Algorithmica (2001) 30: 188–215
DOI: 10.1007/s00453-001-0005-y

Algorithmica

Constraint-Based Processing of Multiway Spatial Joins[1]

D. Papadias,[2] N. Mamoulis,[2] and Y. Theodoridis[3]

Abstract. A multiway spatial join combines information found in three or more spatial relations with respect to some spatial predicates. Motivated by their close correspondence with constraint satisfaction problems (CSPs), we show how multiway spatial joins can be processed by systematic search algorithms traditionally used for CSPs. This paper describes two different strategies, window reduction and synchronous traversal, that take advantage of underlying spatial indexes to prune the search space effectively. In addition, we provide cost models and optimization methods that combine the two strategies to compute more efficient execution plans. Finally, we evaluate the efficiency of the proposed techniques and the accuracy of the cost models through extensive experimentation with several query and data combinations.

Key Words. Spatial databases, Spatial joins, Constraint satisfaction, R-trees.

1. Introduction. Spatial DBMSs and GISs store large amounts of multidimensional data, such as points, lines, or polygons. Popular indexing methods used in relational databases (e.g., B-trees) are not directly applicable for spatial data due to the fact that there is no total ordering of objects in space that preserves proximity. As a result, a number of *multidimensional access methods* [11] have been successfully employed in several domains, including medical information systems [44] and time series databases [10]. The predominant access method for multidimensional data is the R-tree [14] and its variations, which are currently used in many commercial systems (e.g., Informix, Postgress, MapInfo). R-trees have been applied for processing several types of spatial selections such as window [14], relation-based [36], and nearest neighbor queries [47]. In addition, they are effective for (pairwise) spatial joins [5].

This paper deals with processing and optimization of multiway spatial joins using R-trees. A multiway spatial join can be defined as follows: Given a set of n spatial relations $\{R_1, \ldots, R_i, \ldots, R_j, \ldots, R_n\}$, where $R_i = \{u_1^i, \ldots, u_{N_i}^i\}$, and a set of binary spatial predicates $\{C_{ij} \mid 1 \leq i, j \leq n\}$, find all n-tuples $\{(u_p^1, \ldots, u_k^i, \ldots, u_l^j, \ldots, u_r^n) \mid \forall i, j, 1 \leq i, j \leq n, C_{ij}(u_k^i, u_l^j)\}$. In most cases the spatial predicate is *overlap* (*intersect*, *crosses*) but alternatively any predicate, such as *near*, *northeast*, *meet*, could be used. As

[1] A short version of this paper appears in the *Proceedings of the ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems* (*PODS*), Philadelphia, PA, pp. 44–55, 1999. Dimitris Papadias and Nikos Mamoulis were supported by Grants HKUST 6151/98E, HKUST 6090/99E and HKUST 6070/00E from Hong Kong RGC. Yannis Theodoridis was supported by the EC funded project "Chorochronos" under the TMR Programme.

[2] Department of Computer Science, Hong Kong University of Science and Technology, Clearwater Bay, Hong Kong.

[3] Computer Technology Institute, P.O. Box 1122, Patra 26110, Greece.

Received December 21, 1998; revised September 29, 1998. Communicated by M. van Kreveld.
Online publication March 9, 2001.

an example of a multiway spatial join consider the content-based query "find all cities *crossed by* a river which *crosses* an industrial area." The query can be thought of as a case of image similarity retrieval, where similarity is based on spatial predicates and not visual characteristics.

If n is the number of query variables and N is the size of their domain, then in the worst case all n-combinations (or n-permutations if all variables have the same domain) of N objects have to be searched to find configurations that satisfy the query constraints. In order to avoid the large cost of processing, previous image similarity techniques that do not use indexing (e.g., [13] and [30]) have focused on a specific instance of the problem where small images consist of the same set of known (labeled) objects. Petrakis and Faloutsos [44] employ R-trees to solve such queries for images that contain a constant number of labeled objects (e.g., lungs) and a small number of unlabeled ones (e.g., tumors). Although their method is efficient for domains involving numerous small images with few unlabeled objects (e.g., medical databases of X-rays), it is not applicable to large images of unlabeled objects.

This paper proposes a solution to the general problem of multiway spatial joins, where large datasets contain arbitrary numbers of unlabeled objects. The next section outlines the R-trees and the most common types of spatial query processing, i.e., spatial selections and joins. Motivated by a close correspondence between multiway joins and constraint satisfaction problems, we describe, in Section 3, techniques to process multiway spatial joins by integrating systematic search algorithms with R-trees. Section 4 discusses the selectivity of multiway spatial joins and provides analytical cost models. Section 5 describes optimization algorithms based on exhaustive and local search in the space of alternative execution plans. Section 6 evaluates the proposed techniques with extensive experimentation using various datasets and join graph topologies, and Section 7 concludes the paper.

2. Overview of Spatial Query Processing Using R-Trees. The R-tree data structure is a height-balanced tree that consists of intermediate and leaf nodes corresponding to disk pages in secondary memory (R-trees are extensions of B^+-trees [7] to many dimensions). The root is at level $h-1$, where h is the height of the tree, and the leaf nodes are at level 0. The Minimum Bounding Rectangles (MBRs) of the data objects are stored in the leaf nodes and intermediate nodes are built by grouping MBRs of the lower level. We make the distinction between an R-tree node $N[i]$ and its entries s_k, which correspond to MBRs included in $N[i]$; $s_k.ref$ points to the corresponding node $N[k]$ at the next (lower) level. Each R-tree node (except from the root) should contain at least a number m of entries, called minimum R-tree node utilization. Figure 1 illustrates three relations (covering the same workspace) and the corresponding R-trees assuming that $m = 2$ and maximum node capacity C is three rectangles (in real two-dimensional applications C is normally 50–400 depending on the page size). R^+-trees [48] and R^*-trees [3] are improved versions of the original method, proposed to address the problem of performance degradation caused by the overlapping regions and excessive dead-space.

Selection and join queries are fundamental operations in any DBMS. In this section we briefly present the techniques employed by query processors to support spatial selections and joins using R-trees, and describe related analytical models.

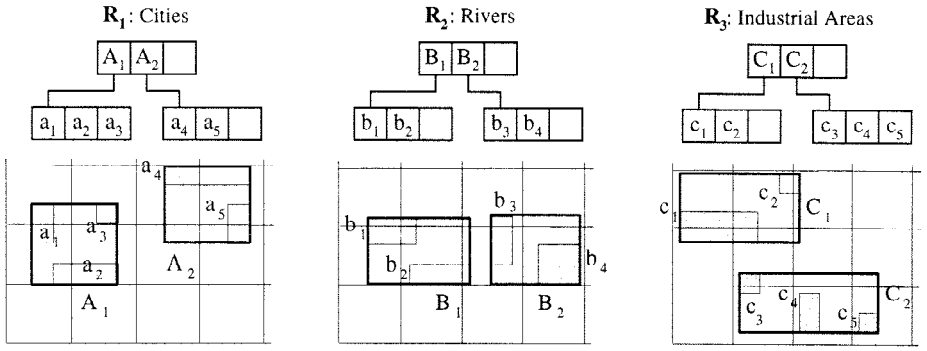


Fig. 1. R-trees.

2.1. *Selection Queries*. A spatial selection retrieves from a dataset, the entries that satisfy some spatial predicate with respect to a reference object q. The most common type of spatial selections are *window queries*, where the predicate is *overlap* and q defines a rectangular window in the workspace. The processing of a window query using R-trees involves the procedure of Figure 2: starting from the root node, exclude the entries that are disjoint with the query window, and recursively search the remaining ones. If, for instance, we are looking for all rivers that intersect city a_1, we retrieve the root entries of the second tree that overlap a_1 (in this case B_1). Then we search inside B_1 for potential solutions (no objects in B_2 can overlap a_1 and the node is not accessed).

When the MBRs of two objects are *disjoint*, the objects that they approximate are also *disjoint*. If the MBRs however share common points, no conclusion can be drawn about the spatial relation between the objects. For this reason, spatial queries involve the following two-step strategy [33]: (i) A *filter step* uses the tree to eliminate rapidly objects that could not possibly satisfy the query. The result of this step is a set of candidates which includes all the results and possibly some false hits. (ii) During a *refinement step* each candidate is examined (by using computational geometry techniques) and false hits are eliminated.

This paper, like most related spatial database literature, focuses on minimizing the cost of filtering. Performance is usually measured in terms of the number of R-tree nodes that should be accessed during the search process. Let d be the dimensionality of the data space and let $[0, 1)^d$ be the d-dimensional unit workspace. Given an R-tree R_i (with height h_{R_i}) and a window q (with $|q|$ average extent on each dimension), the *selectivity*

```
1.  WQ(Rtree_Node N[i], window q)
2.  FOR all s_k ∈ N[i] with s_k ∩ q ≠ ∅ DO
3.      IF N[i] is a leaf node
4.          THEN output (s_k)
5.          ELSE      /* intermediate nodes */
6.              ReadPage(s_k.ref)
7.              WindowQuery(N[k], q)
```

Fig. 2. Window Query (*WQ*) algorithm.

$S(R_i, q, l)$ of q on the entries of R_i at level l is defined as *the ratio of the expected number of entries overlapping q over their total number* (i.e., the probability that a random entry intersects q). Theodoridis and Sellis [52] provide the following formula for selectivity:

$$S(R_i, q, l) = (|s_{R_i,l}| + |q|)^d, \tag{2.1}$$

where $|s_{R_i,l}|$ is the average extent (on each dimension) of an entry $s_{R_i,l}$ of the R-tree R_i at level l. The above formula assumes (i) unit workspace, (ii) square node rectangles, which is a desirable property for "efficient" R-trees [3], [21], (iii) uniformly distributed centers of node rectangles, the so-called "uniformity assumption," and (iv) independent dimensions. These assumptions hold for the analytical formulas presented throughout the paper.

The number $NA(R_i, q, l)$ of node accesses for retrieving entries at level l equals the number of entries intersected by q in the upper level $l + 1$, i.e., the total number of entries at level $l + 1$ (denoted by $N_{R_i,l+1}$) times the probability that an entry intersects q (selectivity):

$$\begin{aligned} NA(R_i, q, l) &= N_{R_i,l+1} \cdot S(R_i, q, l + 1) \\ &= N_{R_i,l+1} \cdot (|s_{R_i,l+1}| + |q|)^d, \qquad 0 \le l \le h - 2. \end{aligned} \tag{2.2}$$

In the previous examples (rivers intersecting city a_1), the number of node accesses depends on how many root entries of the second tree intersect a_1 and not on the number of river MBRs. The total cost of a window query $Cost_{WQ}$ is the sum of node accesses at each level, i.e., the number of entries that intersect q at all intermediate levels plus the access of the root:

$$Cost_{WQ}(R_i, q) = 1 + \sum_{l=0}^{h_{R_1}-2} NA(R_i, q, l) = 1 + \sum_{l=1}^{h_{R_1}-1} N_{R_i,l} \cdot (|s_{R_i,l}| + |q|)^d. \tag{2.3}$$

This formula is based on the performance analysis of [35]. Theodoridis and Sellis [52] define the R-tree properties h_{R_i}, $N_{R_i,l}$, and $|s_{R_i,l}|$ involved in (2.3) as functions of the cardinality and density[4] of the dataset, thus computing $NA(R_i, q, l)$ and $Cost_{WR}(R_i, q)$ by using only data properties, without extracting information from the underlying R-tree structure. Pagel and Six [34] argue that window queries are representative for range queries in general. Papadias et al. [37] show how the above formulas can be applied for any spatial predicate including topological (e.g., inside, meets), direction (e.g., north), and distance relations.

