Processing and Optimization of Multiway Spatial Joins Using R-trees

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~dimitris/

Nikos Mamoulis
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~mamoulis/

Yannis Theodoridis
Computer Technology Institute
P.O. Box 1122
GR-26110 Patras, Greece
http://dias.cti.gr/~ytheod/

ABSTRACT

One of the most important types of query processing in spatial databases and geographic information systems is the *spatial join*, an operation that selects, from two relations, all object pairs satisfying some spatial predicate. A *multiway join* combines data originated from more than two relations. Although several techniques have been proposed for pairwise spatial joins, only limited work has focused on multiway spatial join processing. This paper solves multiway spatial joins by applying systematic search algorithms that exploit R-trees to efficiently guide search, without building temporary indexes or materializing intermediate results. In addition to general methodologies, we propose cost models and an optimization algorithm, and evaluate them through extensive experimentation.

Keywords

Multiway Spatial Joins, R-trees, Constraint Satisfaction.

1. INTRODUCTION

A multiway spatial join can be defined as follows: Given a set of spatial relations $\{R_1,..,R_i,..,R_j,..,R_n\}$, where $R_i = \{u_{i,1},\ldots,u_{i,N}\}$, and a set of binary spatial predicates $\{C_{ij} / 1 \leq i,j \leq n\}$, find all n-tuples $\{(u_{1,w},..,u_{i,x},..,u_{j,y},..,u_{n,z}) / \forall\ i,j,\ 1 \leq i,j \leq n,\ C_{ij}(u_{i,x},u_{j,y})\}$. In most cases the spatial predicate is *overlap* (*intersect*, *crosses*), but alternatively any predicate, such as *near*, *northeast*, *meet* could be used. When n=2, the above definition corresponds to pairwise spatial joins, for which several processing techniques have been developed. Some of these techniques assume the existence of spatial indices (R-trees) on both relations to be joined (e.g., [BKS93]), while others deal with non-indexed inputs (e.g., [LR94; KS97]).

Following the relational database methodology, multiway spatial joins (n>2) could be processed by integration of pairwise join algorithms [MP99a]. Consider the query: "find all cities *crossed by* a river which also *crosses* an industrial area" in an R-tree supported system. The solutions 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., [BKS93]); then joining the resulting rivers with the relation cities employing a method (e.g., [LR94]) applicable when only one R-tree (for cities) is available. An efficient execution plan can be determined using cost models for pairwise spatial joins [TSS98] and optimization methods for relational queries.

This paper follows a different direction, and discusses processing of multiway spatial joins using only R-trees without materializing intermediate results. Papadias et al. [PMD98], motivated by an interesting correspondence between multiway joins and constraint satisfaction problems (CSPs), combine systematic search algorithms (used for CSPs) and R-trees for the retrieval of object combinations matching (exactly or approximately) some input configurations. Mamoulis and Papadias [MP98] employ these methods for a special case of multiway spatial joins where there exists a join condition between all pairs of inputs. Here we apply and extend our work to arbitrary join conditions. In addition, we provide analytical formulae for the expected cost and test accuracy with extensive experimentation. Finally we propose optimization techniques that yield significant improvement over the original algorithms.

The rest of the paper is organized as follows: Section 2 describes R-trees and the most common types of queries for which they have been utilized. Section 3 proposes multiway join processing methodologies using R-trees and section 4 describes cost models and a query optimization algorithm based on data and query properties. Section 5 contains an experimental evaluation using various datasets and join graph topologies, and, finally, section 6 concludes the paper.

2. QUERY PROCESSING USING R-TREES

The R-tree data structure is a height-balanced tree where each node corresponds to a disk page in secondary memory. 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 that correspond to MBRs included in *N[i]*; s_k.*ref*

Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 1999.

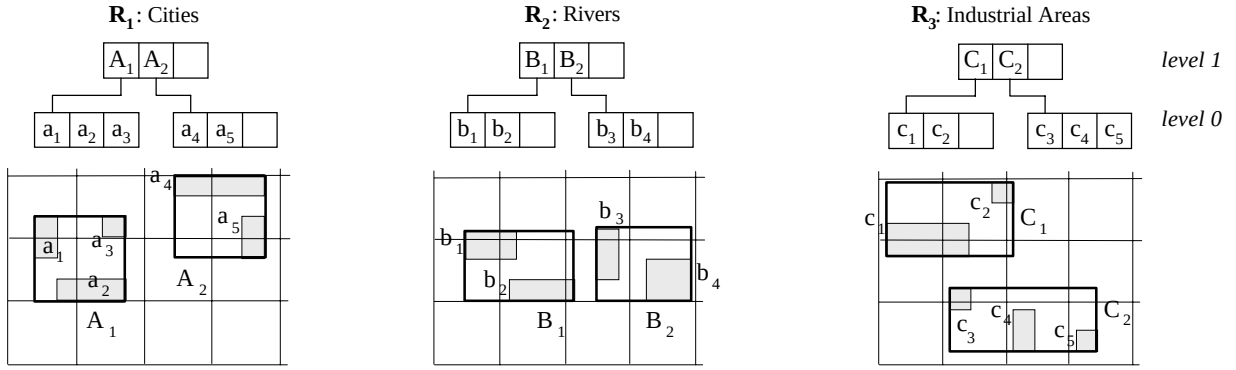

Figure 1 R-trees

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 of entries, called minimum R-tree node utilization *m*. Figure 1 illustrates three relations (covering the same workspace) and the corresponding R-trees, assuming that *m*=2 and maximum node capacity *C* is 3 rectangles (in real 2D applications *C* is normally 50-400 depending on the page size).

Selection and join queries are the fundamental operations in any DBMS, including spatial databases. 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.

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 window in the workspace. The processing of a window query using R-trees involves the following procedure (Figure 2): Starting from the top 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).

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

Figure 2 *WindowQuery*

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 two steps [O86]: (i) a *filter step* uses the tree to rapidly eliminate objects that could not possibly satisfy the query, and (ii) the results are passed through a *refinement step* where false hits are detected.

