Chapter VII

Spatial Joins: Algorithms, Cost Models and Optimization

Nikos Manoulis
University of Hong Kong, Hong Kong

Yannis Theodoridis
University of Piraeus, Greece

Dimitris Papadias
Hong Kong University of Science and Technology, Hong Kong

Abstract

This chapter describes algorithms, cost models and optimization techniques for spatial joins. Joins are among the most common queries in Spatial Database Management Systems. Due to their importance and high processing cost, a number of algorithms have been proposed covering all possible cases of indexed and non-indexed inputs. We first describe some popular methods for processing binary spatial joins and provide models for selectivity and cost estimation. Then, we discuss evaluation of multiway

spatial joins by integrating binary algorithms and synchronous tree traversal. Going one step further, we show how analytical models can be used to combine the various join operators in optimal evaluation plans. The chapter can serve as a comprehensive reference text to the researcher who wants to learn about this important spatial query operator and to the developer who wants to include spatial query processing modules in a Database System.

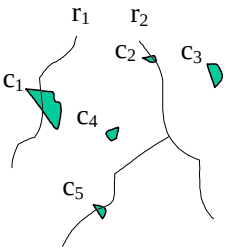
Introduction

Spatial database systems (Güting, 1994) manage large collections of multidimensional data which, apart from conventional features, include special characteristics such as position and extent. That there is no total ordering of objects in space that preserves proximity renders conventional indexes, such as B^+-trees, inapplicable to spatial databases. As a result, a number of *spatial access methods* have been proposed (Gaede & Günther, 1998). A very popular method, used in several commercial systems (for example, Informix and Oracle), is the R-tree (Guttman, 1994), which can be thought of as an extension of B^+-tree in multi-dimensional space. R-trees index object approximations, usually minimum bounding rectangles (MBRs), providing a fast *filter step* that excludes all objects that cannot satisfy a query. A subsequent *refinement step* uses the geometry of the candidate objects (that is, the output of the filter step) to dismiss false hits and retrieve the actual solutions. The R-tree and its variations have been applied to efficiently answer several query types, including spatial selections, nearest neighbors and spatial joins.

As in relational databases, joins play an important role in effective spatial query processing. A *binary* (that is, *pairwise*) spatial join combines two datasets with respect to a spatial predicate (usually *overlap*/*intersect*). A typical example is "find all pairs of cities and rivers that intersect." For instance, in Figure 1 the result of the join between the set of cities $\{c_1, c_2, c_3, c_4, c_5\}$ and rivers $\{r_1, r_2\}$, is $\{(r_1, c_1), (r_2, c_2), (r_2, c_5)\}$.

The query in this example is a spatial *intersection* join. In the general case, the join predicate could be a combination of *topological*, *directional* and *distance* spatial relations. Apart from the intersection join, variants of the *distance* join have received considerable attention because they find application in data analysis tasks (for example, data mining and clustering). Given two sets *R* and *S* of spatial objects (or multidimensional points) and a distance function *dist*(), the *μ-distance* join (or else *similarity* join) (Koudas & Sevcik, 2000) returns the

Figure 1. Graphical example of a spatial intersection join

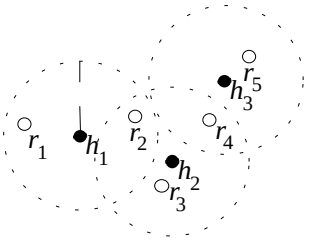


pairs of objects $\{(r, s): r \in R, s \in S, dist(r, s) \leq \mu\}$. A *closest pairs query* (Corral, Manolopoulos, Theodoridis, Vassilakopoulos, 2000) returns the set of closest pairs $CP=\{(r, s): r \in R, s \in S\}$, such that $dist(r, s) \leq dist(r', s')$, for all $r' \in R$, $s' \in S$: $(r', s') \notin CP$. A similar (non-commutative) operator is the *all k-nearest neighbors query* (Böhm & Krebs, 2002), which returns for each object from R its k nearest neighbors in S. Finally, given two datasets R and S, a real number μ and an integer t, the *iceberg distance join* (Shou, Mamoulis, Cao, Papadias, Cheung, 2003) retrieves all pairs of objects from R and S such that: (i) the pairs are within distance ε, and (ii) an object of R appears at least t times in the result (for example, find all regions of R that are within distance 1 km from at least 10 regions of S).

As an example for the spatial join variants, consider Figure 2, which illustrates a set of hotels $\{h_1, h_2, h_3\}$ and a set of restaurants $\{r_1, r_2, r_3, r_4, r_5\}$. The ε-distance join between these two sets returns seven pairs (h_1, r_1), (h_1, r_2), (h_2, r_2), (h_2, r_3), (h_2, r_4), (h_3, r_4) and (h_3, r_5). The 3-closest pairs are (h_2, r_3), (h_3, r_5) and (h_3, r_4). The *all 1-nearest neighbor* operator (for the hotels) returns (h_1, r_2), (h_2, r_3) and (h_3, r_5). Note that ε is not involved in closest pairs and all k-nearest neighbors operations. Finally, the iceberg distance join for t=3 returns (h_2, r_2), (h_2, r_3) and (h_2, r_4). Observe that h_2 is the only hotel with at least 3 nearby restaurants.

In this chapter, we focus on intersection joins by reviewing evaluation algorithms and cost models, as well as techniques for optimizing and integrating them in a spatial database query engine. Other variants of spatial joins can be processed by (trivially or non-trivially) extending algorithms for intersection joins. The interested reader should check Koudas and Sevcik (2000), Corral et al. (2000), Böhm and Krebs(2002) and Shou et al .(2003) for details.

Figure 2. Example of distance join variants



Binary Spatial Joins

Most early spatial join algorithms apply a transformation of objects in order to overcome the difficulties raised by their spatial extent and dimensionality. The first known algorithm (Orenstein, 1986) uses a grid to regularly divide the multidimensional space into small blocks, called *pixels*, and employs a space-filling curve (z-ordering) to order them. Each object is then approximated by the set of pixels intersected by its MBR, that is, a set of z-values. Since z-values are 1-dimensional, the objects can be dynamically indexed using relational index structures, like the B^+-tree, and the spatial join can be performed in a sort-merge join fashion. The performance depends on the granularity of the grid; larger grids can lead to finer object approximations, but also increase the space requirements. Rotem (1991) proposes an algorithm based on a spatial join index similar to the relational join index, which partially pre-computes the join result and employs grid files to index the objects in space.

The most influential algorithm for joining two datasets indexed by R-trees is the *R-tree join* (RJ) (Brinkhoff, Kriegel, Seeger, 1993), due to its efficiency and the popularity of R-trees. RJ synchronously traverses both trees, according to the paradigm of Günther (1993), following entry pairs that overlap; non-intersecting pairs cannot lead to solutions at the lower levels. After RJ, most research efforts focused on spatial join processing for non-indexed inputs. Non-indexed inputs are usually intermediate results of a preceding operator. Consider, for instance, the query "find all *cities* with *population over* 5,000 which are crossed by a *river*." If there are only a few large cities and an index on population, it may be preferable to process the selection part of the query before the spatial join. In such an execution plan, even if there is a spatial index on *cities*, it is not employed by the spatial join algorithm.

The simplest method to process a pairwise join in the presence of one index is by applying a window query to the existing R-tree for each object in the non-

indexed dataset (*index nested loops*). Due to its computational burden, this method is used only when the joined datasets are relatively small. Another approach is to build an R-tree for the non-indexed input using bulk loading (Patel & DeWitt, 1996, Papadopoulos, Rigaux, Scholl, 1999) and then employ RJ to match the trees (*build and match*). Lo and Ravishankhar (1994) use the existing R-tree as a skeleton to build a *seeded tree* for the non-indexed input. The *sort and match* (SaM) algorithm (Papadopoulos et al., 1999) spatially sorts the non-indexed objects but, instead of building the packed tree, it matches each in-memory created leaf node with the leaf nodes of the existing tree that intersect it. Finally, the *slot index spatial join* (SISJ) (Mamoulis & Papadias, 1999, 2003) applies hash-join, using the structure of the existing R-tree to determine the extents of the spatial partitions.

