Complex Spatial Query Processing

Nikos Mamoulis
Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Dinos Arkoumanis
Department of Electrical and Computer Engineering
9 Iroon Polytechnioy Str.,
Zographou 157 73, Athens,
dinosar@dbnet.ntua.gr

Abstract

The user of a Geographical Information System is not limited to conventional spatial selections and joins, but may also pose more complicated and descriptive queries. In this paper we focus on the efficient processing and optimization of complex spatial queries that involve combinations of spatial selections and joins. Our contribution is manifold; we first provide formulae that accurately estimate the selectivity of such queries. These formulae, paired with cost models for selections and joins can be used to combine spatial operators in an optimal way. Second, we propose algorithms that process spatial joins and selections simultaneously and are typically more efficient than combinations of simple operators. Finally we study the problem of optimizing complex spatial queries using these operators, by providing (i) cost models and (ii) rules that reduce the optimization space significantly. The accuracy of the selectivity models and the efficiency of the proposed algorithms are evaluated through experimentation.

Keywords: Spatial Databases, Window Queries, Spatial Joins, Cost Models

1 Introduction

Simple spatial queries, like spatial selections or joins [5], can be evaluated by standalone algorithms, usually applied on spatial access methods like the R-tree [9]. In many cases, however, the users pose complex queries that combine information from more than one spatial (or potentially non-spatial) relations. For instance the query "find all cities within 100 km of Hong Kong *crossed by* a river which *intersects* a forest" combines a selection on cities with the spatial join of cities, rivers and forests. In order to evaluate such queries we need to associate spatial selection and join processing modules in an optimal way [11]. The order by which the simple operations can be evaluated is called *query evaluation plan*. For a particular complex query there could be numerous evaluation plans. Figure 1 illustrates two possible ways of combining spatial join and selection operators in order to process the complex spatial query of the example. The plan of Figure 1a computes the spatial join between rivers and forests and writes the intersecting river-forest pairs in a temporary file T_1. A range query finds the cities that qualify the selection "within 100 km of Hong Kong" and the results are written to another temporary file T_2. T_1 and T_2 are then joined to produce the query

result. On the other hand, the plan of Figure 1b joins the cities close to Hong Kong with the rivers relation and then the qualifying results with the forest relation.

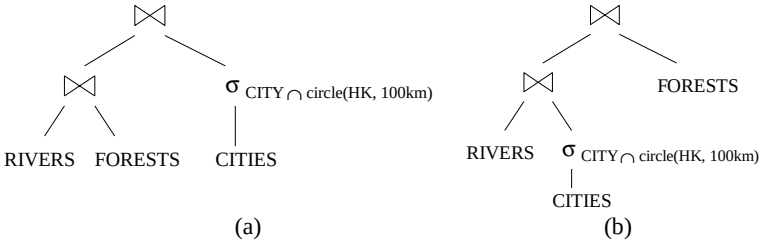
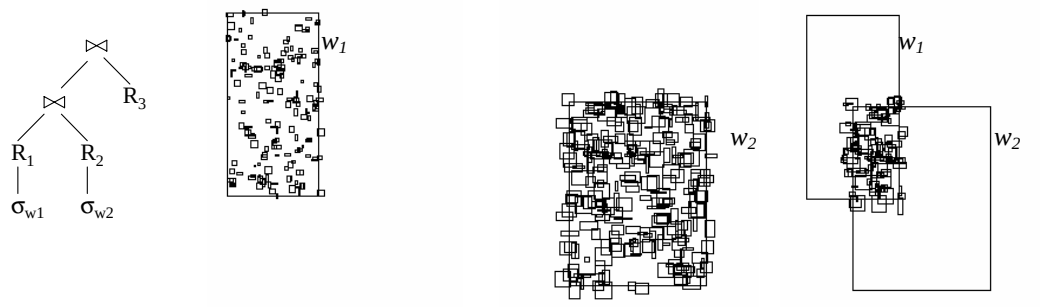


Figure 1: Two evaluation plans for a complex spatial query

The efficient processing of a complex spatial query requires the identification of a plan that minimizes the evaluation cost. The cost of a specific plan can be estimated using (i) cost formulae for the operators involved in the plan and (ii) formulae that estimate the output size of each sub-query of the plan. The first estimate the cost of each node, which accumulate to the total cost of the plan. Selectivity estimation of query sub-plans is essential since it determines the cost of succeeding operators. For example, consider the plan of Figure 1b. The selectivity of the selection operator (i.e., the number of cities that satisfy the selection) determines the cost of the join that follows. Availability of the appropriate formulae allows for query optimization methods (e.g., dynamic programming) that efficiently browse through the space of valid evaluation plans in order to identify one of low cost.

Estimating the selectivity of a complex sub-query seems a trivial task, since selectivity formulae for joins and selections are available (e.g., [31, 25]). However, the relative positions of selection windows affects the skew of the joined rectangles from each dataset. Thus, the selectivity of a spatial join between two datasets restricted by spatial selections depends heavily on the area covered by the selection windows. If the windows are disjoint and far from each other, most probably the query result will be empty. On the other hand, if they overlap the join is likely to have results. Consider, for example, the plan of Figure 2a, where R_1, R_2 are spatial relations, σ_{w1}, σ_{w2} spatial selections (e.g., window queries) and R_3 a third, not necessarily spatial, relation. Figures 2b and 2c illustrate example results of the selections applied to the corresponding dataset. Clearly the cost of the last join depends on the selectivity of the first (spatial) one. The output of the spatial join, however, does not depend solely on the results of the window queries, but also on their relative position. As Figure 2d shows, the output size of the join increases with the intersection area of the windows since only objects inside or near the intersection may participate in the result. If the windows are far apart, the result of the spatial join is expected to be empty.



(a) a query plan (b) $\sigma_{w_1}(R_1)$ (c) $\sigma_{w_2}(R_2)$ (d) $\sigma_{w_1}(R_1) \bowtie \sigma_{w_2}(R_2)$

Figure 2: Example dependency between spatial operators

Notice that this property stems directly from the interdependence between the results produced by spatial operators. The fact that all spatial operators apply on the (same) spatial attribute of the relations introduces intermediate results that are skewed with respect to succeeding operators. This problem seldom arises in relational databases because the selection conditions in most cases apply on different attributes than the join predicates, and the selections do not introduce skew to the join domains.

In this paper we study the processing and optimization of complex spatial queries that involve combinations of spatial selections and joins. Given a query where *n* relations are joined on their spatial attribute and potentially restricted by selection windows, we provide accurate formulae that estimate its output size. Following the common conventions in spatial query optimization, we assume that the input data are uniformly distributed and that the predicate is *intersect* (*overlap*). The proposed formulae use catalog information about the mean sizes of the objects in spatial relations to estimate the query result. Our estimates can be applied for arbitrarily distributed datasets using local statistics like 2D-histograms. They are also appropriate for spatial queries with predicates other than *intersect* (e.g. *meets*, *covers*), since such queries can be transformed to intersection queries (see [24] for a case study). In addition, the methods are not strictly limited to the spatial domain, because dependencies between operators can also be found in other database applications.

Going a step further, we propose novel, composite algorithms that process spatial joins and selections simultaneously and are typically more efficient than combinations of simple operators. The algorithms are extensions of spatial join algorithms [21, 20] that apply on R-trees. Finally, we study the problem of optimizing complex spatial queries using these composite operators, by providing (i) cost models and (ii) rules that reduce the optimization space significantly.

The rest of the paper is organized as follows. Section 2 provides background on selectivity estimation of simple spatial queries and describes spatial join algorithms that operate on R-trees. In Section 3 we provide accurate selectivity estimation formulae for complex spatial queries that involve multiple joins and selections. Section 4 discusses how these models can be used by a query optimizer based on existing spatial join operators. In Section 5 we propose extensions of spatial join algorithms which process spatial selections and joins simultaneously. We also show how they can be included in a spatial query optimizer and prove that not only they are more efficient than combinations of simple operations, but they also reduce the query optimization search space to that of multiway spatial joins. Finally Section 6 concludes the paper with a discussion.

2 Background

Spatial database systems [10] organize and manage large collections of multidimensional data. Spatial relations, apart from conventional attributes, contain one attribute that captures the geometric features of the stored objects. For example, the last attribute in relation City(*CName, PostalCode, Population, CRegion*) is of spatial type *polygon*. In addition to traditional data structures (e.g., B-trees) for alphanumeric attributes, spatial relations are indexed by *multidimensional access methods* [8], usually R-trees [9], for the efficient processing of queries such as *spatial selections* (or window queries) and *spatial joins*. Selections (e.g., “cities in Germany”) apply on a single relation,

while spatial joins (e.g. "cities *crossed by* a river") combine two relations with respect to a spatial predicate (typically *intersect*, which is the counterpart of the relational equi-join).

Complex spatial queries include spatial and non-spatial selections and joins. They can be processed by combining simple operators in a processing tree (plan) like the one illustrated in Figure 1a. The efficiency of an operator depends on whether its input (inputs) is (are) indexed. For instance, the cost of a window query applied on an R-tree is typically linearly related to the size of the window. On the other hand, if the selection applies on intermediate results, the whole input needs to be scanned independently of the selectivity of the operator. The same applies for join operators. The most efficient spatial join method is the R-tree join (RJ) [5], which matches two R-trees. Some methods [18, 20] join an R-tree with some non-indexed dataset (e.g., an intermediate result of another operator). Others [17, 23, 16, 2] organize two non-indexed inputs in intermediate file structures (e.g., hash buckets) in order to join them efficiently in memory.

Selection and join operators are combined to form an evaluation plan for a given complex query. In order to optimize query processing, we need accurate selectivity and cost estimation models for the various operators involved. Typically, a complex query has a large number of potential execution plans with significant cost differences. This number increases exponentially with the number of involved relations (see [21, 29] for an analysis on spatial and non-spatial domains). Optimization algorithms search either in a deterministic (e.g., dynamic programming [21, 29]) or a randomized way (e.g., hill climbing [14]) to find a cheap plan.

In the remainder of this section we review existing selectivity estimation models for simple spatial query types. Next, we describe in detail some R-tree based spatial join algorithms, which we extend in Section 5 to composite operators that process spatial selections and joins simultaneously. Finally, we illustrate a dynamic programming algorithm used to optimize multiway spatial joins.

