Multiway Spatial Joins

Nikos Mamoulis
University of Hong Kong
and
Dimitris Papadias
Hong Kong University of Science and Technology

Due to the evolution of Geographical Information Systems, large collections of spatial data having various thematic contents are currently available. As a result, the interest of users is not limited to simple spatial selections and joins, but complex query types that implicate numerous spatial inputs become more common. Although several algorithms have been proposed for computing the result of pairwise spatial joins, limited work exists on processing and optimization of *multiway spatial joins*. In this paper we review pairwise spatial join algorithms and show how they can be combined for multiple inputs. In addition, we explore the application of *synchronous traversal* (ST), a methodology that processes synchronously all inputs without producing intermediate results. Then, we integrate the two approaches in an engine that includes ST and pairwise algorithms, using dynamic programming to determine the optimal execution plan. The results show that in most cases multiway spatial joins are best processed by combining ST with pairwise methods. Finally, we study the optimization of very large queries by employing randomized search algorithms.

Categories and Subject Decription: H.2.8 [**INFORMATION SYSTEMS**]: Spatial databases and GIS: – Spatial Joins

General Terms: ALGORITHMS

Additional Key Words and Phrases: spatial joins, multiway joins, query processing

Authors' addresses: Nikos Mamoulis, Department of Computer Science and Information Systems, University of Hong Kong, Pokfulam Road, Hong Kong, e-mail: nikos@csis.hku.hk; Dimitris Papadias, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong, e-mail: dimitris@cs.ust.hk.
This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

1. INTRODUCTION

Spatial database systems [Güting 1994] manage large collections of multidimensional data which, apart from conventional features, include special characteristics such as position and spatial extent. The fact that there is no total ordering of objects in space that preserves spatial proximity [Günther 1993] renders conventional indexes, such as B^+-trees, inapplicable to spatial databases. As a result, a number of *spatial access methods* have been proposed [Gaede and Günther 1998]. A very popular spatial access method, used in several commercial systems (e.g., Informix, Illustra), is the R-tree [Guttman 1984]. It 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* during which all objects that cannot satisfy a query are excluded. A subsequent *refinement step*, uses the geometry of the candidate objects (i.e., the output of the filter step) to dismiss false hits and retrieve the actual solutions [Orenstein 1986]. R^+-trees [Sellis et al. 1987] and R^*-trees [Beckmann et al. 1990] are improved versions of the original method, proposed to address the problem of performance degradation caused by overlapping regions and excessive dead-space. The R-tree and its variations have been used to efficiently answer several types of queries including spatial selections, relation-based queries [Papadias et al. 1995] nearest neighbors [Roussopoulos et al. 1995] and spatial joins.

As in relational databases, joins play an important role in effective spatial query processing. A *pairwise spatial join* combines two datasets with respect to some spatial predicate (usually *overlap*). A typical example is "find all cities that are *crossed by* a river". The most influential algorithm for joining two datasets indexed by R-trees is the *R-tree join* (RJ) [Brinkhoff et al. 1993]. RJ traverses synchronously both trees, following entry pairs that overlap; non-intersecting pairs cannot lead to solutions at the lower levels. Several spatial join algorithms have been proposed for the cases that only one of the inputs is indexed by an R-tree [Lo and Ravishankar 1994, Mamoulis and Papadias 2001a, Papadopoulos et al. 1999] or when both inputs are not indexed [Lo and Ravishankar 1996, Patel and DeWitt 1996, Koudas and Sevcik 1997]. Most of these methods deal with the filter step; i.e., they output a set of MBR pairs (candidates) that may enclose intersecting objects.

Multiway spatial joins involve an arbitrary number of spatial inputs. Such queries are important in several applications including Geographical Information Systems (e.g. find all cities *adjacent to* forests which are *intersected* by a river) and VLSI (e.g. find all sub-circuits that formulate a specific topological configuration). Formally, a multiway spatial join can be expressed as follows: Given n datasets $D_1, D_2, \ldots D_n$ and a *query* Q, where Q_{ij} is the spatial predicate that should hold between D_i and D_j, retrieve all n-tuples $\{(r_{1,w},..,r_{i,x},..,r_{j,y},.., r_{n,z}) \mid \forall\, i,j : r_{i,x} \in D_i, r_{j,y} \in D_j$ and $r_{i,x}\ Q_{ij}\ r_{j,y}\}$. Such a 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 *constraint network* [Papadias et al. 1999] 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.

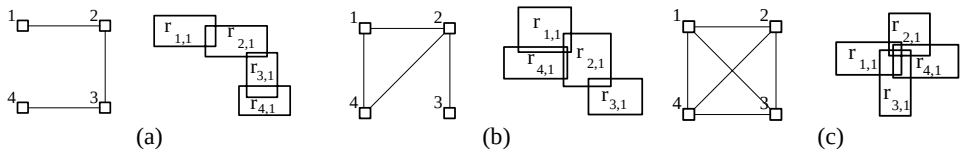


Fig. 1: Three queries of four objects.(a) Chain (tree) query. (b) Query with cycle. (c) Clique query.

Following the standard approach in the spatial join literature, we consider that all datasets are indexed by R-trees on MBRs; we deal with the filter step, assuming that *overlap* is the default join condition, i.e., if Q_{ij} = TRUE, then the rectangles from the corresponding inputs *i*,*j* should overlap. The loosest query has an acyclic (*tree*) graph (Fig. 1a), and the most constrained one has a complete (*clique*) graph (Fig. 1c). For each type of query, Fig. 1 illustrates a solution, i.e., a configuration of rectangles $r_{i,1} \in D_i$ that satisfy the join conditions. We do not consider non-connected query graphs, as these can be processed by solving connected sub-graphs and computing their Cartesian product.

In spite of the importance of multiway spatial joins, limited work has been carried out on their efficient processing. Patel et al., [Patel et al. 1997] apply a pairwise spatial join algorithm [Patel and DeWitt 1996] in a distributed, multi-processor environment to process cascading joins. Spatial data sets are regularly partitioned in space (*spatial declustering*), and the physical resources (disks, processors) are distributed according to the partitions. Papadopoulos et al. [Papadopoulos et al. 1999] perform a two-join case study to evaluate the performance of four spatial join algorithms. In [Mamoulis and Papadias 1999] we have proposed a *pairwise joins method* (PJM) that combines pairwise 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, i.e., all datasets are joined with respect to spatial properties. Motivated by this fact, in this paper we explore the application of *synchronous traversal* (ST), a methodology that traverses synchronously all the R-trees involved in the query, excluding combinations of intermediate nodes that do not satisfy the join conditions. RJ can be thought of as a special case of ST involving two inputs. 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], together with formulae for selectivity (in uniform datasets) and cost estimation (in terms of node accesses).

This work presents a complete solution for processing the filter step of multiway spatial joins: we (i) propose an efficient implementation of ST which takes advantage of the spatial structure of the problem to enhance performance, (ii) present methods for estimating its computational cost which dominates the I/O cost, (iii) combine ST with PJM in an integrated method that outperforms both alternatives, (iv) evaluate our approach through extensive experimentation with real and synthetic datasets, and (v) solve optimization problems for large join queries. The paper is organized as follows: in Section 2 we review algorithms for pairwise joins and discuss their integration using PJM. Section 3 proposes an improved version of ST, studies its behavior and estimates its cost. Section 4 contains a comprehensive experimental evaluation, which confirms the superior performance of the proposed implementation of ST, and a comparison of ST with PJM, which suggests that the two methodologies are complementary in the sense that they perform best under different conditions. Therefore, in Section 5 we study the processing of multiway spatial joins by combining ST with pairwise algorithms and propose selectivity estimation methods for non-uniform datasets. Section 6 deals with optimization of very large queries (where systematic search through the space of alternative plans is not possible) using randomized search algorithms. Finally, Section 7 concludes the paper with a discussion and directions for future work.

2. PAIRWISE SPATIAL JOIN ALGORITHMS

Most early spatial join algorithms apply 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) [Bially 1969] to order them. Each object is then approximated by the set of pixels intersected by its MBR, i.e. 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 is performed in a sort-merge join fashion. The performance of the algorithm depends on the granularity of the grid; larger grids can lead to finer object approximations, but also increase the space requirements. Rotem [Rotem 1991] proposes an algorithm based on a spatial join index, similar to the relational join index [Valduriez 1987], which partially pre-computes the join result and employs grid files to index the objects in space. A method, similar to RJ (*R-tree join*), that joins two PMR quadtrees is presented in [Hoel and Samet 1995]. Currently, the most influential algorithm is RJ due to its efficiency and the popularity of R-trees. Most research after RJ, focused on spatial join processing when one or both inputs are non-indexed.

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 exists 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 and DeWitt 1996] and then employ RJ to match the trees (*build and match*). Lo and Ravishankhar [Lo and Ravishankar 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 and Papadias 2001a] 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 processing. Arge et al. [Arge et al. 1998] propose an algorithm, called *scalable sweeping-based spatial join* (SSSJ), that employs a combination of *plane sweep [Preparata and Shamos 1985]* and space partitioning to join the datasets, and works under the assumption that in most cases the "horizon" of the sweep line will fit in main memory. However, the algorithm cannot avoid external sorting of both datasets, which may lead to large I/O overhead. Patel and DeWitt [Patel and DeWitt 1996] describe a hash-join algorithm, *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 partitions, 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 and Sevcik 1997]. S^3J avoids replication of objects during the partitioning phase by introducing more than one partition layers. 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* (HJ) [Lo and 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 above algorithms. 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 [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

and Papadias 2001a]. 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 HJ and PBSM. HJ 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 (e.g., 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). SSSJ 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 [Patel et al. 1997].

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, Hoel and Samet 1995]	• index nested loops • seeded tree join [Lo and Ravishankar 1994] • build and match [Patel and DeWitt 1996] • sort and match [Papadopoulos et al. 1999] • slot index spatial join [Mamoulis and Papadias 2001a]	• spatial hash join [Lo and Ravishankar 1996] • partition based spatial merge join [Patel and DeWitt 1996] • size separation spatial join [Koudas and Sevcik 1997] • scalable sweeping-based spatial join [Arge et al. 1998]

Table 1: Classification of spatial join methods

In [Mamoulis and Papadias 1999] we have proposed a method (PJM) that processes multiway spatial joins by combining RJ, HJ and SISJ. In this section we outline these algorithms and discuss their application in PJM. Since all algorithms are I/O bound, we also present formulae for their expected cost in terms of page accesses.

2.1 R-tree Join

RJ is based on the *enclosure property* of R-tree nodes: if two intermediate 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. Fig. 2 illustrates the pseudo-code for RJ assuming that the trees are of equal height; the extension to different heights is straightforward.

```
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 then /* N_j is also a leaf node */
        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);
      }
    }
  }
}
```

Fig. 2: R-tree-based spatial join (RJ)

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. Huang et al. [Huang et al. 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.

Theodoridis et al. [Theodoridis et al. 1998] provide an analytical formula that estimates the cost of RJ in terms of node accesses, based on the properties (density, cardinality) of the joined datasets. In their analysis, no buffer, or a trivial buffer scheme is assumed. In practice, however, the existence of a buffer affects the number of page accesses significantly. Here we adopt the formula provided in [Huang et al. 1997], which predicts actual page accesses in the presence of an LRU buffer. Let T_A, T_B be the number of pages in R-trees R_A, R_B, respectively. The cost of RJ in terms of I/O accesses is then estimated by the following formula:

$$C_{RJ} = T_A + T_B + (NA(R_A, R_B) - T_A - T_B) \cdot Prob(\text{node}, M) \quad (1)$$

where $NA(R_A, R_B)$ is the total number of R-tree nodes accessed by RJ and *Prob*(node, *M*) is the probability that a requested R-tree node will not be in the buffer (of size *M*) and will result in a page fault. Details about the computation of $NA(R_A, R_B)$ and *Prob*(node, *M*) can be found in [Huang et al. 1997].

2.2 Spatial Hash Join

Spatial hash-join (HJ) [Lo and Ravishankar 1996], based on the relational hash-join paradigm, computes the spatial join of two inputs, none of which is indexed.

Set A is partitioned into *S* buckets, where *S* 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 B is hashed into buckets with the same extent as A's buckets, but with a different insertion policy; an object is inserted into all buckets that intersect it. Thus, some objects may go into more than one bucket (*replication*), and some may not be inserted at all (*filtering*). The algorithm does not ensure equal sized[1] partitions for A, as sampling cannot guarantee the best possible bucket extents. Equal sized partitions for B cannot be guaranteed in any case, as the distribution of the objects in the two datasets may be totally different. Fig. 3 shows an example of two datasets, partitioned using the HJ algorithm.

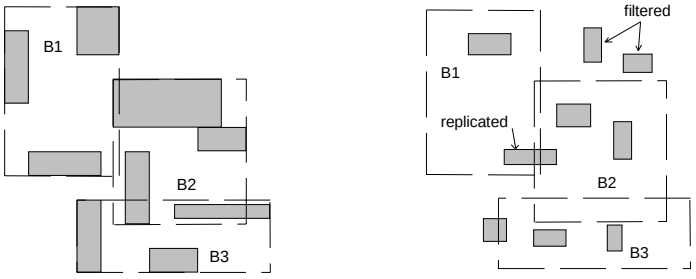


Fig. 3: The partition phase of HJ algorithm: (a) Objects from set A (build input) in three partition buckets. (b) Filtering and replication of objects from set B (probe input).

After hashing set B, the two bucket sets are joined; each bucket from A is matched with only one bucket from B, thus requiring a single scan of both files, unless for some pair of buckets none of them fits in memory. If one bucket fits in memory, it is loaded and the objects of the other bucket are prompted against it. If none of the buckets fits in memory, an R-tree is built for one of them, and the bucket-to-bucket join is executed in an index nested loop fashion.

The I/O cost of HJ depends on the size of the joined datasets and the filtering and replication that occur in set B. Initially, a small number of pages $C_{sampling}$ is read to determine the initial hash buckets. Then both sets are read and hashed into buckets. Let P_A, P_B be the number of pages of the two datasets (stored in sequential files) and r_B, f_B be the replication and filtering ratios of B. The partitioning cost of HJ is given by the following formula:

$$C_{HJ\text{-}part} = C_{sampling} + 2P_A + (2+r_B-f_B)\cdot P_B \quad (2)$$

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 HJ is:

[1] The term "size of partition/slot" denotes the number of objects inside the partition, and not its spatial extent.

$$C_{HJ\text{-}join} = P_A + (1+r_B\text{-}f_B)\cdot P_B \quad (3)$$

considering that the join output is not written to disk. Summarizing, the total cost of HJ is:

$$C_{HJ} = C_{HJ\text{-}part} + C_{HJ\text{-}join} = C_{sampling} + 3P_A + (3+2r_B\text{-}2f_B)\cdot P_B \quad (4)$$

2.3 Slot Index Spatial Join

SISJ [Mamoulis and Papadias 2001a] is similar to HJ, but uses the existing R-tree in order to determine the bucket extents. If *S* 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 *S*. These entries are then grouped into *S* (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. Fig. 4 illustrates a 3-level R-tree (the leaf level is not shown) and a slot index built over it. If *S*=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 *S*, we partition them in 9 (for this example) slots. The grouping policy used by SISJ (see [Mamoulis and Papadias 2001a] for details) 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 *S* 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 et al. 1990].