This paper, like most related spatial database literature, focuses on minimizing the cost of filtering. R-tree performance is usually measured in terms of the number of nodes that should be accessed during the search process. Let *d* be the dimensionality of the data space and $WS = [0,1)^d$ 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* $S(R_i,q,l)$ 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 [TS96] provide the following formula for selectivity, assuming unit workspace and square node rectangles ("*uniformity assumption*" [KF93]):

$$S(R_i,q,l) = (|s_{R_i,l}|+|q|)^d \qquad (1)$$

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 number $NA(R_i,q,l)$ of node accesses 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):

$$NA(R_i,q,l)=N_{R_i,l+1}\cdot S(R_i,q,l+1)=N_{R_i,l+1}\cdot (|s_{R_i,l+1}|+|q|)^d \qquad (2)$$

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:

$$Cost_{WQ}(R_i,q)=1+\sum_{l=0}^{h_{R_i}-2} NA(R_i,q,l)=1+\sum_{l=1}^{h_{R_i}-1} N_{R_i,l}\cdot\left(|s_{R_i,l}|+|q|\right)^d \qquad (3)$$

This formula is based on the performance analysis of [PST+93]. [TS96] defines the R-tree properties $h_{R_i,l}$, $N_{R_i,l}$, and $|s_{R_i,l}|$ involved in Eq. 3 as functions of N_{R_i} and D_{R_i}, denoting the cardinality and density[1] of the dataset, thus computing $NA(R_i,q,l)$ and $Cost_{WQ}(R_i,q)$ by using only data properties, without extracting information from the underlying R-tree structure. Pagel and Six

[1] The density *D* of a set of rectangles in *d*-dimensional space is defined as the average number of rectangles that contain a given point in the workspace. Equivalently, *D* can be expressed as the ratio of the sum of all rectangle areas over the area of the available workspace.

[PS96] argue that window queries are representative for range queries in general. Papadias et al. [PTS97] show how the above formulae 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"). The most influential algorithm for joining inputs indexed by R-trees is the *R-tree-based Spatial Join* (*RJ*) [BKS93], 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. Consider that we want to find 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 where overlapping pairs constitute solutions. Figure 3 illustrates the pseudo-code for *RJ* assuming that the trees are of equal height; the extension to different heights is straightforward. Huang et al. [HJR97] describe a breadth-first search I/O optimized version of the algorithm.

```
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])
```

Figure 3 *R-tree-based Spatial Join*

Initially, *RJ* takes as parameters the roots of the trees to be joined. Then it performs a synchronized 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. Since Eq. 2 calculates the number of node accesses at level l of R_i when a query window q is considered, it can be modified to calculate the cost of a join query by using the corresponding node entries $s_{R_j,l}$ of R_j as query windows. Thus, according to line 7 of the algorithm, the cost for both R-trees at level l is the sum of costs of $N_{R_j,l}$ different window queries on R_i [TSS98]:

$$NA(R_i,R_j,l)=NA(R_j,R_i,l) = N_{R_j,l+1} \cdot N_{R_i,l+1} \cdot (|s_{R_i,l+1}| + |s_{R_j,l+1}|)^d \quad (4)$$

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:

$$Cost_{RJ}(R_i,R_j) = 2 + \sum_{l=0}^{h_R-2} \{NA(R_i,R_j,l) + NA(R_j,R_i,l)\} =$$

$$2 + \sum_{l=1}^{h_R-1} \left\{ 2 \cdot N_{R_j,l} \cdot N_{R_i,l} \cdot \left(|s_{R_i,l}| + |s_{R_j,l}| \right)^d \right\} \quad (5)$$

Theodoridis et al. [TSS98] provide a detailed description of cost formulae 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 non-uniform (e.g., skewed) data distributions, they propose a maintenance of a grid with statistical information about cardinality and density per cell. This approach, applied with reasonably sized grid (50x50), provides good estimations for real datasets with highly skewed data distributions [MP99a].

Table 1 summarizes the symbols and definitions introduced in this section. In the sequel we show how they can be applied for multiway spatial joins.

Symbol	*Definition*
d	number of dimensions
h_{R_i}	height of the R-tree R_i
D_{R_i}	density of data MBRs indexed by 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 ($N_{r_i,l} \equiv N_{R_i}$)
$\|s_{R_i}\|$	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

Table 1 Table of symbols

3. 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. Equivalently, the graph can be viewed as a constraint network corresponding to a binary constraint satisfaction problem. A binary CSP [P93] is defined by:

- A set of n variables, $v_1,..,v_i, ..,v_n$
- For each variable v_i a finite domain $D_i = \{u_{i,1}, \ldots, u_{i,Ni}\}$ 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 simply 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, if city $u_{1,x}$ is crossed by river $u_{2,y}$ which also crosses industrial area $u_{3,z}$. Therefore, in the sequel we use the terms variable/dataset and constraint/join condition interchangeably.

Following the standard approach in the spatial join literature, we consider *overlap* as the default join condition. Furthermore, we focus on two particular types of multiway joins: acyclic (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 4a corresponds to a chain query (e.g., "find all cities *crossed by* a river which *crosses* an industrial area"), while 4b to a clique ("the industrial area should also intersect the city").

(a) Tree query

(b) Clique query

Figure 4 Examples of multiway spatial joins

Since multiway spatial joins can be modelled 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 [P93] for a survey). A naïve backtracking algorithm for processing the example query of Figure 4a (using the datasets of Figure 1) would first instantiate the variable corresponding to cities to some value (e.g., $v_1 \leftarrow a_1$) 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 (a_1,b_1,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.

Obviously the above algorithm performs a large number of redundant consistency checks because it does not exploit the underlying index structures. Several alternatives that take advantage of R-trees to speed-up search are presented in [PMD98]. These algorithms can be classified in two general methodologies which can be utilized for multiway spatial join processing as follows:

(i) The first methodology, called *window reduction* (*WR*), performs systematic search by applying window queries to find the consistent values of uninstantiated variables. For instance, after assigning $v_1 \leftarrow a_1$, 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 indexed nested loop fashion, while the backtracking phase can be based on various CSP algorithms. The order of variables is pre-determined according to some optimization method (see section 4), and is such that every variable after the first one should be directly connected to an instantiated variable (e.g., the order v_1,v_3,v_2 is not valid for the query of Figure 4a, 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 [MP98], since any set of MBRs that mutually overlap has a non-empty intersection. In Figure 4b, 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 a random 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.