2.2. *Spatial Joins*. A spatial join operation selects from two object sets, the pairs that satisfy some spatial predicate, usually *intersect* (e.g., "find all cities that are *crossed by* a river"). Previous work on pairwise spatial join queries can be classified in two categories. The methods of the first category treat nonindexed inputs (e.g., when there is another operation, such as selection, before the spatial join). If there is an R-tree for only one input,

[4] The density of a set of rectangles is defined as the average number of rectangles that contain a given point in the workspace. Equivalently, density can be expressed as the ratio of the sum of the areas of all rectangles over the area of the available workspace.

processing can be done by (i) index nested loops, (ii) building a second R-tree for the nonindexed input using bulk loading [54] and then applying an R-tree-based algorithm (see below), (iii) the *sort and match* algorithm [40], (iv) the *seeded tree* algorithm [24] which works like (ii) but builds the second R-tree using the existing one as a skeleton (seed), and (v) the *slot index spatial join* [27] which is an improved version of (iv). If both inputs are nonindexed, some methods partition the space either regularly [23], [43] or irregularly [25], and distribute the data objects into buckets defined by these partitions. The spatial join is then performed in a relational hash join fashion. Another method [1] first applies external sorting to both files and then uses an adaptable plane sweep algorithm, considering that in most cases the "horizon" of the sweep line will fit in the main memory.

The methods of the second category are applicable when both relations to be joined are indexed on the spatial attributes. The most influential technique in this category is *R-tree-based Join* (*RJ*) [5], which presupposes the existence of R-trees for both relations. *RJ* is based on the *enclosure property*: if two intermediate R-tree nodes do not intersect, there can be no MBRs below them that intersect. Assume that we want to retrieve all pairs of overlapping cities and rivers in Figure 1. The algorithm starts from the roots of the two trees to be joined and finds all pairs of overlapping entries inside them (e.g., (A_1, B_1), (A_2, B_2)). These are the only pairs that may lead to solutions; for instance, there cannot exist any (a_i, b_j) $a_i \in A_1$ and $b_j \in B_2$ such that (a_i, b_j) is a solution, because A_1 does not overlap B_2. For each overlapping pair of intermediate entries, the algorithm is recursively called until the leaf levels. Figure 3 illustrates the pseudocode for *RJ* assuming that the trees are of equal height; the extension to different heights is straightforward.

Two optimization techniques can be used to improve the CPU speed of *RJ*. The first, *search space restriction*, reduces the quadratic number of pairs to be evaluated when two nodes $N[i]$, $N[j]$ are joined. If an entry $s_k \in N[i]$ does not intersect the MBR of $N[j]$ (that is the MBR of all entries contained in $N[j]$), then there can be no entry $s_l \in N[j]$, such that s_k and s_l overlap. In the example of Figure 1, entry a_4 of node A_2 does not intersect node B_2, so it cannot intersect any entry inside B_2. Using this observation, space restriction performs two linear scans in the entries of both nodes before starting the *RJ* procedure, and prunes out from each node the entries that do not intersect the MBR of the other node. The second technique, based on the *plane sweep* paradigm [45], applies sorting in one dimension in order to reduce the computation time of the overlapping pairs between the nodes to be joined. Huang et al. [17] extend *RJ* by introducing an on-the-fly indexing mechanism to optimize, in terms of I/O cost, the execution order of matching at intermediate levels. Papadias et al. [38] shows how *RJ* and related heuristics can be applied for a variety of spatial predicates.

```
1. RJ(Rtree_Node N[i], N[j])
2. FOR all s_l ∈ N[j] DO
3.   FOR all s_k ∈ N[i] with s_k ∩ s_l ≠ ∅ DO
4.       IF N[i] is a leaf node /* N[j] is also a leaf node */
5.           THEN output (s_k, s_l)
6.           ELSE /* intermediate nodes */
7.               ReadPage(s_k.ref); ReadPage(s_l.ref);
8.               RJ(N[k], N[l])
```

Fig. 3. R-tree-based spatial join (*RJ*) algorithm.

Initially, *RJ* takes the roots of the trees to be joined as parameters. Then it performs a synchronous traversal of both R-trees, with the entries of the two structures playing the roles of data rectangles and query windows, respectively, in a series of window queries. According to Theodoridis et al. [53], (2.2), which calculates the number of node accesses at R_i when a window q is considered, can be modified to calculate the cost of a join query by using the corresponding node entries of R_j as a series of query windows on R_i. Thus, the cost for each R-tree at level l is the sum of costs of $N_{R_j,l+1}$ different window queries on R_i:

$$\begin{aligned} (2.4)\quad NA(R_i, R_j, l) &= NA(R_j, R_i, l) \\ &= N_{R_j,l+1} \cdot N_{R_i,l+1} \cdot (|s_{R_i,l+1}| + |s_{R_j,l+1}|)^d, \qquad 0 \le l \le h-2. \end{aligned}$$

For R-trees with equal height h_R, the total cost $Cost_{RJ}(R_i, R_j)$ of a spatial join between R_i and R_j using *RJ* is the sum of node accesses for each level:

$$\begin{aligned} (2.5)\quad Cost_{RJ}(R_i, R_j) &= 2 + \sum_{l=0}^{h_R-2} \{NA(R_i, R_j, l) + NA(R_j, R_i, l)\} \\ &= 2 + \sum_{l=1}^{h_R-1} \{2 \cdot N_{R_j,l} \cdot N_{R_i,l} \cdot (|s_{R_i,l}| + |s_{R_j,l}|)^d\}. \end{aligned}$$

The cost shown in (2.5) is an upper bound where no buffer is considered and every node access in R_i corresponds to a node access in R_j, according to line 7 of the *RJ* algorithm. Theodoridis et al. [53] provide a detailed description of cost formulas for *RJ*, including the case of R-trees with different heights. In correspondence to window query analysis, all the involved parameters can be expressed as functions of dataset properties, namely cardinality and density. Experimental results suggest that the above cost models are accurate for uniform data (where the density remains almost invariant through the workspace). In order to deal with nonuniform data distributions, they propose the maintenance of a grid with statistical information about cardinality and density per cell. This approach, applied with a reasonably sized grid (50×50), provides good estimations for real datasets with highly skewed data distributions [27].

The basic symbols describing the concepts presented in this section are listed in Table 1. In what follows we show how these concepts are applied for multiway spatial joins.

3. Algorithms for Multiway Spatial Joins. A multiway spatial join can be represented by a graph Q where Q[i][j] denotes the join condition between R_i and R_j. Following the standard approach in the spatial join literature, we consider only MBRs, i.e., the filter step, and *overlap* as the default join condition, i.e., if for some i, j, Q[i][j] is TRUE, then there is a intersection-join between R_i and R_j (Q is not directed: Q[i][j] = Q[j][i]). We assume that Q is connected; nonconnected graphs can be solved as independent subproblems. Furthermore, we focus on two particular types of multiway joins: acyclic

Table 1. List of symbols.

Symbol	Definition
d	Number of dimensions
h_{R_i}	Height of the R-tree R_i
N_{R_i}	Number of data MBRs indexed by R_i
$N_{R_i,l}$	Number of entries of R_i at level l ($N_{r_i,0} \equiv N_{R_i}$)
$\|s_{R_i}\|$	Average extent of data rectangles indexed by R_i
$\|s_{R_i,l}\|$	Average extent of entries of R_i at level l ($\|s_{R_i,0}\| \equiv \|s_{R_i}\|$)
$\|q\|$	Average extent of a query window q
$S(R_i, q, l)$	Selectivity of a query window q on the entries of R_i at level l
$Cost_{WQ}(R_i, q)$	Number of node accesses for a window query q on R_i
$Cost_{RJ}(R_i, R_j)$	Number of node accesses for a spatial join between two R-trees R_i and R_j

(trees) and complete graphs (cliques). Figure 4 illustrates two query graphs joining three datasets and two solution tuples $(s_{R_1}, s_{R_2}, s_{R_3})$ such that s_{R_i} is an object in R_i. Figure 4(a) corresponds to a chain query (e.g., "find all cities *crossed by* a river which *crosses* an industrial area"), while Figure 4(b) corresponds to a clique ("the industrial area should also intersect the city").

According to the relational database methodology, multiway spatial joins could be processed by integration of pairwise join algorithms [27]. Solutions to the above queries are obtained by computing the result of one pairwise join (e.g., rivers *crossing* industrial areas) using the corresponding R-trees and an appropriate (pairwise) spatial join algorithm (e.g., [5]); then joining the resulting rivers with the relation cities employing a method (e.g., [24]) applicable when only one R-tree (for cities) is available. An efficient execution plan can be determined using cost models for pairwise spatial joins and optimization methods for relational queries. This paper follows a different direction and discusses processing of multiway spatial joins using systematic search algorithms, traditionally used for constraint satisfaction problems.

3.1. *Multiway Spatial Joins as Constraint Satisfaction Problems*. Numerous problems in a variety of areas (e.g., spatio-temporal reasoning, planning, image processing) can be modeled as constraint satisfaction problems (CSPs). In the context of relational databases, Dechter [8] uses results obtained from the study of CSPs to decompose large relations into trees of binary relations. In the opposite direction, Gyssens et al. [15] apply relational database techniques to decompose a CSP in smaller subproblems whose solutions can be

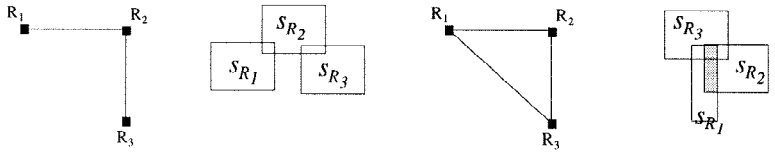


Fig. 4. Examples of multiway spatial join.

combined to generate a solution to the original problem. A binary CSP [46] is defined by:

- A set of n variables, $v_1, \ldots, v_i, \ldots, v_n$.
- For each variable v_i, a finite domain $D_i = \{u_{i,1}, \ldots, u_{i,N_i}\}$ of potential values (where N_i is the cardinality of D_i).
- For each pair of variables v_i, v_j, a binary constraint C_{ij} which is a subset of $D_i \times D_j$.

If $(u_{i,x}, u_{j,y}) \in C_{ij}$, then the assignment $\{v_i \leftarrow u_{i,x}, v_j \leftarrow u_{j,y}\}$ is *consistent*. A *solution* is an assignment $\{v_1 \leftarrow u_{1,w}, \ldots, v_i \leftarrow u_{i,x}, \ldots, v_j \leftarrow u_{j,y}, \ldots, v_n \leftarrow u_{n,z}\}$, such that, for all i, j, $\{v_i \leftarrow u_{i,x}, v_j \leftarrow u_{j,y}\}$ is consistent.

The example query: "find all cities *crossed by* a river which also *crosses* an industrial area" can be mapped to a CSP as follows: (i) There exists a variable v_i for each input, i.e., v_1, v_2, and v_3, for cities, rivers, and industrial areas, respectively. (ii) The domain of each variable v_i consists of the objects in the corresponding relation (e.g., D_1 is the set of cities). (iii) Each join predicate (e.g., "crossed by") corresponds to a binary constraint. An assignment $\{v_1 \leftarrow u_{1,x}, v_2 \leftarrow u_{2,y}, v_3 \leftarrow u_{3,z}\}$ constitutes a solution of the query in Figure 4(a), if city $u_{1,x}$ is crossed by river $u_{2,y}$ which also crosses industrial area $u_{3,z}$. Thus, the join graphs in Figure 4 can be alternatively considered as constraint networks, and in what follows we use CSP and database terminology interchangeably (e.g., variable/dataset, constraint/join condition).