After building the slot index, the second set B is hashed into buckets with the same extents as the slots. As in HJ, if an object from B does not intersect any bucket it is filtered; if it intersects more than one buckets it is replicated. The join phase of SISJ is also similar to the corresponding phase of HJ. All data from R-tree R_A indexed by a slot are loaded and joined with the corresponding hash-bucket from set B using plane sweep. If the data to be joined do not fit in memory they can be joined using external sorting + plane sweep [Arge et al. 1998] or index nested loop join (using as root of the R-tree the corresponding slot). Since 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 [Silberschatz et al. 1997]. During the join phase of SISJ, when no data from B is inserted into a bucket, the sub-tree data under the corresponding slot is not loaded (slot filtering).

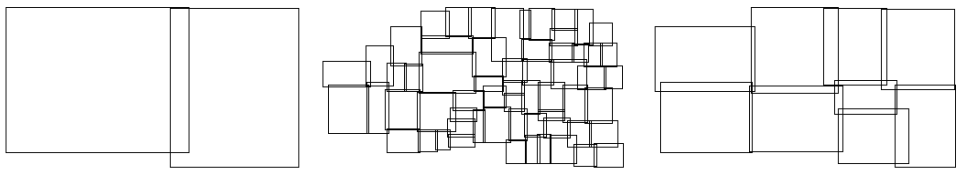


Fig. 4: An R-tree and a slot index built over it: (a) Level 2 (root) entries. (b) Level 1 entries. (c) Slot index over level.

Let T_A be the number of pages (blocks) of R_A, and P_B the number of pages of the sequential file B. Initially, the slots have to be determined from A. This requires loading the top *k* levels of R_A in order to find the appropriate slot level. Let s_A be the fraction of R_A nodes from the root until *k*. The slot index is built in memory, without additional I/Os. Set B is then hashed into the slots requiring P_B accesses for reading, and $P_B + r_B P_B - f_B P_B$ accesses for writing, where r_B, f_B are the replication and filtering ratios of B. Thus, the cost of SISJ partition phase is:

$$C_{SISJ\text{-}part} = s_A \cdot T_A + (2+r_B-f_B) \cdot P_B \quad (5)$$

For the join phase of SISJ we make the same assumptions as for HJ, i.e., for each joined pair at least one bucket fits in memory. The pages from set A that have to be fetched for the join phase are the remaining $(1-s_A) \cdot T_A$, 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:

$$C_{SISJ\text{-}join} = (1-s_A) \cdot T_A + (1+r_B-f_B) \cdot P_B \quad (6)$$

Summarizing, the total cost of SISJ is:

$$C_{SISJ} = C_{SISJ\text{-}part} + C_{SISJ\text{-}join} = T_A + (3+2r_B-2f_B) \cdot P_B \quad (7)$$

2.4 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 exactly one execution plan where: (i) RJ is applied when the inputs are leaves i.e., datasets indexed by R-trees, (ii) SISJ is employed when only one input is indexed by an R-tree and (iii) HJ when both inputs are intermediate results. As an example of PJM, consider the query in Fig. 1a and the plans of Fig. 5. Fig. 5a 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 Fig. 5b applies RJ for $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, and HJ to join the intermediate results.

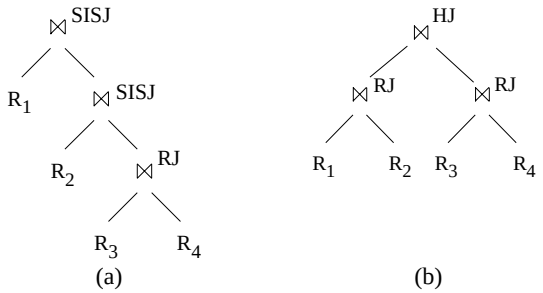


Fig. 5: Alternative plans using pairwise join algorithms: (a) Right-deep plan. (b) Bushy plan.

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 (i.e., 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 1, 4 and 7 to estimate the join cost of the three algorithms. The expected output size (i.e., number of solutions) of a pairwise join determines the execution cost of an upper operator and therefore is crucial of optimization. The size of a join output is determined by:

- The cardinality of the sets to be joined. If $|R_1|$, $|R_2|$ are the cardinalities of two inputs, the join may produce up to $|R_1| \cdot |R_2|$ tuples (Cartesian product).
- The density of the sets. The density of a dataset is formally defined as the sum of areas of all rectangles in the dataset divided by the area of the workspace. Datasets with high density have rectangles with large average area, thus producing numerous intersections.
- The distribution of the rectangles inside the sets. This is the most difficult factor to estimate, as in many cases the distribution is not known, and even if known, its characteristics are very difficult to capture.

According to the analysis in [Theodoridis et al. 1998] and [Huang et al. 1997], the number of output tuples when joining two-dimensional datasets R_1 and R_2 with uniform distribution is:

$$|R_1 \bowtie R_2| = |R_1| \cdot |R_2| \cdot (s_{R_1} + s_{R_2})^2 \quad (8)$$

where s_{R_1} is the average side length of a rectangle in R_1, and the rectangle co-ordinates are normalized to take values from [0,1). The last factor of the product corresponds to *pairwise join selectivity*, i.e. the probability that a random pair of rectangles from the two datasets intersect.

Optimization of multiway spatial joins requires selectivity estimation for each possible decomposition of the query graph (i.e., for each allowable sub-plan). The generalized formula for the output size of a query (sub) graph *Q* with *n* inputs is:

$$\#solutions(Q) = \#(possible\ tuples) \cdot Prob(a\ tuple\ is\ a\ solution) \quad (9)$$

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:

$$Prob(a\ tuple\ is\ a\ solution) = \prod_{\forall i,j:Q(i,j)=TRUE} \left(s_{R_i} + s_{R_j}\right)^2 \quad (10)$$

From Equations 9 and 10, total number of query solutions is:

$$\#solutions(Q) = \prod_{i=1}^{n} |R_i| \cdot \prod_{\forall i,j:Q(i,j)=TRUE} \left(s_{R_i} + s_{R_j}\right)^2 \quad (11)$$

When the query graph contains cycles, the pairwise selectivities are not independent anymore and Eq. 10 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]:

$$Prob(a\ tuple\ is\ a\ solution) = \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} s_{Rj} \right)^2 \quad (12)$$

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

$$\#solutions(Q) = \prod_{i=1}^{n} |R_i| \cdot \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} s_{Rj} \right)^2 \quad (13)$$

The above formulae are applicable for queries that can be decomposed to acyclic and clique graphs (e.g., the one in Fig. 1b). The optimal execution plan can be computed from the estimated output size and the costs of the algorithms involved. Selectivity estimation for real datasets is discussed in Section 5. Next we describe, synchronous traversal, an alternative to PJM for processing multiway spatial joins.

3. SYNCHRONOUS TRAVERSAL

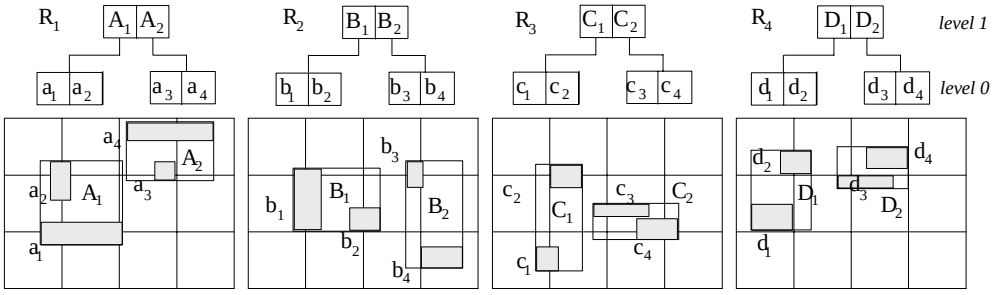


Fig. 6: Example of R-trees

ST synchronously processes the indexes of all joined datasets, following combinations of nodes that satisfy the query constraints. Consider the four R-trees of Fig. 6 and the clique query of Fig. 1c. The query asks for the set of 4-tuples (a_w,b_x,c_y,d_z), such that the four objects mutually overlap (e.g., (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 (i.e.,

(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, intermediate level solutions are recursively followed until the leaves.

A problem of ST is that exhaustive enumeration of all combinations at each level is prohibitive because of their large number. Moreover, the CPU-time optimization techniques for RJ are not readily applicable for multiple inputs. In this section we propose an efficient implementation for ST and provide an accurate formula for its expected cost.

3.1 Description of ST

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. 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 D_i which consists of the entries $\{E_{i,1},.., E_{i,C_i}\}$ 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* iff Q_{ij} = TRUE $\Rightarrow E_{i,x}$ overlaps $E_{j,y}$. A solution of a local problem is a n-tuple $\tau = (E_{1,w},.., E_{i,x},.., E_{j,y},..,E_{n,z})$ such that $\forall i,j$, $\{v_i \leftarrow E_{i,x}, v_j \leftarrow E_{j,y}\}$ is consistent. The goal is to find all *solutions*, i.e., assignments of entries to variables such that all constraints are satisfied.

In the previous example (clique query of Fig. 1c), 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 D_1={A_1,A_2}, D_2={B_1,B_2}, D_3={C_1,C_2} and D_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, i.e., D_1={a_1,a_2}, D_2={b_1,b_2}, D_3={c_1,c_2} and D_4={d_1,d_2}. Notice that an intermediate level solution does not necessarily lead to an actual one. As we show later, the percentage of combinations that constitute solutions increases as we go up the levels of the trees because of the large node extents. Since a part of node area corresponds to "dead space" (space not covered by object MBRs), many high level solutions are *false hits*.

```
ST(Query Q[][], RTNode N[]) {
  for i:=1 to n do {  /*prune domains*/
    D_i := space-restriction(Q, N, i);
    if  D_i = ∅ then RETURN; /*no qualifying tuples may exist for this combination of nodes*/
  }
  for each τ ∈ find-combinations(Q, D) do { /* for each solution at the current level */
    if N are leaf nodes then /*qualifying tuple is at leaf level*/
      Output(τ);
    else /*qualifying tuple is at intermediate level*/
      ST(Q, τ.ref[]); /* recursive call to lower level */
  }
}
Domain space-restriction(Query Q[][], RTNode N[], int i)  {
  read N_i; /* read node from disk */
  D_i := ∅;
  for each entry E_i,x ∈ N_i do {
    valid := true; /*mark E_i,x as valid */
    for each node N_j, such that Q_ij = TRUE do /*an edge exists between N_i and N_j*/ {
      if  E_i,x ∩ N_j.MBR = ∅ then { /* E_i,x does not intersect the MBR of a neighbor node to N_i */
        valid := false; /* E_i,x is pruned */
        break;
      }
    if valid=true then /*E_i,x is consistent with all node MBRs*/
      D_i := D_i ∪ E_i,x;
  }
  return D_i;
}
```

Fig. 7: Synchronous R-tree traversal.

The pseudo-code for ST, assuming R-trees of equal height, is presented in Fig. 7. For each D_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 Fig. 1a and the top-level solution (A_2,B_1,C_1,D_1). At the next level ST is called with D_1={a_3,a_4}, D_2={b_1,b_2}, D_3={c_1,c_2} and D_4={d_1,d_2}. Although A_2 intersects B_1, none of entries (a_3,a_4) does and these entries can be safely eliminated from D_1. Since D_1 becomes empty, (A_2,B_1,C_1,D_1) cannot lead to an actual solution and search is abandoned without loading [2]. *Find-combinations* is the "heart" of ST; i.e., the search algorithm that finds tuples $\tau \in D_1 \times D_2 \times ... \times D_n$, that satisfy Q. In order to avoid exhaustive search of all combinations, several backtracking algorithms applied for constraint satisfaction problems, can be used. One such algorithm is *forward checking* (FC) [Haralick and Elliott 1980]. Forward checking accelerates search by progressively assigning values to variables and pruning the domains of future (uninstantiated) variables. Given a specific order of the problem's variables $v_1, v_2, \ldots, v_n$, 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

[2] In order to avoid redundant page faults, *space-restriction* does not load a node from disk, until its entries have to be compared with the MBRs of the other nodes.

backtracks to v_{i-1} trying a new value for this variable. The algorithm terminates after backtracking from v_1. Several experimental studies (e.g., [Bacchus and Grove 1995]) in different domains have shown the superiority of FC compared to other algorithms. In previous work [Papadias et al. 1998, Papadias et al. 1999], we have applied FC as an implementation of *find-combinations* and called this version of ST, *multilevel forward checking* (MFC).

As we show next, the performance of search can be further improved by exploiting the spatial structure of the problem. In particular, we propose a heuristic that combines *plane sweep* and *forward checking* in order to reduce the number of comparisons required. Furthermore, we describe an optimization method that orders the problem variables according to their *degree* (i.e., the number of adjacent edges in the query graph).

3.2 Optimization of ST: SVO and PSFC

The order of variables in the first implementation of *[Papadias et al. 1998]* was not considered because the structural queries examined there had complete graphs; therefore, all variables/nodes had identical degrees. In the multiway spatial join problem examined here, a pair of variables is not necessarily connected by a graph edge, thus the *static* order in which the variables are considered is important for non-complete queries (e.g. the first two queries in Fig. 1).

We propose a preordering of the problem variables based on a *static variable ordering* (SVO) heuristic, which "places the most constrained variable first" [Dechter and Meiri 1994]. Before running ST, the variables are sorted in decreasing order of their degree. Thus for the chain query in Fig. 1a the order of the variables will be {2,3,1,4} (or any other equivalent order e.g., {3,2,4,1}) and for the query in Fig. 1b {2,1,4,3}. Variable preordering is applied only once and the produced static order is used in *find-combinations* and *space-restriction* at every execution of ST.

Next, we present an alternative search algorithm, PSFC (*plane sweep* combined with *forward checking*), as an improved implementation of *find-combinations*. PSFC does not solve a single local problem (as FC does), but breaks it into a series of small problems, one for each event of the sweep line. In other words, plane sweep is applied for problem decomposition and domain restriction, and then a special version of FC, called *sortFC*, takes advantage of the small domains and order in space to efficiently solve each small sub-problem.

