each *future variable* (un-instantiated variable) continuously shrinks. FC outputs a solution whenever the last variable is given a value. When the domain of the current variable is exhausted, the algorithm backtracks to the previous one.

For ordering of variable instantiations, MFC applies the *dynamic variable ordering* (DVO), which is also mainly used in CSPs. DVO dynamically reorders the future variables after each instantiation so that the variable with the minimum domain size becomes the next variable. Additionally, MFC adopts the *search space restriction* technique to improve performance. A slightly modified version of the space restriction algorithm used in [15] is shown below:

```
BOOLEAN SpaceRestriction_1 (Query_graph Q[][], Rtree_Nodes N[])

(1)  FOR i=0 TO n-1 DO
(2)      ReadPage (N[i]);
(3)      FOR all E_k ∈ N[i] DO
(4)          FOR j=0 TO n-1, i≠j DO
(5)              IF Q[i][j] == TRUE AND E_k.rect ∩ N[j].rect == ∅ THEN
(6)                  N[i] = N[i]-{E_k};
(7)                  BREAK;
(8)      IF N[i]==∅ THEN RETURN FALSE;
(9)  RETURN TRUE;

END SpaceRestriction_1
```

We do not adopt MFC for the following reasons: First, MFC does not apply the plane sweep technique, which is fairly efficient in the rectangle intersection problem [21,3], but uses FC-DVO which is just a special form of the nested loop. Second, during the space restriction, MFC does not consider the space restriction order among $n$ R-tree nodes, i.e., which node should be checked first. In Section 3, we propose a new generalization of the 2-way R-tree join which considers both the space restriction ordering and the plane sweep technique.

**2) Window Reduction** WR maintains an $n * n$ *domain window* (instead of a 3D domain table) that encloses all potential values for each variable. When a variable is instantiated, a domain window for each future variable is shrunk to the intersection between the newly computed window according to the current

variable instantiation and existing domain window. For the instantiation of the current variable, a window query is performed using the current domain window. In WR, the DVO technique was also applied to reorder the future variables, i.e., the future variable with the smallest domain window becomes the next variable to be examined. WR was considered as a special form of the indexed nested loop join. However it does not generate intermediate results. WR must essentially search the whole space in order to instantiate the first variable. To avoid the blind instantiation for the first variable, a hybrid technique called *join window reduction* (JWR) was proposed [15]. JWR applies the R-tree join for the first pair of variables and then WR for the rest of the variables.

In [16], a slightly different WR algorithm was proposed for the multi-way intersection join. In that algorithm, the instantiation order of variables is predetermined according to an optimization method. As a query window, for acyclic queries (tree topology), the rectangle of the variable directly connected to the current variable among instantiated variables becomes the query window. For complete queries (clique topology), the common intersected rectangle of all instantiated variables becomes the query window. In our implementation and experiment, regardless of query types, among instantiated variables which are connected to the current variable, one whose value has the smallest rectangle was selected and the rectangle becomes a query window for the next variable instantiation.

## 3 New Methods for M-Way Spatial Joins Using R-Trees

### 3.1 A New Generalization of the 2-Way R-Tree Join

In this section, we propose a new M-way join algorithm which extends both the search space restriction and the plane sweep optimization techniques of the 2-way R-tree join. We emphasize the ordering of both optimization techniques, assuming only *intersect* (not disjoint) as a join predicate.

**1) Search Space Restriction** Algorithm *SpaceRestriction_1* [15] does not consider ordering among M R-tree nodes. If no entry of an R-tree node passes over the space restriction, we do not have to check other nodes. Especially in an *incomplete join* (no join predicate between some variables), the possibility that no entry of an R-tree node may pass over the space restriction is high. In such a case, Algorithm *SpaceRestriction_1* may result in unnecessary reading of other nodes. Therefore, the space restriction order of the R-tree nodes becomes important. For example, Figure 2 shows an MBR intersection between intermediate nodes of the R-trees for a 4-way spatial join "X *intersect* Y and Y *intersect* Z and Z *intersect* W." Since no entry of node B simultaneously intersects with nodes A and C, the intermediate node-tuple $\langle A, B, C, D \rangle$ cannot pass over the space restriction and becomes a false hit. If the space restriction is performed first on node B, we do not have to check other nodes A, C and D and can save the I/O and CPU time.

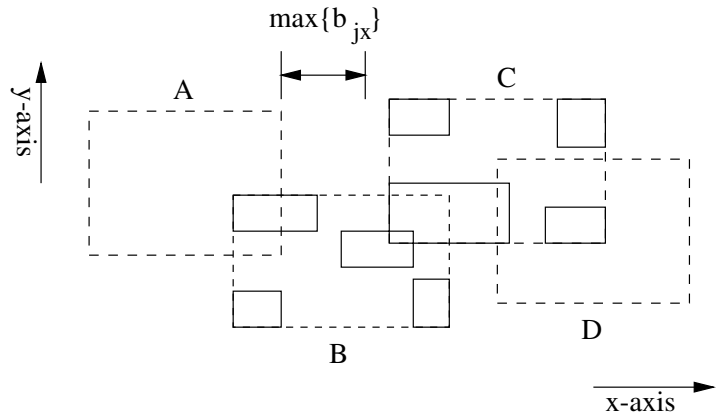



**Fig. 2.** Intersection between intermediate nodes of R-trees

We will explain the space restriction ordering (SRO) in the context of the query graph. For SRO, we use the following two metrics per node $N_i$, $0 \leq i \leq n\text{-}1$:

(1) *normalized common rectangle area* (NCRA): the area of the common intersection of $N_i$ and its adjacent nodes divided by the area of the rectangle of $N_i$. Formally, for all $N_j$ where $i = j$ or $Q_{ij}$ = TRUE, $0 \leq j \leq n\text{-}1$, area($\bigcap N_j.rect$)/area($N_i.rect$).
(2) *maximum inter-rectangle distance* (MIRD): the sum of squares of the maximum of distances per axis between the nodes adjacent to $N_i$. Formally, for all $N_j$ and $N_k$ where $Q_{ij}$ = TRUE and $Q_{ik}$ = TRUE, $0 \leq j, k \leq n\text{-}1$, $j \neq k$, $\Big(\max\{x\_dist(N_j, N_k)\}\Big)^2 + \Big(\max\{y\_dist(N_j, N_k)\}\Big)^2$.

Using the above two metrics, we perform SRO on the basis of the following criteria:

(1) Choose a node with the minimum NCRA.
(2) If the minimum NCRA is zero for more than one node, choose a node such that MIRD is maximal.

In Figure 2, the common intersected rectangles for each node are A.rect ∩ B.rect, A.rect ∩ B.rect ∩ C.rect, B.rect ∩ C.rect ∩ D.rect and C.rect ∩ D.rect. Since nodes B and C have zero NCRA, these two nodes are selected by Criteria (1). Then, since MIRD (only between A and C in this case) of node B is longer than MIRD (between B and D) of node C, we perform the space restriction for node B first by Criteria (2).