(ii) 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. When a legal combination is found at the intermediate levels, the algorithm is recursively called, taking the references to the underlying nodes as parameters, until the leaf level is reached. For the query of Figure 4a, *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 8 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. 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.

The combination of *WR* and *ST* can yield significant performance improvement over the individual methods. *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 example query, *ST* could retrieve pairs of overlapping cities and rivers, and for each such pair *WR* will be called to find qualifying industrial areas. Obviously this technique can be applied with any number of variables. For instance, a query involving ten relations may be processed using *ST* for the first four variables, and *WR* to instantiate the rest. The pseudo-code in Figure 5 illustrates *hybrid*, a hybrid *ST*/*WR* routine which consists of two modules: the outer module is *WR* and the inner one is *ST*. *Hybrid* takes 3 input parameters:

- a n×n boolean array Q that stores the query graph to be executed. If for some i,j Q[i][j] is TRUE, the corresponding variables intersect. We assume that Q is connected; non-

```
1.  hybrid (Query Q[][], Rtree R[], int k)
2.  i := k; /*values for the first k variables come as k-tuples*/
3.  N[] := root nodes of R[];
4.  WHILE (TRUE) {
5.    IF i = k  /*values of first k variables*/
6.      THEN
7.        τ := ST(Q, N, k); /*get next valid  k-tuple output by ST - τ[i] stores the current value of variable v_i*/
8.        IF τ = NULL /*no more k-tuples are output by ST*/
9.          THEN RETURN;/*termination-backtrack from first k variables*/
10.     ELSE  /*values of (k+1)th and subsequent variables*/
11.       τ[i] := Query(R[i], WindowQuery[i]); /*next value from R[i] intersecting queryWindow[i]*/
12.       IF τ[i] = NULL /*empty domain for (k+1)th or later variable*/
13.         THEN i := i-1; GOTO 4; /*backtrack*/
14.         ELSE  /* not empty domain */
15.           FOR j=1 to i-1 /*check consistency of the value w.r.t other instantiated variables*/
16.             IF (Q[j][i]=TRUE) AND (τ[j]∩τ[i]=∅) /*τ[i] is inconsistent because it does not intersect τ[j]*/
17.               THEN GOTO 11; /*select new value of v_i */
18.   IF i = n /*last variable has been instantiated*/
19.     THEN output_solution(τ);
20.     ELSE  /*intermediate variable*/
21.       i = i+1; /*go forward */
22.       Set queryWindow[i];
23. } /* end WHILE */

    /*ST will return only one k-tuple τ every time it is called, or NULL if no more consistent tuples exist*/
24. ST (Query Q[][], RTreeNode N[], int k)
25. T:=find-solutions(Q,N,k); /*calls an algorithm to find all solutions (k-tuples) at current level (S is the # solutions)*/
26. FOR s=1 to S DO /* for each solution at the current level */
27.   FOR j=1 to k DO τ[j]=T[s][j] /* τ[1]… τ[k] holds the current solution */
28.   IF intermediate level
29.     THEN ST(Q, τ.references, k); /*recursively call ST for lower level*/
30.     ELSE PIPELINE(τ); /*leaf level -> return tuples to hybrid when needed until end of tuple_array */
```

Figure 5 The *hybrid* algorithm

connected graphs can be solved as independent sub-problems.

- an array of n R-trees that index the relations to be joined (R_i indexes variable/relation v_i). For simplicity, all R-trees are assumed to be of equal height, although the method can be easily extended for trees of different heights (similarly to *RJ*).
- a parameter k ($1 \leq k \leq n$) that denotes the number of variables to be joined by *ST*. If $k = 1$ ($k = n$), then *hybrid* is actually *WR* (*ST*).

Initially the index i to the current variable is set to k and the pointer of all R-tree nodes is set to the roots. Then *ST* retrieves a consistent tuple of values for the first k variables (lines 6-9). These values are stored in τ[1]… τ[k] (τ holds the current instantiations). When all such k-tuples are exhausted, *hybrid* terminates. Lines 10-17 correspond to instantiations of variables $v_{k+1},\ldots,v_n$. A value for the current variable is retrieved using a query window in the corresponding R-tree (line 11). If such a value cannot be found, the algorithm will backtrack (here we assume chronological backtracking).

Line 18 of the code will be reached only in the case of a successful instantiation. If the last variable has been instantiated, τ contains a complete solution which is output to the user. Otherwise, i is increased and the algorithm proceeds to the next variable. The query window for v_i (line 22), becomes the current value of the single instantiated variable connected to v_i (for acyclic queries), or the intersection of all current values (for cliques). For arbitrary graphs (i.e., v_i is connected to any number of instantiated variables) the value with the smallest query window is chosen and the results are filtered (lines 15-17) with respect to other instantiated variables joined with the current one.

ST is invoked each time there is a need for a new consistent k-tuple. The first time *ST* is called, it takes as parameters the roots of the first k R-trees. It then calls *find-solutions* to retrieve the consistent tuples for the current set of tree nodes. *Find-solutions* can be any CSP algorithm (*forward checking* was used in [PMD98]) enhanced with several heuristics to reduce the number of consistency checks. The solutions at each level (S denotes the total number of solutions) are stored in an array T[S][k]; a row in T corresponds to one solution τ. If *ST* runs for intermediate tree nodes it is recursively called for the lower nodes pointed by each solution. If it runs for the leaf nodes, it outputs one tuple and waits for the next call by *WR* to continue. This pipelining mechanism between *ST* and *WR* is implemented by buffering the paths of the current return of *ST* (the recursion stack), as well as the sets of k-tuples that have been found at the current level.

The application of *hybrid* in case where some or all of the variables have the same domain (i.e., image similarity retrieval applications) is straightforward. Furthermore, it can be effectively employed when only a subset of the solutions needs to be retrieved. For instance, it can be easily modified to terminate after the retrieval of the first solution resulting in significantly smaller execution cost. Multiway join processing based on integration of pairwise spatial join algorithms [MP99a], does not have this feature; spatial hash join algorithms 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 AND QUERY OPTIMIZATION

It is well-known in both the database [G93] and CSP [BvR95] communities that the order in which pairwise joins are performed, or otherwise the order in which variables get instantiated, has a very significant effect on performance. In the sequel we provide analytical formulae for the expected cost of multiway spatial joins and an optimization algorithm that determines the subset of variables to be instantiated by *ST* and the optimal order of the remaining ones to be instantiated by *WR*.