If no indexes exist, both inputs have to be preprocessed in order to facilitate join evaluation. Arge, Procopiuc, Ramaswamy, Suel and Vitter (1998) propose *scalable sweeping-based spatial join* (SSSJ) that employs a combination of *plane sweep* (Preparata & Shamos, 1985) and space partitioning to join the datasets. However, the algorithm cannot avoid external sorting of both datasets, which may lead to large I/O overhead. Patel and DeWitt (1996) describe *partition based spatial merge join* (PBSM) that regularly partitions the space using a rectangular grid, and hashes both inputs into the partitions. It then joins groups of partitions that cover the same area using plane-sweep to produce the join results. Some objects from both sets may be assigned in more than one partition, so the algorithm needs to sort the results in order to remove the duplicate pairs. Another algorithm based on regular space decomposition is the *size separation spatial join* (S^3J) (Koudas & Sevcik, 1997). S^3J avoids replication of objects during the partitioning phase by introducing more than one partition layer. Each object is assigned in a single partition, but one partition may be joined with many upper layers. The number of layers is usually small enough for one partition from each layer to fit in memory; thus, multiple scans during the join phase are not needed. *Spatial hash-join* (SHJ) (Lo & Ravishankar, 1996) avoids duplicate results by performing an irregular decomposition of space based on the data distribution of the build input.

Table 1 summarizes the existing algorithms of all three classes. In general, indexing facilitates efficiency in spatial join processing; an algorithm that uses existing indexes is expected to be more efficient than one that does not consider them. The relative performance of algorithms in the same class depends on the problem characteristics. Günther (1993) suggests that spatial join indices perform best for low join selectivity, while in other cases RJ is the best choice. Among the algorithms in the second class (one indexed input), SISJ and SaM outperform the other methods because they avoid the expensive R-tree construction (Mamoulis & Papadias, 2003). There is no conclusive experimental evaluation for the algorithms in the third class (non-indexed inputs). S^3J is

preferable when the datasets contain relatively large rectangles and extensive replication occurs in SHJ and PBSM. SHJ and PBSM have similar performance when the refinement step is performed exactly after the filter step. In this case, both algorithms sort their output in order to minimize random I/Os and PBSM combines the removal of duplicate pairs with sorting. However, in complex queries (for example, multiway spatial joins) and when the refinement step is postponed after the filter steps of all operators, PBSM may be more expensive, because it can produce larger intermediate results (due to the existence of duplicates). S^3J requires sorting of both datasets to be joined, and therefore it does not favor pipelining and parallelism of spatial joins. On the other hand, the fact that PBSM uses partitions with fixed extents makes it suitable for processing multiple joins in parallel. In the following paragraphs, we review in detail one representative algorithm from each of the three classes, namely the RJ, SHJ and SISJ. SHJ

The R-tree join

The *RJ* (Brinkhoff et al., 1993) is based on the *enclosure property* of R-trees: if two nodes do not intersect, there can be no MBRs below them that intersect. Following this observation, RJ starts from the roots of the trees to be joined and finds pairs of overlapping entries. For each such pair, 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.

Figure 4 illustrates two datasets indexed by R-trees. Initially, RJ receives the two tree roots as parameters. The qualifying entry pairs at the root level are (A_1, B_1) and (A_2, B_2). Notice that since A_1 does not intersect B_2, there can be no object pairs under these entries that intersect. RJ is recursively called for the nodes pointed by the qualifying entries until the leaf level is reached, where the intersecting pairs (a_1, b_1) and (a_2, b_2) are output.

Table 1. Classification of spatial join methods

Both inputs are indexed	One input is indexed	Neither input is indexed
• transformation to z-values (Orenstein, 1986) • spatial join index (Rotem, 1991) • tree matching (Günther, 1993, Brinkhoff et al., 1993)	• index nested loops • seeded tree join (Lo & Ravishankar, 1994) • build and match (Patel & DeWitt, 1996, Papadopoulos et al., 1999) • sort and match (Papadopoulos et al., 1999) • slot index spatial join (Mamoulis & Papadias, 2003)	• spatial hash join (Lo & Ravishankar, 1996) • partition based spatial merge join (Patel & DeWitt, 1996) • size separation spatial join (Koudas & Sevcik, 1997) • scalable sweeping-based spatial join (Arge et al., 1998)

Two optimization techniques can be used to improve the CPU speed of RJ (Brinkhoff et al., 1993). 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 $e_{i,x} \in n_i$ 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 $e_{j,y} \in n_j$, such that $e_{i,x}$ and $e_{j,y}$ overlap. Using this fact, space restriction performs two linear scans in the entries of both nodes before RJ 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*, applies sorting in one dimension in order to reduce the cost of computing overlapping pairs between the nodes to be joined. Plane sweep also saves I/Os compared to nested loops because consecutive computed pairs overlap with high probability. (Brinkhoff et al. 1994) discuss multi-step processing of RJ using several approximations (instead of conventional MBRs). Huang, Jing and Rundensteiner (1997a) propose a breadth-first optimized version of RJ that sorts the output at each level in order to reduce the number of page accesses.

Spatial Hash Join

SHJ (Lo & Ravishankar, 1996) (based on the relational hash-join paradigm) computes the spatial join of two non-indexed datasets *R (build* input*)* and *S (probe* input*)*. Set *R* is partitioned into *K* buckets, where *K* is decided by the system parameters. The initial extents of the buckets are points determined by sampling. Each object is inserted into the bucket that is enlarged the least. Set *S* is hashed into buckets with the same extent as *R*'s buckets, but with a different insertion policy: An object is inserted into all buckets that intersect it. Thus, some objects may be assigned to multiple buckets (*replication*) and some may not be inserted at all (*filtering*). The algorithm does not ensure equal-sized partitions for *R* (that is, with the same number of objects in them), as sampling cannot guarantee the best possible bucket extents. Equal-sized partitions for *S* cannot

Figure 3. R-tree-based spatial join

```
RJ(Rtree_Node n_i, RTNode n_j)
    for each entry e_{j,y}    n_j do {
        for each entry e_{i,x}    n_i with e_{i,x}    e_{j,y}      do {
        if n_i is a leaf node    /* n_j is also a leaf node */
            then Output (e_{i,x}, e_{j,y});
            else  { /* intermediate nodes */
                ReadPage(e_{i,x}.ref); ReadPage(e_{j,y}.ref);
                RJ(e_{i,x}.ref, e_{j,y}.ref); }      }
    } /* end for */
```

be guaranteed in any case, as the distribution of the objects in the two datasets may be different.

Figure 5 shows an example of two datasets, partitioned using the SHJ algorithm. After hashing set *S*, the two bucket sets are joined; each bucket from *R* is matched with only one bucket from *S*, thus requiring a single scan of both files, unless for some pair neither bucket fits in memory. In this case, an R-tree is built for one of them, and the bucket-to-bucket join is executed in an index nested loop fashion.

Slot Index Spatial Join

SISJ (Mamoulis & Papadias, 2003) is applicable when there is an R-tree for one of the inputs (*R*). The algorithm is similar to SHJ, but uses the R-tree on *R* in order to determine the bucket extents. If *K* is the desired number of partitions, SISJ will find the topmost level of the tree such that the number of entries is larger or equal to *K*. These entries are then grouped into *K* (possibly overlapping) partitions called *slots*. Each slot contains the MBR of the indexed R-tree entries, along with a list of pointers to these entries. Figure 6 illustrates a 3-level R-tree (the leaf level is not shown) and a slot index built over it. If $K = 9$, the root level contains too few entries to be used as partition buckets. As the number of entries in the next level is over *K*, we partition them in 9 slots (for this example). The grouping policy of SISJ starts with a single empty slot and inserts entries into the slot that is enlarged the least. When the maximum capacity of a slot is reached (determined by *K* and the total number of entries), either some entries are deleted and reinserted or the slot is split according to the R*-tree splitting policy (Beckmann, Kriegel, Schneider & Seeger, 1990).