In Metric (1), the reason we use the normalized area instead of the (absolute) common intersected rectangle area (CRA) is to choose a node which has large dead space (If CRAs are the same, a larger node is more likely to have more dead space). The dead space of the MBR of an intermediate node may be influenced by many factors such as the number of entries, the distribution of the rectangles

of the entries, the density of the rectangles of the entries, and the MBR size of the node. If the other conditions are fixed, the smaller the number of entries of an intermediate node the more dead space the MBR of an intermediate node may have. Skewed distributions, low density and large MBRs may lead to large dead space. However, we cannot know the above characteristics except the MBR size unless we visit the node. Therefore, we choose only the MBR size among the above characteristics. We expect that NCRA behaves better than CRA especially in the complete query graph because CRAs of all nodes are always the same.

The time complexity of SRO is as follows: It takes $O(M)$ time to compute NCRA and MIRD per node. Therefore, it takes $O(M^2)$ time for all nodes. For sorting NCRA and MIRD, it takes $O(M \log_2 M)$ time. Therefore, the overall time complexity is $O(M^2)$. Algorithm *SpaceRestriction_2* is identical to Algorithm *SpaceRestriction_1* but considers ordering of nodes according to the above criteria.

**2) Plane Sweep** In MFC, FC-DVO was used in a node-tuple join because it was known to be efficient in CSPs. However, the plane sweep algorithm was also known to be fairly efficient in the rectangle intersection problem [21]. Therefore, we use the plane sweep as the second optimization technique rather than FC-DVO. In the 2-way join, the plane sweep algorithm is applied only once. In the M-way join, however, the plane sweep algorithm must be applied multiple times because there are M variables and at least M-1 predicates. In this case, the ordering of plane sweeps among R-tree nodes becomes important.

Our plane sweep ordering (PSO) performs as follows: In PSO, we call the evaluated nodes *inner nodes* and the un-evaluated nodes *outer nodes*. In the following, the *cardinality* of a node is the number of entries in the node, and the *degree* of a node is the number of edges (i.e., the number of predicates) incident on the node. Before PSO starts, all R-tree nodes are initialized to outer nodes.

(1) Choose the first two connected nodes whose sum of the cardinalities / the maximal degree between the two nodes is minimal.
(2) Apply plane sweep between the selected two nodes and make the two nodes inner nodes.
(3) Choose an outer node which is adjacent to one or more inner nodes such that cardinality / degree is minimal.
(4) Choose an inner node which is adjacent to the selected outer node and whose cardinality is minimal.
(5) Apply plane sweep between the selected inner node and the selected outer node.
(6) Check additional predicates, if any, between the selected outer node and other inner nodes.
(7) Make the selected outer node an inner node.
(8) Stop if all nodes are inner nodes, otherwise go to Step (3).

In Step (1) and Step (3), the reason we divide the cardinality by the degree is because the more the number of predicates is, the smaller the intermediate

result size may be. The time complexity of PSO excluding actual plane sweeps is as follows: It takes $O(M^2)$ time to choose the first two nodes. And, for the ordering of the rest variables, it also takes $O(M^2)$ time. Therefore, the overall time complexity of PSO is $O(M^2)$.

The direct application of PSO generates intermediate results M-2 times [19]. Since the number of solutions in a node-tuple join can be up to $C^M$ (C: the maximum number of entries of an R-tree node) for the worst case, we need a main memory buffer which can store $C^M$ tuples per R-tree level. For example, if the node size is 2048 bytes and the entry size is 20 bytes, C and $C^M$ are about 100 and $100^5$ respectively in a 5-way join. Although a much smaller buffer will be sufficient in general, this is a tremendous amount of main memory for the worst case. In order to solve this main memory problem, we can use pipelining.

For both the plane sweep and pipelining, we use M buffers ($Seq[]$ in Algorithm *MwayRtreeJoin_1*) each of which holds intermediate entry-tuples. The buffer size is determined according to the main memory size. We apply plane sweep between the first two nodes selected by PSO. The intermediate entry-tuples produced by the first plane sweep are accumulated in $Seq[1]$. If $Seq[1]$ is full or all entries in the first two nodes are evaluated, we recursively call the plane sweep algorithm taking $Seq[1]$ and the next selected outer node as parameters. In general, the plane sweep between $Seq[m]$ and an outer node accumulates the intermediate result to $Seq[m+1]$. If plane sweep is called for the last outer node, the algorithm backtracks to the previous one. In PSO with pipelining, the actual ordering is determined once per node-tuple join. The M-way R-tree join algorithm using PSO with pipelining is shown below:

```
   MwayRtreeJoin_1 (Query_graph Q[][], Rtree_Nodes N[])

(1)  IF NOT SpaceRestriction_2(Q[][], N[]) THEN RETURN;
(2)  PSO (Q[][], N[], outer_order[], inner_order[]);
(3)  i = outer_order[0]; j = inner_order[0];
(4)  Seq[0] = N[j];
(5)  FOR k=0 TO n-1, k≠j DO Sort (N[k]); /* sort all outer nodes */
(6)  PipelinedPlaneSweep (Q[][], N[], Seq[], i, j, 0);

   END MwayRtreeJoin_1

   PipelinedPlaneSweep (Query_graph Q[][], Rtree_Nodes N[], Entry_Tuple_Buf
Seq[], int i, int j, int m)

(1)  Sort (Seq[m]);
(2)  SortedIntersectionTest_1 (N[i], Seq[m], j, Seq[m+1]); /* plane sweep +
     additional predicate checking until Seq[m+1] is full or all entries
     in N[i] and Seq[m] are evaluated */
(3)  IF m == n-2 THEN /* the last outer node is evaluated */
(4)      FOR all T_k ∈ Seq[m+1] DO
(5)          IF all N[l] are leaf nodes, 0≤l≤n-1 THEN
(6)              output T_k;
(7)          ELSE /* all tree heights are equal */
(8)              MwayRtreeJoin_1 (Q[][], T_k.ref[]); /* go downward */
```

```
 (9)  ELSE
(10)      i = outer_order[m+1]; j = inner_order[m+1];
(11)      PipelinedPlaneSweep (Q[][], N[], Seq[], i, j, m+1);
(12)  IF all entries in N[i] and Seq[m] are evaluated THEN
(13)      RETURN;
(14)  ELSE
(15)      empty Seq[m+1];
(16)      goto Step (2);

   END PipelinedPlaneSweep
```

In Algorithm *MwayRtreeJoin_1*, *SortedIntersectionTest_1* is the same as *SortedIntersectionTest* in Algorithm *RtreeJoin* except for the following: First, one input is a sequence of entry-tuples. Second, additional predicate checks are done between the selected outer node and the non-selected inner nodes. Third, when $Seq[m+1]$ is full, *SortedIntersectionTest_1* exits and the status of both loop counters[3] in the algorithm is saved for the next call. In our implementation of PSO, we did not use pipelining because all intermediate results fitted in main memory.

### 3.2 Consideration of Indirect Predicates

The maximum number of possible predicates in the M-way spatial join is M*(M-1)/2, i.e., all relation pairs have join conditions. We call such a join *complete*. If a join is not complete, i.e., the number of predicates is less than M*(M-1)/2, the join is *incomplete*.

As it was pointed out in [15], the M-way R-tree join may generate many false intersections in intermediate levels. As we can see in Figure 2, especially in an incomplete join, the possibility of a false intersection is high. In this case, if we can detect the false intersections before visiting the intermediate node-tuple, we can further reduce I/O and CPU time. For example, if we know in advance that no entry of node B can simultaneously intersect nodes A and C in Figure 2, we can avoid reading node B and checking the intersection between all entries of node B and other nodes (A and C) during space restriction. In this section, we propose a technique which detects a false intersection in intermediate levels of R-trees before visiting the node-tuple.