4.1 Selectivity of Multiway Spatial Joins

A solution of a query graph Q at level l is a n-tuple $(s_{R_1,l},..,s_{R_i,l},..,s_{R_j,l},..,s_{R_n,l})$ such that: $s_{R_i,l}$ is an entry at level l of R-tree R_i, and $\forall$i,j, $1 \le$ i,j $\le n$, Q[i][j]=TRUE $\Rightarrow$ $s_{R_i,l}$ overlaps $s_{R_j,l}$. 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 =#all n-tuples$\cdot$*Prob*(a n-tuple is a solution) (6)

The first part of the product in Eq. 6 equals the cardinality of the Cartesian product of n domains, while the second part corresponds to the query selectivity which equals the probability that all binary assignments $\{v_i \leftarrow s_{R_i,l}\ ,\ v_j \leftarrow s_{R_j,l}\}\ \forall$ i,j | Q[i][j] = TRUE are consistent. In case of acyclic graphs, and ignoring boundary effects (i.e. rectangles are small with respect to the workspace), these probabilities are independent. Let s_{R_i} be a data object in R_i with extent $|s_{R_i}|$ (equal to the average entry extent at level 0). 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 in Figure 4a is the product of pairwise selectivities:

$$Prob((s_{R_1}, s_{R_2}, s_{R_3}) \text{ is a solution}) = (|s_{R_1}| + |s_{R_2}|)^d \cdot (|s_{R_2}| + |s_{R_3}|)^d \quad (7)$$

In general, the selectivity of an acyclic join graph containing n variables is:

$$Prob(\text{a } n\text{-tuple is a solution}) = \prod_{\forall i,j:Q(i,j)=TRUE} \left(|s_{R_i}| + |s_{R_j}|\right)^d \quad (8)$$

and the total number of solutions at tree level l is:

$$\#solutions(Q,l) = \prod_{i=1}^{n} N_{R_i,l} \cdot \prod_{\forall i,j:Q(i,j)=TRUE} \left(|s_{R_i,l}| + |s_{R_j,l}|\right)^d \quad (9)$$

When the query graph contains cycles, the assignments are not independent anymore and Eq. 8 is an over-estimation of selectivity. 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. As we show in the Appendix, the average intersection area of two rectangles s_{R_1} and s_{R_2} is:

$$\left(\frac{|s_{R_1}| \cdot |s_{R_2}|}{|s_{R_1}| + |s_{R_2}|}\right)^d \quad (10)$$

Consider the instantiations $\{v_1 \leftarrow s_{R_1},\ v_2 \leftarrow s_{R_2}\}$ in the query of Figure 4b. 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}. By applying Eq. 10 for the intersection area of s_{R_1} and s_{R_2}, we derive:

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

$$= \left(|s_{R_1}| \cdot |s_{R_2}| + |s_{R_2}| \cdot |s_{R_3}| + |s_{R_1}| \cdot |s_{R_3}|\right)^d \quad (11)$$

In the general case, it can be shown that the average intersection area of n mutually overlapping rectangles s_{R_1}, .., s_{R_n} is:

$$\left(\frac{\prod_{i=1}^{n} |s_{R_i}|}{\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_{R_j}|}\right)^d \quad (12)$$

and the probability of a random n-tuple (s_{R_1}, .., s_{R_n}) to be a solution of a complete query graph Q with n nodes is:

Prob(a n-tuple is a solution) =

Prob(s_{R_2} overlaps s_{R_1})

$\cdot$*Prob*(s_{R_3} overlaps $s_{R_1} \wedge$ s_{R_3} overlaps s_{R_2}/ s_{R_1} s_{R_2} mutually overlap)

….

$\cdot$*Prob*(s_{R_n} overlaps $s_{R_1} \wedge .. \wedge s_{R_n}$ overlaps $s_{R_{n-1}}/s_{R_1}..s_{R_{n-1}}$ mut. overlap)

$$= \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_{R_j}|\right)^d \quad (13)$$

Detailed proofs of Eq. 12 and 13 can be found in the Appendix. Using Eq. 13 for selectivity, we obtain the number of solutions at level l:

$$\#solutions(Q,l) = \prod_{i=1}^{n} N_{R_i,l} \cdot \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_{R_j,l}|\right)^d \quad (14)$$

The experiments demonstrate that the above formulae are accurate and, therefore, can be applied for optimization of multiway spatial joins independently of the algorithms. In the sequel we show how they can estimate the cost of *hybrid*[2].

4.2 Cost Models for *Hybrid*

A subgraph $Q_{x,y}$ of Q containing x nodes (variables) is called *legal*[3] if it is connected; $V_{x,y}$ is the set of nodes in $Q_{x,y}$. The total number of legal subgraphs is less or equal (in the case of complete graphs) to 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 4a 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 sub-query of size x-1 and then using *WR* to instantiate the x^{th} 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 case of acyclic graphs $v_{x'}$ is connected with a single instantiated variable in $V_{x-1,y'}$ 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:

$$Cost_{WR}(Q_{x-1,y'},v_{x'}) = \#solutions(Q_{x-1,y'},0)\cdot Cost_{WQ}(R_{y'},q_{x'}) \qquad (15)$$

where $Cost_{WQ}$ is computed according to Eq. 3, and the number of solutions according to Eq. 9 or 14, 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 a root level solution. 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. Thus the total cost of *ST* is:

$$Cost_{ST}(Q_{x,y}) = x + \sum_{l=0}^{h-2} x\cdot \#solutions\left(Q_{x,y}, l+1\right) \qquad (16)$$

Let $P = ((v_1,..,v_k),v_{k+1},..,v_n)$ be a plan where the first k variables are instantiated through *ST* and the rest by *WR* in this order, and $Q_{x,p}$ be a sub-graph containing the first x variables of P. The total cost of processing P is:

$$Cost(P) = Cost_{ST}(Q_{k,p}) + \sum_{x=k+1}^{n} Cost_{WR}(Q_{x-1,p}, v_x) \qquad (17)$$

[2] *Hybrid* is applicable for queries containing arbitrary cycles. Optimization of such queries using Eq. 9 and 14 as bounds for the number of solutions, however, is not accurate. Notice that most related literature in relational multiway join processing deals with acyclic graphs.