After building the slot index, the second dataset *S* is hashed into buckets with the same extents as the slots. If an object from *S* does not intersect any bucket, it

Figure 4. Two datasets indexed by R-trees

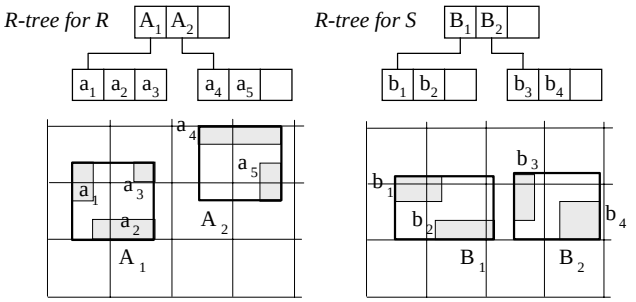
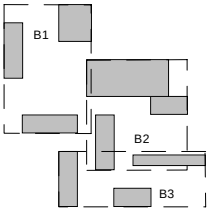
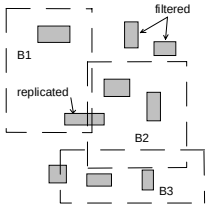


Figure 5. The partitioning phase of SHJ algorithm



(a) Objects from set *R* (build input) in three partition buckets



(b) Filtering and replication of objects from set *S* (probe input)

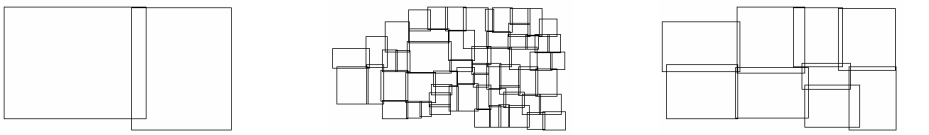
is filtered; if it intersects more than one bucket, it is replicated. The join phase of SISJ is also similar to the corresponding phase of SHJ. All data from the R-tree of *R* indexed by a slot are loaded and joined with the corresponding hash-bucket from *S* using plane sweep. If the data to be joined do not fit in memory, they can be joined using the algorithm of Arge et al.(1998), which employs external sorting and then plane sweep. Another alternative is index nested loop join (using as a root of the R-tree the corresponding slot). These methods can be expensive when the partitions are much larger than the buffer. In such cases SISJ is applied recursively, in a similar way to recursive hash-join. During the join phase of SISJ, when no data from *S* is inserted into a bucket, the sub-tree data under the corresponding slot is not loaded (slot filtering).

Selectivity and Cost Estimation for Spatial Joins

Estimating the cost and the output size of a spatial join is an important and difficult problem. Accurate cost models are necessary for the query optimizer to identify a good execution plan that accelerates retrieval and minimizes the usage of system resources. The output size of a spatial join between datasets *R* and *S* depends on three factors:

- The *cardinalities* $|R|$ and $|S|$ of the datasets. The join may produce up to $|R|\times|S|$ tuples (that is, the Cartesian product).
- The *density* of the datasets. The density of a dataset is formally defined as the sum of areas of all objects in it divided by the area of the workspace1. In other words, it is the expected number of objects that intersect a random

Figure 6. An R-tree and a slot index built over it



(a) level 2 (root) entries (b) level 1 entries (c) slot index over level 1

point in the workspace. Datasets with high density have objects with large average area, and produce numerous intersections when joined.

- The *distribution* of the MBRs. Skewed datasets may produce arbitrary few or many join pairs. Data skew is the most difficult factor to estimate, since in many cases the distribution is not known, and even if known, its characteristics are very difficult to capture.

The I/O cost of the refinement step is determined by the selectivity of the filter step, since for each candidate object (or object pair) a random access that retrieves its exact geometry is required. However, the selectivity of the refinement step is hard to estimate because the arbitrary extents of the actual objects do not allow for the easy computation of quantities like density and complicate the probabilistic analysis of overlapping regions. Although this estimate does not affect the cost of the spatial operator, it can be crucial for the cost estimate of operators that succeed it. For example, for a complex query, where three datasets are joined, the selectivity of the first join determines the input size of the second. Estimating the selectivity of a spatial query after the refinement step is a challenging issue, and to the best of our knowledge, no previous work sufficiently solves this problem. Existing studies focus on the filter step, often assuming that the data are uniformly distributed in the workspace (*uniformity assumption*). Several of these studies are based on selectivity and cost estimation formulae for window queries.

Selectivity and cost estimation for window queries

Given a spatial dataset R consisting of $|R|$ ´-dimensional uniformly distributed rectangles in a rectangular area u (workspace universe), the number of rectangles that intersect a window query w (*output cardinality* - *OC*) is estimated by the following formula,

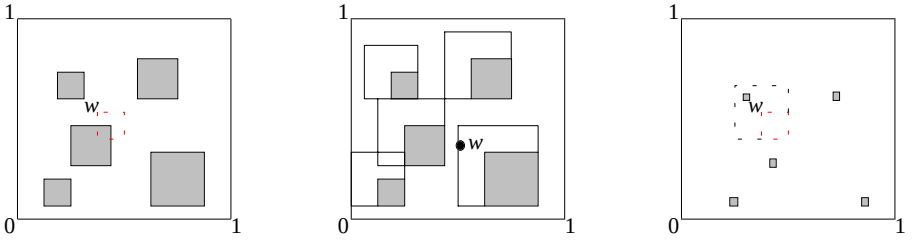
$$OC(R, w) = |R| \cdot \prod_{d=1}^{\delta} \min\left(1, \frac{\overline{r_d} + \overline{w_d}}{\overline{u_d}}\right), \quad (1)$$

where $\overline{r_d}$ is the average length of the projection of a rectangle $r \in R$ at dimension d, and $\overline{w_d}$, $\overline{u_d}$ are the corresponding projections of w, u respectively. The product in Equation 1, called Minkowski sum, depends on the probability that a random rectangle from R intersects w. A graphical example is given in Figure 7. In particular, Figure 7a depicts a dataset R and a window query w. We can think of w as a point query on a dataset that contains rectangles of average projection $\overline{r_d} + \overline{w_d}$ (Figure 7b), in which case the goal is to retrieve all rectangles that contain the query point. Alternatively, the query can be transformed to a rectangle with average side $\overline{r_d} + \overline{w_d}$ on $|R|$ points (Figure 7c), in which case the goal is to retrieve the data points falling in the query window. The *min* function in Equation 1 avoids boundary effects when $\overline{r_d} + \overline{w_d} > 1$ for some dimension d.

The output size for non-uniform data can be estimated by maintaining a histogram that partitions the data space into a set of *buckets*, and assuming that object distribution in each bucket is (almost) uniform. Specifically, each bucket b contains the number $b.num$ of objects whose centroids fall in b, and the average extent $b.len$ of such objects. Figure 8 illustrates an example in the 2D space, where the gray area corresponds to the intersection between b and the extended query region, obtained by enlarging each edge of q with distance $b.len/2$. Following the analysis on uniform data, the expected number of qualifying objects in b approximates $b.num'' I.area/b.area$, where $I.area$ and b.area are the areas of the intersection region and b, respectively (Acharya, Poosala, & Ramaswamy, 1999). The total number of objects intersecting q is predicted by summing the results of all buckets. Evidently, satisfactory estimation accuracy depends on the degree of uniformity of objects' distributions in the buckets. This can be maximized using various algorithms (Muralikrishna & DeWitt, 1988; Poosala & Ioannidis, 1997; Acharya et al., 1999), which differ in the way that buckets are structured. For example, in Muralikrishna & DeWitt (1988), buckets have similar sizes (that is, "equi-width") or cover approximately the same number of objects (that is, "equi-depth"), while in Poosala and Ioannidis (1997) and Acharya et al. (1999) bucket extents minimize the so-called "spatial skew."

When the dataset is not indexed, the cost of a window query (in terms of disk accesses) is equal to the cost of sequentially scanning the entire dataset (that is,

Figure 7. Output size estimation for window queries



(a) a dataset and a window query

(b) a point query on rectangles of average side $\overline{r_d}$ $\overline{w_d}$ in d={1,2}

(c) a window query of side $\overline{r_d}$ $\overline{w_d}$ on points

it is independent of the selectivity). On the other hand, the existence of an R-tree can significantly reduce the query cost. The number of R-tree pages accessed when processing a window query is equal to the expected number of non-leaf node entries that intersect the query window plus the access of the R-tree root.

Let L be the number of R-tree levels and N_l ($\overline{r_{d,l}}$) be the number of entries (average entry projection length) at level l and dimension d (0 is the leaf level and L-1 the root level). The cost of a window query is then given by the following formula (Kamel & Faloutsos, 1993; Theodoridis & Sellis, 1996):

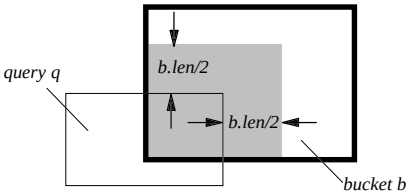
$$Cost(R, w) = 1 + \sum_{l=1}^{L-1} N_l \cdot \prod_{d=1}^{\delta} \left(\min\{1, \frac{\overline{r_{d,l}} + \overline{w_d}}{\overline{u_d}}\}\right) \qquad (2)$$

Theodoridis & Sellis (1996) present formulae for the estimation of $\overline{r_{d,l}}$ for each R-tree level that are based solely on the cardinality of the dataset $|R|$, the average object MBR projection length and the page size, which determines the capacity of the nodes and the height of the tree. The cost for non-uniform datasets can be computed with the aid of histograms (similar to the selectivity case).