**1) Indirect Predicates** In a query "X *intersect* Y and Y *intersect* Z and Z *intersect* W" like the one in Figure 2, it seems that there is no relationship between X and Z (or between Y and W, or between X and W). However, for a data tuple (i.e., a tuple of entries from leaf nodes) $\langle a, b, c, d\rangle$ which satisfies the query, $x\_dist(a, c) \leq b_x$ (or $x\_dist(b, d) \leq c_x$, or $x\_dist(a, d) \leq b_x + c_x$) must be satisfied on x-axis ($b_x$ represents x-length for a data MBR $b$). The same condition holds on y-axis. Consequently, for the data tuple $\langle a, b, c, d\rangle$, $x\_dist(a, c) \leq \max\{b_{jx} \mid b_j \in dom(Y)\}$ (or $x\_dist(b, d) \leq \max\{c_{kx} \mid c_k \in dom(Z)\}$, or

[3] Two internal loops exist in *SortedIntersectionTest* [3].

$x\_dist(a,d) \leq \max\{b_{jx}\} + \max\{c_{kx}\}$) must be satisfied on x-axis (dom($Y$) represents the domain (i.e., relation) of data MBRs for variable $Y$). The same condition holds on y-axis. We call the user predicates in the query such as "X *intersect* Y" and "Y *intersect* Z" the *direct predicates*, and the derived predicates such as $x\_dist(X,Z) \leq \max\{b_{jx}\}$ and $x\_dist(Y,W) \leq \max\{c_{kx}\}$ the *indirect predicates*. In R-trees, the x-length and y-length of MBRs of intermediate nodes may be longer than the max x-length and max y-length of the data MBRs in the domain. In Figure 2, if $x\_dist(A,C) > \max\{b_{jx}\}$ (or $x\_dist(B,D) > \max\{c_{kx}\}$, or $x\_dist(A,D) > \max\{b_{jx}\} + \max\{c_{kx}\}$), we do not have to visit the node-tuple $\langle A,B,C,D\rangle$ because the descendent node-tuples will never satisfy the query. Therefore, if we take advantage of the indirect predicates in intermediate levels of the M-way R-tree join, we can achieve more pruning effects. We call such pruning *indirect predicate filtering* (IPF). The max x-length and y-length can be obtained from the statistical information in the database schema.

**2) Indirect Predicate Paths and Lengths** In Figure 2, we call the paths ABC, BCD and ABCD for indirect predicate pairs AC, BD and AD the *indirect predicate paths* (ipp), and the x-path lengths $\max\{b_{jx}\}$, $\max\{c_{kx}\}$, and $\max\{b_{jx}\} + \max\{c_{kx}\}$ the *indirect predicate x_path lengths* (x_ippl). The *indirect predicate y_path lengths* (y_ippl) are similarly defined. In Figure 2, since there is only one indirect predicate path for each indirect predicate pair, it is easy to compute indirect predicate paths and indirect predicate path lengths. However, there can be several indirect predicate paths for an indirect predicate pair in a general M-way join, and the x_path and y_path for the predicate pair can be different. Therefore, we need a systematic method to compute indirect predicate paths and their lengths.

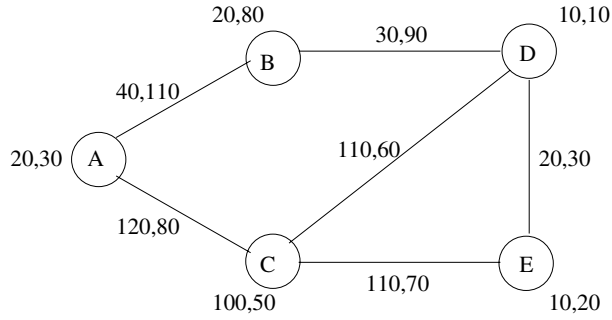
We first draw a query graph whose nodes represent relations and edges represent direct predicates. Then, we assign weights to nodes. The weight of a node is the maximum x-length ($x\_max$) and y-length ($y\_max$) in the relation which the node represents. Since there can be multiple paths between a node pair, we compute the ipp and ippl by using the shortest path algorithm [6]. In order to get the shortest path between a node pair, we need edge weights but we have only node weights now. Therefore, we obtain the edge weights from node weights. The weight of an edge is obtained by summing weights of nodes on which the edge is incident. An example guery graph having both node weights and edge weights for a 5-way join is shown in Figure 3(a). We call this query graph *maximum weighted query graph*.

When there is no direct predicate between two nodes S and D in a maximum weighted query graph, the ipp and ippl between S and D can be obtained as follows: First, we calculate the shortest path and shortest path length per axis. Next, we subtract the weights of both S and D from the shortest path length and then divide the shortest path length by 2. This is because we want to get the sum of the weights of intermediated nodes in the shortest path, but the weights of S and D are included in the edge weights of the shortest path length, and the weights of the intermediated node are included twice. Therefore, the x_ippl

between S and D can be calculated by Expression (1). The y_ippl is similarly defined.

$$x\_ippl(S, D) = (x\_shortest\_path\_length(S, D) - x\_max(S) - x\_max(D))/2 \quad (1)$$

The ipp's and ippl's for all indirect predicate pairs in Figure 3(a) are shown in Figure 3(b). In Figure 3, the x_ipp and y_ipp are different for indirect predicate pairs AD and AE.




(a)

| pairs | x_ipp | x_ippl | y_ipp | y_ippl |
|---|---|---|---|---|
| AD | ABD | 20 | ACD | 50 |
| BC | BDC | 10 | BDC | 10 |
| BE | BDE | 10 | BDE | 10 |
| AE | ABDE | 30 | ACE | 50 |

(b)

**Fig. 3.** Maximum weighted query graph

The indirect predicates can be simultaneously checked with the additional predicates in *SortedIntersectionTest_1* of Algorithm *MwayRtreeJoin_1*. We call the algorithm doing the indirect predicate filtering *MwayRtreeJoin_2*.

**3) Maximum Tagged R-Trees** Until now, we have used only one max x-length and y-length per relation. In this case, if there are several extremely large objects in a relation although other objects are not so large, the effect of indirect predicates can be considerably degenerated. One possible solution for this is to have the max x-length and y-length per R-tree node. A leaf node has the max x-length and y-length for MBRs of all entries in the node, and an intermediate node has the maximum value for the max x-lengths and max y-lengths of its child nodes. In the end, the root node has the max x-length and max y-length for the relation. The max x-length per R-tree node is recursively defined as in Expression (2). The max y-length is similarly defined.

$$x\_max(N) = \begin{cases} \max\{N_1.rect_x \ldots N_n.rect_x\} & \text{for leaf node} \\ \max\{x\_max(N_1.ref) \ldots x\_max(N_n.ref)\} & \text{for intermediate node} \end{cases} \quad (2)$$

where $n$ is the number of entries in node $N$.

We call the max x-length and y-length per relation *domain max information* and those per R-tree node *node max information*. By using the node max information instead of the domain max information, we can have more prunning effects in indirect predicate filtering of the M-way R-tree join. Since only two max values are attached per R-tree node (one for x-length and the other for y-length), we can ignore the storage overhead due to the max lengths. And since the max lengths can be dynamically maintained with the R-tree insertion and deletion, we can always have exact max lengths per R-tree node. We call this R-tree *maximum tagged R-tree*.