2.1 Selectivity estimation of simple spatial query types

There has been extensive research on the accurate estimation of the selectivity and cost of spatial operators. The selectivity of spatial selections has been studied as a prerequisite for the I/O cost estimation of R-tree window queries [15, 28, 31]. Given a spatial dataset R of N d-dimensional uniformly distributed rectangles in a rectangular area r (workspace), the number of rectangles that intersect a window query w (*output cardinality* - *OC*) is estimated by the following formula:

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

where $\overline{s_d}$ is the average length of the projection of a rectangle $s \in R$ at dimension d. $\overline{w_d}$ and $\overline{r_d}$ are the corresponding projections of w, r respectively. The last factor (product) of Equation 1, called Minkowski sum, is the *selectivity* of the window query (i.e., the probability that a random rectangle from R intersects w). This probability at some dimension d equals the sum of projections $\overline{s_d}$ and $\overline{w_d}$ on that dimension normalized to the workspace. Equation 1 can be extended for the output cardinality of an (intersection) spatial join between two relations R_1 and R_2 as follows [32]:

$$OC(R_1, R_2) = N_1 \cdot N_2 \cdot \prod_{d=1}^{k} \min\left(1, \frac{\overline{s_{1,d}} + \overline{s_{2,d}}}{\overline{r_d}}\right) \quad (2)$$

N_1, N_2 denote the cardinalities of the datasets, and $\overline{s_{1,d}}$, $\overline{s_{2,d}}$ correspond to the average length of the projection of rectangles $s_1 \in R_1$ and $s_2 \in R_2$ on dimension d. In other words, the expected number of rectangle pairs that intersect is equal to the number of results after applying N_2 window queries of area s_2 on R_1. The selectivity of multiway spatial joins can be accurately estimated only for acyclic and clique (i.e., complete) query graphs. Let R_1, ..., R_n be n spatial datasets joined through a query graph Q, where Q_{ij} = true, iff rectangle $r_i \in R_i$ should intersect rectangle $r_j \in R_j$. When Q is acyclic, the number of qualifying object combinations can be estimated by:

$$OC(R_1, ..., R_n, Q) = \prod_{i=1}^{n} N_i \cdot \prod_{\forall i,j: Q_{ij}=TRUE} \prod_{d=1}^{k} \min\left(1, \frac{\overline{s_{i,d}} + \overline{s_{j,d}}}{\overline{r_d}}\right) \quad (3)$$

The above formula actually restricts the Cartesian product of the datasets using the selectivities of the query edges, which are independent. When the query graph contains cycles, the pairwise selectivities are no longer independent. For instance, if a intersects b and b intersects c, the probability that a intersects c is relatively high because the rectangles are expected to be close to each other. Thus Equation 3 is not appropriate for such cases. For the special case where Q is complete (clique) a closed formula that estimates the output of the multiway join is proposed in [25]:

$$OC(R_1, ..., R_n, Q) = \prod_{i=1}^{n} N_i \cdot \prod_{d=1}^{k} \frac{1}{(n-1) \cdot \overline{r_d}} \sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} \overline{s_{j,d}} \quad (4)$$

This formula can be derived by the observation that if $\{s_1, s_2, \ldots, s_{n-1}\}$ is a clique then $\{s_1, s_2, \ldots, s_{n-1}, s_n\}$ is also a clique iff s_n intersects the common area of all rectangles in $\{s_1, s_2, \ldots, s_{n-1}\}$. Equations 1 through 4 are accurate for uniform datasets covering the same workspace r. In real life applications these assumptions may not hold, therefore researchers have extended some of them for skewed datasets. A histogram-based method that estimates the selectivity of window queries is presented in [3]. This method decomposes the space irregularly using buckets that cover objects of similar density and keeps statistical information for each bucket, considering that its contents are uniform. A similar technique that divides the objects using a quad-tree like partitioning is presented in [1]. Another method that applies only on point datasets and uses theoretical laws is proposed in [4]. Regarding the selectivity of spatial joins, relatively little work has been done. In [32, 20] the space is decomposed using a regular grid and uniformity is assumed for each cell. The output of the join is then estimated by summing up the estimations from each cell. In [7] an interesting law that governs the selectivity of distance spatial joins (i.e., is joins that return point pairs within a distance parameter) is presented.

As discussed in the introduction, the relative positions of selection windows determine the skew of the joined rectangles from each dataset. Thus existing formula that focus exclusively on spatial selections or joins are not applicable. In Section 3 we study the selectivity of complex spatial queries, where the dependency of operators affects the query result.

2.2 R-tree based spatial join algorithms

The *R-tree join* algorithm (RJ), proposed in [5], computes the spatial join of two inputs provided that they are both indexed by R-trees. The R-tree [9] is a well-known hierarchical index that indexes minimum bounding rectangles of objects in space. RJ synchronously traverses both trees, starting from the roots and following entry pairs that intersect. Let N_A be an intermediate node from R-tree R_A, and N_B be an intermediate node from R-tree R_B. RJ is based on the following observation: if two entries $E_A \in N_A$ and $E_B \in N_B$ do not intersect, there can be no pair (o_A, o_B) of intersecting objects, where o_A, o_B are under the sub-trees pointed by E_A and E_B, respectively. A simple pseudocode for RJ is given in Figure 3. The pseudocode assumes that both trees have the same height, yet it can be easily extended to the general case by applying range queries to the deeper tree when the leaf level of the shallow tree is reached.

```
RJ(Rtree_Node N_A, N_B)
    for each E_A ∈ N_A do
        for all E_B ∈ N_B with E_A ∩ E_B ≠ ∅ do
            if N_A is a leaf node then /* N_B is also a leaf node */
                output (E_A, E_B)
            else  /* N_A, N_B are intermediate nodes */
                ReadPage(E_A.ref); ReadPage(E_B.ref);
                RJ(E_A.ref, E_B.ref);
```

Figure 3: The R-tree join algorithm

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

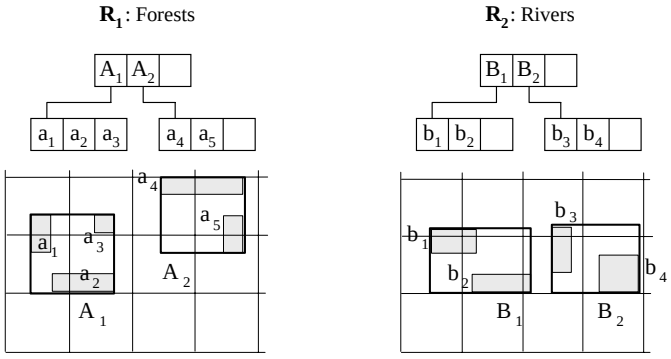


Figure 4: Two spatial datasets indexed by R-trees

Two CPU-time optimization techniques were proposed for RJ in [5]. The first, called *search space restriction*, reduces the quadratic number of pairs to be evaluated when two nodes are joined. If an entry $E_A \in N_A$ does not intersect the MBR of N_B (that is the MBR of all entries contained in N_B), then there can be no entry $E_B \in N_B$, such that E_A and E_B overlap. In Figure 4, entry a_4 of node A_2, does not intersect node B_2, so it cannot intersect any entry inside B_2. Using this observation, space restriction performs two linear scans in the entries of both nodes before applying entry intersection tests, and prunes out from each node the entries that do not intersect the MBR of the other node.

The second technique is based on *plane sweep* [27] and applies sorting in one dimension in order to reduce the computation time of the overlapping pairs between the nodes to be joined.

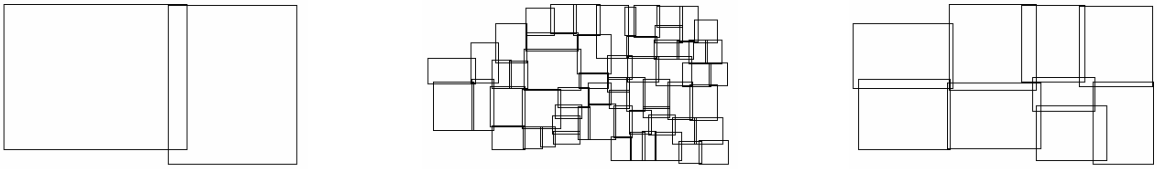
RJ was extended in [21] to process *multiway spatial joins*, i.e., spatial joins between multiple datasets. An example of such a query is "find all cities which intersect a river which crosses a forest" and can be processed by synchronously traversing three R-trees that index relations cities, rivers and forests. A multiway spatial join between n spatial relations is formally defined by a *query graph*, consisting of n nodes (one for each relation) and at least $n-1$ edges representing the join predicates between the relations. In the query graph of the example there is an edge connecting cities and rivers and another on connecting rivers with forests. The generalized extension of RJ for n inputs is called *synchronous traversal* (ST) and can be used as an alternative of combining pairwise spatial join algorithms in a processing tree. A simple pseudocode of the algorithm is given in Figure 5. ST is recursively called for n-tuples $N[]$ of R-tree nodes (initially the roots of the trees), following combinations of entries that may lead to join results until it reaches the leaves of the R-trees where qualifying object tuples are output. Parameter $Q[][]$ is a 2-dimensional array that captures the query graph. For simplicity we assume that all possible join predicates are *intersect*. Thus if $Q[i][j]$=true, there is a join intersection predicate between relations i and j. In our example, Q[cities][rivers]=true, Q[rivers][forests]=true, and Q[cities][forests]=false. The algorithm initially prunes the node domains D_i using the *space restriction* technique described above; if an node entry E_i in node N_i does not intersect the MBR of a node N_j, then it may not intersect any entry below it. Thus if $Q[i][j]$=true, such entries can be pruned, since they cannot lead to query results. *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, ST employs a sophisticated search method based on plane sweep and a backtracking heuristic [12].

```
ST(Query Q[][], RTNode N[])
    for i:=1 to n do
        D_i := space-restriction(Q, N, i);    /*prune domains*/
        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;
```

Figure 5: R-tree synchronous traversal (ST)

Slot Index Spatial Join [20] is a hash-based (pairwise) spatial join algorithm appropriate for the case where only one of the two joined relations is indexed by an R-tree. It uses the existing R-tree to define a set of *hash buckets*. If S is the desired number of partitions (tuned according to the available memory), 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. Figure 6 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, they are partitioned in 9 (for this example) slots. The grouping policy used by SISJ (see [20] for details) is based on the R*-tree insertion algorithm [6]. After building the slot index, all objects from the non-indexed relation are hashed into buckets with the same extents as the slots. If an object does not intersect any bucket it is filtered; if it intersects more than one buckets it is replicated. The join phase of SISJ loads all data from the R-tree under a slot and joins them (in memory) with the corresponding hash-bucket from the non-indexed dataset.