Selectivity and cost estimation for spatial joins

The output size (and selectivity) of a spatial join can be estimated by extending Equation 1 (and Equation 2) in a straightforward way. Let R, S be the joined datasets. The output of the join is the expected number of rectangles retrieved

Figure 8. Estimation using histograms



from R when applying $|S|$ window queries of projection length $\overline{s_d}$. Conversely, it is the expected number of rectangles retrieved from S when applying $|R|$ window queries of projection length $\overline{r_d}$. In either case the output size is:

$$OC(R,S) = |R| \cdot |S| \cdot \prod_{d=1}^{k} \min\left(1, \frac{\overline{r_d} + \overline{s_d}}{\overline{u_d}}\right) \quad (3)$$

For other data distributions where the uniformity assumption does not hold, one can use 2D histograms that divide the space into buckets and summarize local statistics for each bucket (Theodoridis et al., 1998; Mamoulis & Papadias, 1999). The uniformity assumption can then be applied to each region to accumulate the estimate for the join output. An et al. (An et al., 2001) extend this method by maintaining, for each bucket, statistics about the objects' edges and corners that intersect it. Parametric approaches for distance joins, based on the observation that distances between spatial objects follow power laws, were proposed in (Belussi & Faloutsos, 1998, Faloutsos et al., 2000). Approximating the distribution (or object distances) in a real spatial dataset using histograms (or functions) cannot provide worst-case guarantees for query selectivity estimation. As a result, the effectiveness of most of the above methods is evaluated experimentally. An, Yang and Sivasubramaniam (2001) show by experimentation that their method is more accurate compared to the techniques used in Theodoridis et al. (1998)and in Mamoulis and Papadias (1999) for joins between sets of MBRs. Belussi & Faloutsos (1998) and Faloutsos, Seeger, Traina and Traina (2000) provide experimental evidence for the accuracy of their models, though they are incomparable with An et al. (2001), since it is applicable for distance joins between point sets.

Cost of R-Tree Join

Theodoridis et al. (1998) studied the cost of RJ in terms of R-tree node accesses. Let R and S be two datasets indexed by R-trees and assume that the two R-trees have (at level l) average entry side $\overline{r_{d,l}}$, $\overline{s_{d,l}}$ and number of entries $N_{R,l}$, $N_{S,l}$, respectively. The number of node accesses during their spatial join can be estimated by the following formula:

$$Cost_{NA}(\text{RJ}, R, S) = 2 + 2 \cdot \sum_{l=1}^{L-1} N_{R,l} \cdot N_{S,l} \cdot \prod_{d=1}^{\delta} \left(\min\{1, \frac{\overline{r_{d,l}} + \overline{s_{d,l}}}{\overline{u_d}}\}\right) \quad (4)$$

Equation 4 expresses that every pair of intersecting entries at level l is responsible for two node accesses at level l-1, if l>0. Therefore, the sum of the expected number of intersecting entry pairs at the high levels of the trees, plus the two accesses of the tree roots, give an estimate of the total number of node accesses. Nevertheless, this quantity can be considered as an upper bound only, since it does not reflect the actual number of I/Os under the existence of an LRU buffer. When an intersecting pair of entries needs to be loaded, there is a high probability that these pages will be in the system buffer if the buffer is large and if they have been requested before. Huang et al. (1997b) provide an analysis of RJ based on this observation, according to which, the I/O cost of joining R and S in the presence of an LRU buffer is given by the following formula,

$$Cost(\text{RJ}, R, S) = T_R + T_S + (Cost_{NA}(\text{RJ}, R, S) - T_R - T_S) \times Prob(\text{node}, M), \quad (5)$$

where T_R, T_S are the number of nodes in the R-trees for R and S, respectively, and $Prob$(node, M) is the probability that a requested R-tree node will not be in the buffer (of size M), resulting in a page fault. This probability falls exponentially with M, and its estimation is based on an empirical analysis.

Cost of Spatial Hash Join

The I/O cost of SHJ depends on the size of the joined datasets and the filtering and replication that occur in set S. Initially, a small number of pages $Cost_{sam}$ is loaded to determine the initial hash buckets. Then both sets are read and hashed

into buckets. Let P_R, P_S be the number of pages of the two datasets (stored in sequential files) and rep_S, fil_S be the replication and filtering ratios of S. The partitioning cost of SHJ is given by the following formula:

$$Cost_{part}(\text{SHJ}, R, S) = Cost_{sam} + 2\ P_R + (2 + rep_S - fil_S) \times P_S \quad (6)$$

Next, the algorithm will join the contents of the buckets from both sets. In typical cases, where the buffer is large enough for at least one partition to fit in memory, the join cost of SHJ is,

$$Cost_{join}(\text{SHJ}, R, S) = P_R + (1 + rep_S - fil_S) \times P_S, \quad (7)$$

considering that the join output is not written to disk. Summing up, from Equations 6 and 7, the total cost of SHJ is:

$$Cost(\text{SHJ}, R, S) = Cost_{sam} + 3\ P_R + (3 + 2rep_S - 2fil_S) \times P_S \quad (8)$$

Cost of Slot Index Spatial Join

SISJ joins a dataset R indexed by an R-tree with a non-indexed file S. Let T_R be the number of R-tree nodes of R, and P_S the number of pages in S. Initially, the slots have to be determined from R. This requires loading the top ^{o} levels of R's R-tree, in order to find the appropriate slot level. Let $frac_R$ be the fraction of tree nodes from the root until ^{o}. The slot index is built in memory, without additional I/Os. Set S is then hashed into the slots requiring P_S accesses for reading, and $P_S + rep_S P_S - fil_S P_S$ accesses for writing, where rep_S, fil_S are the replication and filtering ratios of S. Thus, the cost of SISJ partition phase is:

$$Cost_{part}(\text{SISJ}, R, S) = frac_R \times T_R + (2 + rep_S - fil_S) \times P_S \quad (9)$$

For the join phase of SISJ, we make the same assumptions as for SHJ; that is, for each joined pair at least one bucket fits in memory. The pages from set R that have to be fetched for the join phase are the remaining $(1\text{-}frac_R) \times T_R$, since the pointers to the slot entries are kept in the slot index and need not be loaded again from the top levels of the R-tree. The number of I/O accesses required for the join phase is:

$$Cost_{join}(\text{SISJ}, R, S) = (1 - frac_R) \times T_R + (1 + rep_S - fil_S) \times P_S \quad (10)$$

Summarizing, the overall cost of SISJ is:

$$Cost(\text{SISJ}, R, S) = T_R + (3 + 2rep_S - 2fil_S) \times P_S \quad (11)$$

We can qualitatively compare the three algorithms from Equations 4, 8 and 11. Given a large enough memory buffer, the cost of RJ is not much higher than $T_R + T_S$, since we expect that every requested R-tree node that has been loaded will remain in the memory buffer with high probability, due to the locality of accessed pages. This assertion is experimentally verified in several studies (for example, Huang et al., 1997b; Mamoulis & Papadias, 1999). Given that in typical R-tree structures nodes have around 67% average utilization (Beckmann et al., 1990), and that the non-leaf R-tree nodes are very few compared to the leaves (due to the large fanouts, 100-200 in practice), the I/O cost of RJ is roughly the cost of reading 150% of the total number of pages occupied by the rectangles (that is, $Cost(\text{SISJ}, R, S) \approx T_R + T_S \approx 1.5(P_R + P_S)$). The cost of SISJ is affected by the filtering and replication ratios, which cannot be easily predicted. From empirical studies on real datasets (Mamoulis & Papadias, 2003), it has been observed that in practice, $rep_S \approx 0.3$ and $fil_S \approx 0$. Considering this and the discussion on average R-tree node occupancy, we can approximate the cost of SISJ with $1.5P_R + 3.6P_S$. With similar assumptions on the filtering and replication ratios (and assuming a negligible sampling cost), the cost of SHJ is reduced to $3P_R + 3.6P_S$. Based on these numbers, we can conclude that RJ is more efficient than SISJ, which is more efficient than SHJ, under usual problem settings. Of course, the application of RJ (SISJ) presumes the existence of two (one) R-trees.