We get only once the ipp's for each axis using the max information in root nodes of R-trees because calculating the shortest path for every node-tuple needs a large CPU time overhead[4]. However, we get the ippl's for every node-tuple based on the ipp's obtained from the root nodes. We call the algorithm using maximum tagged R-trees *MwayRtreeJoin_3*.

## 4 Experiments

To measure the performance of the M-way R-tree joins, we conducted some experiments using real data sets. The experiments were performed on a Sun Ultra II 170 MHz platform on which Solaris 2.5.1 was running with 384 MB of main memory. We implemented the three M-way R-tree join algorithms: *MwayRtreeJoin_1* (MRJ1), *MwayRtreeJoin_2* (MRJ2) and *MwayRtreeJoin_3* (MRJ3). For performance comparisons, we also implemented the multi-level forward checking (MFC) algorithm with the dynamic variable ordering (DVO) and the join window reduction (JWR) algorithm which were proposed in [15,16]. Additionally, we implemented another MFC algorithm (MFC1) which uses our space restriction ordering (SRO) as well as FC-DVO to check the pure effect of SRO.

The real data in our experiments were extracted from the TIGER/Line data of US Bureau of the Census [22]. We used the road segment data of 10 counties of the California State in the TIGER data. The characteristics (statistical information) of the California TIGER data are summarized in Table 1. The original TIGER data of all counties were center-matched to join different county regions, i.e., the x and y coordinates of the original TIGER data were subtracted from those of the center point of each county. The center-matched data were divided by 10 for easy handling.

We implemented the insertion algorithm in [2] to build R*-trees for each county data. The node sizes of the R*-trees considered are 512, 2048 and 4096 bytes. The tree heights for all county data for each node size are 4, 3 and 3, respectively. The LRU buffers are 256 pages in every node size[5].

[4] The complexity of computing all pair's shortest paths is known to be $O(M^3)$ [6].

[5] We assume that an R*-tree node occupies one page.

**Table 1.** Characteristics of the California TIGER data

| county | # of obj | domain area | max length | avg length | density |
|---|---|---|---|---|---|
| Alameda | 49070 | 86222*44995 | 4662*3940 | 102*80 | 0.23 |
| Contra Costa | 40363 | 88025*33808 | 4676*5112 | 100*77 | 0.21 |
| Fresno | 58163 | 233238*151898 | 7190*4633 | 210*167 | 0.09 |
| Kern | 113407 | 257781*100758 | 8204*6497 | 212*169 | 0.26 |
| Monterey | 35417 | 175744*112068 | 9085*6194 | 234*192 | 0.20 |
| Orange | 91970 | 69999*55588 | 3658*6735 | 80*66 | 0.21 |
| Riverside | 91751 | 323725*65389 | 12113*10062 | 158*126 | 0.21 |
| Sacramento | 46516 | 75771*71218 | 6442*4103 | 111*86 | 0.24 |
| San Diego | 103420 | 151241*96476 | 8054*6828 | 122*104 | 0.22 |
| Santa Barbara | 64037 | 99301*58696 | 4541*6460 | 100*81 | 0.22 |

We selected the following 4 query types as input queries: complete, half, ring and chain. Example query graphs for each query type in a 5-way join are shown in Figure 4. The spatial predicate used for our experiments is *intersect*.

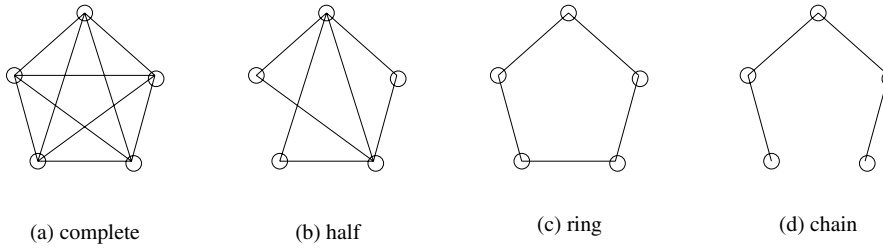



**Fig. 4.** Example query graphs in a 5-way join

First, we measured the total response time (CPU time + I/O time) for various data sets and various query types, and a fixed node size of 2048 bytes. The total response time was measured by "the elapsed CPU time + the number of I/O * the unit I/O time." The unit I/O time was set to 10 ms which is a typical value for a random I/O [7,15]. For this experiment, we extracted the following three data sets from the TIGER data shown in Table 1. An M-way join for each data set was performed for the first M counties of the data set.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Data set 1: | Ora. | Sac. | S.B. | S.D. | Ala. | Kern | Riv. |
| Data set 2: | Ora. | Ala. | Sac. | S.D. | S.B. | Kern | Mon. |
| Data set 3: | C.C. | S.B. | Mon. | Ora. | Sac. | Ala. | Fre. |

The total response time is shown in Table 2. The relative rates of the total response time compared to Algorithm *MwayRtreeJoin_1* (MRJ1) are shown in Figure 5 (only for the algorithms using the synchronous traversal (ST) technique). The numbers of solutions for each data set are also shown in Table 3.

First, we compared the relative performances among the ST algorithms such as MFCs and MRJs. In most cases, SRO considerably reduces the query response time (Compare MFC and MFC1 in Table 2 and Figure 5). FC-DVO has a better performance in complete and half queries while PSO has a better performance in the chain query. In the ring query, both have a similar performance or FC-DVO has a slightly better performance. (Compare MFC1 and MRJ1 in Table 2 and Figure 5.) The reason FC-DVO has a better performance in the complete and half queries is because FC-DVO prunes the entries of the future variables faster with many predicates while PSO does not prune the entries of the outer variables until they are actually evaluated. Since the chain and ring queries are more general in real life and more time consuming than other queries, we think that the optimization for these queries is more important. (According to Table 2, the differences of the query response time between MFC1 and MRJ1 in the complete and half queries are within 10 seconds, but the differences in the chain query reach about 1000 seconds.)

Sometimes, in data set 3, MFC1 does not work as well as MFC. This is due to the locality of LRU buffers and the CPU overhead of SRO. We observed that, in these cases, while MFC1 accessed fewer nodes, MFC performed a smaller or similar number of I/O's. However, in most cases, MFC1 performed a smaller number of I/O's.

Next, we measured the performance of indirect predicate filtering (IPF). In this measurement, we excluded the complete query type because no indirect predicates are in the complete query. In the half query, there is nearly no effect of indirect predicates (Compare MRJ1 and MRJ2 in Table 2 and Figure 5). We do not present the effect of the maximum tagged R-tree (MRJ3) in the half query because it is similar to that of MRJ2 in most cases. IPF has considerable impact on ring and chain queries. As the number of direct predicates decreases, the effect of indirect predicates increases. In summary, the three optimization techniques (SRO, PSO and IPF) improve efficiency. The maximum improvements compared to MFC are about 40%, 80%, 140% and 300%, respectively, for the complete, half, ring and chain queries.