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

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

2.3 Optimization of multiway spatial joins

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, [21] combine ST with pairwise join operators to form query evaluation plans (e.g., similar to the ones of Figure 1). The leaves of the plan correspond to the joined relations and the intermediate nodes to the join algorithms. The algorithms are implemented as *iterators*, i.e., the output of one node of the plan is produced on-demand from the operator above. ST can be used to join two or more indexed inputs, whereas SISJ and HJ (see [17]) are used for pairwise joins where one or both inputs are not indexed.

Given a multiway join query graph Q, a dynamic programming (DP) algorithm is used to compute the optimal plan in a bottom-up fashion. At step i, for each connected sub-graph Q_i with i nodes, DP 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. After finding the optimal plan for each sub-graph of Q level-by-level, eventually DP outputs the optimal plan and estimated cost of the whole

query. In Sections 4 and 5 we show how to extend this methodology for generic complex queries, which may include multiple spatial joins and selections.

The experimental study of [21] suggests that optimal evaluation plan of a multiway spatial join varies depending on the data characteristics; joins of dense datasets are processed best by plans that include ST as a module, whereas joins of sparse datasets are handled best by combinations of pairwise algorithms

3 Selectivity of complex spatial queries

In this section we describe the first contribution of this paper, which is the definition of accurate selectivity estimation formula for complex spatial queries. Such queries involve a number n of spatial relations R_1, R_2, …, R_n joined though a query graph Q, and window queries apply on some relations. When only one selection window w_i that applies on a single dataset R_i exists, selectivity estimation is rather simple. Since the result is the same independently of the order according to which operators are applied, we can assume that the selection follows the join. The output cardinality can then be estimated by multiplying the corresponding join formulae with the selectivity of the window query:

$$OC(Join, w_i) = OC(Join) \cdot \prod_{d=1}^{k} \min\left(1, \frac{\overline{s_d} + \overline{w_d}}{\overline{r_d}}\right), \quad (5)$$

where $OC(Join)$ can be any of Equations 2, 3 or 4.

The problem is more complicated when two or more spatial selections exist on the joined datasets. A simple approach is to assume that the join workspace is the common area of all selections, and apply a single selection on the multiway join result using this area. This, however, would be inaccurate since the common area of selection windows may be empty, while there may exist objects that qualify the query, especially if the non-intersecting windows are close to each other and the data rectangles are large.

3.1 Selectivity of a pairwise join restricted by two selections

Let us first confine our study to the special case where a pair of joined datasets R_1 and R_2 are restricted by two query windows w_1 and w_2, respectively. Based on the extents of w_1 and w_2 we will try to identify the area that should be intersected by rectangles from each dataset in order to participate in the join. Thus, we will compute the query selectivity in three steps: (i) determine a number of candidate join objects from each dataset using w_1, w_2, (ii) estimate the workspace area of the join and (iii) compute the join selectivity using the number of candidates, the estimated workspace area and the average rectangle extents according to Equation 2.

Let $w_{i,d}$ be the projection of w_i on dimension d, $w_{i,d,s}$ and $w_{i,d,e}$ be the starting and the ending point of $w_{i,d}$, respectively, and $\overline{w_{i,d}}$ be the length of $w_{i,d}$. Consider also similar notations for the corresponding properties of the average extents of a rectangle s_i in dataset R_i (e.g., $\overline{s_{i,d}}$ is the average projection of s_i on dimension d). We define two *updated* windows w_1', w_2', as follows:

$$w_{1,d,s}' := \max\{w_{1,d,s}, w_{2,d,s} - \overline{s_{2,d}}\} \quad (6a)$$

$$w_{1,d,e}' := \min\{w_{1,d,e}, w_{2,d,e} + \overline{s_{2,d}}\} \quad (6b)$$

$$w_{2,d,s}' := \max\{w_{2,d,s}, w_{1,d,s} - \overline{s_{1,d}}\} \quad (6c)$$

$$w_{2,d,e}' := \min\{w_{2,d,e}, w_{1,d,e} + \overline{s_{1,d}}\} \quad (6d)$$

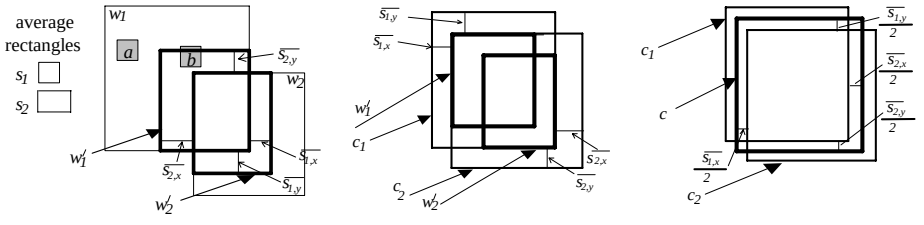
In order for a rectangle from R_i to be candidate for the join *and* intersect w_i, it should intersect w_i'. For instance, consider the selection windows w_1, w_2 and the updated ones w_1', w_2' in Figure 7a. Object *a*, belonging to R_1 and intersecting w_1, cannot participate in the join because it may not overlap some object from R_2 that intersects w_2. On the other hand, object *b* that intersects w_1' is a potential query result.

Intuitively, w_1' and w_2' define the area where the spatial join is restricted. The number of rectangles that participate in the join from dataset R_i is determined by the selectivity of the corresponding w_i'. Notice that the end points set by Equations 6a-d do not always define valid intervals, since the lower end point may be greater than the upper end point. This situation may arise when the actual windows do not intersect and there is a large distance between them. In this case (if $w_{i,d,s}' > w_{i,d,e}'$ for some *i*, *d*), the length of the corresponding projection is negative, and the selectivity of $w_{i,d}'$ is positive only when $\overline{s_{i,d}} > w_{i,d,s}' - w_{i,d,e}'$; otherwise, the selectivity of the complex query is zero.

So far, we have calculated the windows w_1' and w_2' that should be intersected by each rectangle from R_1 and R_2, respectively, in the result. These windows can be used in combination with Equation 1 to determine the number of join candidates from each dataset. The next step is to estimate the *workspace of the join c* , i.e., the area where the query results lie. The rectangles from R_i that may participate in the join are *inside* a window c_i, which is generated by extending $w_{i,d}'$ with $\overline{s_{i,d}}$ at both sides on each dimension. For instance, in Figure 7b we know that join candidates from R_1 are included in c_1. Since c_1 and c_2 do not cover the same area, we need to average them in order to acquire a unique rectangle *c* that reflects best the area where the spatial join is restricted. Figure 7c provides an example of this normalization. Formally:

$$c_{d,s} := \max\{w_{1,d,s}' - \overline{s_{1,d}}, w_{2,d,s}' - \overline{s_{2,d}}\} - |w_{1,d,s}' - \overline{s_{1,d}} - w_{2,d,s}' + \overline{s_{2,d}}|/2 \quad (7a)$$

$$c_{d,e} := \min\{w_{1,d,e}' + \overline{s_{1,d}}, w_{2,d,e}' + \overline{s_{2,d}}\} + |w_{1,d,e}' + \overline{s_{1,d}} - w_{2,d,e}' - \overline{s_{2,d}}|/2 \quad (7b)$$



(a) updated windows w_1', w_2' (b) generation of c_1, c_2 (c) join workspace *c*

Figure 7: Generation of the join workspace

Figure 8 shows some more one-dimensional examples with four representative window configurations over one pair of datasets. In case 3, the workspace is non-empty, although the original windows do not intersect. In case 4, the

updated windows w_1', w_2' are invalid; a rectangle from R_i should intersect both endpoints of the invalid window w_i' in order to be a candidate join object. However, the average length of a rectangle $s_1 \in R_1$ is smaller than the distance of the endpoints of w_1', thus the join is considered to have zero selectivity.

Given the join workspace c, selectivity can be computed using the existing formulae for spatial selections and joins. Summarizing, the output cardinality of a query that includes the spatial join of two datasets R_1, R_2 restricted by window queries w_1, w_2, respectively, is estimated by the following formula:

$$OC(R_1, R_2, w_1, w_2) = OC(R_1, w_1') \cdot OC(R_2, w_2') \cdot \prod_{d=1}^{k} \min\left(1, \frac{\overline{s_{1,d}} + \overline{s_{2,d}}}{\overline{c_d}}\right) \quad (8)$$

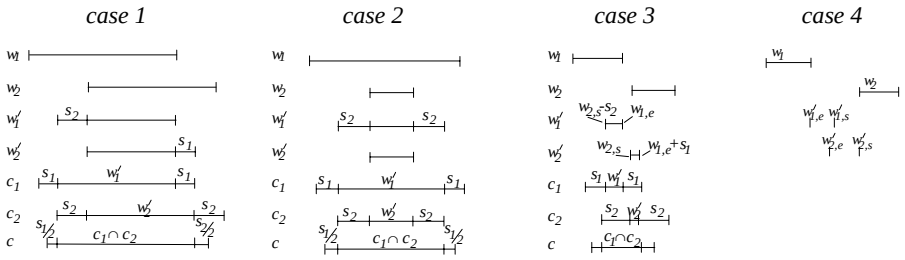


Figure 8: Windows that define the selectivity of a complex pairwise join

3.2 Selectivity of multiway joins with selections

Equation 8 can be extended for complex spatial queries that join n datasets, potentially restricted by spatial selections. Selectivity is again computed in three steps. During the first step, the updated window w_i' is calculated for each dataset R_i, using the windows of the neighbors in the join graph Q. At the second step, depending on Q, the workspace area of each join edge or of the whole graph is computed. Finally, the multiway join selectivity is estimated using (i) the selectivity of w_i' for each R_i, (ii) the workspace area(s) and (iii) Equations 3, 4.