The pseudo-code for PSFC is shown in Fig. 8. First the entries of all nodes are sorted according to their *xl* coordinates (*x* coordinate of the lower left point of the MBR). A set of pointers (*heads*), one for each node, is maintained initially pointing to the first element of the sorted entries array. Let $E_{i,x} \in N_i$ be the entry with the smallest *xl* pointed by a head. PSFC will start by finding all solutions containing the assignment $v_i \leftarrow E_{i,x}$. The domains of the other variables are formulated as follows: if Q_{ij} = TRUE, all entries $E_{j,y}$ after (≥) $head_j$ such that

$E_{j,y}.xl \le E_{i,x}.xu$ are tested for y-intersection with $E_{i,x}$ and added in the domain of v_j. If Q_{ij} = FALSE, the domain consists of all rectangles in N_j after (≥) $head_j$. Now v_i is ignored and sortFC is invoked to solve the sub-problem involving the remaining *n-1* variables. In this way, search is restricted only close to the fixed rectangle $E_{i,x}$, and redundant checks are avoided. After all solutions that contain $v_i \leftarrow E_{i,x}$ are found, $head_i$ is increased and the process is repeated. Notice that no duplicate solutions are retrieved, since the rectangles checked are always at or after the sorted array heads. The heads can be organized in either a priority queue or a sorted array to facilitate fast retrieval of the smallest *xl*-coordinate head.

```
PSFC(Query Q[][], RTNode N[]) {
  for i:=1 to n do {
    sort(N_i); /* sort entries in N_i according to xl co-ordinate */
    head_i := 1; /* hold a pointer to the first entry in each sorted list */
  }
  do { /* do-loop */
    i := variable with min(N_i[head_i].xl);
    τ[i] :=N_i[head_i]; /* fix τ[i] */
    for each j ≠ i do {/* set domains for the rest of the variables */
      if Q_ij=true then /* v_i and v_j are connected in the query graph */
        D_j := ∅;
        while (E_j,y := next entry of N_j after head_j) and (E_j,y.xl ≤ τ[i].xu) do
          if y-intersection(E_j,y, τ[i]) then
            D_j := D_j ∪ E_j,y; /* E_j,y intersects τ[i] */
        if D_j=∅ then { /*no entry of N_j intersects τ[i]*/
          head_i := head_i + 1;
          if (head_i ≤ N_i.size ) then goto beginning of do-loop; /*move to next problem */
          else return;
        }
      else /*v_i and v_j are not connected in the query graph */
        D_j := ∅;
        for each entry E_j,y of N_j after head_j do
          D_j := D_j ∪ E_j,y; /* copy all entries after head_j to D_j */
    } /*end of domains initialization*/
    T := sortFC(Q, D, j ≠ i); /* call sortFC for the sub-problem including all j ≠ i ; T stores all
solution tuples τ'*/
    for each τ' ∈ T do { /* for each solution returned by sortFC */
      τ := τ' ∪ τ[i]; /* concatenate τ[i] */
      Output(τ);
    }
    head_i := head_i + 1; /*move head of fixed variable */
  } while (head_i ≤ N_i.size);
}
```

Fig. 8: PSFC algorithm.

To comprehend the functionality of PSFC consider the query in Fig. 9 applied to four R-tree nodes A, B, C, D. The node with the smallest *xl*-coordinate head is D. PSFC sets $v_4 \leftarrow d_1$ and filters the initial domains of A and B because $Q_{14} = Q_{24}$ = TRUE, setting $D_1 = \{a_1\}$ and $D_2 = \{b_1\}$, whereas D_3 remains $\{c_1,c_2,c_3,c_4\}$. SortFC is then called to solve the (A,B,C) sub-problem, identifying the solution (a_1,b_1,c_2).

By concatenating the current value of the fixed variable ($v_4 \leftarrow d_1$), the first solution (a_1,b_1,c_2,d_1) is generated. *head*$_4$ is moved to d_2 and the algorithm will not consider d_1 again. Each step in Fig. 9 corresponds to a small problem; the first (thick) vertical line illustrates the sweep line, while the second one shows the upper limit of the fixed rectangle on the x-axis, i.e., the two lines show the x-axis ranges for the objects to be tested for intersection. The dark rectangles indicate the position of the head pointers and the grided rectangles are the ones that constitute the domains at the current step.

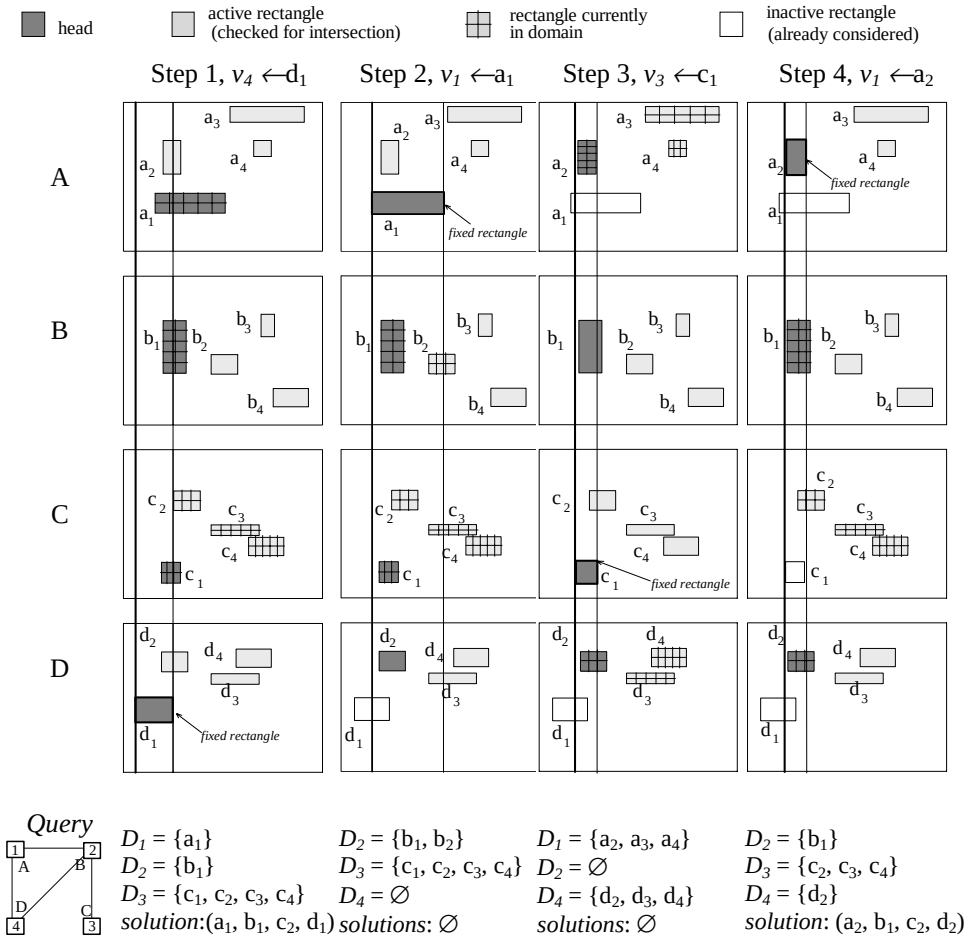


Fig. 9: Four steps of PSFC.

At the second step v_1 is fixed to a_1. The only values from D_2 which are checked for intersection with a_1 are b_1 and b_2; both are included since they overlap a_1. D_3 contains all rectangles because Q_{13} = FALSE. D_4 is eliminated since Q_{14} = TRUE and none of the entries between the vertical lines (d_2 and d_3) intersects a_1. Therefore, $v_1 \leftarrow a_1$ cannot lead to a solution and the algorithm proceeds to the next step (without calling sortFC) fixing v_3 to c_1. The assignment $v_3 \leftarrow c_1$ also eliminates domain D_2 and PSFC moves to the fourth step ($v_1 \leftarrow a_2$). The same

process is repeated until all rectangles are considered and the lists are exhausted. Assuming that each node is full, PSFC solves $n \cdot C$ sub-problems for each local problem.

SortFC uses the sorted domain entries to avoid redundant comparisons during search. After an assignment $v_k \leftarrow E_{k,v}$ ($k \neq i$, where v_i is the variable fixed by PSFC), *check_forward* is called to delete from the domains of future variables v_j ($j>k$) connected to v_k (Q_{kj} =TRUE) those values that do not intersect $E_{k,v}$. Exhaustive search for valid values is prevented, by stopping when an entry $E_{j,y}$ with $E_{j,y}.xl > E_{k,v}.xu$ is reached (all subsequent entries in D_j cannot overlap $E_{k,v}$).

Fig. 10 illustrates the pseudo-code for sortFC. The domains are kept in a 3-dimensional array *D*, in order to facilitate fast restoration of future variable values after the current variable has been unsuccessfully instantiated. *D[k, j]* contains all potential values (entries) of v_j that are consistent with all instantiations of variables prior to v_k. When v_k gets assigned ($v_k \leftarrow E_{k,v}$), *check-forward* copies to *D[k+1, j]* all consistent entries of *D[k, j]*. If Q_{kj} = FALSE, *D[k+1, j]=D[k, j]*; otherwise, *D[k+1, j]*=$\{E_{j,y} \mid E_{j,y} \in D[k, j]$ and $E_{j,y}$ overlaps $E_{k,v}\}$. The domain of v_j is eliminated when there exists a join condition Q_{kj}, but no value in *D[k, j]* that intersects $E_{k,v}$. In this case $v_k \leftarrow E_{k,v}$ cannot lead to a solution, a new value for v_k is chosen, and *D[k+1, j]* is re-initialized to *D[k, j]*. If no future domain gets eliminated the algorithm goes forward to variable v_{k+1} (according to the order determined by SVO).

```
sortFC(Query Q[][], Domain D[][]) {
  k := 1; /* index to the current variable */
  while (k >0) do {
    τ[k] := next value in D[k, k]; /*  τ holds the current instantiations - τ[k]: current value of v_k */
    if τ[k] = NULL then /* no more values in D[k, k] */
      k := k -1; /* backtrack */
    else if k = n then /* last variable instantiated */
      Output (τ);
    else if check_forward(Q,D,k) then /*no future variable eliminated */
      k := k + 1; /* instantiate next variable */
  }
}
boolean check_forward(Query Q[][], Domain D[][], int k) {
  for j := k+1 to n do  /*for all future variables*/
    if Q_kj = true then { /* if there is an edge between v_k and v_j */
      D[k+1, j] := ∅; /* initialize v_j's domain for next instantiation */
      while (E_j,y = next entry ∈ D[k, j]) and (E_j,y.xl ≤ τ[k].xu) do
        if E_j,y ∩ τ[k] ≠ ∅ then /* consistent value */
          D[k+1, j] := D[k+1, j] ∪ E_j,y; /* add value to domain */
      if D[k+1, j] = ∅ then
        return false; /* future variable has been eliminated */
    }
    else /*no edge between v_k and v_j */
      D[k+1, j] := D[k, j]; /*copy whole domain */
  return true;
}
```

Fig. 10: sortFC.

Assuming that all R-trees have the same height, if sortFC is applied for leaf nodes the tuples are output after being concatenated with the current value of the fixed (by PSFC) variable (v_i); otherwise, ST is invoked for each qualifying combination of entries, by following the corresponding pointers. When trees have different heights, some D_i may consist of a single leaf MBR $r_{i,x}$ (if tree R_i is shallower than some others). In this case, *space-restriction* is called only for the other inputs. Assume, for the sake of this example, that the root of R_1 in Fig. 6 is a leaf node (i.e. A_1, A_2 are object MBRs) and the first tree is shallower than the other ones. When the clique query of Fig. 1c is processed, solution (A_1,B_1,C_1,D_1) will be found at the top level. At the next level, *space-restriction* will be executed to prune D_2 (i.e., entries under B_1), D_3 and D_4, whereas D_1={A_1}. Finally PSFC will be called and retrieve solution (A_1,b_1,c_2,d_2).

Summarizing, ST breaks multi-way spatial join processing in local problems throughout the levels of the tree. PSFC in turn, further decomposes each local problem in smaller sub-problems in order to avoid the overhead of searching and backtracking in large domains. Next, we study the behavior of ST and provide a formula for estimating its cost.

3.3 Cost Estimation of ST

ST starts from the top level *h-1* (where *h* 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 the leaves where solutions are output. Thus, the total number of local problems is:

$$N_{PROBLEMS} = 1 + \sum_{l=1}^{h-1} \#\,solutions(Q,l) \qquad (14)$$

where *#solutions(Q, l)* is the number of qualifying entry combinations at level *l*. In order to comprehend the behavior of ST, it is important to scrutinize the number of solutions at each level. We generated four collections of synthetic datasets, with densities[3] 0.1, 0.2, 0.4, 0.8. Each dataset contains 30,000 uniformly distributed rectangles organized in an R*-tree of page size 8K and height 2. For each object, we index/store its MBR, which is described by four 4-byte floats, and an object id (a 4-byte integer). Therefore, R-tree entries require 20 bytes of storage. A 512K system buffer with LRU page replacement policy is employed[4]; R-tree nodes, disk pages and buffer pages have the same size. All

[3] The densities of the real geographic datasets used in this paper range between 0.04 and 0.39. Denser spatial datasets were used in [Koudas and Sevcik 1997, Mamoulis and Papadias 1999]. VLSI data tend to have higher densities, such as 1.2, in the experiments of [Papadias et al. 1999a].

[4] All datasets used have sizes between 350Kb-3Mb, so they fit in the machine's memory. We used a relatively small buffer in order to simulate situations where the datasets are larger than the available buffer (up to an order of magnitude). Analogous buffer sizes are used by other studies in the literature (e.g., [Lo and Ravishankar 1996, Koudas and Sevcik 1997, Huang et al. 1997a]. The underlying assumption (which we evaluated through experiments with large synthetic datasets) is

experiments were run on an Ultrasparc2 workstation (200 MHz) with 256 Mbytes of memory. Because of the caching effects of the machine, it was hard to count the I/O and computational cost accurately. Therefore, we charged 10msec (a typical value [Silberschatz et al. 1997], [Huang et al. 1997a]) for each I/O access, the majority of which are random in R-tree-based algorithms. In the following we denote as "overall cost" the sum of the actual CPU time and the estimated I/O cost (see also [Huang et al. 1997a]).

For each collection of datasets we run ST for chain and clique queries. The first column of Fig. 11 illustrates the number of root level solutions for chains (first row) and cliques (second row) as a function of the number of inputs *n* and the data density. Since the trees have only two levels, the number of root level solutions is equal to the number of problems that have to be solved at the leaf level. As observed in [Theodoridis and Sellis 1996], the size of intermediate node extents is determined mainly by the capacity of the nodes, while density has trivial effect. Thus, the number of solutions at high levels of the trees is almost independent of the density of the datasets for small to medium density values (0.1, 0.2 and 0.4). For the high density case (0.8), the data rectangles are large enough to affect the size of intermediate nodes, resulting in more solutions.