A little later than the early version of this paper [19], other optimization techniques called *static variable ordering* (SVO) and *plane sweep and forward checking* (PSFC) were developed [13]. SVO orders the variables (or nodes) once according to the degrees before the algorithm starts. This static ordering is used both for the search space restriction and the forward checking. PSFC works as follows: The first variable is instantiated by a plane sweep, and a variant of the forward checking, called *sorted forward checking*, is used for the instantiations of remaining variables according to SVO. We believe that SRO is superior to SVO because it uses more sophisticated criteria. Actually, the experimental results in Table 2 and Figure 5 support our opinion. In complete and ring query graphs, the space restriction using SVO is the same as Algorithm *SpaceRestriction_1* used in MFC because the degrees of all nodes are the same. Since the experimental results show that MFC1 outperforms MFC in most cases, SRO will be superior to SVO. On the other hand, when there are many direct predicates, PSFC will

**Table 2.** Total response time for various data sets (node size: 2048, unit: sec)

| | | Data set 1 | | | | | Data set 2 | | | | | Data set 3 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Complete | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 17 | 19 | 10 | 13 | 13 | 8 | 9 | 11 | 14 | 11 | 14 | 21 | 10 | 11 | 15 |
| | MFC1 | 14 | 14 | 8 | 11 | 11 | 6 | 7 | 8 | 11 | 9 | 13 | 22 | 9 | 10 | 14 |
| | MRJ1 | 15 | 17 | 8 | 11 | 12 | 6 | 8 | 9 | 12 | 9 | 13 | 23 | 9 | 10 | 14 |
| | JWR | 36 | 57 | 24 | 25 | 24 | 16 | 22 | 24 | 25 | 17 | 41 | 72 | 39 | 41 | 43 |
| Half | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 22 | 36 | 22 | 28 | | 11 | 21 | 24 | 21 | | 34 | 14 | 26 | 41 | |
| | MFC1 | 19 | 20 | 18 | 21 | | 10 | 14 | 19 | 16 | | 36 | 13 | 18 | 30 | |
| | MRJ1 | 21 | 20 | 20 | 22 | | 11 | 13 | 19 | 17 | | 46 | 14 | 17 | 33 | |
| | MRJ2 | 21 | 20 | 20 | 23 | | 11 | 13 | 20 | 17 | | 47 | 15 | 17 | 33 | |
| | JWR | 78 | 33 | 56 | 285 | | 74 | 29 | 29 | 72 | | 223 | 194 | 58 | 223 | |
| Ring | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 25 | 26 | 106 | 710 | | 12 | 26 | 83 | 295 | | 34 | 45 | 122 | 649 | |
| | MFC1 | 20 | 21 | 81 | 461 | | 10 | 21 | 65 | 213 | | 36 | 34 | 99 | 500 | |
| | MRJ1 | 22 | 21 | 77 | 470 | | 11 | 22 | 68 | 196 | | 37 | 36 | 119 | 623 | |
| | MRJ2 | 22 | 20 | 69 | 364 | | 11 | 22 | 63 | 176 | | 38 | 35 | 111 | 397 | |
| | MRJ3 | 21 | 19 | 63 | 301 | | 11 | 22 | 60 | 152 | | 40 | 34 | 101 | 333 | |
| | JWR | 198 | 159 | 1048 | 228 | | 41 | 97 | 172 | 217 | | 172 | 171 | 209 | 183 | |
| Chain | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 26 | 81 | 335 | 1469 | 3805 | 11 | 32 | 166 | 939 | 2105 | 21 | 251 | 191 | 1738 | 7851 |
| | MFC1 | 24 | 69 | 244 | 969 | 2363 | 9 | 26 | 114 | 632 | 1428 | 19 | 223 | 162 | 1256 | 5396 |
| | MRJ1 | 23 | 62 | 181 | 818 | 2026 | 8 | 22 | 92 | 538 | 1198 | 18 | 184 | 123 | 896 | 4324 |
| | MRJ2 | 23 | 57 | 147 | 580 | 1341 | 8 | 20 | 76 | 396 | 859 | 18 | 196 | 127 | 779 | 3218 |
| | MRJ3 | 23 | 55 | 131 | 454 | 954 | 8 | 20 | 67 | 342 | 719 | 18 | 201 | 126 | 655 | 2815 |
| | JWR | 40 | 212 | 455 | 1091 | 1557 | 17 | 44 | 106 | 185 | 426 | 58 | 277 | 2681 | 841 | 1276 |

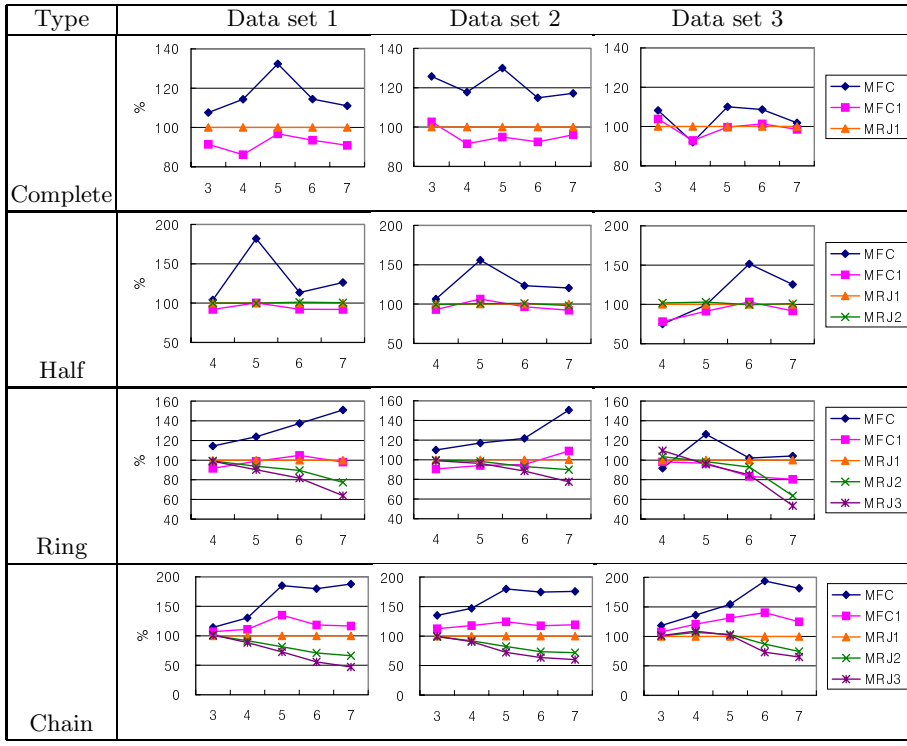



**Fig. 5.** Rates of total response time for various data sets (node size: 2048)

**Table 3.** Number of solutions for various data sets

| M | | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| Data Set 1 | Complete | 16,156 | 3,893 | 435 | 192 | 31 |
| | Half | | 18,897 | 25,578 | 881 | 128 |
| | Ring | | 11,590 | 7,098 | 12,298 | 4,220 |
| | Chain | 131,759 | 329,855 | 440,945 | 475,497 | 81,419 |
| Data Set 2 | Complete | 4,733 | 1,719 | 435 | 192 | 77 |
| | Half | | 15,327 | 2,371 | 825 | 318 |
| | Ring | | 5,209 | 5,916 | 8,724 | 19,574 |
| | Chain | 23,155 | 61,446 | 56,295 | 254,505 | 177,627 |
| Data Set 3 | Complete | 23,188 | 21,880 | 2,506 | 725 | 152 |
| | Half | | 232,068 | 6,152 | 4,558 | 2,074 |
| | Ring | | 161,611 | 72,346 | 51,600 | 42,238 |
| | Chain | 102,327 | 2,753,856 | 530,673 | 1,271,835 | 3,441,939 |

naturally outperform PSO because our experiments show that MFC outperforms PSO for numerous direct predicates.