The updated selection window w_i' for each R_i is estimated using the initial windows and the query graph. It turns out that the calculation process is not simple, since the update of a window w_i to w_i' may restrict the already updated window w_j' of a neighbor R_j. This process is demonstrated in the example of Figure 9, where three datasets are joined by a chain query and three windows restrict the joined rectangles. Assume that we attempt to calculate the updated window w_i' for each w_i using the following formulae:

$$w_{i,d,s}' := \max\{w_{i,d,s}, \max\{w_{j,d,s} - \overline{s_{j,d}}, Q_{ij} = \text{true}\}\} \quad (9a)$$

$$w_{i,d,e}' := \min\{w_{i,d,e}, \min\{w_{j,d,e} + \overline{s_{j,d}}, Q_{ij} = \text{true}\}\} \quad (9b)$$

First w_1 is restricted to w_1' using w_2 and s_2 (Figure 9c). Then w_2 is restricted to w_2' using w_1, s_1, w_3 and s_3 (Figure 9d). Observe that the left point end of w_2 has changed, and this change should be propagated to w_1' (Figure 9e), which is shortened on the left side. In general, each change at a window should be propagated to all neighbor windows in the query graph.

(a) query (b) three windows (c) update of w_1 to w_1' (d) update of w_2 to w_2' (e) second update of w_1'

Figure 9: Selection window update propagation in a network of joined datasets

The problem of window updates is similar to achieving local consistency in constraint satisfaction problems [33]. Therefore, we use a variation of an arc consistency algorithm [19] to estimate the final w_i' for each R_i. The algorithm first places all selection windows in a queue. If a dataset R_i does not have a selection window we set $w_i = r$, i.e., the workspace of the datasets. Then the first window w_f from the queue is picked and updated according to the current windows of the neighbors. If there is a change in w_f, the windows of all neighbors not currently in the queue are inserted in it because the changes need to be propagated to them. The process continues until the queue is empty. The pseudocode of the algorithm is given in Figure 10.

```
window_propagation(window w[], Query Q[][])
    initialize queue;
    for each R_i do
        if w_i does not exist then w_i := r;
        queue.insert(w_i);
    while not queue.empty() {
        w_f = queue.getfirst();
        for each dimension d do
            w_{f,d,s} := max{w_{f,d,s}, max{w_{j,d,s} - s_{j,d} , Q_fj = true}};
            w_{f,d,e} := min{w_{f,d,e}, min{w_{j,d,e} + s_{j,d} , Q_fj = true}};
            if w_f has changed
                for each j, Q_fj = true do
                    if w_j not in queue then queue.insert(w_j);
```

Figure 10: The window_propagation algorithm

After the execution of the *window_propagation* algorithm, each window w_i will be transformed to the minimum intersection window w_i'. Window w_i' is determined by the end points of the most restricted window w_j on each dimension. The path connecting R_i with R_j contains at most n-2 graph nodes (where n is the number of datasets involved in the query), meaning that the end points of the window w_i' can be adjusted at most n-1 times per dimension. Thus, the worst case complexity of the algorithm is $O(d \cdot n^2)$, and its computational overhead in the optimization process is trivial.

The next step of the estimation process is to determine the workspace area c of the multiway join. This process is performed in a similar way as described in the previous paragraph. First the coverage area c_i of each window query w_i' is estimated by extending w_i' with $\overline{s_{i,d}}$ at both sides on each dimension. If the query is acyclic the selectivity of each query edge Q_{ij} is normalized with respect to the corresponding pairwise join workspace. Therefore Equation 3 is modified as follows:

$$OC(R_1,...,R_n,w_1,...,w_n,Q)=\prod_{i=1}^{n}OC(R_i,w_i')\cdot\prod_{\forall i,j:Q_{ij}=TRUE}\prod_{d=1}^{k}\min\left(1,\frac{\overline{s_{i,d}}+\overline{s_{j,d}}}{\overline{c_{i,j,d}}}\right) \quad (10)$$

In Equation 10, $c_{i,j}$ denotes the workspace of the pairwise join between R_i and R_j, which is calculated using w_i', w_j', s_i and s_j and Equations 7a, 7b. In case of clique graphs, we need to consider a unique workspace c for the whole multiway join, since all rectangles in an output tuple mutually overlap. This workspace is defined by extending the common intersection i of all workspaces c_i by the average difference of the c_i's from the common intersection at each side and dimension. Formally:

$$i_{d,s} := \max_{1\le i\le n}\{w_{i,d,s}' - \overline{s_{i,d}}\}$$

$$c_{d,s} := i_{d,s} - \sum_{i=1}^{n}\left(i_{d,s}-w_{i,d,s}'+\overline{s_{i,d}}\right)/n \quad (11a)$$

$$i_{d,e} := \min_{1\le i\le n}\{w_{i,d,e}' + \overline{s_{i,d}}\}$$

$$c_{d,e} := i_{d,e} + \sum_{i=1}^{n}\left(w_{i,d,e}'+\overline{s_{i,d}}-i_{d,e}\right)/n \quad (11b)$$

The output cardinality of a multiway clique join is then estimated by the selectivities of the windows and the multiway join selectivity (see Equation 4), normalized to the join workspace c. Formally:

$$OC(Q,R_1,...,R_n,w_1,...,w_n)=\prod_{i=1}^{n}OC(R_i,w_i')\cdot\prod_{d=1}^{k}\frac{1}{(n-1)\cdot\overline{c_d}}\cdot\sum_{i=1}^{n}\prod_{j=1,j\neq i}^{n}\overline{s_{j,d}} \quad (12)$$

3.3 Experimental evaluation

In this section we evaluate the accuracy of Equations 8, 10 and 12. For this purpose, we generated four series of synthetic datasets with uniformly distributed rectangles in the square workspace $[0,1)^2$. The density[1] of the datasets in the different series is 0.1, 0.2, 0.4 and 0.8. Each dataset consists of 10,000 rectangles. By U*xy* we will denote the y^{th} dataset in the series of density *x*. For instance, U0.1a denotes the first dataset in the series having density 0.1. The lengths of the rectangles are uniformly distributed between 0 and $2\cdot\overline{s_{i,d}}$, where $\overline{s_{i,d}}$ is the rectangle side that leads to the desired density. For instance, in order to achieve 0.1 density in a 10,000 rectangles dataset, the average rectangle side should be $\sqrt{0.1/10{,}000}$.

Table 1 shows analytical and experimental results on complex pairwise spatial joins. Each row corresponds to a different pair of datasets and each column to a representative configuration of selection windows. Clearly, the estimated output is very close to the actual one. If we define the quantity |estimated-real|/min{estimated, real} as estimation error, the median estimation error in the experiment is 8%. The overestimated and underestimated cases are balanced, indicating that the reasoning behind Equation 8 is correct.

[1] The density of a dataset is defined as the total area of the objects in it divided by the area of the workspace, or else as the expected number of objects that intersect a random point in the workspace.

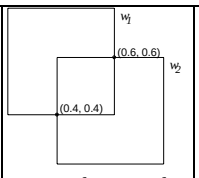
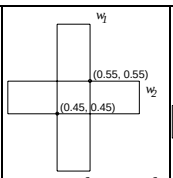
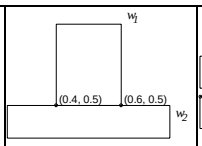
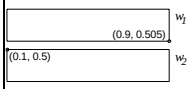
								
	estimated	actual	estimated	actual	estimated	actual	estimated	actual
U0.1a ⋈ U0.8a	633	659	167	179	24	19	46	31
U0.2a ⋈ U0.4a	511	506	134	106	18	12	30	35
U0.1a ⋈ U0.4a	395	406	103	80	12	15	17	17
U0.2a ⋈ U0.8a	780	855	207	252	33	36	69	67
U0.1a ⋈ U0.1b	169	175	43	45	3	3	1	1
U0.8a ⋈ U0.8b	1420	1469	384	433	81	81	199	179

Table 1: Evaluation of the estimation formula for pairwise spatial joins with selections

In the next experiment we test the accuracy of Equations 10 and 12 for multiway spatial joins restricted by selections. We use the uniform datasets described above and several window configurations for chain and clique graphs. Table 2 shows graphically four configurations of windows applied to six join graphs. The assignment of windows to graph nodes is done clockwise, e.g., for the first row w_1 applies to U0.1a, w_2 to U0.2a, w_3 to U0.4a and w_4 to U0.8a. We have experimented with queries that have windows on all datasets (e.g., first column) and queries with windows on some datasets. Typically the estimation is close to the actual result, but the accuracy is not as high as in the case of binary joins (the median error is now 38%). This happens because of (i) the propagation of the error in partial results and (ii) the fact that the intermediate results are more skewed than the input data. However, this is an unavoidable problem, which also exists in query optimization of relational queries involving many operators [13].

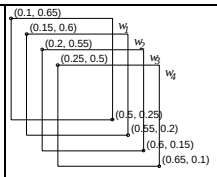
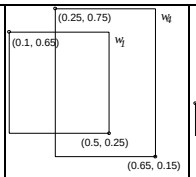
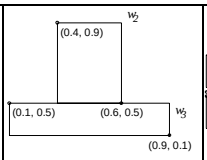
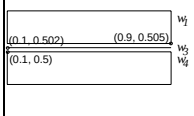
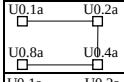
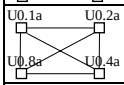
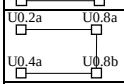
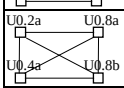
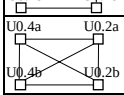
								
	estimated	actual	estimated	actual	estimated	actual	estimated	actual
	1136	1107	1784	2464	52	25	40	86
	279	395	443	542	3	3	5	6
	9352	15577	14734	22604	355	425	438	480
	1203	1602	1931	2399	32	54	18	21
	1611	2506	2554	4168	44	17	51	31
	251	348	404	555	4	5	2	3

Table 2: Evaluation of the estimation formulae for multiway spatial joins with selections

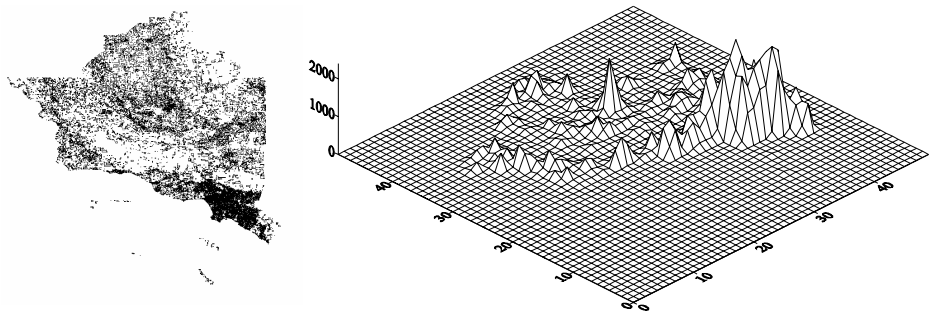
In general, the experiments prove the accuracy of Equations 8, 10 and 12 and support their use for query optimization. However, the importance of this analysis is not only restricted to this task. After proper modification the proposed methods can be used to assure that a query has zero results and, thus avert processing. As explained, if some updated window w_i' is invalid ($w_{i,d,s}' > w_{i,d,e}'$ at some d) the average size of the projection $\overline{s_{i,d}}$ at this dimension

determines whether the query *is expected to have any solutions*. If in the above methodology, instead of the average projections we employ the maximum projections max($\overline{s_{i,d}}$) for each dataset on every axis, we can determine whether the query *definitely has no solution*. Thus if $w_{i,d,s}' - w_{i,d,e}' > \max(\overline{s_{i,d}})$ and $\max(\overline{s_{j,d}})$ is used to derive w_i', processing can be avoided.