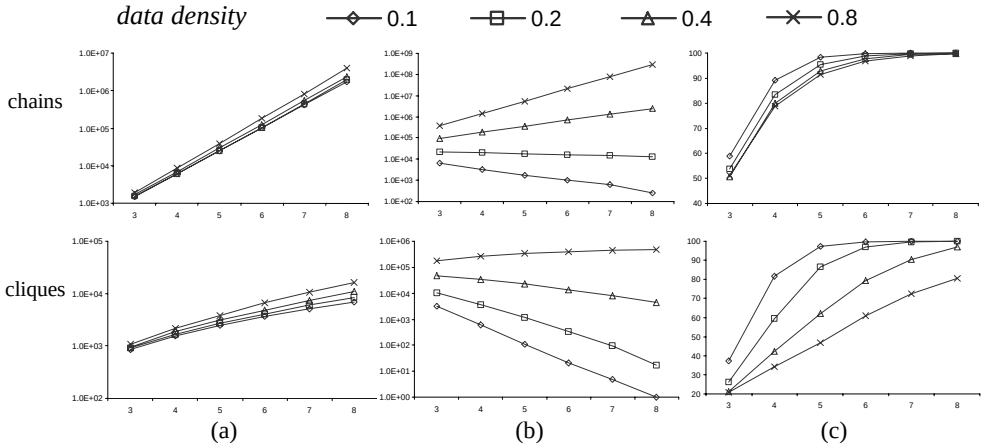


Fig. 11: Effects of data density and *n* on the solutions of ST: (a) Number of root (level 1) solutions. (b) Number of actual (level 0) solutions. (c) Percentage (%) of false hits.

The second column of Fig. 11 presents the number of output tuples (actual solutions at the leaf level). Observe that there is a range of density values (around 0.2-0.4 for chains and 0.4-0.8 for cliques) where the number of solutions does not vary considerably with *n*. Density values above that range result in exponential growth in the number of solutions, while values below the range eventually yield zero solutions. This is in contrast with the root level solutions which always increase with the number of inputs due to the large node extents. The root (and in general intermediate) level solutions are more important than the actual ones for

---

that the various algorithms scale in a similar way, i.e., the relative performance does not change with the size of the data.

the complexity of ST since they determine the number of local problems to be solved. The last column shows the percentage of root solutions that do not lead to any output tuple (*false hits*). This percentage is large for sparse datasets and query graphs and converges to 100% with *n*.

As a result of the large number of intermediate level solutions (most of which are false hits), the cost of ST is expected to grow exponentially with the number of inputs. The large number of local problems affects mainly the CPU-time, because the LRU buffer absorbs the I/O side effects due to the high overlap between consecutive solutions. Fig. 12 presents the CPU-time of ST (using SVO and PSFC) as a percentage of the overall cost for running chain and clique queries using the synthetic datasets with density 0.4. Other densities produce very similar results, independently of the algorithm used for ST. The diagram suggests that ST is indeed CPU bound and, in the sequel, we restrict our attention to its computational cost.

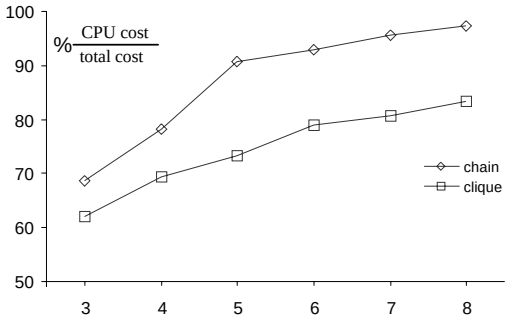


Fig. 12: CPU-time percentage (%) of the total cost as a function of the number of inputs.

The local problems have the same characteristics (i.e., number of variables, constraints and domain size); therefore, it is reasonable to assume that they all have approximately the same computational cost ($C_{PROBLEM}$). Consequently, the total CPU cost (C_{CPU}) of ST equals the number of local problems times the cost of each problem.

$$C_{CPU} = N_{PROBLEMS} \cdot C_{PROBLEM} \quad (15)$$

$N_{PROBLEMS}$ can be estimated by Eq. 14 using the formulae of Section 2 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 Eq. 11 and 13. The remaining factor to compute Eq. 15 is the cost $C_{PROBLEM}$. Although in the worst case (e.g., extremely large intermediate nodes), each local problem is exponential ($O(C^n)$), the average $C_{PROBLEM}$ for typical situations is much lower (actually, 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[5]. Therefore, in order to provide a cost formula for $C_{PROBLEM}$, we perform an empirical study.

The parameters that may have an effect on $C_{PROBLEM}$ are the cardinality and density of the domains, and the query graph (number of variables, graph density). The cardinality depends on the system page size *p*. The next experiment identifies the effect of domain density and query graph using the previous synthetic datasets. Fig. 13 shows the mean cost of a local problem for various data densities when *p* is fixed to 8K (1st row) and the number of variables *n* is fixed to 5 (2nd row) for chain and clique queries. The important observation is that the problem cost is almost constant[6] with the data density, and increases linearly with *n* and *p*. It is also independent of the query graph, since chains and cliques, the two extreme cases of possible graphs, give approximately the same value from $C_{PROBLEM}$.

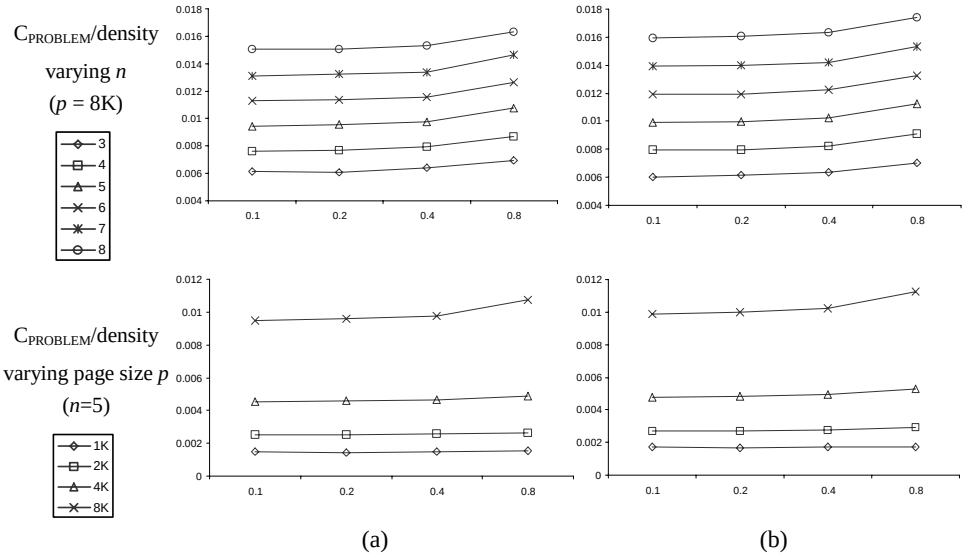


Fig. 13: Average CPU-time of a local problem in ST: (a) Chain queries. (b) Clique queries.

$C_{PROBLEM}$ was also measured for the three queries of Fig. 1 executed over real datasets (Tiger and Germany), described and used in subsequent experimental evaluations. The same execution conditions were used for all six queries (8K

[5] The only existing theoretical analysis [Kondrak and van Beek 1997] compares the relative performance of different search heuristics in terms of consistency checks and cannot be applied for absolute cost estimation.

[6] For densities of 0.8, the problem cost increases slightly due to the high number of actual solutions (see Fig. 13), since search algorithms are, in general, output sensitive. Nevertheless the difference is marginal.

page size, 4 datasets). Table 2 presents the results, suggesting that the values of $C_{PROBLEM}$ are very similar for all query and dataset combinations.

From the above empirical analysis it can be concluded 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*. Thus, we can define:

$$C_{PROBLEM} = F \cdot n \cdot p \quad (16)$$

where F is a factor that depends on the algorithm for ST and the CPU speed. For SVO-PSFC and our experimental settings (see next section), its value is around $2.7 \cdot 10^{-7}$. F can be estimated by Eq. 14, 15 and 16 and the actual cost of a multiway join. The substitution of the above formula for $C_{PROBLEM}$ in Eq. 15 provides a cost prediction with a relative error less than 15% (for uniform data), which is as good as the corresponding models for pairwise joins [Huang et al. 1997, Theodoridis et al. 1998]

	C_{CPU}	$N_{PROBLEMS}$	$\mathbf{C_{PROBLEM}}$
Tiger-chain	177.09	23540	0.007523
Tiger-cycle	100.9	12720	0.007933
Tiger-clique	64.05	8085	0.007922
Germany-chain	35.83	4864	0.007367
Germany-cycle	18.96	2544	0.007454
Germany-clique	12.06	1549	0.007786

Table 2: $C_{PROBLEM}$ for real datasets

4. EXPERIMENTAL EVALUATION OF ST AND COMPARISON WITH PJM

The experiments of this section are grouped in two parts: first we evaluate the effectiveness of SVO and PSFC with respect to MFC and then we compare it against PJM in order to identify the best alternative given the problem characteristics. We used the Tiger files from the US Bureau of the Census [Bureau of the Census 1989] and datasets describing several layers of Germany (available at *http://www.maproom.psu.edu/dcw/*). Details regarding the sizes and densities of the Germany layers can be found in Fig. 18. The experimental settings were the same as in Section 3.3.

4.1 Performance Improvement of SVO and PSFC

In the first experiment, we measure the performance of ST using (i) FC (i.e., MFC [Papadias et al. 1998]) (ii) FC combined with SVO heuristic (SVO-MFC) and (iii) PSFC combined with SVO heuristic (SVO-PSFC). Tiger files T1 and T2, which are common benchmarks of spatial join algorithms [Brinkhoff et al. 1993, Lo and Ravishankar 1996, Huang et al. 1997a], were used. T1 contains 131,461 street segments (density D=0.05), and T2 128,971 railway and river segments

(D=0.39) from California. Two more datasets T3 and T4 were produced from T1 and T2, respectively, by taking as the lower left corner of an MBR the center of the original object and considering the same width and height (a similar method was used in [Koudas and Sevcik 1997]). The three versions were executed for the queries of Fig. 1 and various page sizes. Sizes of 2K-8K resulted in R-trees of three levels, and 1K in R-trees of four levels[7].

The results of Fig. 14 suggest that the improvement after the application of SVO prior to MFC is significant for the first two queries, where the first variable is not the most constrained; there is no difference for the clique query, as all variables have the same degree. Surprisingly, in one case (2nd query, 4K page size) SVO-MFC performed worse than MFC in terms of I/O, although SVO-MFC accessed fewer R-tree nodes. This shows that, in general, there is not high locality between two consecutive solutions produced by MFC. On the other hand, PSFC generates solutions with high locality, and has an advantage in terms of I/O compared to MFC. The performance gain of PSFC in terms of CPU-time is higher for large pages, as sorting and sweeping pay off. As discussed in the previous section, the CPU improvement is more significant than I/O, because ST is CPU bound.

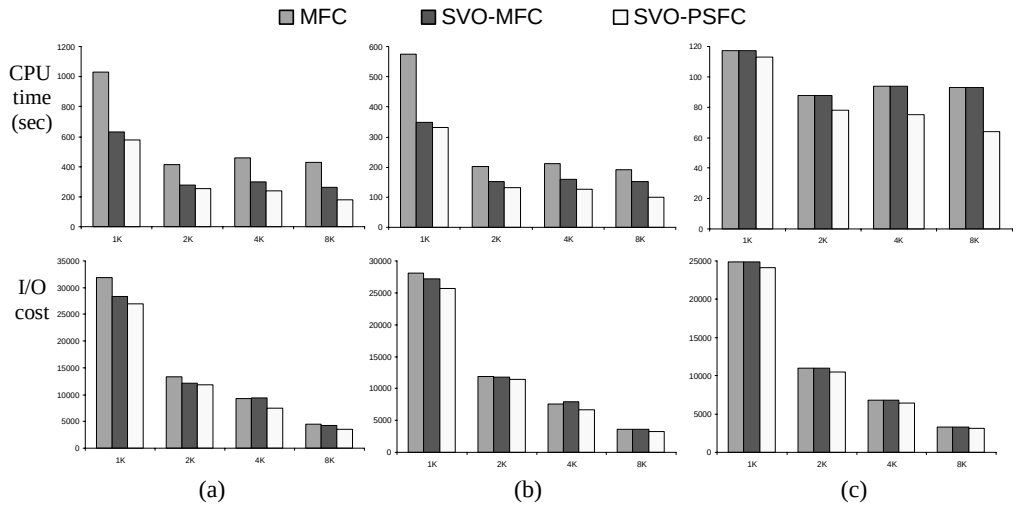


Fig. 14: CPU time (in seconds) and I/O accesses for the three ST versions for several page sizes: (a) Chain query. (b) Query with cycle. (c) Clique query.

[7] The only multiple datasets covering the same area, Germany layers, are not suitable for this experiment because (due to their very different cardinalities) they result in R-trees of variable heights for page sizes less than 8K. Joining trees of variable heights does not allow for a consistent comparison between MFC and PSFC, because the domains of some variables at a large number of problems reduce to 1. Nevertheless, we have run the same experiment with the Germany datasets and found SVO-PSFC also superior to other methods, but the results were less interpretable regarding the effects of the page size.

PSFC (and ST in general) performs best for large page sizes. As the page size increases, so does the capacity of the R-tree nodes, meaning that ST has to solve fewer local problems with larger domains. As a result, the number of I/O accesses drops significantly; on the other hand, the CPU-time difference is not that large. This is because the increase of domain size is compensated by the decreasing number of intermediate level problems. Therefore, there exist fewer false hits due to dead space in the upper tree nodes. The performance improvement is substantial between the 1K and 2K page sizes because of the different tree heights. Notice also that the CPU-difference between PSFC and MFC (when both use SVO) increases with the page size. This can be explained by the fact that PSFC achieves very good pruning in large pages (e.g., 8K) with many rectangles that cover a wider area, compared to the pruning in small pages (e.g., 1K) with few rectangles in a smaller area. In subsequent experiments the page size is fixed to 8K.