Since multiway spatial joins can be modeled as CSPs, CSP algorithms could be employed for their processing. Such algorithms perform systematic search by applying the basic idea of backtracking and trying to improve the backward (e.g., *backjumping* and *dynamic backtracking*) or the forward step (e.g., *forward checking*; see [46] for a survey). A naïve backtracking algorithm for processing the query of Figure 4(a) (using the datasets of Figure 1) would first instantiate the variable corresponding to cities to some value (e.g., $v_1 \leftarrow \mathrm{a}_1$) and then proceed to the next variable (v_2) for rivers. Assume that v_2 is first instantiated to b_1 which overlaps a_1. The algorithm will then proceed another step forward and will assign v_3 (industrial area) with value c_1. Because c_1 overlaps b_1, the first solution $(\mathrm{a}_1, \mathrm{b}_1, \mathrm{c}_1)$ has been found. Then the algorithm would try all other industrial areas before it determines that there is no other value that overlaps b_1, and will backtrack assigning a new value to v_2.

The above algorithm performs a large number of redundant consistency checks because it does not exploit the underlying index structures. Papadias et al. [38] combine systematic search algorithms and R-trees for the retrieval of object combinations matching (exactly or approximately) some input configurations. Mamoulis and Papadias [26] employ these methods for a special case of multiway spatial joins where there exists a join condition between all pairs of inputs. In what follows we apply and extend this work to arbitrary query graphs. In addition, we propose optimization techniques that yield significant improvement over the original algorithms.

3.2. *Window Reduction*. Window reduction (*WR*) performs systematic search by applying window queries to find the consistent values of uninstantiated variables. For instance, after assigning $v_1 \leftarrow \mathrm{a}_1$, a_1 becomes the query window for rivers that will constitute the domain of v_2, avoiding unnecessary consistency checks. In other words, the forward phase of *WR* works in an index nested loops fashion, while the backtracking phase can be based on various CSP algorithms. The overhead of algorithms (e.g., back-

```
1.  WR (Query Q[][], Rtree R[])
2.  i := 1;
3.  queryWindow[1] = U; /*universal window*/
4.  WHILE (i>0) {
5.      τ[i] := WQ(root(R[i]), queryWindow[i]); /*apply window query to tree R_i */
6.      IF τ[i] = NULL /*empty domain for i^th variable*/
7.          THEN i := i-1; /*backtrack*/
8.          ELSE  /* not empty domain */
9.              FOR j=1 to i-1  DO /*check consistency of the value w.r.t other instantiated variables*/
10.                 IF (Q[j][i]=TRUE) AND (τ[j]∩τ[i]=∅) /* τ[i] is inconsistent-does not intersect τ[j]*/
11.                         THEN GOTO 5; /*select new value of v_i */
12.             IF i = n /*last variable has been instantiated*/
13.                     THEN output_solution(τ);
14.                     ELSE  /*intermediate variable*/
15.                         i = i+1; /*go forward */
16.                         set queryWindow[i];
17.     } /* end WHILE */
```

Fig. 5. Window Reduction (*WR*) algorithm.

jumping) that direct the backward step according to information about inconsistencies does not pay-off for the current problem. This is because, due to the large domain sizes and the limited tightness of *overlap*, the instantiated variable that causes an inconsistency with a value of the current one is almost certainly the last. Figure 5 illustrates a nonrecursive version of *WR* based on chronological backtracking.

Initially the index to the current variable v_i is set to $i = 1$ and the query window for v_1 is the whole workspace (the first variable will be instantiated to all values in its domain). Array τ holds the current instantiations ($\tau[i]$ stores the current value of v_i). A value for v_i is retrieved using a query window in the corresponding R-tree (line 5). If such a value cannot be found, the algorithm will backtrack; it will terminate when it attempts to backtrack from v_1. Line 8 will be reached only in the case of a successful instantiation. If v_i is the last variable ($i = n$), τ contains a complete solution that is output to the user. Otherwise, i is increased and the algorithm proceeds to the next variable.

The order of variables is predetermined according to some optimization method (see Section 5), and is such that every variable after the first one should be directly connected to some instantiated variable(s) (e.g., the order v_1, v_3, v_2 is not valid for the query of Figure 4(a), since there is no edge between v_3 and v_1). For acyclic queries, the current variable v_i is directly connected to a single instantiated variable whose value becomes the query window for search in R_i, e.g., for the order v_1, v_2, v_3, s_{R_1} is the query window for v_2, s_{R_2} for v_3, and so on. For clique queries, v_i is connected to all instantiated variables that mutually intersect. In this case the query window for R_i is the common area of instantiated variables [26], since any set of MBRs that mutually overlap has a nonempty intersection. In Figure 4(b), for instance, v_3 should overlap the common intersection (gray area) of s_{R_1} and s_{R_2}. For arbitrary queries, i.e., when v_i is connected to an arbitrary number of instantiated variables, the value of one is chosen as the query window and filtering with respect to the other variables takes place in main memory (lines 9–11).

3.3. *Synchronous Traversal.* The second methodology, synchronous traversal (*ST*), can be thought of as the generalization of *RJ* for an arbitrary number of inputs. In

particular, *ST* starts from the roots of the trees and attempts to find solutions, i.e., combinations of entries that satisfy the input constraints. For the query of Figure 4(a), *ST* would find all triplets (A_i, B_j, C_k) of entries at the roots such that (A_i, B_j) and (B_j, C_k) intersect. Out of the eight possible combinations (i.e., (A_1, B_1, C_1), (A_1, B_1, C_2), (A_1, B_2, C_1), ..., (A_2, B_2, C_2)), only three, (A_1, B_1, C_1), (A_1, B_1, C_2), and (A_2, B_2, C_2), could potentially lead to solutions. For each solution found the algorithm is recursively called, taking the references to the underlying nodes as parameters, until the leaf level is reached.

The calculation of combinations of the qualifying nodes for each level is expensive, as their number can be as high as C^n (where C is the node capacity). Finding the subset of node combinations that is consistent with the input query can be treated as a local CSP at each level in order to avoid exhaustive search. Similarly to *WR*, *ST* can be applied with a variety of search algorithms and optimization techniques. Here we employ *forward checking* (*FC*) [16], a backtracking-based algorithm which prunes the domain of future variables based on the current instantiations. Several studies [2], [31] have shown that it performs very well for a variety of CSPs. Furthermore, we combine *FC* with a space restriction/ordering heuristic that minimizes the number of intersection checks.

A three-dimensional $(n \cdot n \cdot C)$ array keeps the versions of variable domains for each instantiation step: $D[i+1][j]$ stores the potential values of variable v_j, after v_i is instantiated. Initially the domain $D1[1][j]$ of each variable v_j consists of all entries of a node $N[j]$ (in the beginning, the root of the corresponding R-tree). A multivariable variation of the space restriction heuristic (Figure 6) is applied before each execution of *FC*, to reduce the size of domains. Every entry in $N[j]$ which does not intersect the MBR of another node $N[i]$ where $Q[i][j] = \text{TRUE}$ is pruned from the domain of $N[j]$. As an example consider that the solution (A_1, B_1, C_1) for the query of Figure 4(a) has been found at the top level. The domains at the next call of *ST* consist of the entries in these nodes, i.e., $D[1][1] = \{a_1, a_2, a_3\}$, $D[1][2] = \{b_1, b_2\}$, and $D[1][3] = \{c_1, c_2\}$. Space restriction will remove b_2 from $D[1][2]$ (because it does not intersect the MBR of C_1) and c_2 from $D[1][3]$ (because it does not intersect the MBR of B_1). Finally, the remaining entries in the variable domains are sorted with respect to the lower x-coordinate of their MBR (*x_low sorting*).

After the application of space restriction, *FC* (Figure 7) retrieves solutions at the current level. Each variable v_i is assigned values from $D[i][i]$. If $D[i][i]$ has been exhausted the algorithm will backtrack to the previous variable (lines 6–7). A one-dimensional array τ holds the current instantiations. After an instantiation $v_i \leftarrow s_k$, τ ($\tau[i] = s_k$) contains a solution where the constraints between the first i variables are satisfied. When a complete ($i = n$) solution is found, it is either output (at leaf level) or recursively followed at the lower level (lines 9–15).

```
1. space restriction(int j)
2. D[1][j] := all entries s_m ∈ N[j] /* D[1][j] is the initial domain for v_j */
3. FOR each entry s_m ∈ D[1][j] DO
4.       FOR i:=1 to n , i≠j DO /* for each other variable v_i*/
5.           IF Q[i][j]=TRUE and s_m∩N[i].mbr = ∅ THEN /* s_m does not intersect the MBR of N[i] */
6.                D[1][j]: = D[1][j] - s_m; /* remove s_m from the domain of v_j*/
7.                BREAK; /* next s_m - go to line 3*/
8. Sort remaining entries in D[1][j] w.r.t. the x_low point;
```

Fig. 6. Multivariable space restriction.

```
1.  ST (Query Q[][], RTreeNode N[])
2.  FOR j:=1 to n DO space restriction(j); /* space restriction for all variables */
3.  i := 1; /* index to the current variable */
4.  WHILE (i >0) {
5.      τ[i] := next value from D[i][i];
6.      IF τ[i] = NULL /* D[i][i] has been exhausted */
7.         THEN i := i -1; /* backtrack */
8.         ELSE /* τ[i]not null*/
9.              IF i = n /* last variable instantiated */
10.                  THEN
11.                     IF N[] are leaf nodes /*a solution was found for leaf R-tree nodes */
12.                          THEN output (τ);
13.                          ELSE  /* a solution was found for intermediate R-tree nodes */
14.                               FOR k:=1 to n DO ReadPage(τ[k].ref);
15.                               ST (Q, τ[].ref); /* ST is recursively called for the references */
16.                  ELSE /* i < n, intermediate variable instantiated */
17.                       IF check_forward(i) /* no future variable was eliminated */
18.                            THEN i: = i + 1; /* instantiate next variable */
19. } /* end WHILE */

20. boolean check_forward(int i)
21. FOR j := i+1 to n   /*for all future variables*/
22.      IF Q[i][j] = TRUE /* if there is an edge between v_i and v_j */
23.           THEN
24.                D[i+1][j] := ∅; /* initialize v_j's domain for next instantiation */
25.                WHILE (s_m = next entry ∈ D[i][j]) AND (s_m.x_low ≤ τ[i].x_up) {
26.                     IF s_m ∩ τ[i] ≠ ∅ /* consistent value */
27.                          THEN D[i+1][j] := D[i+1][j] ∪ {s_m}; /* copy value to the next domain */
28.                } /* end WHILE */
29.                IF D[i+1][j] = ∅ THEN RETURN FALSE; /* a future variable is eliminated */
30.           ELSE /* Q[i][j] = FALSE, no edge between v_i and v_j */
31.                D[i+1][j] := D[i][j];
32. RETURN TRUE;
```

Fig. 7. Synchronous Traversal (*ST*) with *forward checking* (*FC*).

For partial solutions ($i < n$) *check_forward* is called to remove from the domains of all future variables v_j ($j > i$) such that Q[i][j] = TRUE, those values that do not intersect $\tau[i]$ ($\tau[i] = s_k$). In other words, $D[i+1][j]$ (the domain of v_j after the instantiation of v_i) contains the subset of MBRs in $D[i][j]$ that intersect $\tau[i]$ (lines 21–28). The order of MBRs (generated by space restriction) is used to avoid redundant checks: when the first entry s_m such that $s_m.x\_low > \tau[i].x\_up$ is encountered, searching the domain is aborted because there cannot exist subsequent entries that intersect $\tau[i]$.