3.4 Extension to real data

In real life applications data are not usually uniform, but the distribution and size of the objects may vary in different areas of the workspace. In such cases, we need models that take advantage of information about the distribution of the objects to estimate the cost of complex queries. Formulae for range queries and distance joins [4, 7] on point datasets are not readily applicable for intersection joins of datasets containing objects with spatial extent. Techniques that keep statistics using irregular space decomposition are not appropriate either, due to the fact that two (or more) joined datasets may not have the same distribution and the space partitions can be totally different. Another limitation of such methods is that information cannot be maintained incrementally, because they need to read the whole dataset in order to update statistics. Thus, in order to deal with real data, we adopt a regular space partitioning and consider that the distribution in each partition is almost uniform, so that our models can be applied locally within partitions. Similar *local uniformity assumptions* have been extensively applied for multi-dimensional histograms (e.g., [3]) and cost models for node accesses (e.g., [32]).

In particular, we divide the space using a uniform C×C grid and for each cell we keep track of the number of objects and the total length of their MBR per axis. Assuming that the distribution in each cell is uniform, we can use this information to estimate the selectivity of spatial queries. For example, the selectivity of a range query can be estimated by applying Equation 1 for each cell that is partially covered by the window, summing the results, and adding the number of objects in cells that are totally covered. The selectivity of a pairwise or multiway spatial join [32, 20] can be estimated by applying Equations 2, 3 and 4 (depending on the query) for each combination of cells from the joined datasets that cover the same area, and summing up the results. Figure 11a illustrates a real dataset (T1) that contains road segments from an area in California. Figure 11b presents a regular 50×50 grid that keeps statistical information about the dataset. The z-axis shows the number of objects per cell. Since the characteristics of the dataset vary significantly between cells, application of uniformity-based selectivity formulae is not expected to provide good estimates. The distribution of objects in each cell may not be uniform; however, the skew is not expected to have major effects in the total estimate. This was demonstrated in a previous study [20] where the estimation error for pairwise and multiway joins was within acceptable limits. Moreover, the experimental study in [3] suggests that the accuracy of our method (called *Equi-Area* in this paper) increases significantly with the number of cells. Some histogram-based selectivity estimation methods [26] suggest approximating the distribution of data in a cell by a deviation function instead of assuming uniformity. However, these methods work for (multidimensional) points; the distribution of objects with spatial extents can hardly be described by simple functions. Another important advantage of our method is that statistical information can be maintained incrementally with trivial cost at each insertion/deletion of a rectangle.



(a) dataset T1 (b) number of rectangles per cell in a 50x50 grid

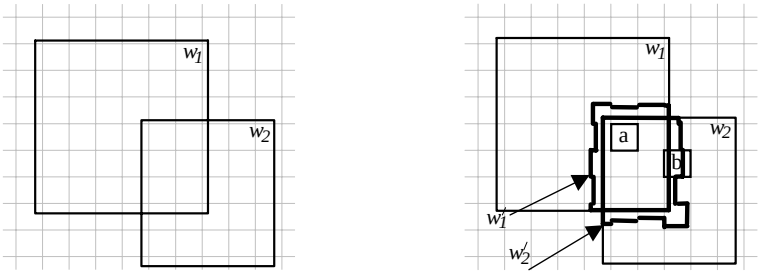
Figure 11: Skew in real dataset T1

In this section we show how the methodology of the previous section for uniform data can be applied to estimate the selectivity of complex spatial queries involving real-life, skewed datasets. We provide formulae that are based on the existence of the 2-dimensional uniform grid and the assumption that rectangles in each cell are uniformly distributed.

3.4.1 Selectivity of pairwise joins restricted by selections

We estimate the selectivity of pairwise joins involving skewed datasets that are restricted by selection windows using the methodology described in Section 3.1. Figure 12a shows a configuration of selection windows w_1, w_2 and a statistical grid. We first compute the updated w_1', w_2' for each cell, as illustrated in Figure 12b. Instead of using the global statistics about the average rectangle in a dataset we use the information kept in each cell. Thus the updated windows are not regular rectangles but their length may vary between grids. After the update there might be cells which are totally covered by both windows (e.g., cell *a* in Figure 12b) or partially intersected by them (e.g., cell *b*).

Selectivity is then estimated by summing the join result for each such cell. When a cell is totally covered by both windows, its selectivity is estimated using Equation 2 and considering the area of the cell as the workspace. Otherwise, we apply the methodology described in Section 3.1. Thus we (i) estimate the selectivity of w_1', w_2' in the cell, (ii) estimate the join workspace *c* and (iii) apply Equation 8.



(a) two windows and a 2D histogram (b) irregular updated windows using grid information

Figure 12: Two selection windows and a grid

3.4.2 Selectivity of multiway joins restricted by selections

As in Section 3.2, we will study two configurations of multiway join queries that are restricted by selections; acyclic and clique (complete) join graphs. The first stage of the estimation involves the computation of the updated

windows w'. This is done by applying the window_propagation algorithm of Figure 10. Notice that the updated windows to be considered at each step of the algorithm may be irregular, depending on the rectangle extents at each cell (see Figure 12).

We estimate the output of acyclic queries with selections incrementally using the algorithm of Figure 13. The algorithm first orders the nodes in the query graph, such that each R_i, $i>1$, in the order is connected to some R_j, $j<i$. Then at each step i it computes the selectivity of the subquery that includes nodes $\{R_1,R_2,...,R_i\}$ for each cell $g_{x,y}$ of the grid. The number of rectangles that participate in the join at $g_{x,y}$ are estimated by the selectivity of the previous step $OC(g_{x,y},R_1,R_2,...,R_{i-1},w_1',w_2',...,w_{i-1}')$ and the selectivity of w_i', $OC(g_{x,y},R_i,w_i')$. The join result at this step is determined by edge (R_i, R_j), thus, the join workspace $c_{x,y,i,j}$ at cell $g_{x,y}$ is determined by extending w_i', w_j' at both edges and all dimensions by the respective average rectangle extents $\overline{s_{x,y,i,d}}$, $\overline{s_{x,y,j,d}}$ and averaging as described in Section 3.1. The resulting $c_{x,y,i,j}$ is adjusted in all dimensions to be no longer than the respective cell extents, i.e., $\overline{c_{x,y,i,j,d}} = \min\left(\overline{c_{x,y,i,j,d}}, \overline{g_{x,y,d}}\right) \forall\ d$.

Let $OC(g_{x,y},i) = OC(g_{x,y},R_1,R_2,...,R_i,w_1',w_2',...,w_i')$ be the output of the query at step i for cell $g_{x,y}$. In summary, $OC(g_{x,y},i)$ is estimated by multiplying the selectivities of w_i' and the previous sub-query $OC(g_{x,y},i\text{-}1)$ and the join selectivity based on the estimated workspace:

$$OC(g_{x,y},i)=OC(g_{x,y},i\text{-}1)\cdot OC(R_i, g_{x,y}, w_i')\cdot \prod_{d=1}^{k} \min\left(1, \frac{\overline{s_{x,y,i,d}} + \overline{s_{x,y,j,d}}}{\overline{c_{x,y,i,j,d}}}\right) \quad (13)$$

For clique queries the process is simpler. We assume that the multiway join has a unique workspace which does not vary between join edges, as explained in Section 3.2. Thus, for each dimension we estimate the common intersection of all workspaces c_i and extend it by the average difference of the c_i's from it all each side and dimension. Naturally, like in the previous cases the common workspace might not be a regular window, but we estimate each extent at each cell of the grid. The selectivity of the multiway clique join is then estimated by Equation 12 for each cell after performing the appropriate normalization of the workspace according to the cell's extent at each dimension.

```
selectivity_estimation(window w[], Query Q[][])
    window_propagation(w,Q);
    order datasets: ∀ i>1, R_i connected to some R_j, j<i;
    estimate the selectivity of the first edge (R_1, R_2);
    for i = 3 to n do
        let R_j be the node connected to R_i, j<i;
        for each cell g_{x,y} of the grid do
            compute OC(g_{x,y}, R_i, w_i'); /* OC(R_i, w_i')on g_{x,y} */
            compute the join workspace c_{i,j,x,y};
            estimate the join results using Equation 13;
    sum up estimations for each cell and return result;
```

Figure 13: Selectivity estimation for acyclic queries

abv.	*Description*	*cardinality (N)*	*density*
T1	California roads	131461	0.05
T2	California rivers and railroads	128971	0.39
G1	German utility network	17790	0.12
G2	German roads	30674	0.08
G3	German railroads	36334	0.07
G4	German hypsography	76999	0.04

Table 3: Description of real data used in the experiments

3.4.3 Experiments

We evaluated the accuracy of the proposed extension of our methodology to handle skewed data by using some real datasets from Geographical Information Systems. The characteristics of the data used in the experiments are provided in Table 3. T1 and T2 are two layers of an area in California with large density differences. The last four datasets capture layers of Germany's map. In the first experiment we test the accuracy of our methodology for pairwise joins restricted by selections. We compare the accuracy of Equation 8 which assumes uniformity with the method of Section 4.1 using a 50×50 statistical grid. In Equation 8, instead of the actual average rectangles sides, we used normalized averages taking under consideration that the query workspace is not rectangular, but depends on the area covered by the joined datasets.