In order to measure the efficiency of SVO-PSFC against MFC under a wide range of conditions, we also used the uniform datasets described in Section 3. Fig. 15 presents the number of times that SVO-PSFC outperforms MFC (i.e., cost MFC/cost SVO-PSFC) in terms of overall normalized cost as a function of *n* for chain and clique queries. The improvement factor increases with the data density, while the difference is greater for chains (and sparse query graphs, in general), an important fact because such queries are common and have large computational overhead. Even for joins of low-density datasets the improvement is significant (i.e., 50%-300%) when the number of datasets is small (<6). As we will see in Section 5.3, ST is especially useful for joining small groups of relations, a fact that increases the value of PSFC. As a general conclusion, SVO and PSFC contribute equally to the performance improvement of ST for reasonably large page sizes (i.e., 4K-8K) and we adopt both in the standard implementation of ST for the rest of the paper.

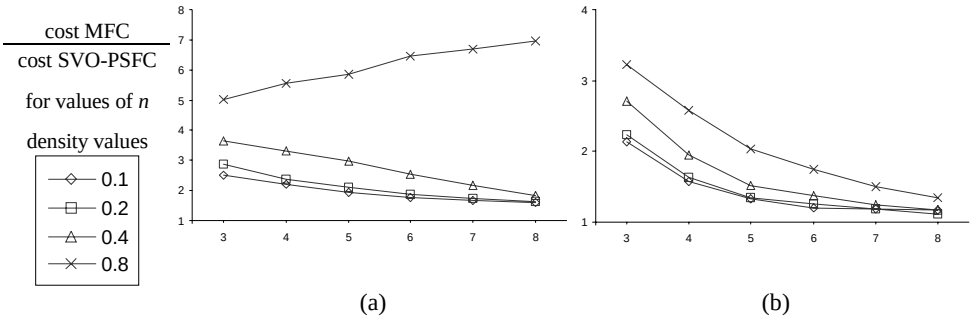


Fig. 15: Improvement factor for synthetic datasets, as a function of the number of inputs: (a) Chain queries. (b) Clique queries.

An alternative approach to PSFC would break each sub-problem (instead of calling sortFC), into smaller ones by recursively calling the main body of PSFC, until only two variables remain, where plane sweep can be applied. We have experimented with such a technique, but found it inferior to PSFC in most cases. We believe that this is due to the overhead of tracing the domains multiple times

in order to hopefully reduce them, without this to pay-off the high initialization constants at each recursive call of PSFC.

4.2 ST vs. Pairwise Spatial Join Algorithms

As discussed in Section 3.3, the data density affects little the CPU-time of ST (which is the main factor for the total cost). On the other hand, since density determines the size of intermediate results of pairwise algorithms, it is of major importance to the performance of PJM. The first experiment uses the synthetic datasets to unveil the effects of data density and number of inputs on relative performance of ST and PJM. Fig. 16 shows the CPU-time, page accesses and total cost of the optimal PJM plan divided by the corresponding numbers for ST in case of chain and clique queries. In other words, the diagram shows how many times ST outperforms (or, is outperformed by) PJM.

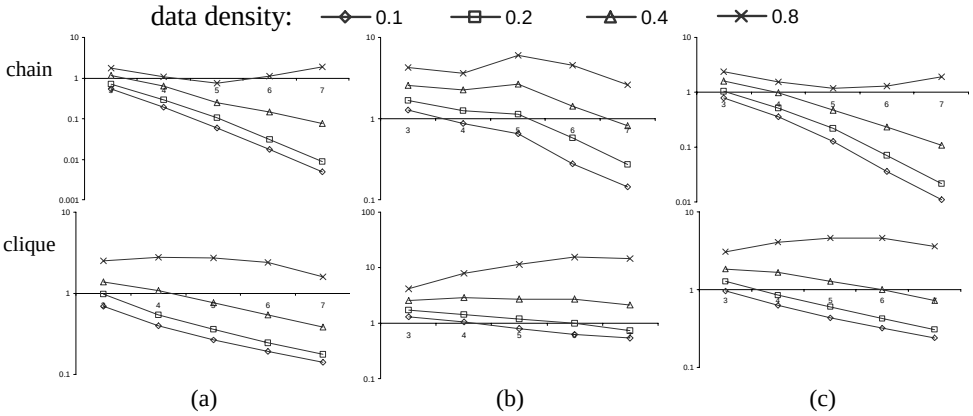


Fig. 16: Overall cost of PJM/ overall cost of ST as function of *n*: (a) CPU time. (b) I/O accesses. (c) Overall cost.

Dense datasets generate a large number of intermediate results and are best processed by ST. This is especially true for dense query topologies (e.g. cliques), where a significant percentage of solutions at the high tree levels lead to actual solutions. On the other hand, sparse datasets and queries do not produce many intermediate solutions and favor PJM. ST in this case suffers from the large number of false hits. It is worth noticing that in most cases ST is better in terms of I/O. For large numbers of variables, however, the CPU cost outweighs the I/O cost substantially and I/O savings do not pay off. We have also compared the relative efficiency of PJM and ST in processing multiway joins of real datasets using datasets T1-T4 and geographic layers of Germany. Due to the sparseness of the data, PJM was superior to ST in all query configurations (see Fig. 24). The difference is very small for the clique of T1-T4, because these datasets are relatively dense.

In the next experiment, we test the performance of the two alternatives when not all solutions are required. We ran the three queries of Fig. 1 for the T1-T4

datasets and terminated the algorithms after a percentage of the output size was found. The first row of Fig. 17 shows the CPU time and the second row the number of I/O accesses as a function of the percentage of solutions retrieved. ST has a significant advantage for small percentages since PJM requires a certain amount of time to compute the intermediate results before the final join. The difference grows with the number of variables. This feature of ST is important since during expensive queries users may browse through the results almost immediately.

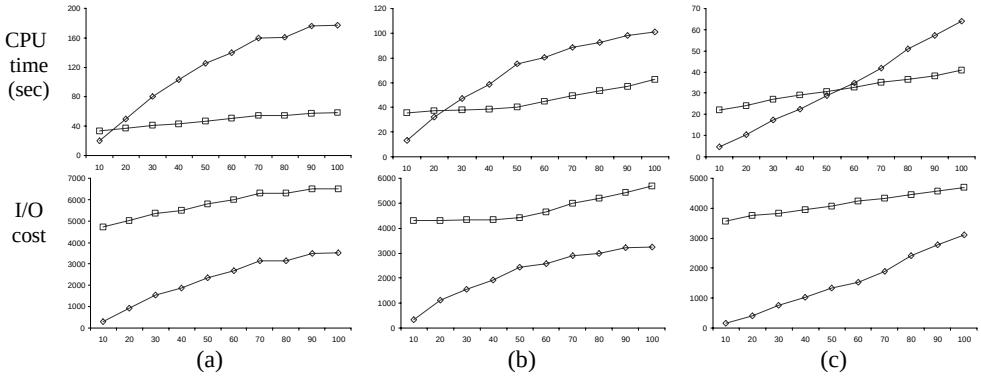


Fig. 17: CPU and I/O of ST and PJM as a function of percentage of solutions retrieved (for T1-T4 datasets): (a) Chain query. (b) Query with cycle. (c) Clique query.

The above experiments suggest that ST and PJM are complementary in the sense that they perform best under different conditions. The winner depends on the data and query densities, restrictions on system resources (CPU vs. I/O) and the percentage of required solutions. Therefore, an approach that combines the two methods could outperform both for real life situations (e.g., when the densities of the joined inputs vary significantly). In the next section we study the integration of ST with pairwise join algorithms.

5. COMBINING ST WITH PAIRWISE JOIN ALGORITHMS

Since ST is essentially a generalization of RJ it can be easily integrated with other pairwise join algorithms to effectively process complex spatial queries. We have implemented all algorithms (Table 3) as iterator functions [Graefe 1993] in an execution engine running on a centralized, uni-processor environment that applies pipelining. ST (RJ for two inputs) just 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 executes the join passing the results to the upper operator. HJ 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 hash buckets. Then the results from the intermediate file are read and hashed to the buckets. The results of the probe input are immediately hashed to buckets.

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 and return next tuple	close tree file; de-allocate slot index and hash buckets
HJ (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 the extents of the 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 and return next tuple	de-allocate hash buckets

Table 3: Iterator functions

Notice that in this implementation, the system buffer is shared between at most two operators. *Next* functions never run concurrently; when join is executed at one operator, only hashing is performed at the upper one. Thus, given a memory buffer of M pages, the operator which is currently performing a join uses M-S pages and the upper operator, which performs hashing, uses S pages, where S is the number of slots/buckets. In this way, the utilization of the memory buffer is maximized.

In the rest of the section we provide methods that estimate the selectivity of multiway spatial joins involving skewed inputs, describe an optimization method based on dynamic programming, and evaluate our proposal, by comparing the integration of ST and pairwise algorithms with each individual method.

5.1 Selectivity Estimation for Real Datasets

Accurate join selectivity estimation is essential for query optimization. In particular, optimization of multiway spatial joins requires two estimations: (i) the number of solutions at intermediate R-tree levels, in order to compute the cost of ST (ii) the output size of each possible query sub-graph. The formulae of Section 2.4, however, are meant for uniform data, and are not expected to be precise for real datasets. Real datasets do not necessarily cover a rectangular area, while in most cases their density varies significantly throughout the covered space.

Eq. 11 and 13 assume a square $[0,1)\times[0,1)$ workspace, and use the average area of the rectangles in the datasets (actually the square root of this quantity which is the average rectangle side) to estimate join selectivity. Even if the minimum and maximum coordinates of the rectangles are normalized to take values within this range, typically large parts of the square workspace are empty. Given a series of layers of the same region (e.g. rivers, streets, forests), we define as *real workspace* the total area covered by all layers. In order to estimate this area, we use a rectangular grid (bitmap), and each time a rectangle is inserted into a dataset, the intersected cells are marked. The real workspace is estimated by computing the area covered by the marked cells. This information is stored and maintained as meta-data. Fig. 18 illustrates four layers of Germany (G1-G4) and

the corresponding real workspace using a 50×50 bitmap. The *normalized average area* for the rectangles of a dataset (to be used in the above equations) is their average area divided by the real workspace. When there is dead space, the real workspace is less than 1 and the normalized average area is larger than the actual average area. The same method, called *workspace normalization* (WN), can be applied for the MBRs of intermediate R-tree levels to estimate the number of local problems solved by ST.

Germany utility network (17790, D=0.12) | Germany road lines (30674, D=0.08) | Germany railroad lines (36334, D=0.07) | Germany hypsography (76999, D=0.04) | real workspace

Fig. 18: Germany datasets and workspace.

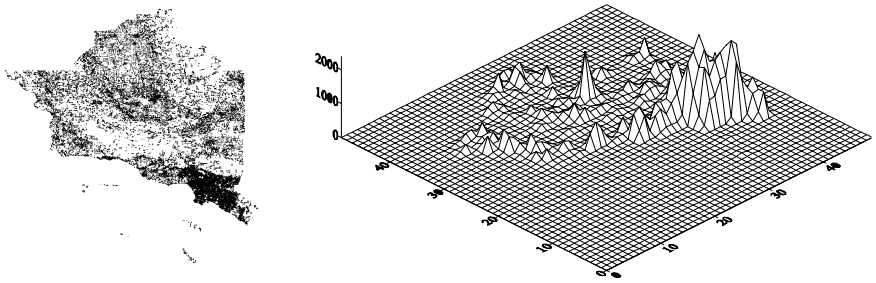
WN is expected to work well for (relatively) uniform data in some irregular workspace. Since intermediate nodes are not usually as skewed as the leaf rectangles, we may assume that they are uniformly distributed in the real workspace. Therefore WN can be used to predict the number of intermediate level solutions (which is needed for the estimation of local problems). In the next experiment we test the method by applying the queries of Fig. 1 using G1-G4 datasets. The first column in Table 4 illustrates the number solutions at the root of a two-level R-tree, while the second and third columns show the estimations before and after WN. The last row shows the average relative estimation error which is defined as |*estimated output* - *actual output*|/min{*actual output*, *estimated output*}. In other words this metric shows how many times the estimated result is larger or smaller than the actual. WN improves accuracy significantly; without normalization of the workspace, the join result is underestimated because the average area of the objects does not reflect their effects in the actual space they cover. By shrinking the workspace, the normalized area of the rectangles increases providing a better estimate for the join result.

Unlike intermediate nodes, real object rectangles can be highly skewed. To deal with this problem, previous work on relational databases [Mannino et al. 1998, Ioannidis and Poosala 1995] has considered the use of histograms. A histogram-based method that estimates the selectivity of range queries in two dimensional space is presented in [Achaya et al. 1999]. This method irregularly decomposes the space using histogram buckets that cover points of similar density. Obviously datasets with different distributions have different bucket extents and therefore they cannot be matched (at least in a trivial way) to estimate the result of a spatial join.

	Actual solutions	Without WN	WN
chain	9460	1438	11510
cycle	3842	414	3352
clique	832	294	2355
avg. Error	0	5.23	0.73

Table 4: Estimation of intermediate level solutions

In order to overcome this problem, we use a regular grid similar to [Theodoridis et al. 1998]. Each cell in the grid can be thought of as a histogram bucket that keeps information about the number of rectangles and the normalized average rectangle size. The criterion for assigning a rectangle to a cell is the enclosure of the rectangle's center, thus no rectangle is assigned to more than one cells[8]. The estimation of the join output size is then done for each cell and summing up the results, i.e. the contents of each cell are assumed to be uniformly distributed. Fig. 19a depicts the T1 dataset and 19b shows the number of rectangles per cell when a 50×50 grid is used. Observe that the south-east part of the dataset contains very dense data.



(a) (b)

Fig. 19: Skew in dataset T1: (a) T1 dataset. (b) Number of rectangles per cell in a 50x50 grid.

Table 5 presents the actual (column 1) and estimated output size of several join pairs using various grids. The average accuracy improves with the size of the grid used. Nevertheless, as the area of each cell decreases, the probability that a random rectangle intersects more than one cells increases introducing boundary effects; this is why in some cases accuracy drops with the grid size. Moreover, maintaining large grids is space demanding especially for large join queries where the number of plans is huge, and statistical information needs to be maintained for a high percentage of the sub-plans. For our experimental settings 50×50 grids provide reasonable accuracy and they can be efficiently manipulated in main memory.

[8] We have also experimented with a different strategy, where each rectangle is assigned to all grid cells that intersect it. However, this turned out to be less accurate resulting in a large overestimation of the results almost in all cases.

Table 6 shows the error propagation effects of the estimations for multiway joins. The output of the three example queries is estimated using datasets G1-G4 and various grids. As expected the error in estimates increases with the query size. The results produced without WN are clearly unacceptable, whereas the granularity of the grid increases the accuracy, as expected. Notice that histograms are applicable when the rectangles are much smaller than the cells, and boundary effects are negligible. Since the extents of intermediate tree nodes are large, the method cannot be applied to estimate the number of intermediate level solutions at ST.