After *check_forward*, the domain of each future variable contains only values which are consistent with all current instantiations. If some domain has been eliminated completely, *check_forward* will return FALSE signaling that $\tau[i]$ cannot lead to a solution. Domain elimination for a variable v_j happens when there exists a join condition Q[i][j], but no value for v_j that intersects $\tau[i]$. Continuing the previous example, after space restriction the domains become $D[1][1] = \{a_1, a_2, a_3\}$, $D[1][2] = \{b_1\}$, and $D[1][3] = \{c_1\}$. During the instantiation $v_1 \leftarrow a_3$, *check_forward* will remove b_1 and $D[2][2]$ becomes empty. Therefore, $v_1 \leftarrow a_3$ cannot lead to a solution and v_1 must

be assigned another value. The three-dimensional structure of the domain array is used as a stack mechanism to simplify the domain restoration procedure, i.e., when v_i is unassigned value $\tau[i]$, $D[i+1][j]$ is re-initialized to $D[i][j]$ for all future variables.

The application of *ST* and *WR* in cases where some or all of the variables have the same domain (i.e., image similarity retrieval applications) is straightforward. Furthermore, both algorithms can be effectively employed when only a subset of the solutions needs to be retrieved. For instance, they can be easily modified to terminate after retrieval of the first solution resulting in significantly smaller execution cost (see the experiments in Section 6). Traditional multiway join processing, based on integration of pairwise spatial join algorithms, does not have this feature. For instance, spatial hash join algorithms (e.g., [23] and [25]) applied for joining intermediate outputs must read and write the whole build input, even if pipelining is used for passing the results to the next operator.

4. Cost Models. *WR* essentially searches the whole space in order to instantiate the first variable, but after doing so it performs only window queries which are cheap operations in R-trees. The disadvantage of blindly instantiating the first variable in the whole universe could be avoided by an algorithm that applies *ST* to instantiate multiple initial variables which will then be input to *WR* through pipelining. In the query of Figure 4(a), for instance, *RJ* could retrieve pairs of overlapping cities and rivers, and for each such pair *WR* will be called to find qualifying industrial areas. Figure 8 illustrates all the alternative plans for the query, where joins to be processed by *ST* are shown in rectangles. The last four plans correspond to *WR* where the leftmost variable is instantiated first.

Obviously this technique can be applied with any number of variables, e.g., a query involving ten relations may be processed using *ST* for the first four variables, and *WR* to instantiate the rest. The combination of *ST* and *WR* for multiway spatial join processing results in plans of a certain "left-deep" form, which is different from left-deep trees in relational join processing [12] in the sense that the leftmost (deepest) leaf nodes are synchronously traversed (plans are not necessarily binary trees). In order to determine the optimal plan we need analytical formulas for the selectivity of multiway spatial joins and for the cost of the algorithms.

4.1. *Selectivity of Multiway Spatial Joins.* As in the case of spatial selections and pairwise joins, the expected number of solutions determines the cost and is crucial for the optimization of multiway spatial joins. The total number of solutions is given by the following formula:

$$\#solutions = \#(\text{all possible } n\text{-tuples}) \cdot Prob(\text{an } n\text{-tuple constitutes a solution}). \quad (4.1)$$

The first part of the product in (4.1) equals the cardinality of the Cartesian product of n

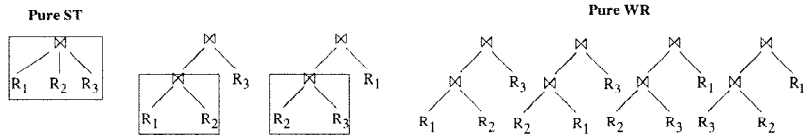


Fig. 8. Possible plans for the query of Figure 4(a).

domains, while the second part corresponds to query selectivity. If s_{R_i} is a data object in R_i, selectivity is defined as the probability that all binary assignments $\{v_i \leftarrow s_{R_i}, v_j \leftarrow s_{R_j}\} \forall i, j \mid Q[i][j] = \text{TRUE}$ are consistent. In the case of acyclic graphs, and ignoring boundary effects (i.e., rectangles are small with respect to the workspace), the pairwise probabilities are independent. For instance, in the query of Figure 4(a), the event that "s_{R_1} overlaps s_{R_2}" is independent of the event "s_{R_2} overlaps s_{R_3}." Thus the probability of a triplet satisfying the join conditions is the product of pairwise selectivities, which are computed by (2.1):

$$Prob((s_{R_1}, s_{R_2}, s_{R_3}) \text{ is a solution}) = (|s_{R_1}| + |s_{R_2}|)^d * (|s_{R_2}| + |s_{R_3}|)^d. \tag{4.2}$$

Extending to n variables, the selectivity of an acyclic join graph is

$$Prob(\text{an } n\text{-tuple is a solution}) = \prod_{\forall i,j: Q(i,j)=\text{TRUE}} (|s_{R_i}| + |s_{R_j}|)^d \tag{4.3}$$

and (4.1) and (4.3) imply that the total number of solutions at tree level l is

$$\#solutions(Q, l) = \prod_{i=1}^{n} N_{R_i,l} \cdot \prod_{\forall i,j: Q(i,j)=\text{TRUE}} (|s_{R_i,l}| + |s_{R_j,l}|)^d. \tag{4.4}$$

When the query graph contains cycles, the assignments are not independent anymore and (4.3) does not accurately estimate the probability that a random tuple constitutes a solution. For cliques, it is possible to provide a formula for selectivity based on the fact that if a set of rectangles mutually overlap, then they must share a common area. The common intersection area of a set of rectangles is computed by the following lemma.

LEMMA 1. *Given a set of n ($n \geq 2$) mutually overlapping rectangles s_i, $i = 1, \ldots, n$, with extent $|s_i|$ on each direction, the common intersection area is a rectangle q_n of average extent $|q_n|$ defined as follows*:

$$|q_n| = \frac{\prod_{i=1}^{n} |s_i|}{\sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} |s_j|}. \tag{4.5}$$

PROOF (BY INDUCTION ON n). *Step* 1. For $n = 2$, it is sufficient to prove that

$$|q_2| = \frac{|s_1| \cdot |s_2|}{|s_1| + |s_2|}.$$

Without loss of generality, we assume $|s_1| \leq |s_2|$. Since the two rectangles overlap, their projections (line segments) on each direction also overlap; let δ be their intersection and let $s_{i.\,start}$ ($s_{i.\,end}$) be their projections' start (end) points, $i = 1, 2$. Figure 9 sketches the three possible configurations between two overlapping line segments, representing the following sets of conditions:

- Case I: $s_{1.\,start} < s_{2.\,start} < s_{1.\,end} < s_{2.\,end}$.
- Case II: $s_{2.\,start} \leq s_{1.\,start} < s_{1.\,end} \leq s_{2.\,end}$.
- Case III: $s_{2.\,start} < s_{2.\,start} < s_{2.\,end} < s_{1.\,end}$.

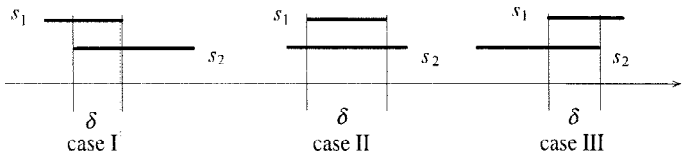


Fig. 9. Possible configurations of overlapping intervals.

Recall that it is always: $s_{2.\,start} \leq s_{1.\,end}$ and $s_{1.\,start} \leq s_{2.\,end}$ since the two projections overlap.

Assuming that the address space is discrete, with very fine granularity (the case of continuous space will be the limit for infinitely fine granularity) [20], the probability that a configuration may be the case for two overlapping segments s_1 and s_2 corresponds to the different relative positions of s_1 with respect to s_2. Formally:

$$\begin{aligned} &Prob\,(s_{1.\,start} < s_{2.\,start} < s_{1.\,end} < s_{2.\,end} / s_{2.\,start} \leq s_{1.\,end} \wedge s_{1.\,start} \leq s_{2.\,end}) \\ &\quad = \frac{|s_1|}{|s_1| + |s_2|} \\ &\quad = Prob\,(s_{2.\,start} < s_{1.\,start} s_{2.\,end} < s_{1.\,end} / s_{2.\,start} \leq s_{1.\,end} \wedge s_{1.\,start} \leq s_{2.\,end}) \end{aligned}$$

and

$$\begin{aligned} &Prob\,(s_{2.\,start} < s_{1.\,start} < s_{1.\,end} \leq s_{2.\,end} / s_{2.\,start} \leq s_{1.\,end} \wedge s_{1.\,start} \leq s_{2.\,end}) \\ &\quad = \frac{|s_2| - |s_1|}{|s_1| + |s_2|}. \end{aligned}$$

The first two probabilities correspond to cases I and III and the latter one corresponds to case II. For each of the three cases, the average δ size equals the average portion of s_1 intersecting s_2, i.e., $\delta = |s_1|/2$ for cases I and III and $\delta = |s_1|$ for case II. In turn, the average extent $|q_2|$ equals the weighted average δ size, i.e.,

$$|q_2| = 2 \cdot \left(\frac{|s_1|}{|s_1| + |s_2|} \cdot \frac{|s_1|}{2} \right) + \frac{|s_2| - |s_1|}{|s_1| + |s_1|} \cdot |s_1|,$$

where the first part of the summation represents the (equal) weighted average of δ for cases I and III while the second part corresponds to case II. Since the above value of $|q_2|$ equals $|s_1| \cdot |s_2| / (|s_1| + |s_2|)$, step 1 of the proof has been completed.

Step 2. We assume that (4.5) holds for $n = k$, i.e.,

$$|q_k| = \frac{\prod_{i=1}^{k} |s_i|}{\sum_{i=1}^{k} \prod_{j=1,\, j \neq i}^{k} |s_j|}.$$

Step 3 (*induction step*). We will prove that (4.5) holds for $n = k + 1$, i.e.,

$$|q_{k+1}| = \frac{\prod_{i=1}^{k+1} |s_i|}{\sum_{i=1}^{k+1} \prod_{j=1,\, j \neq i}^{k+1} |s_j|}.$$

Proof of the induction step: Since rectangle s_{k+1} overlaps all $s_1, \ldots, s_k$ rectangles that are mutually overlapping, it overlaps their common intersection area, denoted by q_k. Furthermore, the common intersection area of all $s_1, \ldots, s_k, s_{k+1}$ rectangles, denoted by q_{k+1}, is identical to the common intersection area between q_k and s_{k+1}. According to steps 1 and 2:

$$|q_{k+1}| = \frac{|s_{k+1}| \cdot |q_k|}{|s_{k+1}| + |q_k|} = \cdots = \frac{\prod_{i=1}^{k+1} |s_1|}{|s_{k+1}| \cdot \sum_{i=1}^{k} \prod_{j=1, j\neq i}^{k} |s_j| + \prod_{i=1}^{k} |s_i|}$$

$$= \frac{\prod_{i=1}^{k+1} |s_1|}{\sum_{i=1}^{k} \left(|s_{k+1}| \cdot \prod_{j=1, j\neq i}^{k} |s_j| \right) + \sum_{i=k+1}^{k+1} \prod_{j=1, j\neq i}^{k} |s_j|}$$

$$= \frac{\prod_{i=1}^{k+1} |s_i|}{\sum_{i=1}^{k+1} \prod_{j=1, j\neq i}^{k+1} |s_j|}. \qquad \square$$

Consider the instantiations $\{v_1 \leftarrow s_{R_1}, v_2 \leftarrow s_{R_2}\}$ in the query of Figure 4(b). The probability that a tuple $(s_{R_1}, s_{R_2}, s_{R_3})$ is a solution is $Prob(s_{R_1}$ overlaps $s_{R_2}) \cdot Prob(s_{R_3}$ overlaps s_{R_1} *and* s_{R_3} overlaps s_{R_2}/s_{R_1} overlaps $s_{R_2})$. The conditional probability in the second part of the product is equal to the probability that s_{R_3} intersects the common area of s_{R_1} and s_{R_2}, i.e., $Prob(s_{R_3}$ overlaps $q_2)$. By applying (4.5) for the intersection area of q_2 and (2.1) for pairwise selectivities, we derive

$$(4.6)\quad Prob((s_{R_1}, s_{R_2}, s_{R_3}) \text{ is a solution }) = \left(|s_{R_1}| + |s_{R_2}|\right)^d * \left(|s_{R_3}| + \frac{|s_{R_1}| \cdot |s_{R_2}|}{|s_{R_1}| + |s_{R_2}|}\right)^d$$

$$= \left(|s_{R_1}| \cdot |s_{R_2}| + |s_{R_2}| \cdot |s_{R_3}| + |s_{R_1}| \cdot |s_{R_3}|\right)^d.$$

The selectivity of complete query graphs involving an arbitrary number of inputs is computed by the following lemma.