In the next sections, we discuss how the cost estimation formulae for RJ, SHJ and SISJ can be used in combination with the selectivity estimation models discussed earlier to optimize complex queries that include spatial join operators. The experimental study of Mamoulis & Papadias (2001) suggests that these estimates are indeed accurate and usually lead to optimal plans.

Multiway Spatial Joins

Multiway spatial joins involve an arbitrary number of spatial inputs. Such queries are important in several applications, including Geographical Information Systems (for example, "find all cities *adjacent to* forests, which are *intersected* by a river") and VLSI (for example, "find all sub-circuits that formulate a

specific topological configuration"). Formally, a multiway spatial join can be expressed as follows: Given n datasets R_1, R_2, ... R_n and a *query* Q, where Q_{ij} is the spatial predicate that should hold between R_i and R_j, retrieve all n-tuples $\{(r_{1,w},..,r_{i,x},..,r_{j,y},.., r_{n,z}) \mid \forall i,j : r_{i,x} \in R_i, r_{j,y} \in R_j \text{ and } r_{i,x}\ Q_{ij}\ r_{j,y}\}$. The query can be represented by a graph, where nodes correspond to datasets and edges to join predicates. Equivalently, the graph can be viewed as a spatial *constraint network*, where the nodes correspond to problem variables and edges to binary spatial constraints. In the sequel we use the terms variable/dataset and constraint/join condition interchangeably.

We consider that all datasets are indexed by R-trees (on MBRs) and we deal with the filter step, assuming that *overlap* is the default join condition; that is, if Q_{ij} = True, then the rectangles from the corresponding inputs i,j should overlap. The loosest query is the one that corresponds to an acyclic (*tree*) graph (for example, the one illustrated in Figure 9a), while the most constrained consists of a complete (*clique*) graph (for example, the one in Figure 9c). For each type of query, Figure 9 illustrates a solution; that is, a configuration of rectangles $r_{i,1} \hat{I} R_i$ that satisfies the join conditions. We do not consider non-connected query graphs, as these can be processed by solving connected sub-graphs and then computing their Cartesian product.

Patel and DeWitt (1996) apply PBSM in a distributed, multi-processor environment to process cascading joins. Spatial datasets are regularly partitioned in space (*spatial declustering*), and the physical resources (disks, processors) are distributed according to the partitions. Papadopoulos et al. (1999) perform a two-join case study to evaluate the performance of four spatial join algorithms. Mamoulis and Papadias (1999) propose a *pairwise joins method* (PJM) that combines binary join algorithms in a processing tree where the leaves are input relations indexed by R-trees and the intermediate nodes are join operators.

Processing multiway joins by integration of pairwise join algorithms is the standard approach in relational databases where the join conditions usually relate different attributes. In spatial joins, however, the conditions refer to a single spatial attribute for all inputs; that is, all object sets are joined with respect to their spatial features. Motivated by this fact, *synchronous traversal* (ST) traverses top-down2 all the R-trees involved in the query, excluding combinations of intermediate nodes that do not satisfy the join conditions. The first general application of ST to an arbitrary number of inputs appeared in Papadias et al. (1998) for retrieval of database images matching some input configuration. The employment of the method in multi-way spatial join processing is discussed in Papadias et al. (1999) and in Mamoulis and Papadias (2001), together with formulae for selectivity (in uniform datasets) and cost estimation (in terms of node accesses). Next, we present in detail PJM and ST. Finally, we discuss the optimization of processing multiway spatial joins based on dynamic programming.

Integration of Pairwise Join Algorithms for Processing Multiple Inputs

As in the case of relational joins, multiway spatial joins can be processed by combining pairwise join algorithms. PJM considers a join order that is expected to result in the minimum cost (in terms of page accesses). Each join order corresponds to a single execution plan, where:

(i) RJ is applied when the inputs are leaves; that is, datasets indexed by R-trees,

(ii) SISJ is employed when only one input is indexed by an R-tree,

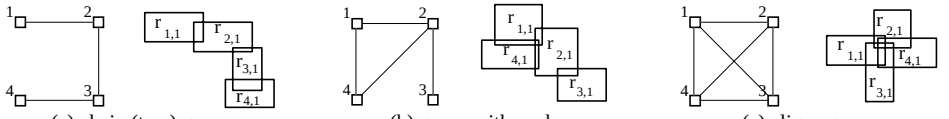
(iii) SHJ is used when both inputs are intermediate results.

As an example of PJM, consider the query in Figure 9a and the plans of Figure 10. Figure 10a involves the execution of RJ for determining $R_3 \bowtie R_4$. The intermediate result, which is not indexed, is joined with R_2 and finally with R_1 using SISJ. On the other hand, the plan of Figure 10b applies RJ for $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, and SHJ to join the intermediate results.

Queries with cycles can be executed by transforming them to tree expressions using the most selective edges of the graph and filtering the results with respect to the other relations in memory. For instance, consider the cycle (R_1 *overlap* R_2), (R_2 *overlap* R_3), (R_3 *overlap* R_1) and the query execution plan $R_1 \bowtie (R_2 \bowtie R_3)$. When joining the tuples of ($R_2 \bowtie R_3$) with R_1 we can use either the predicate (R_2 *overlap* R_1), or (R_3 *overlap* R_1) as the join condition. If (R_2 *overlap* R_1) is the most selective one (that is, results in the minimum cost), it is applied for the join, and the qualifying tuples are filtered with respect to (R_3 *overlap* R_1).

PJM uses Equations 5, 8 and 11 to estimate the join cost of the three algorithms. The expected output size of a pairwise join determines the execution cost of an upper operator and therefore is crucial for optimization. Selectivity estimation for a pairwise join has already been discussed. Optimization of multiway spatial joins requires selectivity estimation for each possible decomposition of the query

Figure 9. Multiway join examples



(a) chain (tree) query (b) query with cycle (c) clique query

graph (that is, for each allowable sub-plan). The generalized formula for the output size of a query (sub) graph Q with n inputs is:

$$OC(R_1,R_2, \ldots, R_n, Q) = \#(\text{possible tuples}) \times Prob(\text{a tuple is a solution}) \quad (12)$$

The first part of the product equals the cardinality of the Cartesian product of the n domains, while the second part corresponds to *multiway join selectivity*. In case of acyclic graphs, the pairwise probabilities of the join edges are independent and selectivity is the product of pairwise join selectivities (Papadias et al., 1999):

$$Prob(\text{a tuple is a solution}) = \prod_{\forall i,j:Q_{ij}=TRUE} \prod_{d=1}^{\delta} \min\left(1, \frac{\overline{r_{i,d}} + \overline{r_{j,d}}}{\overline{u_d}}\right) \quad (13)$$

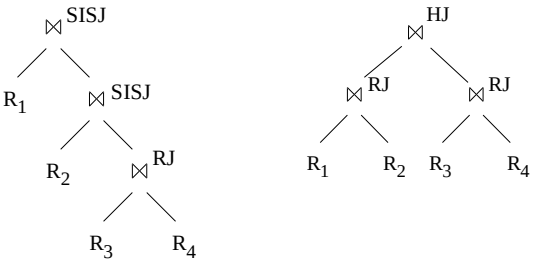
From Equations 12 and 13, the total number of query solutions is:

$$OC(R_1,\ldots, R_n, Q) = \prod_{i=1}^{n} |R_i| \cdot \prod_{\forall i,j:Q_{ij}=TRUE} \prod_{d=1}^{\delta} \min\left(1, \frac{\overline{r_{i,d}} + \overline{r_{j,d}}}{\overline{u_d}}\right) \quad (14)$$

When the query graph contains cycles, the pairwise selectivities are no longer independent and Equation 14 is not accurate. For cliques, it is possible to provide a formula for multiway join selectivity based on the fact that if a set of rectangles mutually overlap, then they must share a common area. Given a random n-tuple of rectangles, the probability that all rectangles mutually overlap is (Papadias et al., 1999):

$$Prob(\text{a tuple is a solution}) = \prod_{d=1}^{\delta} \frac{1}{(n-1)\cdot \overline{u_d}} \sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} \overline{r_{j,d}} \quad (15)$$

Figure 10. Alternative plans using pairwise join algorithms



(a) right-deep plan (b) bushy plan

Thus, in case of clique queries Q the number of solutions is:

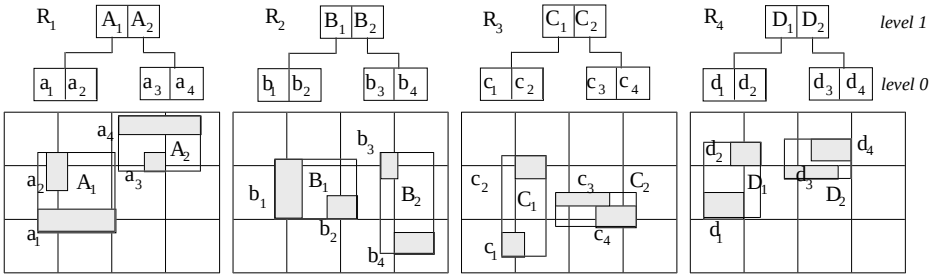
$$OC(R_1,...,R_n,Q) = \prod_{i=1}^{n} |R_i| \cdot \prod_{d=1}^{\delta} \frac{1}{(n-1)\cdot \overline{u_d}} \sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} \overline{r_{j,d}} \qquad (16)$$

The above formulae are applicable for queries that can be decomposed to acyclic and clique graphs (for example, Figure 9b). The optimal execution plan can be computed from the estimated output size and the costs of the algorithms involved. Selectivity estimation for real datasets can be performed using histograms. Next we describe ST, an alternative to PJM for processing multiway spatial joins.

Synchronous Traversal

ST processes the indexes of all joined datasets, following combinations of nodes that satisfy the query constraints. Consider the four R-trees of Figure 11 and the clique query of Figure 9c. The query asks for the set of 4-tuples (a_w, b_x, c_y, d_z), such that the four objects mutually overlap (for example, (a_2, b_1, c_2, d_2)). ST starts from the roots of the R-trees, searching for entries that satisfy the join conditions. In this example, out of the 16 combinations of root entries (that is, (A_1, B_1, C_1, D_1), (A_1, B_1, C_1, D_2), .., (A_2, B_2, C_2, D_2)), only (A_1, B_1, C_1, D_1) may lead to actual solutions. For instance, the combination (A_2, B_1, C_1, D_1) does not satisfy the query constraints because A_2 does not intersect C_1 (or D_1); therefore, there cannot be any pair of overlapping objects (a_w, c_y), a_w pointed by A_2 and c_y pointed by C_1. As in the case of RJ, for each intermediate level solution, the algorithm is called for the pointed R-tree nodes, recursively, until the leaves, where solutions are output.

Figure 11. Example of four R-trees



In the worst case, the total number of combinations of data MBRs that have to be checked for the satisfaction of the join conditions is $|R|^n$, where n is the number of inputs and $|R|$ the cardinality of the datasets (assumed to be the same). ST takes advantage of the hierarchical decomposition of space preserved by R-trees to break the problem in smaller *local* ones at each tree level. A local problem has to check C^n combinations in the worst case (C is the R-tree node capacity), and can be defined by:

- A set of n variables, v_1, v_2,.., v_n, each corresponding to a dataset.
- For each variable v_i, a domain Δ_i consisting of the entries $\{e_{i,1},.., e_{i,Ci}\}$ of a node n_i (in tree R_i).
- Each pair of variables (v_i, v_j) is constrained by *overlap*, if Q_{ij} is True.

A binary assignment $\{v_i \leftarrow e_{i,x}, v_j \leftarrow e_{j,y}\}$ is *consistent* if Q_{ij} = True $\Rightarrow e_{i,x}$ overlaps $e_{j,y}$. A solution of a local problem is a n-tuple t = $(e_{1,w},.., e_{i,x},.., e_{j,y},..,e_{n,z})$ such that $\forall i,j$, $\{v_i \neg e_{i,x}, v_j \neg e_{j,y}\}$ is consistent. The goal is to find all *solutions*; that is, assignments of entries to variables such that all constraints are satisfied. In the previous example (clique query of Figure 9c), there exist four variables $v_1,..,v_4$, and for each (v_i,v_j), $i \neq j$, the constraint is *overlap*. At level 1 the domains of the variables are $\Delta_1=\{A_1,A_2\}$, $\Delta_2=\{B_1,B_2\}$, $\Delta_3=\{C_1,C_2\}$ and $\Delta_4=\{D_1,D_2\}$. Once the root level solution (A_1,B_1,C_1,D_1) is found, ST will recursively search for qualifying tuples at the lower level, where the domains of $v_1,..,v_4$ consist of the entries under $A_1,..,D_1$, respectively; that is, $\Delta_1=\{a_1,a_2\}$, $\Delta_2=\{b_1,b_2\}$, $\Delta_3=\{c_1,c_2\}$ and $\Delta_4=\{d_1,d_2\}$. Notice that an intermediate-level solution does not necessarily lead to an actual one. Since a part of the node area corresponds to "dead space" (space not covered by object MBRs), many high-level solutions are *false hits*. The pseudo-code for ST, assuming R-trees of equal height, is presented in Figure 12.

For each Δ_i, *space-restriction* prunes all entries that do not intersect the MBR of some n_j, where Q_{ij} = True. Consider the chain query of Figure 9a and the top-

level solution (A_2, B_1, C_1, D_1). At the next level ST is called with $\Delta_1 = \{a_3, a_4\}$, $\Delta_2 = \{b_1, b_2\}$, $\Delta_3 = \{c_1, c_2\}$ and $\Delta_4 = \{d_1, d_2\}$. Although A_2 intersects B_1, none of entries (a_3, a_4) do and these entries can be safely eliminated from Δ_1. Since Δ_1 becomes empty, (A_2, B_1, C_1, D_1) cannot lead to an actual solution and the search is abandoned without loading the nodes pointed by B_1, C_1 and D_1. *Find-combinations* is the "heart" of ST; that is, the search algorithm that finds tuples $t \in \Delta_1 \times \Delta_2 \times ... \times \Delta_n$, that satisfy Q. In order to avoid exhaustive search of all combinations, several backtracking algorithms applied for constraint satisfaction problems can be used. The implementation of Mamoulis and Papadias (2001) uses *forward checking* (FC) (Haralick & Elliott, 1981), which accelerates search by progressively assigning values to variables and pruning the domains of future (non-instantiated) variables. Given a specific order of the problem's variables v_1, v_2,…, v_n, when v_i is instantiated, the domains of all future variables v_j, $j > i$, such that Q_{ij} = True, are revised to contain only rectangles that intersect the current instantiation of v_j (*check forward*). If during this procedure some domain is eliminated, a new value is tried for v_i until the end of D_i is reached. Then FC *backtracks* to v_{i-1}, trying a new value for this variable.

Figure 12. Synchronous R-tree traversal

```
ST(Query Q[][], RTNode n[])
    for i:=1 to n do {  /*prune domains*/
            Δi := space-restriction(Q, n[], i);
            if Δi =     then return;  /*no qualifying tuples exist for this combination of nodes*/}
    for each      find-combinations(Q, Δ) do { /* for each solution at the current level */
            if n[] are leaf nodes then /*qualifying tuple is at leaf level*/
                    Output( );
            else /*qualifying tuple is at intermediate level*/
                    ST(Q,   .ref[]); /* recursive call to lower level */}

Domain space-restriction(Query Q[][], RTNode n[], int i)
    read ni; /* read node from disk */
    Δi :=    ;
    for each entry ei,x    ni  do {
            valid := True; /*mark ei,x as valid */
            for each node nj such that Qij = True do {/*an edge exists between ni and nj*/
                    if  ei,x    nj.MBR =     then { /* ei,x does not intersect the MBR of node ni */
                            valid := false; /* ei,x is pruned */
                            break;}
        if valid = True then /*ei,x is consistent with all node MBRs*/
                Δi := Δi    ei,x;
    }
    return Δi;
```