[3] We use index y to distinguish different legal subgraphs of x nodes.

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 [IK91] in the sense that the leftmost (deepest) leaf nodes are synchronously traversed (plans are not necessarily binary trees). Figure 6 illustrates the alternative plans for the query of Figure 4a, 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.

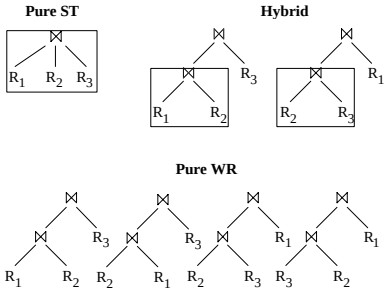

Figure 6 Possible plans for the query of Figure 4a

Let $p(x)$ be a function that returns the number of plans for a legal subgraph of x nodes, and $d(x)$ a function that returns the number of legal decompositions. We assume 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:

$$p(x)=d(x)\cdot p(x\text{-}1)+1 \text{ and } p(1)=1 \qquad (18)$$

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 18, we derive that the number of alternative plans for chain queries is: 2^n-1. Eq. 18 cannot be applied for arbitrary trees, because $d(x)$ may be different for two sub-graphs 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. Similarly, 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:

$$n!\cdot\sum_{x=1}^{n}\frac{1}{x!} < n!\cdot e \qquad (19)$$

This is significantly smaller than the corresponding number in relational queries, i.e., (2(n-1))! / (n-1)! [SKS97], because there do not exist right-deep or bushy plans. In the next section we describe a dynamic programming algorithm that determines the optimal execution plan by searching through the whole plan space.

4.3 Optimization with Dynamic Programming

Dynamic programming has been successfully applied for optimization of relational queries involving a small number of inputs [I96]. *Hybrid-plan* computes the best execution strategy incrementally, based on the optimal plans of its subgraphs. The recursive equation implemented by the algorithm is:

$$Cost(Q_{x,y})=\min\{Cost_{ST}(Q_{x,y}),\ \min_{\forall decomposition\ y'}(Cost(Q_{x-1,y'})+Cost_{WR}(Q_{x-1,y'},v_{x'}))\} \quad (20)$$

In general, at each level *hybrid-plan* decomposes every $Q_{x,y}$ into all legal combinations $(Q_{x-1,y'},v_{x'})$, and finds the best decomposition 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 sub-graphs of size x+1.

```
1.  Hybrid-plan(Query 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;
```

Figure 7 *Hybrid-plan*

Cost[$Q_{1,y}$] is initially the number of leaf nodes in each R-tree R_y (i.e., the number $N_{Ry,1}$ 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 (indexed nested loop). For all pairwise joins, the best of the 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 *hybrid-plan*, *bestPlan*[*Q*] will contain the optimal plan for executing *Q*, and *Cost*[*Q*] its expected cost.

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 $Q_{x,y}$, and for each $Q_{x,y}$ it will perform x decompositions. Thus. the total running time (assuming constant table writing and look-up) is:

$$\sum_{x=2}^{n}\frac{n!}{(x-1)!(n-x)!} \quad (21)$$

Only the optimal cost and the number of solutions[4] of each sub-graph with size x-1 has to be maintained for the calculation of the optimal costs of sub-graphs with size x; thus, the space requirements of *hybrid-plan* at iteration x of the outer loop are $C(x\text{-}1,n) + C(x,n)$. The time and space requirements of the algorithm renders exhaustive optimization inapplicable for queries involving numerous relations. In [PMT99] we present two local search techniques (based on *iterative improvement* and *simulated annealing*) that efficiently generate nearly optimal plans for large number of inputs.

[4] Lines 4 and 7 use Eqs. 16 and 15 which require the expected number of solutions. This number is also stored, but, for simplicity, is omitted in the pseudo-code.

5. 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 [PTS97]. Following the standard experimental methodology in the spatial join literature, in this section we evaluate them by assuming that the spatial predicate is always *overlap*.

All experiments were executed on an Ultrasparc2 workstation (200 MHz) with 256 Mbytes of memory. The implementation of *WR* is based on chronological backtracking (as in Figure 5). The overhead of algorithms (e.g., *backjumping* [D90]) 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. *Find-solutions* (the core of *ST*) is based on forward checking [HE80; BG95], but uses the basic idea of plane sweep [PS85] to reduce the number of consistency checks. Both implementations of *WR* and *ST* are basic in the sense that we did not include heuristics (such as space restriction [BKS93; PMD98]) to speed-up search.

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 ran tree and clique queries involving 7 variables using datasets of various densities. The cardinality of all datasets is fixed to 10,000 uniformly distributed rectangles[5], while the density D has four potential values: 0.05, 0.20, 0.35, and 0.50. There is a total of 4x2 (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. The cost of optimization was less than 1% of the cost of processing the optimal plan.

Table 2 illustrates actual (NA), estimated[6] (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). We 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. Several observations can be made based on the results[7]:

1. 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%.

[5] Page size is set to 1KB resulting in R*-trees [BKS+90] with node capacity 50 and height 3.

[6] For the estimation of node extents $|s_{R_i}|$ we use statistical information from the tree (rather than the analytical formulae of [TS96]) because they provide higher accuracy.

[7] Notice that 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.