Although after the employment of WN and histograms the error is still non-trivial, the methods provide significant improvement compared to estimations based on uniformity assumptions, which are inapplicable in most real-life cases. Similar problems exist in optimization of relational queries [Ioannidis and Christodoulakis 1991]; however, the actual goal of optimization is not to find the best plan, but rather, to avoid expensive ones. In the future, we plan to investigate the application of methods that apply irregular space decomposition (e.g., [Achaya et al. 1999]) for more accurate selectivity estimation of spatial joins.

	Actual output	**without WN**	**WN**	**20×20**	**50×50**	**100×100**
T1 ⋈ T2	86094	53228	94084	69475	78312	83706
G1 ⋈ G2	12888	7779	9625	10814	12098	13481
G1 ⋈ G3	12965	7797	10342	11140	12419	13802
G1 ⋈ G4	14012	11668	15098	16813	16962	17372
G2 ⋈ G3	20518	8151	10050	11483	13514	15534
G2 ⋈ G4	13435	10914	12712	16312	16392	16609
G3 ⋈ G4	12672	9545	12573	14792	14993	15621
avg. error	0	0.60	0.26	0.28	0.19	0.17

Table 5: Output size estimation of pairwise spatial joins using grids.

	actual output	**without WN**	**WN**	**20×20**	**50×50**	**100×100**
chain	7007	426	1089	1878	2390	3961
cycle	1470	144	354	697	846	1243
clique	832	117	303	541	681	1003
avg. error	0	10.25	3.44	1.46	0.96	0.38

Table 6: Output size estimation of multiway spatial joins using grids.

5.2 Optimization with Dynamic Programming

Dynamic programming (DP) is the standard technique for relational query optimization. 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 (Fig. 20) finds the best decomposition of Q_i to two connected components, based on the optimal cost of executing these components and their sizes. 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, HJ 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. The I/O cost of pairwise algorithms is given in Section 2. The CPU-time of ST is estimated using Eq. 15. The two costs are transformed to the same scale and compared to determine the best alternative.

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. Due to the bottom-up computation of the optimal plans, the cost and size for a specific query sub-graph is computed only once. The price to pay concerns the storage requirements of the algorithm, which are manageable for moderate query graphs. The execution cost of dynamic programming depends on (i) the number of relations n, (ii) the number of valid node 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 7 illustrates the above parameters for three special cases of join graphs. Notice that combinations of 2 nodes do not have valid decompositions because the they can be processed only by RJ.

```
DP(Query Q, int n) { /*n = number of inputs*/
  for each connected sub-graph Q2 ∈ Q of size 2 do {
    Q2.cost := CRJ(A, B); /*Eq. 1*/
    Q2.size := |A ⋈ B|; /*Eq. 8*/
  }
  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 := CST(Qi); /*Eq. 15)*/
      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 + CSISJ(Qk, Qi-k); /*Eq. 7*/
        else /*both components are sub-plans; HJ will be used*/
          {Qk, Qi-k}.cost := Qk.cost + Qi-k.cost + CHJ(Qk, Qi-k); /*Eq.4*/
        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 := |Qi.plan|;
    }
  }
}
```

Fig. 20: Dynamic programming for optimization of multiway spatial joins.

	clique	chain	Star
$comb_k$	C_n^k	n-k+1	$\begin{cases} n, & k=1 \\ C_{n-1}^{k-1}, & \text{otherwise} \end{cases}$
$decomp_k$	$\begin{cases} 0, & 1 \le k \le 2 \\ k + \sum_{2 \le i < k-1} C_k^i, & \text{otherwise} \end{cases}$	$\begin{cases} 0, & 1 \le k \le 2 \\ 2, & \text{otherwise} \end{cases}$	$\begin{cases} 0, & 1 \le k \le 2 \\ k-1, & \text{otherwise} \end{cases}$

Table 7: Number of plans and optimization cost parameters for different query graphs.

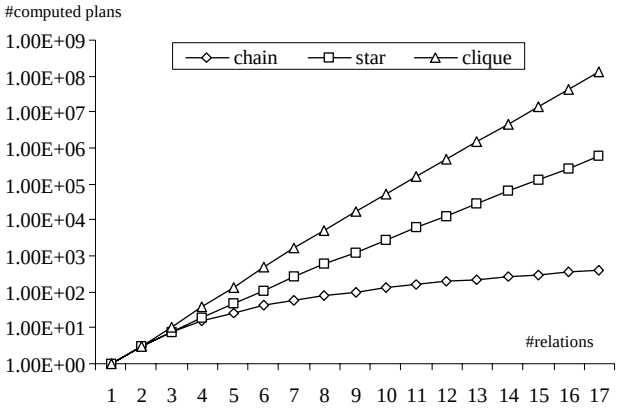


Figure 21: Cost of dynamic programming (in terms of computed plans) as a function of *n*.

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:

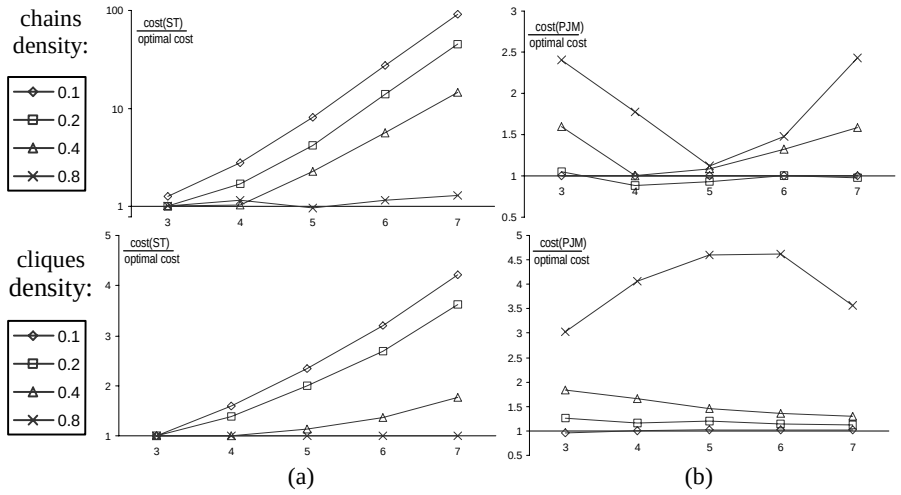
$$C_{DP} = \sum_{1 \le k \le n} comb_k \cdot (1 + decomp_k) \qquad (17)$$

Fig. 21 demonstrates the cost of dynamic programming (in terms of computed plans) as a function of *n*, for the three types of queries. The cost of DP for most other topologies is expected to be between those for the clique and star graphs with the same number of inputs.

5.3 Experimental Evaluation

In order to identify the form and quality of the plans suggested by DP, we first experimented using the synthetic datasets. Fig. 22 presents the improvement factor of the optimal plan over the best PJM plan and the pure ST plan, as a function of the density, the number of inputs and the query topology (cliques and chains). The proposed plan was right-deep (1 application of RJ, and *n*-2 of SISJ) for chain queries over 0.1 density files, and pure ST for cliques over 0.8 density files. In all other cases, a combination of ST and PJM was suggested. Observe that in some cases the proposed plan was not the optimal, and this was due to

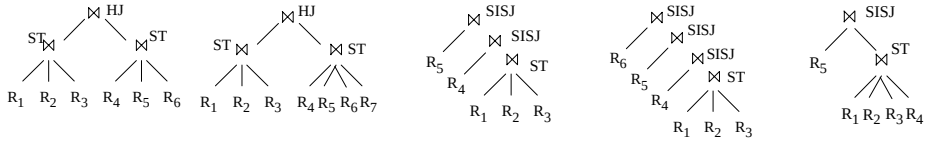
errors in join size estimations. However, these plans were only slightly more expensive than the actual optimal. For instance in the worst case (chain query over four 0.2 density datasets), the suggested plan is ST for the first three inputs and SISJ for the final join, which is only 12% worse than the actual optimal (right-deep) PJM plan. The cost of DP was very small for the tested queries compared to the evaluation cost of the optimal plan. Consider, for instance, cliques of 7 inputs (the most expensive queries to optimize in this experiment). The optimization cost is around 5% of the execution time for the 0.1 density datasets (cheapest to process) and a mere 0.8% for the 0.8 density datasets (most expensive to process).



(a) (b)

Fig. 22: Improvement factor of optimal plans as a function of *n*: (a) Performance gain over ST. (b) Performance gain over PJM.

Fig. 23 presents some representative plans proposed by the algorithm. In general, "bushy" combinations of ST sub-plans (Fig. 23a, 23b) are better than deep plans for chain queries over dense datasets, where the intermediate results are large. They are also better than pure ST because of the degradation of ST performance for large chain queries. Deep plans (Fig. 23c, 23d) are best for chain and clique queries of sparse datasets and for some cliques of dense datasets, because the size of intermediate results is not large and SISJ can be used efficiently. For clique queries over dense datasets, a ST sub-plan with a large number of inputs (Fig. 23e) is better than combinations of small ST plans.



(a) chain D=0.4 (b) chain D=0.8 (c) chain D=0.2 (d) clique D=0.1 (e) clique D=0.4

Fig. 23: Representative optimal plans

The effectiveness of optimization was also tested with queries over real datasets. We applied various queries to the Tiger (T1-T4) and Germany (G1-G4) datasets, and compared the suggested optimal plans with alternative ones. Fig. 24 illustrates some execution examples. The numbers under each plan show its overall cost (in seconds). In most cases the best plan was a hybrid one. This shows that ST is a valuable component of a query execution engine.

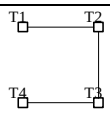
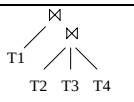
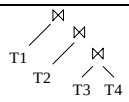
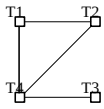
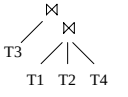
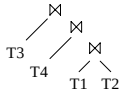
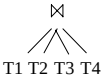
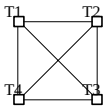
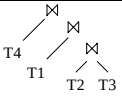
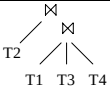
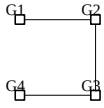
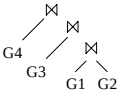
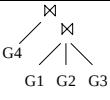
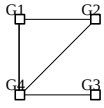
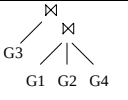
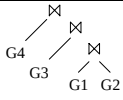
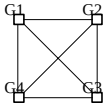
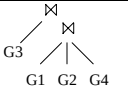
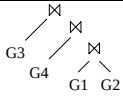
Query	*optimal plan*	*alternative plan*	*ST*
	 122.1	 124.6	 212.2
	 102.5	 118.9	 133.2
	 89.2	 93.1	 95.2
	 17.6	 19.65	 39.38
	 14.1	 17.6	 25.5
	 14.1	 14.5	 18.4

Fig. 24: Overall cost (in seconds) of plans that include real datasets

Notice that the cost differences among alternatives are not very large due to the small number of inputs and because whenever the best plan is a hybrid one, the second column gives the optimal PJM plan. Therefore, the table shows also the relative performance of ST compared to PJM for the real datasets used in this paper. As we show in the next section, for many inputs and arbitrary plans, the difference between different plans can be orders of magnitude. For more than 10 inputs, however, dynamic programming is inapplicable due to its very high space and time requirements. Such queries may be imposed in GIS (e.g., overlays of multiple geographical layers) or VLSI/CAD applications (e.g., configurations of numerous components). In the sequel we study the application of randomized search algorithms for the optimization of multiway spatial joins involving numerous datasets.

6. OPTIMIZATION OF LARGE SPATIAL QUERIES

Randomized search heuristics have been applied to optimize large relational joins. [Swami and Gupta 1988, Ioannidis and Kang 1990, Galindo-Legaria et al. 1994]Such algorithms search in the space of alternative plans trying to find good, but possibly sub-optimal plans, within limited time. The search space of query optimization can be thought of as an undirected graph, where nodes (also called *states*) correspond to valid execution plans. Two plans are connected through an edge, if each can be derived from the other by a simple *transformation*. In case of relational joins, transformation rules apply join commutativity and associativity [Swami and Gupta 1988, Ioannidis and Kang 1990]; in multiway spatial join queries, the implication of ST calls for the definition of special rules. In the rest of this section, we study the search space for different query topologies, propose heuristic algorithms that quickly identify good plans by searching only a small part of the space, and evaluate their effectiveness for several experimental settings.

6.1 Space of execution plans

In case of pairwise algorithms, the maximum number of join orders is (2(n-1))! / (n-1)! [Silberschatz et al. 1997], i.e., the possible permutations of all complete binary trees. If ST is included as a processing method this number is higher, since join orders are not necessarily binary trees, but multiple inputs to be processed by ST are represented as leaves with a common father node. A join order may correspond to multiple plans that differ in the join processing method, when more than one methods exist for each type of join. For the current problem each valid join order generates exactly one plan because the processing method is determined by whether the inputs are indexed or not. Furthermore, some plans are computationally equivalent, since order is not important for ST and SISJ.

Let P_n denote the number of distinct plans (valid plans that are not equivalent) involving n inputs. In case of clique queries, P_n is given by the following formula:

$$P_n = 1 + n \cdot P_{n-1} + \sum_{2 \le k < n-1} C_n^k \cdot P_k P_{n-k} \tag{18}$$

where C_n^k denotes the combinations of k out of n objects. In other words, a query involving n variables can be processed in three ways: (i) by ST (1 plan), (ii) by choosing an input which will be joined with the result of the remaining ones using SISJ ($n \cdot P_{n-1}$ plans), or (iii) by breaking the graph in all combinations of two smaller ones (with at least 2 inputs each) and then joining them with HJ.

The number of valid plans is smaller for arbitrary queries since some plans correspond to Cartesian products and can be avoided. Chain queries have the minimum P_n, which is defined as follows:

$$P_n = 1 + 2 \cdot P_{n-1} + \sum_{2 \le k < n-1} P_k P_{n-k} \tag{19}$$

In this case only the first or the last input can be chosen to generate an SISJ join at the top level ($2 \cdot P_{n-1}$). If any other input is used, the remaining sub-graph will consist of two disconnected components (joining them requires a Cartesian product). Similarly when decomposing in plans to be joined by HJ, for each value of *k* there is only one valid plan, i.e., the one where the first part contains the first *k* inputs and the second part the remaining *n-k* ones.

The same methodology can be employed to compute the possible plans for several graph topologies; e.g., for *star* queries all (*n-1*) but the central node can be chosen for the generation of an SISJ plan. On the other hand, no HJ sub-plan can be applied since any decomposition in two disjoint sub-graphs with more than one node implies that one of the two sub-graphs does not contain the central node (therefore each of its nodes is a disconnected component). Thus, for star queries the number of distinct plans is:

$$P_n = 1 + (n-1) \cdot P_{n-1} \tag{20}$$

Fig. 25 illustrates the number of distinct plans as a function of the number of inputs for chain, star and clique queries (computed by Eq. 18, 19, and 20 respectively). In general, the number of plans increases with the query graph density since every added edge in a graph validates some plans.