LEMMA 2. *Given a random n-tuple of rectangles* $(s_1, \ldots, s_n)$*, the probability that all rectangles mutually overlap is*

$$(4.7)\qquad Prob(\text{rectangles } s_1, \ldots, s_n \text{ mutually overlap}) = \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j\neq i}}^{n} |s_j| \right)^d.$$

PROOF. Since all rectangles mutually overlap, without loss of generality we assume that the instantiation order is $s_1, \ldots, s_n$. Thus, the left part of (4.7) is equal to a product of independent probabilities:

$$Prob(\text{rectangles } s_1, \ldots, s_n \text{ mutually overlap}) = Prob(s_2 \text{ overlaps } s_1)$$
$$\cdot Prob(s_3 \text{ overlaps } s_1 \wedge s_3 \text{ overlaps } s_2/s_1, s_2 \text{ mutually overlap})$$
$$\ldots$$
$$\cdot Prob(s_n \text{ overlaps } s_1 \wedge \cdots \wedge s_n \text{ overlaps } s_{n-1}/s_1, \ldots, s_{n-1} \text{ mutually overlap}).$$

In general, in order for a rectangle s_{k+1} to overlap $s_1, \ldots, s_k$ mutually overlapping rectangles, it should overlap their common intersection, which is denoted by q_k. Hence, the above equation is equivalent to

$$\begin{aligned} &Prob(\text{rectangles } s_1, \ldots, s_n \text{ mutually overlap}) \\ &\quad = Prob(s_2 \text{ overlaps } s_1) \cdot Prob(s_3 \text{ overlaps } q_2) \cdot \cdots \cdot Prob(s_n \text{ overlaps } q_{n-1}) \\ &\quad = (|s_2| + |s_1|)^d \cdot (|s_3| + |q_2|)^d \cdot \cdots \cdot (|s_n| + |q_{n-1}|)^d . \end{aligned}$$

Using Lemma 1 for the area of each q_i, we obtain

$$\begin{aligned} &Prob(\text{rectangles } s_1, \ldots, s_n \text{ mutually overlap}) \\ &= (|s_2| + |s_1|)^d \cdot \left(|s_3| + \frac{|s_1| \cdot |s_2|}{(|s_1| + |s_2|)} \right)^d \cdot \cdots \cdot \left(|s_n| + \frac{\prod_{i=1}^{n-1} |s_i|}{\sum_{i=1}^{n-1} \prod_{j=1, j\neq i}^{n-1} |s_j|} \right)^d \\ &= \left((|s_2| + |s_1|) \cdot \left(\frac{|s_1| \cdot |s_2| + |s_1| \cdot |s_3| + |s_2| \cdot |s_3|}{(|s_1| + |s_2|)} \right) \cdot \cdots \cdot \left(\frac{\sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} |s_j|}{\sum_{i=1}^{n-1} \prod_{j=1, j\neq i}^{n-1} |s_j|} \right) \right)^d \\ &= \sum_{i=1}^{n} \prod_{\substack{j=1 \\ j\neq i}}^{n} |s_j|. \end{aligned}$$

□

Thus, in the case for clique queries the numbers of solutions at level l is

$$\#solutions(Q, l) = \prod_{i=1}^{n} N_{Ri,l} \cdot \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j\neq i}}^{n} |s_{Rj,l}| \right). \tag{4.8}$$

The experiments of Section 6 demonstrate that previous formulas are accurate and, therefore, can be applied for optimization of multiway spatial joins independently of the algorithms. In what follows we show how they can be used to estimate the cost of *WR* and *ST*.

4.2. *Cost Models for WR and ST.* Like the approaches described in Section 2, we use node accesses (*NA*) as a measure of the cost of *WR* and *ST*. This is because (i) *NA* are relatively simple to estimate, (ii) they do not depend on buffer size and page replacement policy, and (iii) they provide an indication for the CPU overhead since *NA* are directly related to the number of consistency checks performed by both algorithms. As we show in the experimental evaluation, the CPU-time plays an important role in the cost of multiway spatial joins (whereas selections and pairwise joins are I/O bound).

A subgraph $Q_{x,y}$ of Q containing x nodes (variables) is called *legal* if it is connected (we use index y to distinguish different legal subgraphs of x nodes). $V_{x,y}$ is the set of nodes in $Q_{x,y}$. The total number of legal subgraphs is less than or equal to (in the case of complete graphs) the number of x *combinations* of n objects $C(x, n)$. $(Q_{x-1,y'}, v_{x'})$ denotes a *decomposition* of $Q_{x,y}$ into a legal subgraph $Q_{x-1,y'}$ (with $x-1$ nodes), and a single variable $v_{x'}$, such that $v_{x'} = V_{x,y} - V_{x-1,y'}$. For instance, the graph in Figure 4(a) can be decomposed into a subgraph $Q_{2,1}$ with $V_{2,1} = \{v_1, v_2\}$ and variable v_3. On the

other hand, a decomposition into $\{v_1, v_3\}$ and v_2 is not allowed since v_1 and v_3 are not directly connected. A legal subgraph $Q_{x,y}$ can be processed in two ways: either by applying *ST*, or by executing a subquery of size $x-1$ and then using *WR* to instantiate the xth variable.

Let $Cost_{WR}(Q_{x-1,y'}, v_{x'})$ be the cost (in terms of node accesses) of executing *WR* to find all consistent instantiations of $v_{x'}$, when $Q_{x-1,y'}$ has been solved. For each solution we have to perform a window query in index $R_{x'}$ in order to retrieve the consistent instantiations of $v_{x'}$. As discussed previously, in the case of acyclic graphs $v_{x'}$ is connected with a single instantiated variable $V_{x-1,y'}$ whose value becomes the query window $q_{x'}$. For cliques, $q_{x'}$ is the common intersection area of the values of all variables in $V_{x-1,y'}$. The total number of window queries corresponds to the number of solutions of $Q_{x-1,y'}$ at level 0. Thus,

$$Cost_{WR}(Q_{x-1,y'}, v_{x'}) = \#solutions(Q_{x-1,y'}, 0) \cdot Cost_{WQ}(R_{y'}, q_{x'}), \tag{4.9}$$

where $Cost_{WQ}$ is computed according to (2.3), and the number of solutions according to (4.4) or (4.8), for acyclic and clique queries, respectively.

Let $Cost_{ST}(Q_{x,y})$ be the cost of processing $Q_{x,y}$ using *ST*. The x roots of the R-trees must be accessed in order to find root level solutions. Each solution will lead to x accesses at the next (lower) level. In general, at level l, there will be $x \cdot \#solutions(Q_{x,y}, l+1)$ node accesses. The total cost of *ST* is

$$Cost_{ST}(Q_{x,y}) = x + \sum_{l=0}^{h-2} x \cdot \#solutions(Q_{x,y}, l+1). \tag{4.10}$$

Note that although both algorithms are applicable for queries containing arbitrary cycles, optimization of such queries using (4.4) and (4.8) as bounds for the number of solutions is not accurate. Let $P = ((v_1, \ldots, v_k), v_{k+1}, \ldots, v_n)$ be a plan where the first k variables are instantiated through *ST* and the rest by *WR* in this order, and let $Q_{x,p}$ be a subgraph containing the first x variables of P. The total cost of processing P is

$$Cost(P) = Cost_{ST}(Q_{k,p}) + \sum_{x=k+1}^{n} Cost_{WR}(Q_{x-1,p} v_x). \tag{4.11}$$

It is well known in both the database [12] and CSP [2] communities that the choice of an appropriate plan, or an order in which variables get instantiated, has a very significant effect on performance. In what follows we provide optimization algorithms that determine the subset of variables to be instantiated by *ST* and the optimal order of the remaining variables to be instantiated by *WR*.

5. Query Optimization. Let $p(x)$ be a function that returns the number of plans for a legal subgraph of x nodes, and let $d(x)$ be a function that returns the number of legal decompositions. Assuming that all $Q_{x,y}$ can result in the same number of decompositions and each decomposition has the same number of plans, then the total number of plans is described by the following recurrence:

$$p(1) = 1 \quad \text{and} \quad p(x) = d(x) \cdot p(x-1) + 1, \tag{5.1}$$

where the additional plan is for processing $Q_{x,y}$ using *ST*. For chain queries (minimal number of plans), $d(x) = 2$ since $Q_{x-1,y'}$ can be generated from $Q_{x,y}$ only by removing the first or the last variable. By substituting this value in recurrence (5.1), we derive that the number of alternative plans for chain queries is $2^n - 1$. Equation (5.1) cannot be applied for arbitrary trees, because $d(x)$ may be different for two subgraphs with x nodes. Among all acyclic queries, the one that results in the largest number of plans is the star graph. In this case $Q_{x-1,y'}$ can be generated from $Q_{x,y}$ by removing any variable except for the one at the center, thus $d(x) = x - 1$. For cliques (maximum number of plans) any variable can be removed during a decomposition, resulting in $d(x) = x$, and a total number of plans equal to

$$n! \cdot \sum_{x=1}^{n} \frac{1}{x!} < n! \cdot e. \tag{5.2}$$

This is significantly smaller than the corresponding number in relational queries, i.e., $(2(n-1))!/(n-1)!$) [49], because there do not exist right-deep or bushy plans. In this section we first describe a dynamic programming algorithm that searches through the whole plan space and can be used for joins involving relatively few variables. Next we apply hill climbing techniques for the case of numerous inputs.