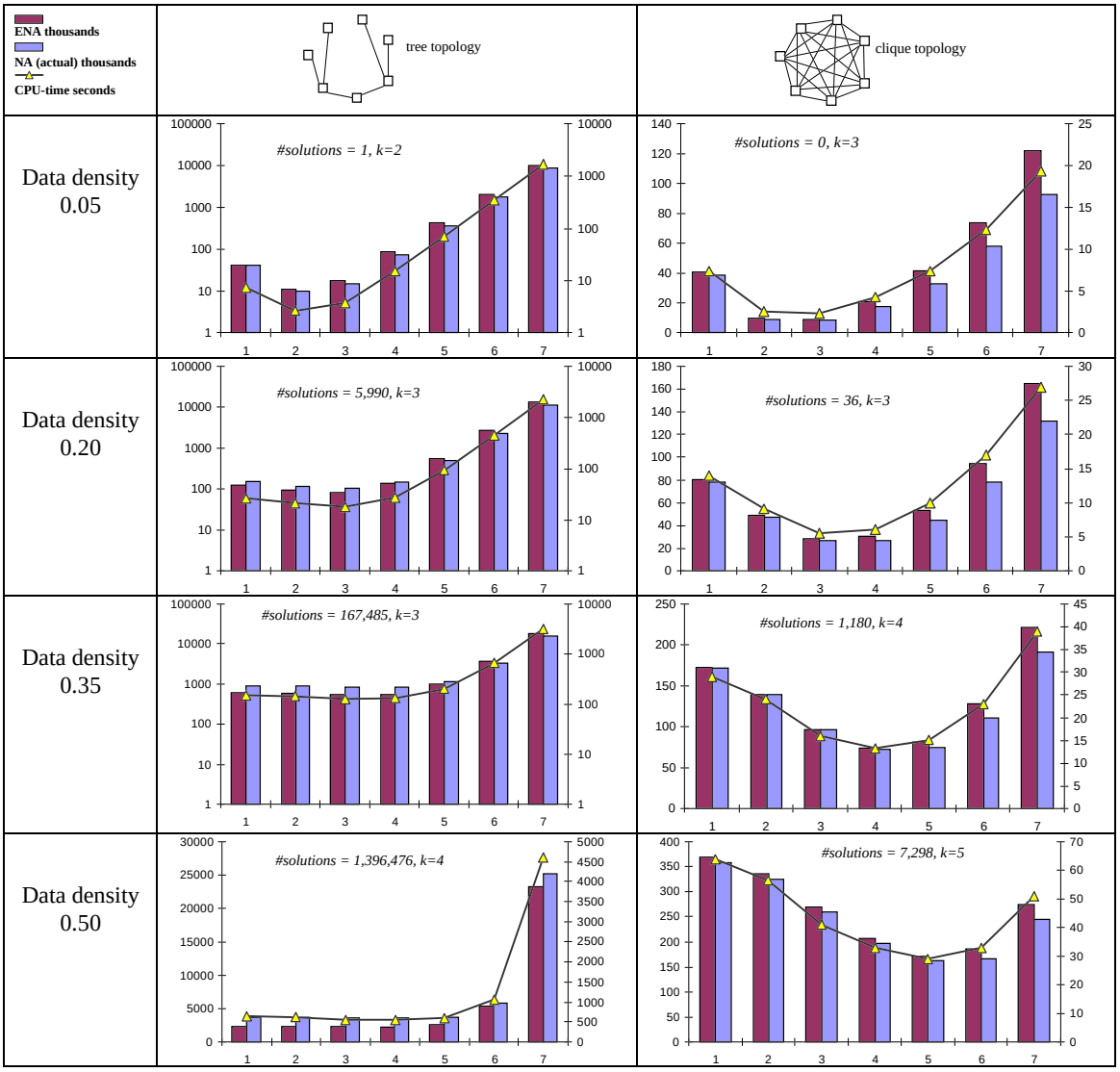

Table 2 Actual and estimated node accesses and CPU-time for various combinations of data / query densities

2. 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.
3. There are vast performance differences (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 the following experiments we use the suggested optimal plan and measure the effect of the data size and the number of inputs on the performance of *hybrid*. Firstly, we keep the number of variables and density fixed, and investigate the cost of multiway spatial joins as a function of the input cardinality. Table 3 (first row) 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 (on top of the NA columns). The cost, as well as the number of solutions, increases linearly with the size of the datasets. Notice that we chose different densities (D=0.2) for acyclic, and for clique (D=0.5) queries, because these values give a reasonable number of solutions. D=0.5 for acyclic queries (with n=7, N=10K) generates more than 10^6 solutions, while for cliques D=0.2 results in only 36 solutions.

For the last set of experiments data sizes and densities are fixed, and the number of variables ranges from 3 to 21. For queries involving more than 12 inputs, the optimal plan was computed using the local search techniques of [PMT99] because exhaustive optimization was prohibitively expensive. Nevertheless, for most practical applications, the number of relations is less than 10, and dynamic programming suffices.

Table 3, second row, illustrates the NA and CPU-time as a function of n. 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, cliques queries with 18 or more variables do not have

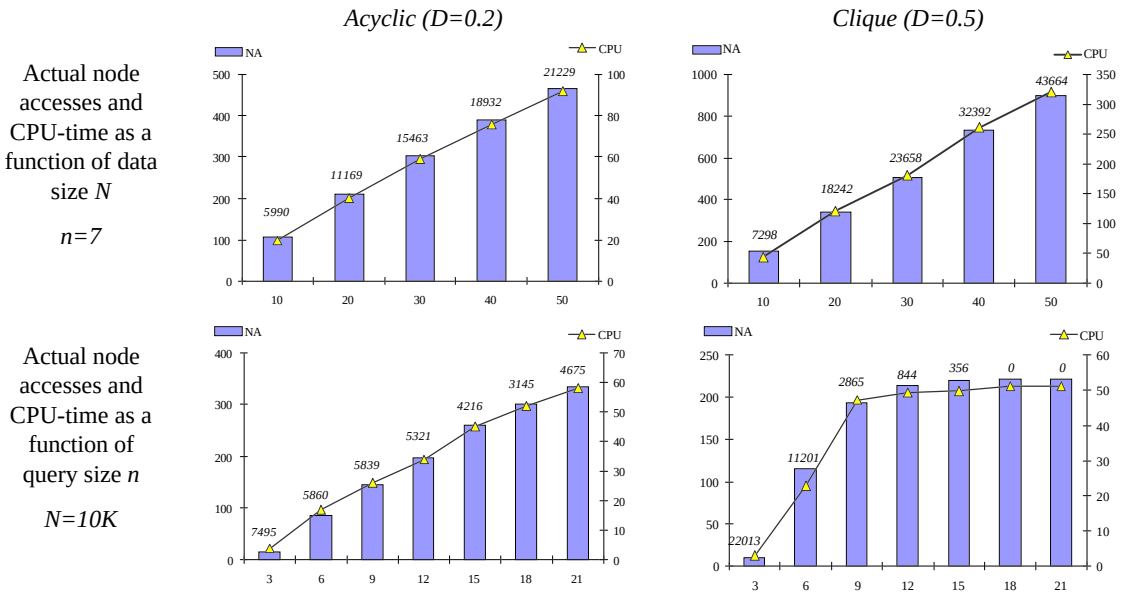

Table 3 NA (in thousands) and CPU time (in seconds) as a function of *n* and *N*

solutions. As a result, the cost almost stabilizes since search is abandoned when no solution can be found for a subset of variables.