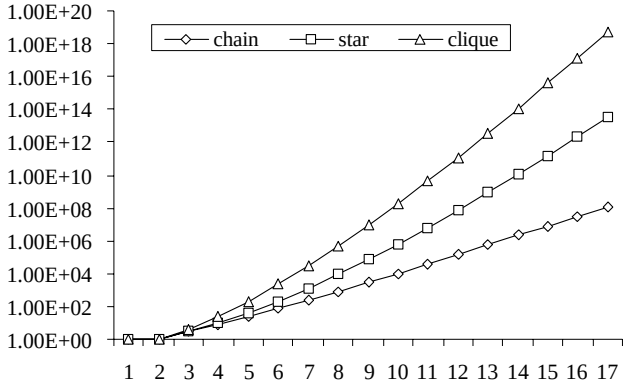


Figure 25: Number of potential plans as a function of *n.*

Several sets of transformations can be applied to link plans in the search space. The transformation rules should generate a connected graph, i.e., there should be at least one path between any two states. Different sets of rules have different properties; adding redundant rules, for example, produces direct paths between several plans, bypassing intermediate nodes. For our problem we use the set of rules shown in Fig. 26. R denotes spatial relations (leaf nodes of the plan), J denotes intermediate results (non-leaf nodes), and X denotes any of the previous cases.

Rules 1, 2 and 3 have also been used in relational query optimization [Swami and Gupta 1988, Ioannidis and Kang 1990]. Commutativity is applied only in the case of HJ, because swapping the sub-plans of a node results in an equivalent plan for all other methods. Rules 4 and 5 are special for the case of ST. Rule 4 breaks or

composes an ST plan, while rule 5 changes the inputs of ST without going through composition/decomposition. Rules 3 and 5 are redundant; they are applied to escape faster from areas with similar cost.

1. commutativity (HJ only)
   $J_1 \bowtie_{HJ} J_2 \leftrightarrow J_2 \bowtie_{HJ} J_1$
2. associativity
   $(X_1 \bowtie X_2) \bowtie X_3 \leftrightarrow X_1 \bowtie (X_2 \bowtie X_3)$
3. left/right pairwise join exchange
   $(X_1 \bowtie X_2) \bowtie X_3 \leftrightarrow (X_1 \bowtie X_3) \bowtie X_2$
   $X_1 \bowtie (X_2 \bowtie X_3) \leftrightarrow X_2 \bowtie (X_1 \bowtie X_3)$
4. ST composition/decomposition
   $ST(R_1, ..., R_{i-1}, R_i, R_{i+1}, ...,R_n) \leftrightarrow R_i \bowtie_{SISJ} ST(R_1, ..., R_{i-1}, R_{i+1}, ...,R_n)$, $1 \le i \le n$
   $ST(R_1, ..., R_{i-1}, R_i, R_{i+1}, ...,R_n) \leftrightarrow ST(R_1, ..., R_{i-1}, R_i) \bowtie_{HJ} ST(R_{i+1}, ..., R_n)$, $2 \le i \le n-2$
5. ST element exchange
   $R_L \bowtie_{SISJ} ST(R_1, R_2, ..., R_i, ...,R_n) \leftrightarrow R_i \bowtie_{SISJ} ST(R_1, R_2, ..., R_L, ...,R_n)$, $1 \le i \le n$
   $ST(R_{L1}, ..., R_{Li}, ...,R_{Ln}) \bowtie_{HJ} ST(R_{R1}, ..., R_{Rj}, ...,R_{Rm}) \leftrightarrow$
   $ST(R_{L1}, ..., R_{Rj}, ...,R_{Ln}) \bowtie_{HJ} ST(R_{R1}, ..., R_{Li}, ...,R_{Rm})$, $1 \le i \le n$, $1 \le j \le m$

Fig. 26: Transformation rules for multiway spatial join plans.

6.2 Randomized Search Algorithms

Randomized search algorithms traverse the space of alternative plans by performing *moves* (or *transitions*) from one state to another by applying one of the allowed transformations at random.[9] A simple, yet efficient heuristic is iterative improvement (II). [Nahar et al. 1986]Starting from a random initial state (called *seed*), it performs a random series of moves and accepts all downhill ones, until a local minimum is detected. This process is repeated until a time limit is reached, each time with a different seed. The pseudo-code of the algorithm is given in Fig. 27. Since the number of neighbors of a state may be large and the algorithm needs to retain its randomness, a state is conventionally considered as local minimum if a long sequence of consecutive uphill moves are attempted from it.

Simulated annealing [Kirkpatrick et al. 1983] follows a procedure similar to II, but it also accepts uphill moves with some probability. This probability is gradually decreased with time and finally the algorithm accepts only downhill moves leading to a good local minimum. The intuition behind accepting uphill moves is led by the fact that some local minima may be close to each other, separated by a small number of uphill moves. If only downhill moves were

[9] First a valid transformation rule from Fig. 26 is chosen at random. Then a valid application of the transformation is chosen at random. For instance, assume that the current state is a right-deep plan $R_1 \bowtie_{SISJ} ST(R_2,R_3,R_4)$. One of the valid rules 4,5 is chosen at random, e.g., rule 5. Then, one of the applications of rule 5 is chosen at random, e.g., exchange of R_1 with R_3 and the resulting plan is $R_3 \bowtie_{SISJ} ST(R_2,R_1,R_4)$.

accepted (as in II) the algorithm would stop at the first local minimum visited, missing a subsequent (and possibly better) one. The pseudo-code of SA is given in Fig. 28.

In order to comply with the original version of SA, we use the same terminology. The initial temperature T_0 corresponds to a (usually high) probability of accepting an uphill move. The algorithm tries a number of moves (inner iterations) for each temperature value T, which gradually decreases, allowing SA to accept uphill moves less frequently. When the temperature is small enough, the probability of accepting an uphill move converges to zero and SA behaves like II. The "system" is then said to be *frozen* and the algorithm terminates. Notice that since an uphill move increases the cost, the probability to accept it should be associated to the cost difference. The number of inner iterations for each value of T is typically proportional to the size of the problem.

```
II {
   mins := NULL; /* the global minimum */
   while time limit is not reached do {
      pick a random state s; /*pick seed */
      while local minimum not reached do {
         s′ := move(s); /* go to a random neighbor state */
         if cost(s′) < cost(s) then
            s := s′;
      }
      if cost(mins) > cost(s) then
         mins := s;
   }
}
```

Fig. 27: Iterative Improvement.

```
SA {
   pick a random state s; /*pick seed */
   mins := s; /* initialize the global minimum */
   T := T0; /* initialize temperature */
   while "system not frozen" do {
      for a number of inner iterations do{
         s′ := move(s); /* go to a random neighbor state */
         if  cost(s′)≤cost(s) then s := s′;
         else {
            ΔC := cost(s)-cost(s′);
            s := s′ with probability e^-ΔC/T;
         }
         if  cost(mins) > cost(s) then mins := s;
      }
      reduce(T);
   }
}
```

Fig. 28: Simulated Annealing.

Another alternative is random sampling (RA). Given a time limit, sampling selects plans at random and returns the one with the minimum cost. In general, sampling is efficient when the local minima are not much worse than a random state. The experiments for relational query optimization in [Galindo-Legaria et al. 1994] suggest that sampling often converges to an acceptable solution faster than II and SA. This shows that RA can be a useful approach when very limited time is given for searching, e.g. when verifying the cost of one state is very expensive. In most problems, the other two techniques find better solutions when they are given enough time to search. II works well if there are many "deep" local minima, the quality of which does not differ much, whereas SA is better when there are large cost differences between local minima connected by a few uphill moves. In the latter case a hybrid method, *two-phase optimization* (2PO) [Ioannidis and Kang 1990], that combines II and SA can be more efficient than both algorithms. This method uses II to quickly locate an area where many local minima exist, and then applies SA with a small initial temperature to search for the global minimum in this area. Thus, 2PO avoids unnecessary large uphill moves at the early stages of SA and the non-acceptance of uphill moves in II.

6.3 Experimental Evaluation

We have implemented RA, II, SA and 2PO using the set of transformation rules in Fig. 26. The parameters for II, SA and 2PO, summarized in Table 8, were fine-tuned through several experiments (the presentation of these experiments is out of the scope of this paper) so that algorithms had small running cost (up to 10% compared to the execution time of the produced plans) while providing plans of good quality. The performance of the algorithms was tested only for clique query graphs, since they are the most expensive to optimize. We experimented with joins involving from 10 to 50 relations. The datasets in each query have between 10000 and 100000 uniformly distributed rectangles, in order to produce problems with a large variety of costs in the space of possible plans.

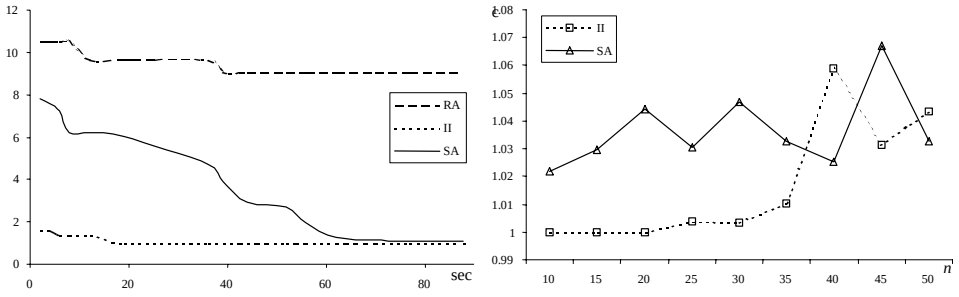
II	time limit: equal time to SA local minimum condition: $10 \cdot n$ consecutive uphill moves
SA	$T_0 = 2 \cdot$(random plan cost) frozen: $T<1$ and mins unchanged for the last four values of T inner_iterations: $20 \cdot n$ reduce(T): $T := 0.95 \cdot T$
2PO	initial state: best solution after 5 local optimizations of II $T_0 = 2 \cdot$(initial state cost)

Table 8: Parameters of II and SA.

In the first set of experiments the densities of all datasets randomly range between 0.1 and 0.5. Fig. 29a illustrates the cost of the best plan found by RA, II and SA

over time (in seconds) for a query with 30 inputs divided by the cost of the minimum cost found. The corresponding cost of 2PO coincides with that of II, i.e., 2PO did not converge to a solution better than its seed (generated by II), and therefore we did not include it in the diagrams. The behavior of the algorithms for this query is typical for all tested cases. RA has very low performance implying that random states are much more expensive than local minima (notice that the best plan found is 10 times worse than the one produced by the other algorithms). II manages to find fast a good local minimum, whereas SA wanders around high cost states until it converges to a local minimum. In contrast to the results in [Ioannidis and Kang 1990], for this experimental setting SA does not typically converge to a better solution than the one found by II. Both algorithms find plans with very similar costs, indicating that there are no great differences between most of the local minima which are scattered in the search space.

The above observation is also verified in the following experiment. We generated 10 queries for each of several values of *n* ranging from 10 to 50, and executed II and SA. II was left to run until the convergence point of SA. Fig. 29b shows the average scaled cost of the algorithms[10] as a function of the number of inputs. II found a plan better than SA in most cases, but the costs between the plans did not differ much. On the other hand, in the few exceptions that SA found the best plan, it was much cheaper than the one found by II. This explains the anomaly for queries of 40 relations; in one out of the ten queries, SA generated a plan which was about 60% better than II. In general II, is the best choice for sparse datasets, since in addition to achieving better quality plans, it requires less time to search.



(a) (b)

Fig. 29: Performance of randomized search algorithms for large joins of low-density datasets: (a) Scaled cost of plans over time for a 30-relation query. (b) Average scaled cost of the produced plan.

Fig. 30 shows the average time required by the algorithms to find the best solution as a function of the number of inputs. In addition to the search space, this depends on the implementation parameters: (i) for SA (and 2PO) T_0, number of inner iterations, freezing condition and temperature reduction (ii) for II, local minimum condition. II is the only alternative when limited time is available for optimization since its cost does not grow significantly with query size.

[10] The *scaled cost* of a plan is defined as the cost of the plan divided by the cost of the best plan found for the specific query [Swami and Gupta 1988].

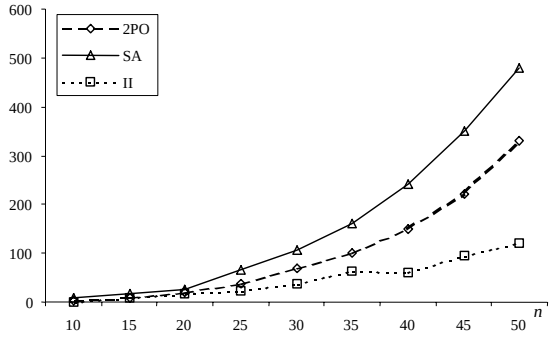


Fig. 30: Average time point (in seconds) where the algorithms find the best plan

As the query size grows, the output shrinks and large cliques do not have any solutions for low density datasets (recall Fig. 11). Such queries produce small intermediate results for the tested dataset densities, which are best handled by SISJ; for this reason the optimal plan is almost always deep involving only a chain of SISJs and an ST of 2 to 4 relations (e.g., see plans for clique queries in Fig. 23d, 23e). In order to test the efficiency of optimization in different cases, in the next experiment we tuned the density of the datasets to be between 0.5 and 1. This caused the intermediate results to be large and the optimal plans to be shallower (bushy plans involving HJ).

Fig. 31a illustrates the performance of the algorithms over time for a typical 40-relation query. Interestingly, in this case the quality of the plans produced by SA was better than the ones found by II. The same is true for 2PO, which converges to a good solution much faster than SA. We also performed an experiment by varying the number of inputs (similar to the one in Fig. 29b). As shown in Fig. 31b, the difference in the quality of plans produced by SA (and 2PO) with respect to II grows with the number of inputs. SA and 2PO find plans of similar cost, but 2PO should be preferred because of its speed. The effectiveness of RA is worse than for low density queries, therefore we omitted it from the comparison.