5.1. *Optimization Using Dynamic Programming.* Dynamic programming has been successfully applied for optimization of relational queries involving a small number of inputs [18]. *DP_plan* algorithm (Figure 10) computes the best execution strategy for a query incrementally, based on optimal plans of its subgraphs. The recursive equation implemented by the algorithm is

$$\begin{aligned} &Cost(Q_{x,y}) \\ &\quad = \min\{Cost_{ST}(Q_{x,y}), \min_{\forall \text{ decomposition } y'}(Cost(Q_{x-1,y'}) + Cost_{WR}(Q_{x-1,y'}, v_{x'}))\}. \end{aligned} \tag{5.3}$$

In general, at each level *DP_plan* decomposes every $Q_{x,y}$ into all legal combinations $(Q_{x-1,y'}, v_{x'})$, and finds the best demcomposition using the cost for $Q_{x-1,y'}$ which was computed at the previous execution level $x - 1$. Either this decomposition or $ST(Q_{x,y})$ will be marked as $Q_{x,y}$'s optimal plan, to be used when computing the optimal cost for query subgraphs of size $x + 1$.

```
1.  DP-plan(Qucry Q, int n)
2.  FOR x=2 to n DO
3.     FOR each connected subgraph y of size x DO
4.        Cost[Q_{x,y}] = Cost_ST(Q_{x,y});
5.        bestPlan[Q_{x,y}] = ST;
6.        FOR each legal decomposition y' of Q_{x,y} DO
7.              minCost = Cost[Q_{x-1,y'}] + Cost_WR(Q_{x-1,y'},v_{x'});
8.              IF minCost < Cost[Q_{x,y}] THEN
9.                    bestPlan[Q_{x,y}] = WR(Q_{x-1,y'},v_{x'});
10.                   Cost[Q_{x,y}] = minCost;
```

Fig. 10. Dynamic programming optimization algorithm (*DP_plan*).

Cost$[Q_{1,y}]$ is initially the number of leaf nodes in each R-tree R_y (i.e., the number $N_{R_y,1}$ of entries at level 1). Then the algorithm will calculate the plans and corresponding costs for all pairwise joins, i.e., all $Q_{2,y}$ such that $Q_{2,y}$ is connected. First the cost of each pairwise join is computed using *ST*. Then for both decompositions of $Q_{2,y}$ to two subgraphs (containing one variable each), it will calculate the cost of *WR* for instantiating one variable first and then the second one (index nested loops). For all pairwise joins, the best of three options (*ST* and two *WR* plans) and their costs are stored in two tables (*bestPlan* and *Cost*, respectively) and used for calculating the costs of processing subgraphs of three nodes. At the end of *DP_plan*, *bestPlan*$[Q]$ will contain the optimal plan for executing Q, and *Cost*$[Q]$ its expected cost. Notice that lines 4 and 7 use (4.10) and (4.9) which require the expected number of solutions. This number is also stored for each decomposition, but, for simplicity, is omitted in the pseudocode.

If the query is clique (worst case), at each iteration of the outer loop the algorithm will test $C(x, n) = n!/x!(n-x)!$ subgraphs of $Q_{x,y}$, and for each $Q_{x,y}$ it will perform x decompositions. Thus, the total running time (assuming constant table writing and look-up) is

$$\sum_{x=2}^{n} \frac{n!}{(x-1)!\,(n-x)!}. \tag{5.4}$$

Only the optimal cost and the number of solutions of each subgraph with size $x-1$ has to be maintained for the calculation of the optimal costs of subgraphs with size x; thus, the space requirements of *DP_plan* at iteration x of the outer loop are $C(x-1, n) + C(x, n)$. The time and space requirements of the algorithm renders exhaustive optimization inapplicable for queries involving numerous relations.

5.2. *Optimization Using Hill Climbing Algorithms.* Hill climbing algorithms operate on a graph performing random walks between the nodes based on a certain movement (transition) mechanism. In the current problem, each node corresponds to a plan P and the transition mechanism defines a neighborhood of all the nodes that can be reached from P. The neighborhood of a plan $P = ((v_1, \ldots, v_k), v_{k+1}, \ldots, v_n)$ contains $((v_1, \ldots, v_k, v_{k+1}), \ldots, v_n)$, $((v_1, \ldots, v_{k-1}), \ldots, v_n)$ and all plans that can be derived from by P swapping the positions of any two variables v_i and v_j, $k \leq i$, $j \leq n$. Notice that starting with P, any plan can be reached after a finite number of moves. A move is called *downhill*, if it leads to a plan with a lower cost, or *uphill*, if it leads to a plan with higher cost. We implemented two hill climbing methods for optimization of multiway spatial joins involving large queries.

The first one, based on *iterative improvement* (*II*) [32], starts with a randomly chosen plan P and applies the above transition mechanism, trying to find a legal plan with lower cost in the neighborhood of P (i.e., a downhill move). If such a plan is found, P is replaced by the new one; otherwise the algorithm keeps P and continues with another move, until a local minimum is found. This iterative local optimization is repeated a number of times starting from a different random plan. As time approximates ∞, the probability that iterative improvement will find the global minimum approximates to 1 [32]. Given a finite amount of time, the algorithm terminates in a local minimum.

The second method implements *simulated annealing* (*SA*). *SA* performs random walks just like iterative improvement but in addition to downhill, it also accepts uphill moves

with a certain probability, trying to avoid local minima. Initially this probability has a relatively high value, which is gradually reduced. Detailed descriptions of simulated annealing can be found in [6] and [22]. After running *SA* under various conditions we chose the following set of control parameters for the current problem: the initial probability of accepting an uphill move is 0.4, the temperature reduce factor is 0.975, and the equilibrium condition is n. Similar values were obtained for the optimization of relational joins [19], [51].

We also implemented versions of the above heuristics, called the *II-sortDN* and *SA-sortDN*, respectively, which, instead of using a random initial plan, choose a "good" seed by applying the following heuristic: based on the fact that the join cost depends on the data density D and the cardinality N, we sort the variables with respect to the product $D \cdot N$. The variables with the smallest value of $D \cdot N$ should be processed first, decreasing the probability that the size of intermediate results will be large. Then, keeping the order fixed, we test all possible values ($= n$) of k and set as the initial plan the one that gives the minimum cost. The experiments in the next section suggest that hill climbing algorithms combined with the above heuristic provide nearly optimal plans.

6. Experimental Evaluation. The previous algorithms and optimization methods are independent of the underlying predicates, so they could be used with a variety of spatial constraints. In these cases, the equation parameters (e.g., number of solutions, cost of window query) need to be modified using appropriate cost models [37]. Here we follow the standard experimental methodology and evaluate them by assuming that the spatial predicate is always *overlap*. We classify the experiments in two subsections: the first deals with optimization issues, while the second one studies the effects of various parameters on the cost of multiway spatial joins. All experiments were executed on an Ultrasparc2 workstation (200 MHz) with 256 MB of memory.

6.1. *Evaluation of Optimization*. The first set of experiments shows the accuracy of the cost models, and studies how data and query density affect the optimal value for k (i.e., the number of variables to be instantiated by *ST*). We run tree and clique queries involving seven variables using datasets of various densities. The cardinality of all datasets is fixed to 10,000 uniformly distributed rectangles. Page size is set to 1 KB resulting in R*-trees [3] with node capacity 50 and height 3. Data density has four potential values: 0.05, 0.20, 0.35, and 0.50. There is a total of 4×2 (data density times graph topology) experimental settings. For each setting the value of k ranges from 1 (pure *WR*) to 7 (pure *ST*); every run corresponds to the best plan given the value of k. In all cases the cost of optimization was less than 1% of the cost of processing the optimal plan.

Figure 11 illustrates the actual (*NA*), estimated (*ENA*) node accesses, and CPU time for each setting. Node accesses are shown on the left y-axis and CPU time on the right one (sometimes in logarithmic scale). The first column contains the density values of all datasets in each query. The diagrams also include the optimal k and the number of actual solutions retrieved; obviously, the number of solutions increases with the data density and decreases with the query density. The results are very similar for all acyclic topologies so we do not include special cases (i.e., chains or stars); the behavior of such queries can be derived from the general diagrams for trees. For the estimation of

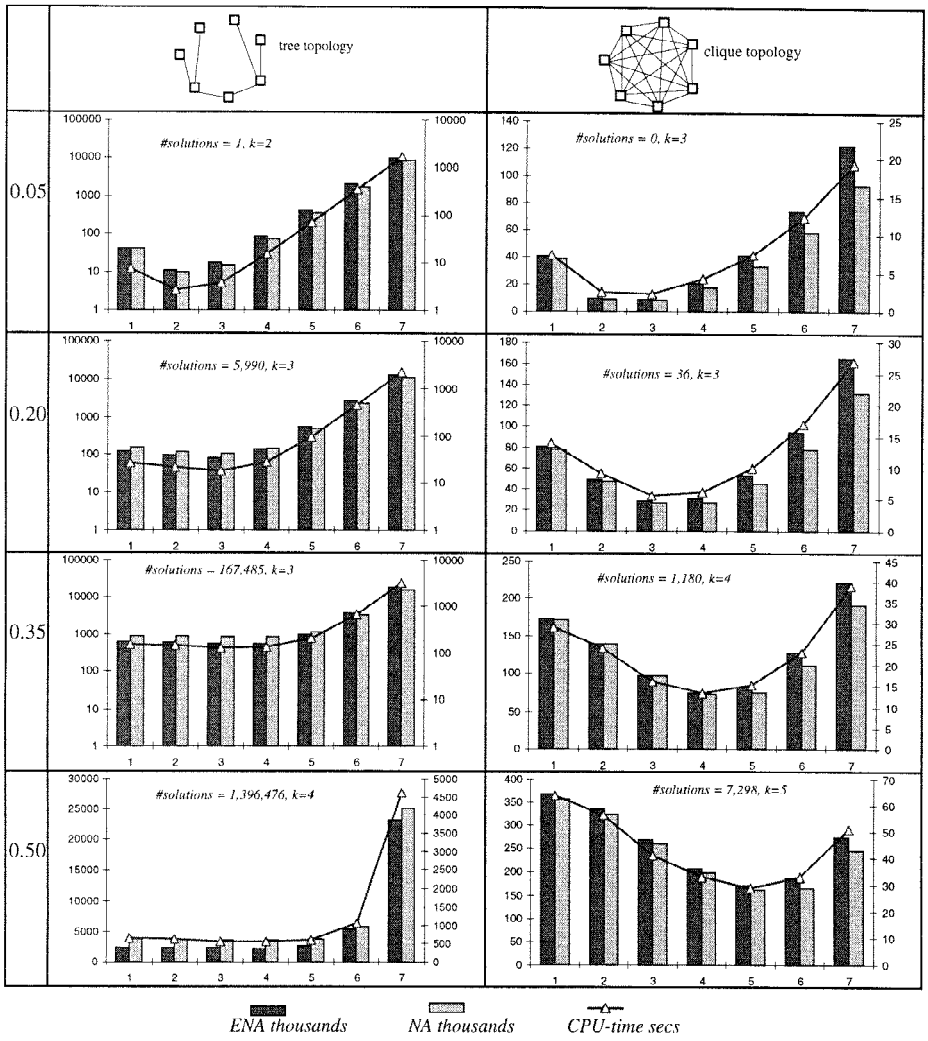


Fig. 11. Actual and estimated node accesses and CPU time for various combinations of data/query densities.

node extents $|s_{R_i}|$ we use statistical information from the tree (rather than the analytical formulas of [52]) because they provide higher accuracy.

Several observations can be made based on the results:

- Estimated node accesses are close to the actual number. In the worst case, the relative error is below 25%, whereas the average difference between *ENA* and *NA* is 8%.
- The diagrams for CPU time are very similar to the ones for node accesses, and the cheapest plan in terms of CPU time is always the one with the fewest accesses. This confirms the fact that *ENA*, based on the models of Section 4, is a good measure for the cost of multiway spatial joins.

Fig. 12. Germany datasets and workspace.

- There are vast performance differences (in some cases, orders of magnitude) for the different choices of k (although for each k the best plan was used). In particular, the optimal k increases with the data and query density. In all cases, intermediate values of k achieved the best performance (no pure *WR* or *ST* plans).