Next, we compared the query response time between ST algorithms and JWR. As the variable instantiation order of JWR, we used the same as in PSO. According to the result shown in Table 2, ST algorithms have better performances in all Ms of complete and half queries and in most Ms of other queries. When M is high (6 or 7), JWR has a better performance than MFC for some data sets in ring and chain queries, which is similar to the result in [16]. There are some cases that JWR has a better performance than MFC for some data sets, but has a worse performance than MRJs. For example, see Table 2 for M=6,7 and data set 1, M=5 and data set 2, and M=6 and data set 3 in the chain query. Therefore, unlike the experimental results in [15,16], we can use our M-way R-tree join algorithms for a higher range of M.

Sometimes, the costs of JWR are abruptly increased (for example, M=7 in the half query of data set 1, M=6 in the ring query of data set 1, and M=5 in the chain query of data set 3). We think this is due to the evaluation order of variables. While real data sets are highly skewed, PSO does not consider the data distribution. However, the variable ordering worked properly in most other cases.

Next, we conducted an experiment for various node sizes. Table 4 shows the total response time of all algorithms for various node sizes and a fixed data set 2. Figure 6 illustrates the performance rates of the total response time compared to MRJ1. According to Figure 6, SRO has large effects in most cases. And the smaller the node size is, the better the performance of FC-DVO is. In other words, the larger the node size, the better the performance of PSO. (See the performance rate of MFC1 compared to MRJ1.) In particular, PSO has a better performance than FC-DVO for node size 4096 of the ring query although both have a similar performance for node size 2048. As the node size increases, the effect of IPF slightly decreases in ring and chain query types. When the node size is 4096, there is nearly no difference between the effect of indirect predicates using domain max information (MRJ2) and that using node max information

**Table 4.** Total response time for various node sizes (data set 2, unit: sec)

| | | 512 | | | | | 2048 | | | | | 4096 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Complete | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 29 | 32 | 34 | 47 | 31 | 8 | 9 | 11 | 14 | 11 | 5 | 6 | 7 | 10 | 9 |
| | MFC1 | 20 | 23 | 24 | 36 | 26 | 6 | 7 | 8 | 11 | 9 | 4 | 5 | 6 | 9 | 8 |
| | MRJ1 | 20 | 24 | 28 | 40 | 28 | 6 | 8 | 9 | 12 | 9 | 4 | 5 | 6 | 9 | 8 |
| | JWR | 39 | 49 | 47 | 50 | 34 | 16 | 22 | 24 | 25 | 17 | 14 | 20 | 22 | 23 | 16 |
| Half | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 38 | 77 | 77 | 65 | | 11 | 21 | 24 | 21 | | 8 | 17 | 20 | 21 | |
| | MFC1 | 32 | 50 | 63 | 54 | | 10 | 14 | 19 | 16 | | 7 | 11 | 15 | 15 | |
| | MRJ1 | 34 | 54 | 65 | 56 | | 11 | 13 | 19 | 17 | | 8 | 11 | 14 | 14 | |
| | MRJ2 | 33 | 51 | 68 | 59 | | 11 | 13 | 20 | 17 | | 7 | 11 | 15 | 15 | |
| | JWR | 148 | 57 | 60 | 121 | | 74 | 29 | 29 | 72 | | 69 | 26 | 29 | 71 | |
| Ring | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 42 | 87 | 281 | 791 | | 12 | 26 | 83 | 295 | | 9 | 21 | 78 | 323 | |
| | MFC1 | 32 | 66 | 236 | 622 | | 10 | 21 | 65 | 213 | | 8 | 17 | 62 | 222 | |
| | MRJ1 | 34 | 73 | 243 | 629 | | 11 | 22 | 68 | 196 | | 7 | 14 | 51 | 189 | |
| | MRJ2 | 34 | 70 | 222 | 537 | | 11 | 22 | 63 | 176 | | 7 | 14 | 47 | 155 | |
| | MRJ3 | 34 | 63 | 186 | 423 | | 11 | 22 | 60 | 152 | | 7 | 14 | 47 | 149 | |
| | JWR | 81 | 172 | 273 | 371 | | 41 | 97 | 172 | 217 | | 43 | 108 | 200 | 246 | |
| Chain | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 41 | 93 | 472 | 3521 | 6692 | 11 | 32 | 166 | 939 | 2105 | 8 | 27 | 147 | 774 | 1816 |
| | MFC1 | 30 | 66 | 322 | 2383 | 4702 | 9 | 26 | 114 | 632 | 1428 | 7 | 23 | 107 | 528 | 1177 |
| | MRJ1 | 30 | 69 | 303 | 2309 | 4647 | 8 | 22 | 92 | 538 | 1198 | 5 | 14 | 62 | 362 | 876 |
| | MRJ2 | 29 | 65 | 252 | 1687 | 3467 | 8 | 20 | 76 | 396 | 859 | 5 | 13 | 48 | 259 | 600 |
| | MRJ3 | 28 | 59 | 180 | 1027 | 2064 | 8 | 20 | 67 | 342 | 719 | 5 | 13 | 48 | 263 | 592 |
| | JWR | 40 | 85 | 168 | 292 | 604 | 17 | 44 | 106 | 185 | 426 | 15 | 45 | 113 | 210 | 446 |

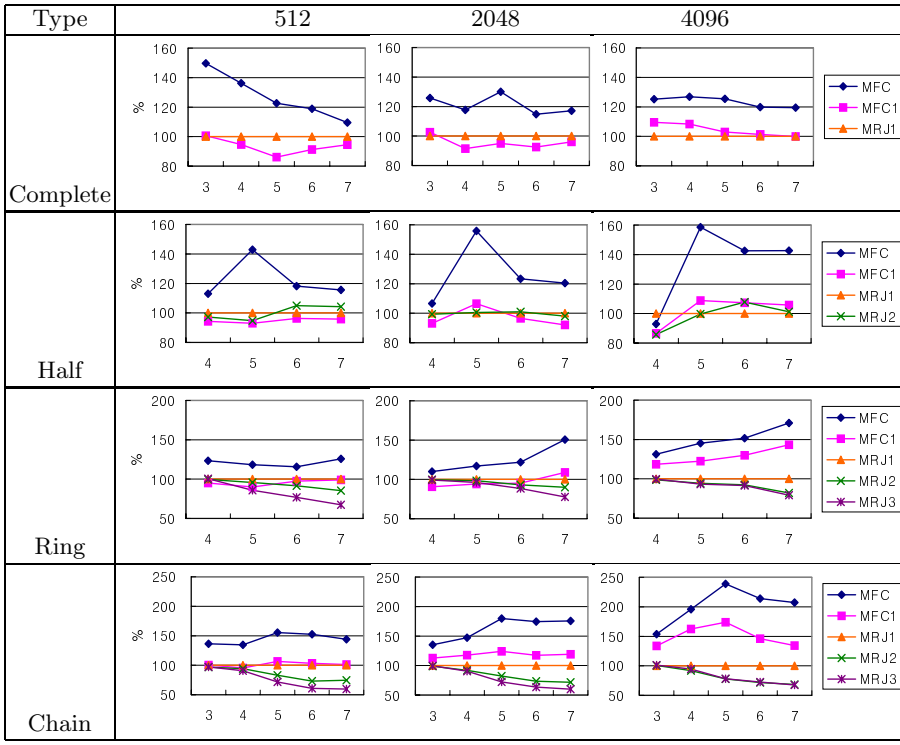



**Fig. 6.** Rates of total response time for various node sizes (data set 2)

(MRJ3). This is because the large node size leads to many entries per node and increases the node max information. However, still for a large node size (4096), the effect of IPF is large for the ring and chain queries.