Table 4 shows the estimates of these methods and the actual query results for various joined pairs and the window configurations of Table 1. The first column for each configuration of windows shows the estimation of Equation 8, the second the estimation of the histograms method and the third the actual result of the query. The results show that both methods are not as accurate as in the case of uniform datasets. Observe that for queries where the windows have some overlap (first and second), applying the grid method is better than assuming uniformity, whereas in queries with trivial window overlap, using histogram information has small effect. The reason is that the number of results is very small and estimations are more error sensitive.

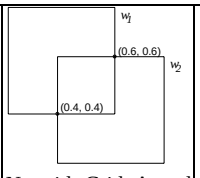
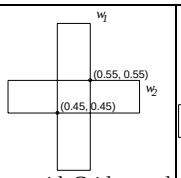
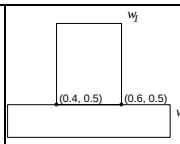
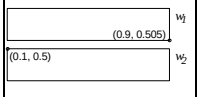
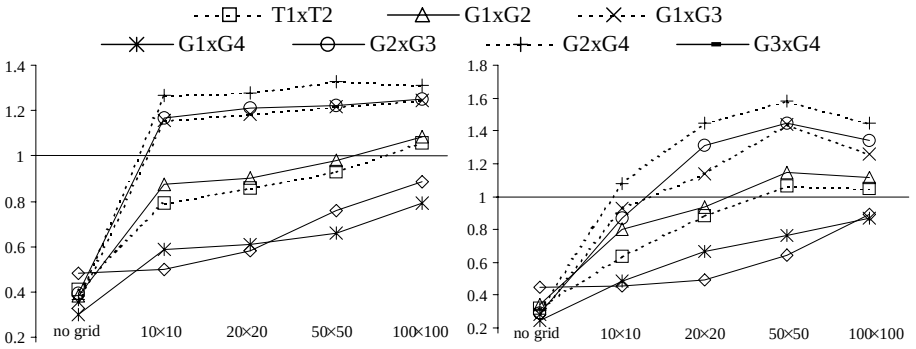
												
	No grid	Grid	Actual	no grid	Grid	actual	no grid	grid	actual	no grid	grid	Actual
T1 ⋈ T2	3629	5722	7548	917	1333	2061	27	21	27	0	0	2
G1 ⋈ G2	393	893	958	100	329	309	6	5	3	1	0	2
G1 ⋈ G3	422	1085	1105	107	356	310	6	8	4	2	0	1
G1 ⋈ G4	606	2069	1703	154	778	541	6	16	8	0	0	0
G2 ⋈ G3	407	893	1353	103	319	418	4	4	6	0	0	0
G2 ⋈ G4	505	1573	1284	127	630	436	4	7	6	0	0	0
G3 ⋈ G4	498	1823	1370	125	639	404	3	8	5	0	0	4

Table 4: Evaluation of the accuracy of the 50×50 grid on pairwise joins with selections

In the next experiment we study how accuracy is affected by the granularity of the grid. For the first two window configurations and various grid sizes we estimated the output of various joins. Figure 14 presents for each join pair and grid size the estimated selectivity divided by the actual query output. Observe that typically the accuracy

increases with the detail of the grid, although this is not a rule (see for instance G1 ⋈ G3 in Figure 14a). This is expected, since the more detailed the grid is, the best skew is handled. Nevertheless, very large grids are expensive to store and maintain.



(a) grid accuracy: first window configuration (b) grid accuracy: second window configuration

Figure 14: Accuracy of grids for various joins

We also studied the accuracy of grids for multiway join queries and the estimates were less precise to the effects of error propagation. Table 5 presents some results when uniformity is assumed and when a 50×50 grid is used. We experimented with two multiway join configurations of the Germany's layers and with the four selection window configurations of Table 2. In general, using the grid is better than assuming uniformity. The estimates are inaccurate for windows with overlap, but the error is not extreme; it is within expected bounds given the increased deviation in pairwise joins and the propagation. On the other hand, in the last two queries where the results are small, error propagation may have large effects (see the last query of the first raw).

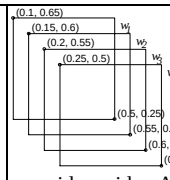
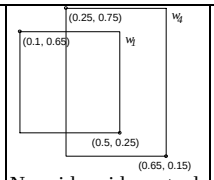
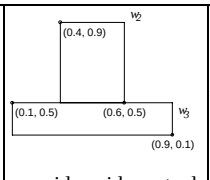
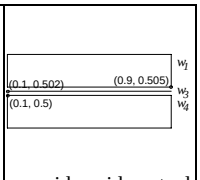
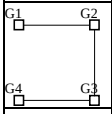
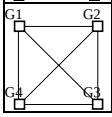
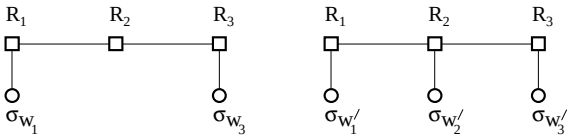
												
	no grid	grid	Actual	No grid	grid	actual	no grid	grid	actual	no grid	grid	actual
	68	319	1036	108	650	1856	2	1	0	9	28	0
	15	88	266	25	164	264	0	2	0	0	3	0

Table 5: Evaluation of the accuracy of the 50x50 grid on multiway joins with selections

We expect windows in typical queries to have some overlap. Thus, the grid can be used without major errors in optimization. On the other hand, when the actual query result is very small, the relative estimation error can be too large. Notice, however, that large relative errors in small results do not significantly affect much estimates in the cost of query operators, since the actual difference translates to few page accesses. The computation cost of the output estimates was negligible. For multiway join queries with selections when the grid is used (the most expensive case) the running time did not exceed few milliseconds, indicating that the estimates can be used for efficient query optimization.

4 Optimization of complex spatial queries using existing operators

The selectivity formulae for complex queries can be used in combination with cost formulae of spatial query operators to optimize queries that involve multiple joins and selections. This can be achieved by extending the dynamic programming (DP) algorithm described in Section 2.3 and [21] to consider potential selection operators. First the join graph Q is enriched with edges of the type (R_i, w_i) from each dataset R_i to a newly created node that symbolizes the selection window on R_i. Figure 15a shows how a complex spatial query can be transformed to an extended graph. The adapted DP begins by computing the cost of each edge. At step i, it computes the cost and selectivity of connected sub-graphs with i nodes based on the costs and selectivity of sub-graphs computed in the previous steps. The only difference with the version of DP for multiway joins is that for sub-graphs, which include selection nodes, ST is not considered as a processing method.



(a) extended graph that includes selections

(b) extended graph with updated windows

Figure 15: Graphs for the complex query $\sigma_{w_1}(R_1) \bowtie R_2 \bowtie \sigma_{w_3}(R_3)$

4.1 Using the updated windows to restrict the join inputs

As discussed in Section 3, there is interdependence between windows that restrict the joined relations. We have shown how from the initial windows w_i we can derive a series of updated windows w_i' which are smaller. More importantly, it may occur that the rectangles of an unrestricted dataset R_i in the query expression are in fact restricted because of the implicit relation of R_i with the selection window w_j of some dataset R_j with which it is joined. Therefore it is possible to accelerate the execution of a complex query by using the updated window w_i' for each R_i instead of the initial w_i. Notice that in order to replace some w_i with the corresponding w_i' in query processing, w_i' should be defined using the *maximum* projection $\max(\overline{s_{i,d}})$, $\forall R_i, d$ as explained in the previous section. Figure 15b shows how the query graph is transformed when the updated windows w_i' for each R_i are used. Even though w_2 does not exist, an implicit w_2' can be applied to restrict the rectangles from R_2.

The introduction of updated windows can increase the optimization space, since some datasets may not be initially restricted by some condition. However (i) the new optimization space can include plans which are cheaper than the cheapest plan that uses original windows, and (ii) certain rules can reduce the optimization space significantly, as will be explained in the following.

4.2 Window redundancy

Selections, when applied after the join of some datasets, filter the join results accordingly. Theoretically a complex query can be processed by first computing the multiway join of all datasets and then applying the corresponding selections to each join tuple. Some of these selections may be redundant, in the sense that the query result is the

same if they are not applied at all. An example of a redundant selection is window w_2', in Figure 15b. The execution of this selection after all joins is redundant since it is implied by the other selections. On the other hand, as explained in the previous paragraph, w_2' can be useful when applied *before* the join, because it can reduce its input significantly.

In general a window w_i' is redundant if for each dimension d, $w_{2,d,s}' > w_{2,d,s}$ and $w_{2,d,e}' < w_{2,d,e}$, i.e. the original window contains the updated window. For example in the query "find all cities in Germany crossed by a European river" the condition that the river should be European is redundant, since all rivers which go through Germany are known (or expected) to be European, due to the geographic position of Germany.

4.3 Rules that reduce the optimization space

Summarizing, the following rules may reduce the space of query optimization significantly:

i) The order of selections applied on join results is not important. After the join results the selection operators are actually filter operators with trivial cost. This rule applies also for relational query processing plans.

ii) A selection condition w_i' for some R_i is either applied before the participation of R_i in some join, or *right after* the join. Clearly, the result of a join is non-indexed and its filtering by a selection operator does not affect negatively the cost of a successive join. For instance plan $(\sigma_{w_1}(R_1 \bowtie R_2)) \bowtie R_3$ is always at most as expensive as $\sigma_{w_1}((R_1 \bowtie R_2) \bowtie R_3)$, because the result of $R_1 \bowtie R_2$ is at least as large as the result of $\sigma_{w_1}(R_1 \bowtie R_2)$ and the last join with R_3 will be executed using the same operator (e.g., SISJ).

iii) A redundant window w_i' needs not be used to filter results *after a join* if the selections on which w_i' depends have been processed before, or are processed in combination with it. Consider for example the query of Figure 15b, and assume that all joins have been processed using ST. Plan $\sigma_{w_1', w_2', w_3'}(R_1 \bowtie R_2 \bowtie R_3)$ is equivalent to $\sigma_{w_1', w_3'}(R_1 \bowtie R_2 \bowtie R_3)$, since w_1' and w_3' imply w_2'.

5 Composite spatial operators

Until now we have assumed that selections and joins are evaluated by different operators, combined in a processing tree. However, it is possible to process multiple logical operators simultaneously in the same algorithm. In this section we show how R-tree based join algorithms can be extended to process both selections and joins.