6. CONCLUSION

In this paper, we propose a complete method for multiway spatial join processing and optimization which is motivated by a close correspondence between multiway joins and CSPs. The advantages of our approach are: i) it is efficient, ii) it does not materialize intermediate results, iii) its cost is predictable by accurate analytical formulae, iv) it is relatively simple to implement on top of an R-tree supported system, v) it can be easily extended to capture any spatial predicate, and vi) it can be modified for other spatial access methods that are based on hierarchical decomposition of space.

Some preliminary comparisons with methods based on integration of (pairwise) spatial join algorithms [MP99a] indicate that constraint-based methods perform better for dense queries and datasets, because they take advantage of multiple joins to restrict the search space. An interesting direction for future work is the combination of our techniques with pairwise join algorithms; for instance, we could split a query graph in two (or more) subgraphs to be processed by *ST* (or another method) and then combine the intermediate results using spatial join algorithms for non-indexed inputs (e.g., [KS97]). Such a methodology would efficiently support parallel processing of multiway spatial joins.

Another direction is the development of more efficient algorithms and pre-processing heuristics. Assume, for instance, that *ST* or *WR* outputs the solutions (a_1,b_1), (a_2,b_1) and (a_3,b_1) for the first two inputs of the chain query in Figure 4a. Instead of immediately performing a window query on the third input for each solution, we could partially materialize the intermediate results and process the three solutions together. In this way only one query window is needed for the instantiation $v_2 \leftarrow b_1$.

Finally, the proposed techniques can be applied in other application domains. Mamoulis and Papadias [MP99b] integrate the basic idea of *WR* with backtracking and forward checking to solve temporal CSPs where the variable domains consist of numerous intervals. Their experimental comparison suggests that indexing can speed up search several times compared to traditional CSP algorithms. The same ideas could also be applied with other types of CSP problems involving large domains.

ACKNOWLEDGEMENTS

Dimitris Papadias and Nikos Mamoulis were supported by grant HKUST 6151/98E from Hong Kong RGC and grant DAG97/98.EG02. Part of this work was done while Yannis Theodoridis was with the National Technical University of Athens, supported by the EU project *"Chorochronos:* A research network for spatiotemporal database systems". We would like to thank Sophie Lamacq and Panos Kalnis for their useful comments.

REFERENCES

[BG95] Bacchus, F., Grove, A. On the Forward Checking Algorithm. *Principles and Practice of Constraint Programming*, 1995.

[BvR95] Bacchus, F., van Run, P. Dynamic Variable Ordering in CSPs. *Principles and Practice of Constraint Programming*, 1995.

[BKS+90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD*, 1990.

[BKS93] Brinkhoff, T., Kriegel, H. Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *ACM SIGMOD*, 1993.

[D90] Dechter, R. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41: 273-312, 1990.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD*, 1984.

[G93] Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2): 73-170, 1993.

[HJR97] Huang, Y.-W., Jing, N., Rundensteiner, E. Spatial Joins using R-trees: Breadth First Traversal with Global Optimizations. *VLDB*, 1997.

[IK91] Ioannidis, Y., Kang, Y. Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implication for Query Optimization. *ACM SIGMOD*, 1991.

[I96] Ioannidis, Y. Query Optimization. *ACM Computing Surveys*, 28(1), 121-123, 1996.

[KF92] Kamel, I., Faloutsos, C., Parallel R-trees. *ACM SIGMOD*, 1992.

[KS97] Koudas, N., Sevcik, K., Size Separation Spatial Join. *ACM SIGMOD*, 1997.

[LR94] Lo, M.-L., Ravishankar, C.V. Spatial Joins Using Seeded Trees. *ACM SIGMOD*, 1994.

[MP98] Mamoulis, N., Papadias, D. Constraint-based Algorithms for Computing Clique Intersection Joins. *ACM-GIS*, 1998.

[MP99a] Mamoulis, N., Papadias, D. Integration of Spatial Join Algorithms for Processing Multiple Inputs. *ACM SIGMOD*, 1999.

[MP99b] Mamoulis, N., Papadias, D. Improving Search Using Indexing: a Study with Temporal CSPs. *IJCAI*, 1999.

[O86] Orenstein, J. A. Spatial Query Processing in an Object-Oriented Database System. *ACM SIGMOD*, 1986.

[P93] Prosser, P. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3): 268-299, 1993.

[PS96] Pagel, B.-W., Six, H. Are Window Queries Representative for Arbitrary Range Queries? *ACM PODS*, 1996.

[PST+93] Pagel, B.-W., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance. *ACM PODS*, 1993.

[PMD98] Papadias, D., Mamoulis, N., Delis, B. Algorithms for Querying by Spatial Structure. *VLDB*, 1998.

[PMT99] Papadias, D., Mamoulis, N., Theodoridis, Y. Constraint-based Processing of Multiway Spatial Joins. *TR-HKUST-CS99-02*, 1999. Available via ftp from http://www.cs.ust.hk/~dimitris/

[PTS97] Papadias, D., Theodoridis, Y., Stefanakis, E. Multidimensional Range Query Processing with Spatial Relations. *Geographical Systems*, 4(4): 343-365, 1997.

[PS85] Preparata, F., Shamos, M. *Computational Geometry*. Springer, 1988.

[SKS97] Silberschatz, A., Korth, H. F., Sudarshan, S. *Database System Concepts*. McGraw-Hill, 1997.

[TS96] Theodoridis, Y., Sellis, T. A Model for the Prediction of R-tree Performance. *ACM PODS*, 1996.

[TSS98] Theodoridis, Y., Stefanakis, E., Sellis, T. Cost Models for Join Queries in Spatial Databases, *IEEE ICDE*, 1998.

APPENDIX

In this Appendix we prove Eqs 12 and 13, used for the calculation of the number of solutions in case of clique queries.