The comparison between ST algorithms and JWR shows that ST algorithms perform better for large node sizes. This is due to the index probing overhead in JWR. Since there is no global ordering in multi-dimensional non-point objects, we should check all entries of a node during an R*-tree search. In addition, while ST algorithms have the best performance for all query types in node size 4096 compared to other node sizes, JWR has the best performance for the ring and chain queries in node size 2048.

Finally, we measured the I/O time (see Table 5 and Table 6). MRJs consume more I/O time than MFCs and JWR in high Ms. From Table 5 and Table 6, however, we found an important fact: the higher the value of M, the lower the rate of I/O time compared to the total response time. For ring and chain queries, the rate of I/O time considerably decreases as M increases. Therefore, the I/O time becomes less important and the M-way R-tree join becomes CPU-bound. The I/O rate also decreases along the node size. (According to Table 6, when the node size is 4096 and M is 7, the I/O rates in the ring and chain query types are less than or equal to 5%.)

**Table 5.** I/O time for data set 2 (node size: 2048)

| | | # of I/O | | | | | I/O rate (%) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Complete | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 553 | 629 | 712 | 787 | 563 | 72 | 69 | 64 | 57 | 52 |
| | MFC1 | 439 | 469 | 495 | 612 | 448 | 70 | 66 | 61 | 55 | 51 |
| | MRJ1 | 435 | 542 | 537 | 689 | 476 | 71 | 70 | 63 | 57 | 52 |
| | JWR | 575 | 631 | 609 | 657 | 541 | 35 | 29 | 25 | 26 | 31 |
| Half | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 692 | 799 | 835 | 617 | | 61 | 39 | 35 | 30 | |
| | MFC1 | 564 | 601 | 765 | 561 | | 57 | 43 | 41 | 35 | |
| | MRJ1 | 604 | 595 | 845 | 694 | | 56 | 45 | 44 | 40 | |
| | MRJ2 | 602 | 595 | 841 | 649 | | 57 | 45 | 43 | 38 | |
| | JWR | 1429 | 602 | 739 | 1039 | | 19 | 21 | 25 | 14 | |
| Ring | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 720 | 858 | 1076 | 955 | | 58 | 33 | 13 | 3 | |
| | MFC1 | 567 | 774 | 1127 | 1193 | | 55 | 37 | 17 | 6 | |
| | MRJ1 | 720 | 1036 | 1776 | 1308 | | 64 | 46 | 26 | 7 | |
| | MRJ2 | 716 | 1040 | 1714 | 1149 | | 64 | 47 | 27 | 7 | |
| | MRJ3 | 716 | 1029 | 1671 | 1090 | | 64 | 48 | 28 | 7 | |
| | JWR | 652 | 714 | 777 | 763 | | 16 | 7 | 5 | 4 | |
| Chain | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 632 | 775 | 956 | 1347 | 1467 | 57 | 24 | 6 | 1 | 1 |
| | MFC1 | 520 | 685 | 986 | 2014 | 2662 | 56 | 27 | 9 | 3 | 2 |
| | MRJ1 | 514 | 845 | 1316 | 3351 | 6367 | 62 | 39 | 14 | 6 | 5 |
| | MRJ2 | 514 | 829 | 1224 | 2800 | 4501 | 63 | 41 | 16 | 7 | 5 |
| | MRJ3 | 514 | 821 | 1212 | 2510 | 4176 | 62 | 41 | 18 | 7 | 6 |
| | JWR | 584 | 654 | 704 | 764 | 794 | 34 | 15 | 7 | 4 | 2 |

In overall summary, we recommend the following based on the experimental results: First, always use SRO. Second, if there are many direct predicates as in

**Table 6.** I/O time for various node sizes (data set 2)

| | | # of I/O (512) | | | | | I/O rate (%) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Complete | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 2563 | 2679 | 2655 | 3396 | 1796 | 87 | 82 | 78 | 72 | 59 |
| | MFC1 | 1658 | 1782 | 1733 | 2372 | 1402 | 84 | 79 | 73 | 65 | 53 |
| | MRJ1 | 1653 | 1902 | 2091 | 2687 | 1536 | 84 | 80 | 75 | 67 | 55 |
| | JWR | 2382 | 2591 | 2287 | 2449 | 1629 | 61 | 53 | 48 | 49 | 48 |
| Half | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 2931 | 3414 | 4216 | 2661 | | 76 | 45 | 55 | 41 | |
| | MFC1 | 2309 | 2285 | 3429 | 2287 | | 72 | 46 | 55 | 42 | |
| | MRJ1 | 2474 | 2427 | 3710 | 2564 | | 73 | 45 | 57 | 46 | |
| | MRJ2 | 2451 | 2427 | 3953 | 2701 | | 74 | 48 | 58 | 46 | |
| | JWR | 5702 | 2342 | 2887 | 4109 | | 39 | 41 | 48 | 34 | |
| Ring | M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| | MFC | 3106 | 4767 | 7918 | 8860 | | 74 | 55 | 28 | 11 | |
| | MFC1 | 2326 | 3528 | 7318 | 7644 | | 72 | 53 | 31 | 12 | |
| | MRJ1 | 2496 | 4332 | 8256 | 8258 | | 74 | 59 | 34 | 13 | |
| | MRJ2 | 2498 | 4032 | 7954 | 8039 | | 74 | 57 | 36 | 15 | |
| | MRJ3 | 2489 | 3773 | 7283 | 7229 | | 73 | 60 | 39 | 17 | |
| | JWR | 2673 | 2943 | 3214 | 3095 | | 33 | 17 | 12 | 8 | |
| Chain | M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| | MFC | 3031 | 4740 | 7858 | 18627 | 29918 | 73 | 51 | 17 | 5 | 4 |
| | MFC1 | 2239 | 3249 | 6149 | 16064 | 23149 | 74 | 49 | 19 | 7 | 5 |
| | MRJ1 | 2269 | 3732 | 5855 | 20176 | 34121 | 75 | 54 | 19 | 9 | 7 |
| | MRJ2 | 2255 | 3539 | 5659 | 17228 | 28152 | 77 | 55 | 22 | 10 | 8 |
| | MRJ3 | 2250 | 3388 | 5384 | 14865 | 21579 | 79 | 58 | 30 | 14 | 10 |
| | JWR | 2378 | 2690 | 2900 | 3184 | 3334 | 59 | 32 | 17 | 11 | 6 |