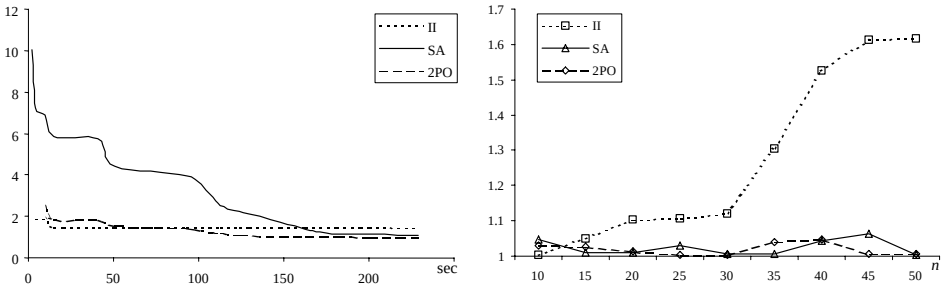


Fig. 31: Behavior of the algorithms for high density datasets. (a) Scaled cost of plans over time for a 40-relation query. (b) Average scaled cost of the produced plan.

We have also compared the cost of the plans computed by the randomized search algorithms to the cost of the optimal plan (produced by DP), for joins of up to 15

datasets where DP finishes within reasonable time (i.e., around 10 minutes). We observed that the plans produced by II, SA, and 2PO were only slightly more expensive than the optimal one (at most 10% and typically around 3% for 10-15 inputs). This indicates that there exist numerous plans in the optimization space with cost similar to the optimal and it is worth applying randomized search for large problems.

Concluding, when the number of solutions is large, i.e. the intermediate results are large, the optimal plan is harder to find, since it is expected to be shallow and the bushy plans are many more than the right deep ones. In this case, 2PO is the best optimization method because it converges faster than SA to a plan typically 50% cheaper than the one found by II. On the other hand, when the query output is very small the optimal plan is expected to be right deep and II manages to find a good plan much faster than SA. These observations are consistent with experiments in relational query optimization. Swami and Gupta [Swami and Gupta 1988], which consider only left-deep plans, conclude that II is the best method for optimization of large join queries. The consideration of bushy plans in [Ioannidis and Kang 1990] causes the search space to explode and SA (and 2PO) to become more effective than II.

7. CONCLUSIONS

This paper contains several significant contributions to spatial query processing. It surveys existing pairwise spatial join algorithms, analyzes the cost of three representative methods and discusses their application in processing queries with multiple inputs. Although PJM is a natural way to process multiway spatial joins, it has the disadvantage of generating temporary intermediate results between blocking operators. Therefore, we also explore the application of an R-tree traversal algorithm (ST) that processes all spatial inputs synchronously and computes the join without producing intermediate results. We propose two methods that enhance the performance of ST; a *static variable ordering* heuristic which optimizes the order in which the join inputs are considered, and a combination of *plane sweep* and *forward checking* that efficiently finds qualifying node combinations. An empirical analysis suggests that the cost of solving a local problem in ST is analogous to the number of variables and page size and independent of the data density and query graph. Based on this observation, we provide a formula that accurately estimates its computational cost.

A comparison of ST with pairwise join algorithms indicates that they perform best under different problem characteristics. High data and query densities favor ST, whereas PJM is the best choice for sparse datasets and query graphs. Numerous inputs usually favor pairwise algorithms, especially for chain queries, due to the explosion of the solutions at high level nodes during ST. Consequently, the combination of ST with pairwise join algorithms, using dynamic programming to compute the best execution plan, outperforms both individual approaches in most case. In order to optimize very large queries, we adapt two

randomized search methods, namely iterative improvement (II) and simulated annealing (SA), as well as a hybrid two phase optimization method (2PO) for the current problem.

Several future directions could enhance the current work. One very important issue refers to complex spatial query processing involving several spatial and possibly non-spatial operations. Consider for instance the query "find all cities within 200 km of Hong Kong *crossed by* a river which *intersects* an industrial area". This corresponds to a combination of spatial selection and joins, which gives rise to a variety of alternative processing methods. The choice depends on selectivities, data properties and underlying spatial indexes. The situation becomes even more complicated when there also exist non-spatial conditions (e.g., the population of a city must be over a million) since non-spatial indexes must be taken into account. Despite the importance of efficient spatial information processing in many applications, currently there does not exist any system that performs query optimization involving several operations. In [Mamoulis and Papadias 2001] we have proposed selectivity estimation formulae that can be used by query optimizers to estimate the cost of such complex queries. We are currently working towards defining composite operators that process spatial joins and spatial selections simultaneously.

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 [Park et al. 1999]. Going back to the example query, assume that we know that there are only 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. A similar situation applies for non-spatial selections where if the selectivity is high it may be better to fetch actual object records and check if they satisfy the selection conditions before executing subsequent joins. On the other hand, low selectivities can result in significant overhead in both cases since fetching records and checking for intersections between actual objects may be expensive. The execution plan should be determined using information about selectivities at the actual object level. This, however, requires appropriate analytical formulae for output size estimation using arbitrary shapes, which is a difficult problem.

The processing of multiway spatial joins can be enhanced by faster algorithms and optimization techniques. An *indirect predicate heuristic [Park et al. 1999a]*, that detects false hits at the intermediate tree levels, can be used by ST to further improve performance. More pairwise algorithms can be included in the execution engine in order to be applied in cases where they are expected to be more efficient than the current ones. Index nested loops, for example, may be more efficient than SISJ when the intermediate result is very small. Topics not covered, but well worth studying, include alternative forms of joins (e.g., distance joins [Hjaltason and Samet 1998]), inclusion of multiple spatial data structures (e.g., quadtrees for raster and R-trees for vector data [Corral et al. 1999]), parallel

processing issues (e.g., similar to the study of [Brinkhoff et al. 1996] for RJ), and optimal memory sharing between cascading operators.[Bouganim et al. 1998]

ACKNOWLEDGMENTS

This research was conducted while Nikos Mamoulis was with the Hong Kong University of Science and Technology. The authors were supported by grants HKUST 6070/00E and HKUST 6090/99E from Hong Kong RGC.

REFERENCES

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., VITTER, J.S. 1998. Scalable Sweeping-Based Spatial Join. *In Proceedings of the VLDB Conference*, August 1998, 570-581.

ACHAYA, S., POOSALA, V., RAMASWAMY, S. 1999. Selectivity Estimation in Spatial Databases. *In Proceedings of the ACM SIGMOD Conference*, June 1999, 13-24.

BIALLY, T. 1969. Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. *IEEE Transactions on Information Theory*, no 15, vol. 6, 1969, 658-664.

BUREAU OF THE CENSUS 1989. *Tiger/Line Precensus Files: 1990 Technical Documentation*. Washington, DC, 1989

BACCHUS, F., GROVE, A. 1995. On the Forward Checking Algorithm. *In Proceedings of the Conference on Principles and Practice of Constraint Programming (CP)*, September 1995, 292-308.

BRINKHOFF, T., KRIEGEL, H.P., SEEGER, B. 1993. Efficient Processing of Spatial Joins Using R-trees. *In Proceedings of the ACM SIGMOD Conference*, May 1993, 237-246.

BRINKHOFF T., KRIEGEL, H.P., SEEGER, B. 1996. Parallel Processing of Spatial Joins Using R-trees. *In Proceedings of the International Conference on Data Engineering (ICDE)*, March 1996, 258-265.

BECKMANN, N., KRIEGEL, H.P. SCHNEIDER, R., SEEGER, B. 1990. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *In Proceedings of the ACM SIGMOD Conference*, May 1990, 322-331.

BOUGANIM, L., KAPITSKAIA, O., VALDURIEZ, P. 1998. Memory Adaptive Scheduling for Large Query Execution. *In Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, May 1998, 105-115.

CORRAL, A., VASSILAKOPOULOS, M., MANOLOPOULOS, 1999. Y. Algorithms for Joining R-trees with Linear Region Quadtrees. *In Proceedings of the Symposium on Large Spatial Databases (SSD)*, July 1999, 251-269.

DECHTER, R., MEIRI, I. 1994. Experimental Evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, no 68, vol. 2, 1994, 211-241.

GUTTMAN, A. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. *In Proceedings of the ACM SIGMOD Conference*, June 1984, 47-57.

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

GÜNTHER, O. 1993. Efficient Computation of Spatial Joins. *In Proceedings of the International Conference on Data Engineering (ICDE)*, April 1993, 50-59.

GÜTING, R.H. 1994. An Introduction to Spatial Database Systems. *VLDB Journal*, no 3, vol. 4, 1994, 357-399.

GAEDE, V., GÜNTHER, O. 1998. Multidimensional Access Methods. *ACM Computing Surveys*, no 30, vol. 2, 1998, 123-169.

GALINDO-LEGARIA, C., PELLENKOFT, A., KERSTEN, M. 1994. Fast, Randomized Join-Order Selection - Why Use Transformations?. *In Proceedings of the VLDB Conference*, September 1994, 85-95.

HARALICK, R., ELLIOTT, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, no 14, 1980, 263-313.

HUANG, Y.W., JING, N., RUNDENSTEINER, E. 1997. Spatial Joins using R-trees: Breadth First Traversal with Global Optimizations. *In Proceedings of the VLDB Conference*, August 1997, 395-405.

HUANG, Y.W., JING, N., RUNDENSTEINER, E. 1997. A Cost Model for Estimating the Performance of Spatial Joins Using R-trees. *SSDBM*, August 1997, 30-38.

HOEL, E.G., SAMET, H. 1995. Benchmarking Spatial Join Operations with Spatial Output. *In Proceedings of the VLDB Conference*, September 1995, 606-618.

HJALTASON, G., SAMET, H. 1998. Incremental Distance Join Algorithms for Spatial Databases. *In Proceedings of the ACM SIGMOD Conference*, June 1998, 237-248.

IOANNIDIS, Y., CHRISTODOULAKIS, S. 1991. On the Propagation of Errors in the Size of Join Results. *In Proceedings of the ACM SIGMOD Conference*, May 1991, 268-277.

IOANNIDIS, Y., KANG, Y. 1990. Randomized Algorithms for Optimizing Large Join Queries. *In Proceedings of the ACM SIGMOD Conference*, May 1990, 312-321.

IOANNIDIS Y., POOSALA, V. 1995. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *In Proceedings of the ACM SIGMOD Conference*, May 1995, 233-244.

KIRKPATRICK, S., GELAT, C., VECCHI M. 1983. Optimization by Simulated Annealing. *Science*, no 220, 1983, 671-680.

KOUDAS, N., SEVCIK, K. 1997. Size Separation Spatial Join. *In Proceedings of the ACM SIGMOD Conference*, May 1997, 324-335.

KONDRAK Q., VAN BEEK, P. 1997. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, no 89, 1997, 365-387.

LO, M-L., RAVISHANKAR, C.V. 1994. Spatial Joins Using Seeded Trees. *In Proceedings of the ACM SIGMOD Conference*, May 1994, 209-220.

LO, M-L., RAVISHANKAR, C.V. 1996. Spatial Hash-Joins. *In Proceedings of the ACM SIGMOD Conference*, June 1996, 247-258.

MANNINO, M., CHU, P., SAGER, T. 1988. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, no 20, vol. 3, 1988, 192-221.

MAMOULIS, N., PAPADIAS, D., 1999. Integration of Spatial Join Algorithms for Processing Multiple Inputs. *In Proceedings of the ACM SIGMOD Conference*, June 1999, 1-12.

MAMOULIS, N., PAPADIAS, D. 2001. Slot Index Spatial Join. *To appear in IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

MAMOULIS, N., PAPADIAS, D. 2001. Selectivity Estimation of Complex Spatial Queries. *In Proceedings of the Symposium on Large Spatial and Temporal Databases (SSTD)*, July 2001, 155-174.

NAHAR, S., SAHNI, S., SHRAGOWITZ, E. 1986. Simulated Annealing and Combinatorial Optimization. *23rd Design Automation Conference*, 1986.

ORENSTEIN, J. 1986. Spatial Query Processing in an Object-Oriented Database System. *In Proceedings of the ACM SIGMOD Conference*, May 1986, 326-336.

PARK, H., CHA, G., CHUNG, C. 1999. Multiway Spatial Joins using R-trees: Methodology and Performance Evaluation. *In Proceedings of the Symposium on Large Spatial Databases (SSD)*, July 1999, 229-250.

PATEL, J.M., DEWITT, D.J. 1996. Partition Based Spatial-Merge Join. *In Proceedings of the ACM SIGMOD Conference*, June 1996, 259-270.

PARK, H., LEE, C-G., LEE, Y-J., CHUNG, C. 1999. Early Separation of Filter and Refinement Steps in Spatial Query Optimization. *In Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, April 1999, 161-168.

PAPADIAS, D., MAMOULIS, N., DELIS, 1998. V. Algorithms for Querying by Spatial Structure. *In Proceedings of the VLDB Conference*, August 1998, 546-557.

PAPADIAS, D., MANTZOUROGIANNIS, M., KALNIS, P., MAMOULIS, N., AHMAD, I. 1999. Content-Based Retrieval Using Heuristic Search. *In Proceedings of the International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, August 1999, 168-175.

PAPADIAS, D., MAMOULIS, N., THEODORIDIS, Y. 1999. Processing and Optimization of Multiway Spatial Joins Using R-trees. *In Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, July 1999, 44-55.

PAPADOPOULOS, A.N., RIGAUX, P., SCHOLL, M. 1999. A Performance Evaluation of Spatial Join Processing Strategies. *In Proceedings of the Symposium on Large Spatial Databases (SSD)*, July 1999, 286-307.

PREPARATA, F., SHAMOS, M. 1985. *Computational Geometry*. Springer, 1985.

PAPADIAS, D., THEODORIDIS, Y., SELLIS, T., EGENHOFER, M. 1995. Topological Relations in the World of Minimum Bounding Rectangles: a Study with R-trees. *In Proceedings of the ACM SIGMOD Conference*, May 1995, 92-103.

PATEL, J., YU, J., KABRA, N., ET AL. 1997. Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. *In Proceedings of the ACM SIGMOD Conference*, June 1997, 336-347.

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

ROUSSOPOULOS, N., KELLEY, F., VINCENT, F. 1995. Nearest Neighbour Queries. *In Proceedings of the ACM SIGMOD Conference*, May 1995, 71-79.

SWAMI, A., GUPTA, A. 1988. Optimization of Large Join Queries. *In Proceedings of the ACM SIGMOD Conference*, June 1988, 8-17.

SILBERSCHATZ, A., KORTH, H. F., SUDARSHAN, S. 1997. *Database System Concepts*. McGraw-Hill, 1997.

SELLIS, T., ROUSSOPOULOS, N., FALOUTSOS, C. 1987. The R^+-tree: A Dynamic Index for Multidimensional Objects. *In Proceedings of the VLDB Conference*, September 1987, 507-518.

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

THEODORIDIS, Y., STEFANAKIS, E., SELLIS, T. 1998. Cost Models for Join Queries in Spatial Databases, *In Proceedings of the International Conference on Data Engineering (ICDE)*, February 1998, 476-483.

VALDURIEZ, P. 1987. Join Indices. *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, 1987, 218-246.