5.1 A synchronous traversal algorithm that processes spatial selections

Synchronous traversal (ST) can be extended to process spatial selections on the joined relations simultaneously with the (multiway) spatial join. A pseudocode of a straightforward extension of ST (ST_SS) is given in Figure 16. ST_SS, in addition to the query graph Q, takes as parameter an array w of selection windows, one for each variable involved in the join. If a relation R_i does not have a selection condition, the corresponding condition w_i in ST_SS is void. Clearly, if an entry $E_i \in N_i$ does not intersect the selection window, no data indexed by this entry can participate in query results. This spatial selection is applied in combination with *space-restriction*.

```
/*version 1*/
ST_SS(Query Q[][], SelConditions w[], RTNode N[])
    for each i do
        D_i = apply-selection(w_i, N_i);
        D_i = space-restriction(Q, D_i, i); /*prune domains*/
        if (D_i = ∅) RETURN; /*no qualifying tuples for this combination of nodes*/
    for each τ ∈ find-combinations(Q, D) do   /* for each current level solution */
        if N are leaf nodes /*qualifying tuple is at leaf level*/
            Output(τ);
        else /*qualifying tuple is at intermediate level*/
            ST_SS(Q, w, τ.ref[]); /* recursive call to lower level */
```

Figure 16: An extension of ST that processes spatial selections

In the next sub-section we investigate methods for optimizing ST_SS. We first propose the reduction of the original selection windows w_i to revised ones based on catalog information about the sizes of the rectangles in the joined datasets. We propose a method that filters qualifying combinations of entries at a specific level based on implicit constraints and revises the selection windows at the next level according to these constraints. Finally, we optimize the order in which nodes are considered in *apply-selection* and *space-restriction*.

5.1.1 Optimization of ST_SS

The first way to improve the performance of ST_SS is to replace the original window queries w_i posed by the user with the updated windows w_i' derived by the implicit object relations through the join conditions. These windows can be calculated using the query graph and the maximum rectangle projections per axis, as described in the previous sections. ST_SS is expected to profit from the window replacement since the new windows are expected to be smaller than the original and restrict more the search space. We will refer to this version of the algorithm as ST_SS v.2. After a combination of entries that satisfy the query constraints is found at level l, ST_SS is executed at the next level l-1 taking as parameters the nodes pointed to by these entries. However the combination of nodes at level l-1 may not have any solution. Provided that catalog information for the maximum size $\overline{s_{i,d}}$ of rectangles at each dataset R_i and axis d is available, we can identify qualifying entry combinations at level l that *implicitly* cannot lead to solutions. This is achieved by restricting the windows w_i' at the current level using the boundaries of the entries E_j, such that Q_{ij} = true. To clarify this, consider the example query of Figure 17. Assume that ST_SS is run for the root nodes of the trees and (E_1, E_2, E_3) is a qualifying entry tuple returned by *find-combinations*. Clearly, each rectangle under E_i that participates in a solution under this triple of entries should intersect, apart from w_i', all E_j such that Q_{ij} = true. However, this is not possible for any rectangle under E_2, because its maximum length max(s_2) is smaller than the distance between E_1 and E_3.

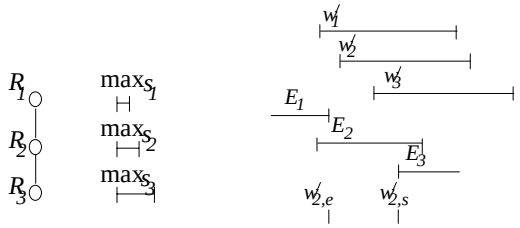


Figure 17: A qualifying triple of entries defining an invalid window at the next level

This *implicit filtering* (*IF*) technique can be implemented in three steps. Given an *n*-tuple of qualifying entries at level *l*:

(i) update each intersection rectangle w_i' with respect to the boundaries of the neighbor entries:

$$w_{i,d,s}' := \max\{w_{i,d,s}', \max\{E_{j,d,s}, Q_{ij} = \text{true}\}\}$$

$$w_{i,d,e}' := \min\{w_{i,d,e}, \min\{E_{j,d,e}, Q_{ij} = \text{true}\}\}$$

(ii) propagate window changes using the algorithm of Figure 10.

(iii) if after some window change $\max(\overline{s_{i,d}}) < w_{i,d,s}' - w_{i,d,e}'$ at some dimension *d*, consider the combination of entries disqualified and proceed to the next combination. In a different case call ST_SS passing as parameters the references to the respective nodes and the newly created windows w'.

Observe that the newly created w' can replace the windows for *apply-selection* and *space-restriction* at the next level. Assume that we follow the links of a qualifying triple ($E_{1,l}$, $E_{2,l}$, $E_{3,l}$) to a next level combination of nodes ($N_{1,l-1}$, $N_{2,l-1}$, $N_{3,l-1}$). If an entry $E_{i,l-1} \in N_{i,l-1}$ intersects the newly created window w_i' it will also intersect *both* the w_i' of the previous level and all the MBRs of $N_{j,l-1}$, where Q_{ij} = true. Thus updating the windows at each run of ST_SS improves the performance of the method since multiple intersection tests are replaced by one; space restriction is not required. More importantly, the newly created windows w' for a specific combination are calculated only once, by applying IF at the previous level, and are passed as parameters. A method similar to IF which optimizes ST for the processing of multiway spatial joins is proposed in [22]. Figure 18 shows version 3 of ST_SS that uses IF and replaces the multiple intersection tests at *apply-selection* and *space-restriction* by one. Initially, the algorithm takes as parameters the roots of the R-trees and the updated windows w'.

```
/*version 3*/
ST_SS(Query Q[][], SelConditions w'[], RTNode N[])
    for each i do
        D_i = apply-selection(w_i', N_i); /*prune domains*/
        if (D_i = ∅) RETURN; /*no qualifying tuples for this combination of nodes*/
    for each τ ∈ find-combinations(Q, D) do  /* for each current level solution */
        if N are leaf nodes /*qualifying tuple is at leaf level*/
            Output(τ);
        else  /*qualifying tuple is at intermediate level*/
            create nextw' from τ and w' using Implicit Filtering;
            if τ is still consistent /*no nextw_i' is invalid*/
                ST(Q, nextw', τ.ref[]); /* recursive call to lower level */
```

Figure 18: ST_SS with implicit filtering

The order in which the nodes will be considered in *apply-selection* and *space-restriction* may crucially affect the performance of ST_SS in terms of both I/O cost and CPU time. An R-tree node is not loaded until its entries need to be checked for consistency by these functions. If for some N_i all entries are inconsistent, the nodes N_j not examined yet ($j>i$ in the examination order) need not to be loaded because any configuration of objects under the current combination of nodes is not consistent. If the order is chosen in a way that the domains with the highest probability

to be pruned are placed first, inconsistent node combinations are detected early and fewer nodes need to be loaded and scanned.

ST orders the nodes using a simple heuristic that places the most constrained dataset in the query graph first [21]. The *degree* of a graph node is defined by the number of edges adjacent to it. The datasets are ordered in decreasing order of their degree and the most constrained ones are examined first in *space-restriction*. The involvement of selections in ST_SS changes the definition of variable restriction, since it is not anymore dependent solely on the query graph, but also on the selection condition of the variables. Luckily, the replacement of multiple intersection tests in the initial version of ST_SS with a single test using the updated w' simplifies the definition of an order in which the nodes can be optimally considered in the combined *apply-selection* and *space-restriction*. Therefore we define a fourth version of ST_SS which, before each execution of the recursive function sorts the nodes with respect to increasing selectivity of the windows w_i' to the entries of the current level.

We evaluated the efficiency of the optimization techniques for ST_SS by implementing the four versions of the algorithm and running the queries of Table 6, using the datasets described in Section 3. For each dataset an R*-tree [6] was built with page size 8K. ST_SS was supported by an LRU buffer of size 128K. All experiments were run on a Pentium III workstation (450 MHz) with 128Mbytes of main memory.

Table 6 shows for each query and each version of the algorithm four measures: (i) the CPU cost in seconds, (ii) the number of node accesses (i.e. the I/O cost assuming a zero buffer scheme), (iii) the number of page accesses in the presence of the LRU buffer, and (iv) the overall cost which is estimated by summing the CPU cost with the I/O cost, calculated by charging 10ms for each page access (a typical value [29]). The experiments show that the optimization techniques typically improve the performance of the method significantly. Especially for chain queries the performance gain of ST_SS v.4 over the initial version of the algorithm is large (the difference factor in the total cost is on the average 2.6 and reaches 6.1 in the best case).

Observe that the node accesses reflect the CPU time of the algorithm. The number of local problems, or else the number of executions of the recursive function, determines the CPU cost, as explained in Chapter 5. The difference between the computational costs of versions 1 and 4 is typically greater than the respective difference in I/O accesses. This is due to the existence of the LRU buffer, which absorbs the large difference in node accesses. For example for the configuration of the first row and the windows of the third column, the node accesses due to ST_SS v.4 are 33 times fewer than the ones of the initial version, whereas the respective difference in I/Os is just 3.4.

Although the optimization techniques can improve significantly the performance of ST_SS, they are based on catalog information which may not be available, or may be hard to maintain. For instance, the maximum rectangle projection max($\overline{s_{i,d}}$) in a dataset is not incrementally maintainable, as opposed to the average projection $\overline{s_{i,d}}$. In many cases, however, the datasets are static, or such information seldom changes, rendering the optimization of ST_SS feasible.