In order to test the applicability of the methods in real-life situations, we also ran experiments with four real datasets containing different layers of Germany (available at *http://www.maproom.psu.edu/dcw/*). The previous selectivity formulas refer to uniform data normalized to a [0, 1) workspace. Since intermediate nodes are not usually as skewed as the leaf rectangles, the uniformity assumption can be applied to real datasets. However, the *workspace* should be normalized because the data objects do not necessarily cover a rectangular area. Given a series of different layers of the same region (e.g., rivers, streets, forests), its workspace is defined as the total area covered by all layers, including holes if any. In order to estimate this area, we use a rectangular grid, where each cell is marked if it intersects some rectangle(s) in any dataset. The workspace is the area covered by the marked cells. The selectivity formulas of Section 4 are then applied by considering that the normalized node area at a specific R-tree level is equal to the average node area divided by the workspace. Figure 12 illustrates two (of the four) layers of Germany used in the experiments, with their cardinalities and densities, and the corresponding workspace using a 50×50 grid.

The four datasets were indexed by R*-trees with 2 KB page size (1 KB size results in trees of different heights due to the different cardinalities). An LRU page replacement policy was used and the buffer size was set to 512 KB. We executed the chain and clique queries of Figure 13 using all the possible plans. In addition to the optimal plan, Figure 13 illustrates the cheapest *WR*, and *ST* plans. For each plan we include the *ENA*, (actual) *NA*, CPU time (seconds), page accesses, and the percentage of CPU time in the overall cost of processing (we charge 10 ms for each page access [49]).

The estimated cost of all plans (*ENA*) is in all cases smaller than the actual one and the accuracy drops with respect to uniform data. This is because some areas (e.g., residential) have high density data which increase the actual number of overlaps. However, the plan suggested by *DP_plan* was still the actual optimal; in most cases the suggested plan, even if not the best, is expected to be nearly optimal. Another observation refers to the CPU time which plays an important role in the overall cost. This is particularly true for *WR* plans where the buffer reduces the I/O cost since consecutive window queries are likely to access similar pages (notice that in both pure *WR* plans the number of page accesses

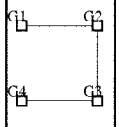
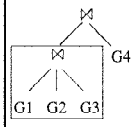
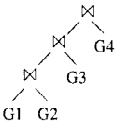
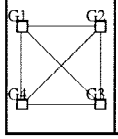
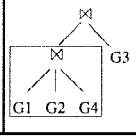
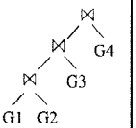
Query	optimal plan		cheapest WR		ST	
		ENA: 25338 NA: 54075 CPU: 22.61 I/O: 2468 %CPU: 47.8		ENA: 96168 NA: 135348 CPU: 48.55 I/O: 2360 %CPU: 67.3		ENA: 112984 NA: 133096 CPU: 60.56 I/O: 4085 %CPU: 59.7
		ENA: 15464 NA: 25865 CPU: 11.69 I/O: 2083 %CPU: 36		ENA: 93301 NA: 114071 CPU: 41.14 I/O: 2212 %CPU: 65		ENA: 24759 NA: 27316 CPU: 13.27 I/O: 2809 %CPU: 49.1

Fig. 13. Experimental results for real datasets.

is almost two orders of magnitude smaller than *NA*). For *ST* also, the CPU cost increases significantly with the number of variables because of the false hits at the high levels of the trees; for queries involving more than seven variables and normal buffer sizes, CPU time is more than 80% of the total cost [29].

Although *DP_plan* is applicable with trivial cost for small number of spatial inputs ($n \leq 10$), for large values of n its combinatorial explosion makes the method inefficient. Figure 14 demonstrates the cost of *DP_plan* (in seconds) as a function of n, for chain, star, and clique queries. If $n \geq 15$ the method is inapplicable for star and clique queries, because the optimization cost exceeds that of executing the optimal plan. Local search methods should be engaged for optimization in this case.

In order to test the performance of the hill climbing methods of Section 5.2, we optimized 50 random clique queries for each of the following values of n: 10, 15, ..., 50. The datasets were randomly generated with cardinality between 5,000 and 40,000, and density between 0.05 and 1.0. Setting runtime to 2 seconds for $n = 10$ as a base, the time limit of the algorithms for the various values of n was chosen using the function: $runtime(n) = (n/10)^2 \cdot runtime(10)$ (in accordance with similar experiments for relational joins [51]). Figure 15(a) illustrates the quality of the plans produced by simulated annealing (*SA*), iterative improvement (*II*), and their sorted variants. Quality

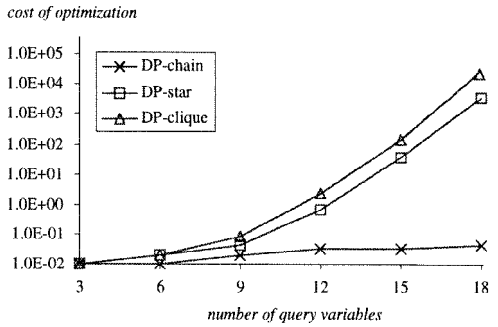
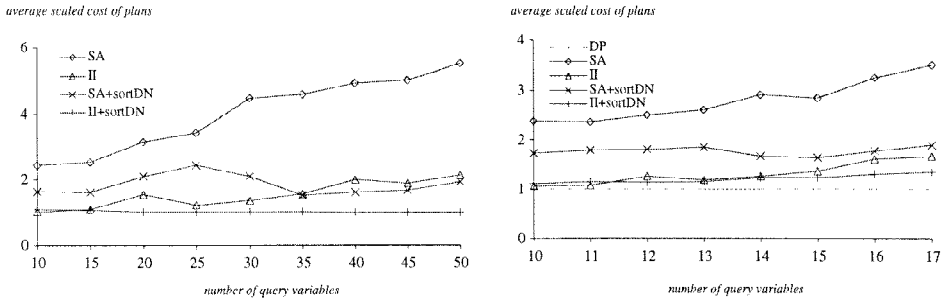


Fig. 14. Cost of *DP_plan*.



(a) Plans of hill climbing algorithms ($10 \leq n \leq 50$) (b) Comparison with the optimal plan

Fig. 15. Cost of optimization and quality of plans.

is measured in terms of *average scaled cost*, i.e., the cost of the output plan divided by the minimum cost produced by any algorithm for the same query. *II* performs better than *SA* in all cases, while the sorted versions of both algorithms produce better plans than the original ones. *II-sortDN* is consistently the most effective algorithm.

The set of experiments in Figure 15(b) compares the cost of plans produced by hill climbing algorithms scaled with respect to the cost of the optimal plan produced by *DP_plan*. Because of the combinatorial explosion of *DP_plan*, we were able to test the quality of the local optimization methods only for $n < 18$. Observe that *II-sortDN* finds a plan which is just 10–20% more expensive than the optimal; this percentage is not expected to grow significantly for larger values of n.

6.2. *Cost Parameters.* In this subsection we use the plan computed by *DP_plan* (for $n \leq 10$) and *II-sortDN* (for $n > 10$) to test how several parameters affect the performance of the proposed algorithms. The first experiment demonstrates the effect of data size; in particular we keep the number of variables and density fixed and measure the cost of multiway spatial joins by increasing the size of datasets. Figure 16 illustrates the actual node accesses (in thousands) and CPU time (in seconds) for datasets with 10k, 20k, . . . , 50k rectangles. For each dataset we also include the number of solutions retrieved (on top of the NA columns).

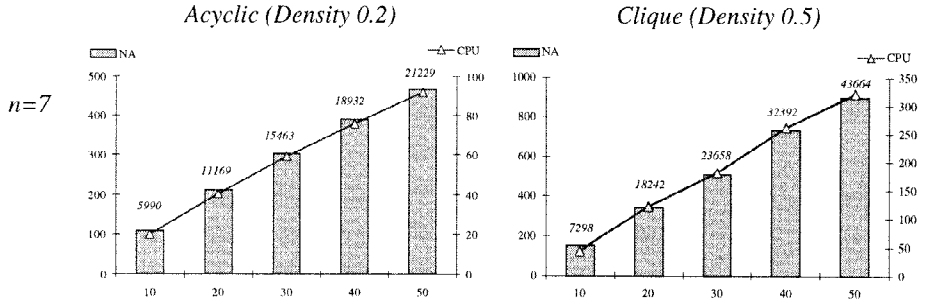


Fig. 16. Actual node accesses and CPU time as a function of data size N.

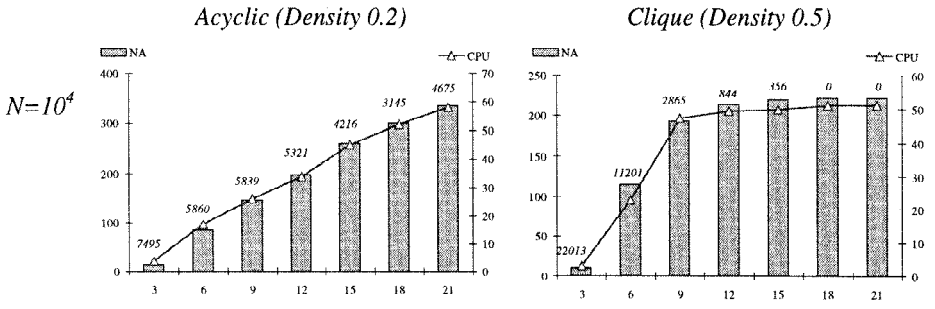


Fig. 17. Actual node accesses and CPU time as a function of query size n.

The cost, as well as the number of solutions, increases linearly with the size of the datasets. Notice that we chose different density values for acyclic (0.2) and for clique (0.5) queries, because these values give a reasonable number of solutions. Recall from Figure 11, that density 0.5 for acyclic queries (with seven variables) generates more than a million solutions. On the other hand, for cliques density 0.2 results in only 36 solutions.

For the second set of experiments data sizes and densities are fixed, and the number of variables ranges from 3 to 21 (Figure 17). There is a range of density values where the number of solutions does not vary considerably as a function of the number of variables. Density values above that range result in exponential growth in the number of solutions, while values below result in no solutions for large queries. As shown in the diagram for trees, when there is no significant change in the number of solutions, the cost increases linearly with the number of variables. On the other hand, density 0.5 for cliques is below the aforementioned threshold and queries with 18 or more variables do not have solutions. As a result, the cost almost stabilizes since search is abandoned when no solution can be found for a subset of variables.

The last set of experiments demonstrates the effectiveness of the algorithms for partial retrieval. We use the same settings ($N = 10{,}000$, $n = 7$) and measure the cost when only a subset (i.e., 10%, 20%, ...) of solutions is to be retrieved. As shown in Figure 18, the algorithms again demonstrate an output-sensitive behavior, the cost increasing linearly as a function of the percentage of retrieved solutions. This feature is important not only in the cases where a subset of the solutions is needed, but also when the results of the join

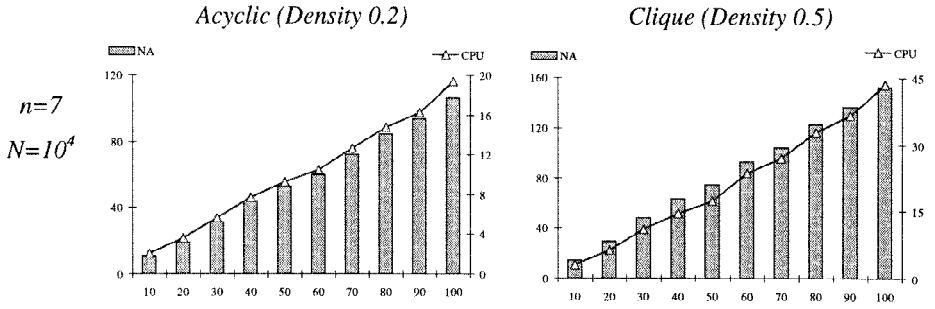


Fig. 18. Actual node accesses and CPU time as a function of the percentage of the retrieved solutions.

are passed to another operation through pipelining, or when the user wants to visualize the output during processing.