ST Cost Estimation

ST starts from the top level *L-1* (where *L* is the height of the trees), and solves one local problem in order to find solutions at the roots. Each solution generates one problem at the next level until it reaches the leaves where solutions are output. Thus, the total number of local problems is,

$$N_{PROBLEMS} = 1 + \sum_{l=1}^{L-1} \#solutions(Q,l) , \quad (17)$$

where *#solutions(Q, l)* is the number of qualifying entry combinations at level *l*. An experimental study in Mamoulis and Papadias (2001) suggests that ST is CPU bound, due to the huge number of local problems and the fact that tree nodes are visited with high locality; thus, the LRU buffer serves the majority of I/O requests. Therefore, it is crucial to estimate the CPU cost of the algorithm. This depends on the cost of the local problems, all of which have the same characteristics (that is, number of variables, constraints and domain size); therefore, it is reasonable to assume that they all have approximately the same cost ($C_{PROBLEM}$). Consequently, the total CPU cost ($Cost_{CPU}$) equals the number of local problems times the cost of each problem:

$$Cost_{CPU}(\text{ST}, Q) = N_{PROBLEMS} \times C_{PROBLEM} \quad (18)$$

$N_{PROBLEMS}$ can be estimated by Equation 17 using Equation 12 for the number of solutions at each level of the tree. The only difference is that instead of object MBRs, intermediate nodes are used in Equations 14 and 16. The remaining factor is the cost $C_{PROBLEM}$. Although in the worst case (for example, extremely large intermediate nodes) each local problem is exponential ($O(C^n)$), the average $C_{PROBLEM}$ for typical situations is much lower (actually, it increases linearly with *n* and page size). Unfortunately, the nature of backtracking-based search algorithms (including forward checking) does not permit theoretical average case analysis (Kondrak & van Beek, 1997). Therefore, an empirical analysis was conducted in Mamoulis and Papadias (2001) to isolate this cost. The result of this analysis is that *the CPU-time for each local problem is linear to the number of variables n and the page size p, independently of the domain density or the structure of the graph*, and we can define,

$$C_{PROBLEM} = F \times n \times p , \quad (19)$$

Table 2. Iterator functions

Iterator	***Open***	***Next***	***Close***
ST (RJ for two inputs)	– open tree files	– return next tuple	– close tree files
SISJ (assuming that left input is the R-tree input)	– open left tree file; – construct slot index; – *open* right (probe) input; – call *next* on right input and hash results into slots; – *close* right input	– perform hash-join; – return next tuple	– close tree file; – de-allocate slot index; – hash buckets
SHJ (assuming that left input is the build input and right input the probe input)	– *open* left input; – call *next* on left and write the results into intermediate file while determining extents of hash buckets; – *close* left input; – hash results from intermediate file into buckets; – *open* right input; – call *next* on right and hash all results into right buckets; – *close* right input	– perform hash-join; – return next tuple	– de-allocate hash buckets

where F is a factor that depends on the algorithm for ST and the CPU speed and can be estimated by Equations 17, 18, 19 and the actual cost of a multiway join. The experiments of Mamoulis and Papadias (2001) suggest that this method has low average error (below 15%) for various multiway joins on synthetic datasets.

Combining ST with Pairwise Join Algorithms

Since ST is essentially a generalization of RJ, it easily can be integrated with other pairwise join algorithms to effectively process complex spatial queries. Table 2 shows how ST, SISJ and SHJ can be implemented as *iterator functions* (Graefe, 1993) in an execution engine running on a centralized, uni-processor environment that applies pipelining.

ST (RJ for two inputs) executes the join and passes the results to the upper operator. SISJ first constructs the slot index, then hashes the results of the probe (right) input into the corresponding buckets and finally performs the join, passing the results to the upper operator. SHJ does not have knowledge about the initial buckets where the results of the left join will be hashed; thus, it cannot avoid writing the results of its left input to disk. At the same time it performs sampling to determine the initial extents of the buckets. Then, the intermediate file is read and hashed to the buckets. The results of the probe input are immediately hashed to buckets. Notice that in this implementation, the system buffer is shared between at most two operators and *next* functions never run concurrently; when

Table 3. Number of plans and optimization cost parameters for different query graphs

	clique	chain	star
$comb_k$	$\binom{n}{k}$	$n\text{-}k\text{+}1$	$\begin{cases} n, k=1 \\ \binom{n-1}{k-1}, \text{otherwise} \end{cases}$
$decomp_k$	$\begin{cases} 0, 1 \le k \le 2 \\ k + \sum_{2 \le i < k\text{-}1} \binom{k}{i}, \text{otherwise} \end{cases}$	$\begin{cases} 0, 1 \le k \le 2 \\ 2, \text{otherwise} \end{cases}$	$\begin{cases} 0, 1 \le k \le 2 \\ k-1, \text{otherwise} \end{cases}$

join is executed at one operator, only hashing is performed at the upper one. Thus, given a memory buffer of M pages, the operator that is currently performing a join uses $M - K$ pages and the upper operator, which performs hashing, uses K pages, where K is the number of slots/buckets. In this way, the utilization of the memory buffer is maximized.

Optimization of Multiway Spatial Joins

Given a set of binary (for example, SISJ, SHJ) and *n*-ary (for example, ST) join operators, and the corresponding selectivity/cost estimation formulae, the spatial query optimizer aims at finding a fast execution plan. *Dynamic programming* (DP), the standard technique for relational query optimization, can also be applied for multiway spatial joins. The optimal plan for a query is computed in a bottom-up fashion from its sub-graphs. At step *i*, for each connected sub-graph Q_i with *i* nodes, DP (Figure 13) finds the best decomposition of Q_i to two connected components, based on the optimal cost of executing these components and their sizes. We assume that all join inputs are indexed by R-trees. When a component consists of a single node, SISJ is considered as the join execution algorithm, whereas if both parts have at least two nodes, SHJ is used. The output size is estimated using the size of the plans that formulate the decomposition. DP compares the cost of the optimal decomposition with the cost of processing the whole sub-graph using ST, and sets as optimal plan of the sub-graph the best alternative. Since pairwise algorithms are I/O bound and ST is CPU-bound, when estimating the cost for a query sub-plan, DP takes under consideration the dominant factor in each case.

At the end of the algorithm, *Q*.plan will be the optimal plan, and *Q*.cost and *Q*.size will hold its expected cost and size. The execution cost of dynamic programming depends on: (i) the number of relations *n*, (ii) the number of valid node

Figure 13. Dynamic programming for optimization of multiway spatial joins

```
DP(Query Q, int n) /*n = number of inputs*/
  for each connected sub-graph Ri-Rj Q2    Q of size 2 do {
      Q2.cost := Cost(RJ, Ri, Rj); /*Equation 5*/
      Q2.size := OC(Ri, Rj); /*Equation 3*/    }
  for i:=3 to n do
      for each connected sub-graph Qi    Q with i nodes do {/*Find optimal plan for Qi*/
          Qi.plan := ST; Qi.cost := Cost_CPU(ST, Qi); /*Equation 18*/
          for each decomposition Qi    {Qk, Qi-k}, such that Qk, Qi-k connected do {
              if (k=1) then /*Qk is a single node; SISJ will be used*/
                  {Qk, Qi-k}.cost := Qi-k.cost + Cost(SISJ, Qk, Qi-k); /*Equation 11*/
              else /*both components are sub-plans; SHJ will be used*/
                  {Qk, Qi-k}.cost := Qk.cost + Qi-k.cost + Cost(SHJ, Qk, Qi-k); /*Equation8*/
              if {Qk,Qi-k}.cost<Qi.cost then { /*better than former optimal*/
                  Qi.plan := {Qk, Qi-k}; /*mark decomposition. as Qi's optimal plan*/
                  Qi.cost := {Qk, Qi-k}.cost; /*mark so far optimal cost of Qi*/}
          } /*decomposition*/
          /*Estimate Qi's output size from optimal decomposition*/
          Qi.size := OC(Qi.plan);
      }}
```

combinations $comb_k$ (that formulate a connected sub-graph) for each value of n, and (iii) the number of decompositions $decomp_k$ of a specific combination. Table 3 illustrates the above parameters for three special cases of join graphs. Note that combinations of 2 nodes do not have valid decompositions because they can be processed only by RJ.

The running cost of the optimization algorithm is the number of input combinations for each value of n times the number of valid decompositions plus 1 for the cost of ST:

$$Cost_{CPU}(\text{DP}, Q) = \sum_{1 \le k \le n} comb_k \cdot (1 + decomp_k) \qquad (20)$$

Equation 20 suggests that DP can be too expensive for joins with a large (for example, >10) number of inputs. For such cases, randomized algorithms can find a good (but sub-optimal) plan within limited time (Mamoulis & Papadias, 2001).