| | # of I/O (4096) | | | | | I/O rate (%) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| MFC | 298 | 331 | 365 | 420 | 383 | 59 | 56 | 50 | 41 | 40 |
| MFC1 | 240 | 266 | 295 | 356 | 290 | 55 | 53 | 49 | 41 | 37 |
| MRJ1 | 240 | 268 | 298 | 352 | 292 | 60 | 58 | 51 | 41 | 37 |
| JWR | 315 | 350 | 340 | 372 | 329 | 22 | 17 | 15 | 16 | 20 |
| M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| MFC | 339 | 415 | 446 | 388 | | 44 | 25 | 23 | 19 | |
| MFC1 | 301 | 341 | 400 | 376 | | 42 | 30 | 27 | 25 | |
| MRJ1 | 315 | 349 | 420 | 391 | | 38 | 33 | 31 | 27 | |
| MRJ2 | 315 | 349 | 515 | 384 | | 44 | 33 | 35 | 26 | |
| JWR | 689 | 333 | 413 | 527 | | 10 | 13 | 14 | 7 | |
| M | 4 | 5 | 6 | 7 | | 4 | 5 | 6 | 7 | |
| MFC | 371 | 439 | 549 | 551 | | 43 | 21 | 7 | 2 | |
| MFC1 | 321 | 410 | 583 | 663 | | 41 | 24 | 9 | 3 | |
| MRJ1 | 326 | 411 | 678 | 750 | | 49 | 28 | 13 | 4 | |
| MRJ2 | 325 | 402 | 752 | 704 | | 50 | 29 | 16 | 5 | |
| MRJ3 | 325 | 402 | 753 | 701 | | 49 | 30 | 16 | 5 | |
| JWR | 352 | 392 | 421 | 422 | | 8 | 4 | 2 | 2 | |
| M | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 |
| MFC | 345 | 404 | 519 | 874 | 1106 | 42 | 15 | 4 | 1 | 1 |
| MFC1 | 286 | 328 | 434 | 979 | 1338 | 40 | 15 | 4 | 2 | 1 |
| MRJ1 | 286 | 327 | 519 | 1041 | 1298 | 54 | 24 | 8 | 3 | 1 |
| MRJ2 | 286 | 327 | 504 | 1100 | 1172 | 53 | 26 | 10 | 4 | 2 |
| MRJ3 | 286 | 327 | 499 | 1097 | 1160 | 53 | 25 | 10 | 4 | 2 |
| JWR | 313 | 351 | 382 | 418 | 433 | 21 | 8 | 3 | 2 | 1 |

the complete and half queries, use FC-DVO and no IPF. Third, if the number of direct predicates is small as in the ring and chain queries, use PSO and IPF. Fourth, if the node size is small and M is high, use JWR; otherwise, use ST algorithms.

## 5 Conclusions

In this paper, we study the generalization of the 2-way R-tree join. We proposed the following three optimization techniques: space restriction ordering (SRO), plane sweep ordering (PSO) and indirect predicate filtering (IPF). Through experiments using real data, we showed that our three optimization techniques have a great impact on improving the performance of synchronous traversal (ST) algorithms.

After completing the M-way R-tree join, an oid pair may appear several times in the resulting oid-tuples. If the oid-tuples are read in the combined refinement step without scheduling, it may access the same page several times and perform the same refinement operation several times. However, this can be solved by extending scheduling methods for oid pairs such as [23] to oid-tuples. In future studies, first, we will develop an efficient combined refinement algorithm for the M-way spatial join. Second, although we found that the I/O rate of the total response time decreases as M increases, the I/O rate is still high for a small M. Therefore, we will develop I/O optimization techniques for the M-way R-tree join. Last, we will combine the optimization techniques proposed in this paper with our rule-based optimization technique for spatial and non-spatial mixed queries called ESFAR (Early Separated Filter And Refinement) [17,18].

## Acknowledgement

We would like to thank Dimitris Papadias and Nikos Mamoulis for their careful reading and valuable comments. We also thank the anonymous referees for their suggestions to improve the quality of this paper. This research was supported by the National Geographic Information Systems Technology Development Project and the Software Technology Enhancement Program 2000 of the Ministry of Science and Technology of Korea.

## References

1. L. Arge, O. Procopiue and S. Ramaswary, "Scalable Sweeping-Based Spatial Join," Proc. of VLDB, 570-581, 1998.
2. N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. of ACM SIGMOD, 322-331, 1990.
3. T. Brinkhoff, H.-P. Kriegel and B. Seeger, "Efficient Processing of Spatial Joins Using R-trees," Proc. of ACM SIGMOD, 237-246, 1993.
4. A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. of ACM SIGMOD, 47-57, 1984.
5. R. H. Güting, "An Introduction to Spatial Database Systems," VLDB Journal, Vol. 3, No. 4, 357-399, 1994.
6. E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms," Computer Science Press, 1978.
7. Y.-W. Huang, N. Jing and E. A. Rundensteiner, "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations," Proc. of VLDB, 396-405, 1997.
8. Y. E. Ioannidis and Y. C. Kang, "Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization," Proc. of ACM SIGMOD, 168-177, 1991.
9. N. Koudas and K. C. Sevsik, "Size Separation Spatial Join," Proc. of ACM SIGMOD, 324-355, 1997.
10. M. L. Lo and C. V. Ravishankar, "Spatial Joins Using Seeded Trees," Proc. of ACM SIGMOD, 209-220, 1994.
11. M. L. Lo and C. V. Ravishankar, "Spatial Hash-Joins," Proc. of ACM SIGMOD, 247-258, 1996.
12. N. Mamoulis and D. Papadias, "Integration of Spatial Join Algorithms for Processing Multiple Inputs," to appear in Proc. of ACM SIGMOD'99.
13. N. Mamoulis and D. Papadias, "Synchronous R-tree Traversal," Technical Report HKUST-CS99-03, 1999.
14. J. A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," Proc. of ACM SIGMOD, 326-336, 1986.
15. D. Papadias, N. Mamoulis and V. Delis, "Algorithms for Querying by Spatial Structure," Proc. of VLDB, 546-557, 1998.
16. D. Papadias, N. Mamoulis and Y. Theodoridis, "Processing and Optimization of Multi-way Spatial Joins Using R-trees," to appear in Proc. of ACM PODS'99.
17. H.-H. Park, C.-G. Lee, Y.-J. Lee and C.-W. Chung, "Separation of Filter and Refinement Steps in Spatial Query Optimization," KAIST, Technical Report, CS/TR-98-122, 1998. See also: http://islab.kaist.ac.kr/~hhpark/eng_tr_sfro.ps

18. H.-H. Park, C.-G. Lee, Y.-J. Lee and C.-W. Chung, "Early Separation of Filter and Refinement Steps in Spatial Query Optimization," Proc. of DASFAA, 161-168, 1999.
19. H.-H. Park, G.-H. Cha and C.-W. Chung, "Multi-way Spatial Joins Using R-trees: Methodology and Performance Evaluation," KAIST, Technical Report, CS/TR-99-135, 1999.
20. J. M. Patel and D. J. DeWitt, "Partition Based Spatial-Merge Join," Proc. of ACM SIGMOD, 259-270, 1996.
21. F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, 1985.
22. U.S. Bureau of the Census, Washington, DC., "TIGER/Line Files, 1995, Technical Documentation."
23. P. Valduriez, "Join Indices," ACM Transactions on Database Systems, Vol.12, No. 2, 218-246, 1987.