Lemma A1: Given a set of n ($n \geq 2$) mutually overlapping rectangles s_i, $i = 1, \ldots, n$, with average extent $|s_i|$ on each direction and assuming uniformity and independence, the common intersection area is a rectangle q_n of average extent $|q_n|$ defined as follows:

$$|q_n| = \frac{\prod_{i=1}^{n} |s_i|}{\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_j|} \qquad \text{(A.1)}$$

Proof (*by induction on n*):

Step 1: For $n = 2$, it is sufficient to prove that

$$|q_2| = \frac{|s_1| \cdot |s_2|}{|s_1| + |s_2|} \qquad \text{(A.2)}$$

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 $s_{i.start}$ ($s_{i.end}$) be their projections' start (end) points, $i = 1, 2$. Figure A.1 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): $s_{2.start} < s_{1.start} < s_{2.end} < s_{1.end}$

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

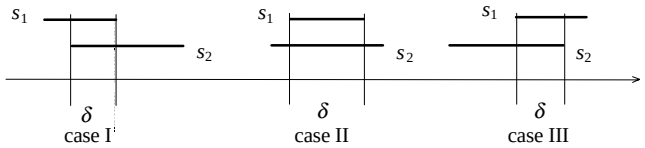

Figure A.1 Possible configurations of overlapping intervals

Assuming that the address space is discrete, with very fine granularity (the case of continuous space will be the limit for infinitely fine granularity) [KF92] the probability for each specific configuration corresponds to the different relative positions of s_1 with respect to s_2. Formally:

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

and

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

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 (i) and (iii) and $\delta = |s_1|$ for (ii). In turn, the average extent $|q_2|$ equals the weighted average δ size, i.e.,

$$|q_2| = 2 \cdot \left(\frac{|s_1|}{|s_1|+|s_2|} \cdot \frac{|s_1|}{2} \right) + \frac{|s_2|-|s_1|}{|s_1|+|s_2|} \cdot |s_1| \qquad \text{(A.3)}$$

where the first part of the summation represents the (equal) weighted average δ for cases (i) and (iii) while the second part corresponds to (ii).

Since Eq. A.3 implies Eq. A.2, step 1 of the proof has been completed.

Step 2: We assume that Eq. A.1 holds for $n = k$, i.e.,

$$|q_k| = \frac{\prod_{i=1}^{k} |s_i|}{\sum_{i=1}^{k} \prod_{\substack{j=1 \\ j \neq i}}^{k} |s_j|} \qquad \text{(A.4)}$$

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

$$|q_{k+1}| = \frac{\prod_{i=1}^{k+1} |s_i|}{\sum_{i=1}^{k+1} \prod_{\substack{j=1 \\ j \neq i}}^{k+1} |s_j|} \qquad \text{(A.5)}$$

Proof of the induction step: Since rectangle s_{k+1} overlaps all s_1, …, 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, …, 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 Eqs A.2 and A.4:

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

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

q.e.d. •

Corollary: Given a random n- tuple of rectangles (s_1, .., s_n), the probability that all rectangles mutually overlap is:

$$Prob(\text{rectangles } s_1 \ldots s_n \text{ mutually overlap}) = \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_j| \right)^d \qquad \text{(A.6)}$$

Proof: Since all rectangles mutually overlap, without loss of generality we assume that the instantiation order is s_1, …, s_n. Thus, the left part of Eq A.6 is equal to a product of independent probabilities:

$$\begin{aligned} &Prob(\text{rectangles } s_1 \ldots s_n \text{ mutually overlap}) = \\ &= Prob(s_2 \text{ overlaps } s_1) \\ &\cdot Prob(s_3 \text{ overlaps } s_1 \wedge s_3 \text{ overlaps } s_2 / s_1, s_2 \text{ mutually overlap}) \\ &\cdot \ldots \\ &\cdot Prob(s_n \text{ overlaps } s_1 \wedge \ldots \wedge s_n \text{ overlaps } s_{n-1} / s_1, \ldots, s_{n-1} \text{ mut. overlap}) \end{aligned} \qquad \text{(A.7)}$$

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 and its area is calculated according to Lemma A1. Hence Eq. A.7 is equivalent to

$$\begin{aligned} &Prob(\text{rectangles } s_1 \ldots s_n \text{ mutually overlap}) = \\ &= Prob(s_2 \text{ overlaps } s_1) \cdot Prob(s_3 \text{ overlaps } q_2) \\ &\cdot \ldots \cdot Prob(s_n \text{ overlaps } q_{n-1}) \end{aligned} \qquad \text{(A.8)}$$

As discussed in subsection 2.1, the probability that a member of a set of rectangles overlaps a given rectangle q is, by definition, equal to the selectivity of q on the set of rectangles, which is computed according to Eq. 1. Thus Eq. A.8 is equivalent to

$$\begin{aligned} &Prob(\text{rectangles } s_1 \ldots s_n \text{ mutually overlap}) = \\ &= (|s_2| + |s_1|)^d \cdot (|s_3| + |q_2|)^d \cdot \ldots \cdot (|s_n| + |q_{n-1}|)^d \end{aligned} \qquad \text{(A.9)}$$

Substituting Eq A.1 in Eq A.9 we obtain:

$$Prob(\text{rectangles } s_1 \ldots s_n \text{ mutually overlap}) =$$

$$= (|s_2| + |s_1|)^d \cdot \left(|s_3| + \frac{|s_1| \cdot |s_2|}{(|s_1| + |s_2|)} \right)^d \cdot \ldots \cdot \left(|s_n| + \frac{\prod_{i=1}^{n-1} |s_i|}{\sum_{i=1}^{n-1} \prod_{\substack{j=1 \\ j \neq i}}^{n-1} |s_j|} \right)^d$$

$$= \left((|s_2| + |s_1|) \cdot \left(\frac{|s_1| \cdot |s_2| + |s_1| \cdot |s_3| + |s_2| \cdot |s_3|}{(|s_1| + |s_2|)} \right) \cdot \ldots \cdot \left(\frac{\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_j|}{\sum_{i=1}^{n-1} \prod_{\substack{j=1 \\ j \neq i}}^{n-1} |s_j|} \right) \right)^d$$

$$= \sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |s_j|$$

q.e.d.•