Summary

In this chapter we review some of the most significant research results related to spatial join processing. In particular, we describe: (i) binary algorithms that can be used in different cases, depending on whether the joined inputs are indexed or not; (ii) selectivity and cost estimation models; and (iii) techniques for the efficient processing of multiway joins based on integration of binary algorithms and synchronous traversal. Although we attempted to provide an extensive coverage of the literature, several issues related to spatial joins — for example, parallel join processing (Brinkhoff et al., 1996; Luo, Naughton, & Ellman, 2002) and join variants (Koudas & Sevcik, 2000; Corral et al., 2000; Böhm & Krebs, 2002; Shou et al., 2003) — were omitted due to space constraints.

There are several issues related to spatial joins that still need to be addressed. First, it is a common belief that intersection join algorithms can be straightforwardly applied for other types, like distance joins. However, practice (for example, see Corral et al., 2000; Shou et al., 2003) has already shown that direct extensions (usually of RJ) may be inefficient, and several optimizations can potentially enhanceperformance. Thus the application and optimization of different intersection algorithms to other join variants is an interesting topic of future work. Furthermore, although current systems only consider the standard "first filter, then refinement step" strategy, a spatial query processor should allow the interleaving of filter and refinement steps. For example, consider the query "find all cities *adjacent to* forests, which are *intersected* by a river" and assume that we know there are only a few rivers that intersect cities, although there are numerous such MBR pairs. Then, it would be preferable to execute the refinement step after the first join before we proceed to the next one. However, this knowledge presumes that we have accurate selectivity formulae for the refinement step, which is a difficult, open problem for future work.

References

Acharya, S., Poosala, V., & Ramaswamy, S. (1999). Selectivity Estimation in Spatial Databases. *Proceedings of the ACM SIGMOD Conference*, 13-24.

An, N., Yang, Z., & Sivasubramaniam, A. (2001). Selectivity Estimation for Spatial Joins. *Proceedings of the IEEE ICDE Conference*, 368-375.

Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., & Vitter, J.S. (1998). Scalable Sweeping-Based Spatial Join. *Proceedings of the VLDB Conference*, 570-581.

Beckmann, N., Kriegel, H.P., Schneider, R., & Seeger, B. (1990). The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *Proceedings of the ACM SIGMOD Conference*, 322-331.

Belussi, A., & Faloutsos, C. (1998). Self-spatial Join Selectivity Estimating Using Fractal Concepts. *ACM TOIS, 16*(2), 161-201.

Böhm C., & Krebs F. (2002). High Performance Data Mining Using the Nearest Neighbor Join. *Proceedings of the IEEE International Conference on Data Mining*, 43-50.

Brinkhoff, T., Kriegel, H.P., Schneider, R., & Seeger, B. (1994). Multi-Step Processing of Spatial Joins. *Proceedings of the ACM SIGMOD Conference*, 197-208.

Brinkhoff, T., Kriegel, H.P., & Seeger, B. (1993). Efficient Processing of Spatial Joins Using R-trees. *Proceedings of the ACM SIGMOD Conference*, 237-246.

Brinkhoff, T., Kriegel, H.P., & Seeger, B. (1996). Parallel Processing of Spatial Joins Using R-trees. *Proceedings of the ICDE Conference*, 258-265.

Corral, A., Manolopoulos, Y., Theodoridis, Y., & Vassilakopoulos, M., (2000). Closest Pair Queries in Spatial Databases. *Proceedings of the ACM SIGMOD Conference*, 189-200.

Faloutsos, C., Seeger, B., Traina, A., & Traina, C. (2000). Spatial Join Selectivity Using Power Laws. *Proceedings of the ACM SIGMOD Conference*, 177-188.

Gaede, V. & Günther, O. (1998). Multidimensional Access Methods. *ACM Computing Surveys, 30*(2), 123-169.

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

Günther, O. (1993) Efficient Computation of Spatial Joins. *Proceedings of the ICDE Conference*, 50-59.

Güting, R.H. (1994). An Introduction to Spatial Database Systems. *VLDB Journal*, *3*(4), 357-399.

Guttman, A. (1984). R-trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the ACM SIGMOD Conference*, 47-57.

Haralick, R., & Elliott, G. (1981). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14,* 263-313.

Huang, Y.W., Jing, N., & Rundensteiner, E. (1997a). Spatial Joins using R-trees: Breadth First Traversal with Global Optimizations. *Proceedings of the VLDB Conference*, 395-405.

Huang, Y.W., Jing N., & Rundensteiner, E. (1997b). A Cost Model for Estimating the Performance of Spatial Joins Using R-trees. *Proceedings of the SSDBM Conference*, 30-38.

Kamel, I., & Faloutsos, C. (1993). On Packing R-trees. *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 490-499.

Koudas, N., & Sevcik, K. (1997). Size Separation Spatial Join. *Proceedings of the ACM SIGMOD Conference*, 324-335

Koudas, N., & Sevcik, K. (2000). High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *IEEE Transactions in Knowledge and Data Engineering, 12*(1), 3-18.

Kondrak Q., & van Beek, P. (1997). A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence, 89*, 365-387.

Lo, M-L., & Ravishankar, C.V. (1994). Spatial Joins Using Seeded Trees. *Proceedings of the ACM SIGMOD Conference*, 209-220.

Lo, M-L., & Ravishankar, C.V. (1996). Spatial Hash-Joins. *Proceedings of the ACM SIGMOD Conference*, 247-258.

Luo, G., Naughton, J., & Ellman, C. (2002). A Non-Blocking Parallel Spatial Join Algorithm. *In Proceedings of the ICDE Conference*, 697-705.

Mamoulis, N., & Papadias, D. (1999). Integration of Spatial Join Algorithms for Processing Multiple Inputs. *Proceedings of the ACM SIGMOD Conference*, 1-12.

Mamoulis, N., & Papadias, D. (2001). Multiway Spatial Joins. *ACM Transactions on Database Systems (TODS), 26*(4), 424-475.

Mamoulis, N., & Papadias, D. (2003). Slot Index Spatial Join. *IEEE Transactions on Knowledge and Data Engineering (TKDE), 15*(1), 211-231.

Muralikrishna, M., & DeWitt, D. (1988). Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. *Proceedings of the ACM SIGMOD Conference*, 28-36.

Orenstein, J. (1986). Spatial Query Processing in an Object-Oriented Database System. *Proceedings of the ACM SIGMOD Conference*, 326-336.

Papadias, D., Mamoulis, N., & Delis, V. (1998). Algorithms for Querying by Spatial Structure. *Proceedings of the VLDB Conference*, 546-557.

Papadias, D., Mamoulis, N., & Theodoridis, Y. (1999) Processing and Optimization of Multiway Spatial Joins Using R-trees. *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 44-55.

Papadopoulos, A.N., Rigaux, P., & Scholl, M. (1999). A Performance Evaluation of Spatial Join Processing Strategies. *Proceedings of the Symposium on Large Spatial Databases (SSD)*, 286-307.

Patel, J.M., & DeWitt, D.J. (1996). Partition Based Spatial-Merge Join. *Proceedings of the ACM SIGMOD Conference*, 259-270.

Poosala, Y., & Ioannidis, Y. (1997) Selectivity Estimation without the Attribute Value Independence Assumption. *Proceedings of the VLDB Conference*, 486-495.

Preparata, F., & Shamos, M. (1985). *Computational Geometry*. Springer, New York.

Rotem, D. (1991). Spatial Join Indices. *Proceedings of the International Conference on Data Engineering (ICDE)*, 500-509.

Shou, Y., Mamoulis, N., Cao, H., Papadias, D., & Cheung, D.W. (2003). Evaluation of Iceberg Distance Joins. *Proceedings of the 8th International Symposium on Spatial and Temporal Databases, (SSTD)*, 270-288.

Theodoridis, Y., & Sellis, T. (1996). A Model for the Prediction of R-tree Performance. *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 161-171.

Theodoridis, Y., Stefanakis, E., & Sellis, T. (1998). Cost Models for Join Queries in Spatial Databases. *Proceedings of the ICDE Conference*, 476-483.

Endnotes

* supported by grant HKU 7149/03E from Hong Kong RGC

** also with the Data and Knowledge Engineering Group, Computer Technology Institute, Greece [http://dke.cti.gr]

*** supported by grant HKUST 6180/03E from Hong Kong RGC

1 Given a series of different layers of the same region (for example, rivers, streets, forests), its *workspace* is defined as the total area covered by all layers (not necessarily rectangular) including holes, if any.

2 RJ can be thought of as a special case of ST involving two inputs.