7. Conclusion. This paper describes a constraint-based approach for processing multiway spatial joins by combining systematic search algorithms with R-trees to guide search. In addition to methodologies, we propose cost models and optimization techniques and evaluate their effectiveness with extensive experimentation. Although we focused on joins involving *overlap*, the proposed techniques can be applied with a variety of spatial or temporal predicates. In [39] *ST* is compared with traditional systematic and local search algorithms using topological relations, while [28] shows how CSP algorithms can be applied with *WR* in the context of temporal constraint networks. In both cases the indexed versions clearly outperform the original algorithms for most experimental settings.

The current work can be extended in many ways. Park et al. [41] propose several optimization methods to speedup *ST*. These include a *space restriction ordering* which minimizes the pages to be loaded during space restriction, a new *plane sweep-based algorithm* which outperforms *FC* for some experimental settings, and an *indirect predicate heuristic* that detects false hits at the intermediate levels. In a recent work Mamoulis and Papadias [29] propose another efficient algorithm for *ST*, which combines plane sweep with *FC* and works by dividing each problem into smaller ones reducing the amount of backtracking. Furthermore, they compare *ST* with methods based on the integration of pairwise join algorithms for processing multiway spatial joins [27]. The results show that *ST* is preferable for dense data and queries, while in most cases the most efficient plan is one which combines *ST* with some pairwise join algorithm(s).

Another area with interesting open problems involves cost models. The selectivity formulas of Section 4 can be extended to arbitrary query graphs and nonuniform data. A more difficult extension refers to selectivity involving actual objects rather than MBRs. Given such formulas the filter and refinement steps could be interleaved as suggested in [42]. For instance, after a k-tuple has been returned by *ST* in a plan $P = ((v_1, \ldots, v_k), v_{k+1}, \ldots, v_n)$, it may be preferable to check if the tuple corresponds to an actual solution (i.e., refinement) before proceeding to *WR*. Furthermore, cost models for algorithms can be elaborated to capture page accesses in the presence of LRU buffers or other replacement policies.

Efficient multidimensional information processing becomes increasingly important as the availability of information in various forms (e.g., satellite images, digital video, multimedia documents) and the complexity of related applications increase continuously. Large systems must handle massive volumes of multidimensional data and answer on-line queries from numerous users. Several such systems (i.e., the DBMS for satellite images in [4]) and query languages for GIS (*Query-by-Sketch* [9]) and multimedia databases (*VisualSeek* [50]) already provide ways of expressing content-based queries that can be processed as multiway spatial joins. Therefore, the proposed techniques have a wide range of potential applications.

Acknowledgments. We thank Sophie Lamacq for her useful comments.

References

[1] Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S. Scalable Sweeping-Based Spatial Join. *Proc. VLDB*, pp. 570–581, 1998.

[2] Bacchus, F., van Run, P. Dynamic Variable Ordering in CSPs. *Principles and Practice of Constraint Programming*, pp. 258–275. LNCS 976, Springer-Verlag, Berlin, 1995.

[3] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. ACM SIGMOD*, pp. 322–331, 1990.

[4] Bergman, L., Castelli, V., Li, C.-S. Progressive Content-Based Retrieval from Satellite Image Archives. *D-Lib Magazine*, October 1997 (http://mirrored.ukoln.ac.uk/lis-journals/dlib/dlib/dlib/october97/ibm/10li.html).

[5] Brinkhoff, T., Kriegel, H., Seeger, B. Efficient Processing of Spatial Joins Using R-Trees. *Proc. ACM SIGMOD*, pp. 237–246, 1993.

[6] Cerny, V. Thermodynamical APproach to the Travelling Salesman Problem: An Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, 45: 41–51, 1985.

[7] Comer, D. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2): 121–138, 1979.

[8] Dechter, R. Decomposing a Relation into a Tree of Binary Relations. *Journal of Computer and System Sciences*, Special Issue on the Theory of Relational Databases, 41: 2–24, 1990.

[9] Egenhofer, M. Query Processing in Spatial-Query-by-Sketch. *Journal of Visual Languages and Computing*, 8: 403–424, 1997.

[10] Faloutsos, C., Ranganathan, M., Manolopoulos, Y. Fast Subsequence Matching in Time Series Databases. *Proc. ACM SIGMOD*, pp. 419–429, 1994.

[11] Gaede, V., Guenther, O. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2): 123–169, 1998.

[12] Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2): 73–170, 1993.

[13] Gudivada, V., Raghavan, V. Design and Evaluation of Algorithms for Image Retrieval by Spatial Similarity. *ACM TOIS*, 13(1): 115–144, 1995.

[14] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[15] Gyssens, M., Jeavons, P., Cohen, D. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artificial Intelligence*, 66(1): 57–89, 1994.

[16] Haralick, R.M., Elliott, G.L. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14: 263–313, 1980.

[17] Huang, Y.-W., Jing, N., Rundensteiner, E. Spatial Joins Using R-Trees: Breadth First Traversal with Global Optimizations. *Proc. VLDB*, pp. 396–405, 1997.

[18] Ioannidis, Y. Query Optimization. *ACM Computing Surveys*, 28(1): 121–123, 1996.

[19] Ioannidis, Y., Kang, Y. Randomized Algorithms for Optimizing Large Join Queries. *Proc. ACM SIGMOD*, pp. 312–321, 1990.

[20] Kamel, I., Faloutsos, C. Parallel R-Trees. *Proc. ACM SIGMOD*, pp. 195–204, 1992.

[21] Kamel, I., Faloutsos, C. On Packing R-Trees. *Proc. ACM CIKM*, pp. 490–499, 1993.

[22] Kirkpatrick, S., Gelat, C., Vecchi, M. Optimization by Simulated Annealing. *Science*, 220: 671–680, 1983.

[23] Koudas, N., Sevcik, K. Size Separation Spatial Join. *Proc. ACM SIGMOD*, pp. 324–333, 1997.

[24] Lo, M.-L., Ravishankar, C.V. Spatial Joins Using Seeded Trees. *Proc. ACM SIGMOD*, pp. 209–220, 1994.

[25] Lo, M.-L., Ravishankar, C.V. Spatial Hash-Joins. *Proc. ACM SIGMOD*, pp. 247–258, 1996.

[26] Mamoulis, N., Papadias, D. Constraint-Based Algorithms for Computing Clique Intersection Joins. *Proc. ACM-GIS*, pp. 118–123, 1998.

[27] Mamoulis, N., Papadias, D. Integration of Spatial Join Algorithms for Processing Multiple Inputs. *Proc. ACM SIGMOD*, pp. 1–12, 1999.

[28] Mamoulis, N., Papadias, D. Improving Search Using Indexing: A Study with Temporal CSPs. *Proc. IJCAI*, pp. 436–441, 1999.

[29] Mamoulis, N., Papadias, D. Synchronous R-Tree Traversal. Technical Report HKUST-CS99-03, February 1999. Available via ftp from http://www.cs.ust.hk/∼dimitris/

[30] Nabil, M., Ngu, A., Shepherd, J. Picture Similarity Retrieval using 2d Projection Interval Representation, *IEEE TKDE*, 8(4): 533–539, 1996.
[31] Nadel, B. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5: 188–224, 1989.
[32] Nahar, S., Sahni, S., Shragowitz, E. Simulated Annealing and Combinatorial Optimization. *Proc*. 23*rd Design Automation Conference*, pp. 293–299, 1986.
[33] Orenstein, J.A. Spatial Query Processing in an Object-Oriented Database System. *Proc*. *ACM SIGMOD*, pp. 326–336, 1986.
[34] Pagel, B.-W., Six, H. Are Window Queries Representative for Arbitrary Range Queries? *Proc*. *ACM PODS*, pp. 150–160, 1996.
[35] Pagel, B.-W., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance. *Proc*. *ACM PODS*, pp. 214–221, 1993.
[36] Papadias, D., Theodoridis, Y., Sellis, T., Egenhofer, M. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-Trees. *Proc*. *ACM SIGMOD*, pp. 92–103, 1995.
[37] Papadias, D., Theodoridis, Y., Stefanakis, E. Multidimensional Range Query Processing with Spatial Relations. *Geographical Systems*, 4(4): 343–365, 1997.
[38] Papadias, D., Mamoulis, N., Delis, B. Algorithms for Querying by Spatial Structure. *Proc*. *VLDS*, pp. 547–557, 1998.
[39] Papadias, D., Kalnis, P., Mamoulis, N. Hierarchical Constraint Satisfaction in Spatial Databases. *Proc*. *AAAI*, pp. 142–147, 1999.
[40] Papadopoulos, A.N., Rigaux, P., Scholl, M. A Performance Evaluation of Spatial Join Processing Strategies. *Proc*. *SSD*, pp. 286–307. LNCS 1651, Springer-Verlag, Berlin, 1999.
[41] Park, H., Cha, G., Chung, C.J.M. Multiway Spatial Joins Using R-Trees: Methodology and Performance Evaluation. *Proc*. *SSD*, pp. 231–250. LNSC 1651, Springer-Verlag, Berlin, 1999.
[42] Park, H., Lee, C.-G., Lee, Y.-J., Chung, C.-W. Early Separation of Filter and Refinement Steps in Spatial Query Optimization. *Proc*. *DASFAA*, pp. 161–168, 1999.
[43] Patel, J.M., DeWitt, D.J. Partition Bases Spatial-Merge Join. *Proc*. *ACM SIGMOD*, pp. 259–270, 1996.
[44] Petrakis, E., Faloutsos, C. Similarity Searching in Medical Image Databases. *IEEE TKDE*, 9(3): 435–447, 1997.
[45] Preparata, F., Shamos, M. *Computational Geometry*. Springer-Verlag, New York, 1988.
[46] Prosser, P. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3): 268–299, 1993.
[47] Roussopoulos, N., Kelley, F., Vincent, F. Nearest Neighbor Queries. *Proc*. *ACM SIGMOD*, pp. 71–79, 1995.
[48] Sellis, T., Roussopoulos, N., Faloutsos, C. The R^+-Tree: A Dynamic Index for Multidimensional Objects. *Proc*. *VLDB*, pp. 507–518, 1987.
[49] Silberschatz, A., Korth, H.F., Sudarshan, S. *Database System Concepts*. McGraw-Hill, New York, 1997.
[50] Smith, J., Chang, S.-F. VisualSEEk: A Fully Automated Content-Based Image Query System. *Proc*. *ACM Multimedia*, pp. 87–98, 1996.
[51] Swami, A., Gupta, A. Optimization of Large Join Queries. *Proc*. *ACM SIGMOD*, pp. 8–17, 1988.
[52] Theodoridis, Y., Sellis, T. A Model for the Prediction of R-Tree Performance. *Proc*. *ACM PODS*, pp. 161–171, 1996.
[53] Theodoridis, Y., Stefanakis, E., Sellis, T. Cost Models for Join Queries in Spatial Databases. *Proc*. *IEEE ICDE*, pp. 476–483, 1998.
[54] van der Bercken, J., Seeger, B., Widmayer, P. A Generic Approach to Bulk Loading Multidimensional Index Structures. *Proc*. *VLDB*, p. 406–415, 1997.