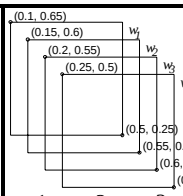
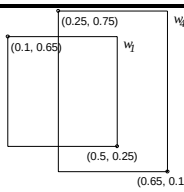
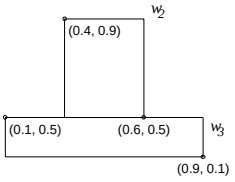
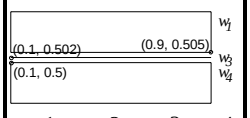
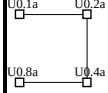
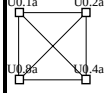
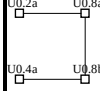
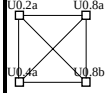
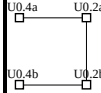
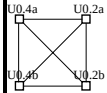
		 v.1	v.2	v.3	v.4	 v.1	v.2	v.3	v.4	 v.1	v.2	v.3	v.4	 v.1	v.2	v.3	v.4
	CPU	0.89	0.54	0.32	0.32	1.11	0.69	0.36	0.36	0.77	0.07	0.07	0.07	0.53	0.17	0.17	0.15
	NA	1017	587	349	329	1328	794	433	405	963	43	34	29	596	191	123	120
	I/O	71	38	36	36	129	69	51	49	45	13	13	13	37	29	29	29
	total	1.6	0.92	0.68	0.68	2.4	1.38	0.87	0.85	1.22	0.2	0.2	0.2	0.9	0.46	0.46	0.44
	CPU	0.35	0.25	0.25	0.25	0.4	0.35	0.35	0.3	0.19	0.1	0.1	0.09	0.17	0.17	0.17	0.16
	NA	382	261	261	243	453	334	334	325	203	33	33	30	151	118	118	113
	I/O	56	34	34	31	60	49	49	45	28	13	13	13	29	29	29	29
	total	0.91	0.59	0.59	0.56	1	0.84	0.84	0.75	0.47	0.23	0.23	0.22	0.46	0.46	0.46	0.45
	CPU	0.8	0.55	0.36	0.31	1.23	0.87	0.46	0.46	0.33	0.14	0.14	0.14	0.46	0.19	0.18	0.15
	NA	892	581	334	317	1492	946	498	478	330	90	83	77	500	189	153	136
	I/O	57	53	40	34	81	111	71	70	25	17	17	17	75	30	30	29
	total	1.37	1.08	0.76	0.65	2.04	1.98	1.17	1.16	0.58	0.31	0.31	0.31	1.21	0.49	0.48	0.44
	CPU	0.3	0.3	0.3	0.25	0.47	0.39	0.39	0.35	0.14	0.13	0.13	0.1	0.22	0.19	0.19	0.19
	NA	325	276	276	254	503	386	386	368	91	58	58	47	214	158	158	155
	I/O	46	33	33	34	60	55	55	47	19	15	15	15	37	33	33	32
	total	0.76	0.63	0.63	0.59	1.07	0.94	0.94	0.82	0.33	0.28	0.28	0.25	0.59	0.52	0.52	0.51
	CPU	0.7	0.37	0.29	0.27	1.22	0.65	0.39	0.39	0.31	0.11	0.1	0.1	0.52	0.2	0.18	0.17
	NA	835	424	290	276	1441	721	460	437	325	35	32	32	572	225	171	138
	I/O	78	35	35	30	98	58	53	56	16	12	12	12	42	30	30	29
	total	1.48	0.72	0.64	0.57	2.2	1.23	0.92	0.95	0.47	0.23	0.22	0.22	0.94	0.5	0.48	0.46
	CPU	0.31	0.22	0.22	0.21	0.38	0.29	0.29	0.29	0.09	0.08	0.08	0.08	0.13	0.13	0.13	0.13
	NA	267	195	195	190	424	297	297	300	52	24	24	24	115	92	92	92
	I/O	34	29	29	29	62	52	52	45	14	12	12	12	28	28	28	28
	total	0.65	0.51	0.51	0.5	1	0.81	0.81	0.74	0.23	0.2	0.2	0.2	0.41	0.41	0.41	0.41

Table 6: Evaluation of the optimization techniques for ST_SS

5.1.2 Cost estimation of ST_SS

For the buffer scheme used in the experiments of the previous section, in most of the cases the I/O cost of ST_SS dominates its computational cost, but the difference is not large. If a larger buffer (e.g., 512K) is used the I/O cost is lower, and the algorithm becomes I/O bound. The presence of the LRU buffer complicates the estimation of the I/O cost. On the other hand, the computational cost can be predicted using the formulae of Section 3 and catalog information about the R-trees. The number of solutions at an R-tree level determines the number of problems to be solved at the next level, or else the number of node accesses. Thus, if we assume that each solution at level *l* causes *n* node accesses at level *l*-1 and a local problem to be solved, the estimation of the computational cost (and the I/O cost if no buffer is used) is easy. Thus Equations 10 and 12 can used in order to determine the number of problems solved by ST_SS as follows:

$$N_{PROBLEMS\ (ST\_SS)} = 1 + \sum_{l=1}^{h-1} Sel(Q, R_{1,l},..., R_{n,l}, w_1,..., w_n) \quad (13)$$

In the above equation l denotes an R-tree level, $R_{i,l}$ the number of entries of R-tree R_i at level l and h the height of the trees[2]. The selectivity factor at the right side of the equation is calculated using the formulae of Section 3 and the average length of the entries of the trees *at each level* and dimension. Thus in order to estimate the number of solutions at level l we need for each R-tree the number of entries at this level and the average length of their projections at each dimension. $N_{PROBLEMS}$ multiplied by n provides an estimate for the number of node accesses of ST_SS. The computational cost can be estimated following the analysis of [21] for each local problem of ST.

The optimization techniques affect the cost estimation accuracy for ST_SS since (i) a significant part of solutions at a specific level can be pruned due to implicit constraints and (ii) a significant part of problems may finish early due to the elimination of a node N_i when its entries do not intersect a (potentially very strict) updated window w_i'. As shown in the experiments of the previous section, the number of node accesses and the CPU cost can be significantly reduced after applying the optimization techniques.

The current formula is accurate only for the initial version of the algorithm (ST_SS). We can easily adapt it to estimate the number of problems run by ST_SS v.2 by simply replacing in the window propagation algorithm, the average rectangle sides at the high levels with the maximum rectangle sides of the actual data (leaf level). Thus $\max(\overline{s_{i,d}})$ is used for the computation of w', and the average entry extents per level are used for the selectivity of the query. The cost of ST_SS v.3 and ST_SS v.4 is hard to estimate due to the fact that there is no closed formula that captures the implicit relationships of an arbitrary pair of nodes in an acyclic (and incomplete in general) graph. However when the join graph is a clique the cost of ST_SS v.2 is expected to be approximately the same as that of ST_SS v.3 (and ST_SS v.4) due to a lack of implicit constraints. The experimental results of Table 6 justify this statement.

5.2 Extending SISJ to include spatial selections

SISJ can be extended to process a pairwise join where the indexed dataset is restricted by a selection window w. The adapted SISJ_SS algorithm consists of the following steps:

i) find the topmost R-tree level where the number of entries that intersect w is at least as large as the number of slots S, by following only entries that intersect w.

ii) divide the entries into S slots.

iii) hash the rectangles from the non-indexed set into buckets having the same extents as the slots.

iv) load the data under each slot, filter them using w, and match them with the corresponding bucket.

There are two issues to be discussed regarding the above extension. The first is whether we can improve its performance using catalog information about the maximum rectangle side in the indexed dataset. We can achieve larger degree of filtering while hashing the non-indexed dataset by restricting the boundaries of the slots, using w and $\max(\overline{s_d})$. Moreover, as explained in Section 3, we can restrict the selection window w to w' using its implicit relationships with other selection windows in the join graph. The second issue is what happens when w is small

[2] Notice that in ST_SS we have assumed that all R-trees have the same height. The extension of the algorithms and selectivity estimation for R-trees of different height is straightforward.

enough for the slot level to be the leaf level of the tree. In this case we expect the memory to be large enough to fit all rectangles that intersect *w*. Therefore, we have a single partition for the indexed set. The rectangles of the non-indexed set are filtered using the MBR of this partition and are matched in memory using plane sweep.

The cost of SISJ_SS is simple to estimate. Using catalog information about the extents of the R-tree entries we can estimate the slot level and the number of I/Os required to reach this level and find the slots. This is the cost of executing a window query *w* against the R-tree until a specific level. Then by extending *w* using max($\overline{s_d}$) we can estimate the number of rectangles from the non-indexed dataset that will not be filtered and will be hashed into the buckets. This number is equal to the selectivity of the extended window *ext(w)* on the non-indexed data. The cost of the match phase is the remaining cost of the window query on the R-tree plus the cost to load the hash buckets from the second dataset. Thus the total cost of SISJ_SS consists of (i) the cost of the window query *w* on the R-tree, (ii) the cost of reading the non-indexed data and (iii) the cost of writing and reading back the hashed data.

5.3 Query optimization using composite operators

Composite join algorithms that process non-indexed datasets (e.g., an extension of the spatial hash join algorithm [17]) are meaningless, because there is no index that needs to be preserved by them. The selection can be executed before the join as a filter of the inputs instead of being combined with the join. The introduction of ST_SS and SISJ_SS complicates optimization since the number of potential plans for a complex query increases. In this section we discuss some properties that reduce the search space and show the superiority of composite operators.

Lemma 1: A plan where a spatial selection on a dataset is executed *after* a join that involves the dataset has *at least* the cost of a plan where the selection is executed concurrently with the join using a composite operator.

Proof: It suffices to show that ST_SS and SISJ_SS are at most as expensive as ST and SISJ, respectively. This statement is true, since the cost of the join algorithms can only be decreased when a selection is included in them. The methods having trivial computational overhead (the filtering of R-tree entries/nodes using the selection windows) usually reduce the search space while executing the join. □

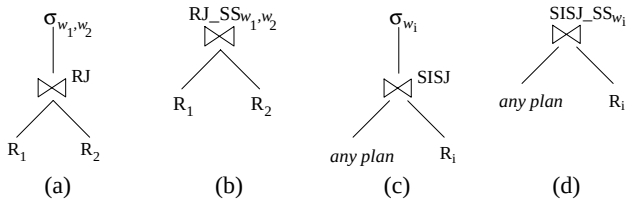


Figure 19: Combining selections with joins is always more efficient than evaluating them after the joins

As a result, the plans of Figure 19b and Figure 19d are superior to those of Figure 19a and Figure 19c, respectively (notice that we use RJ_SS to denote ST_SS with only two inputs). The lemma reduces the optimization space to exactly the one that the problem had before the inclusion of composite operators; each combination of selections after some join is replaced by the composite join operator that includes the selections. Now let us see whether we can further reduce the optimization space by investigating the superiority of composite operators in comparison with

selections followed by simple join operators. Consider the pairwise join of two datasets R_1, R_2, restricted by selection conditions w_1, w_2, respectively. The query can be executed using one of the plans depicted in Figure 20. Notice that we use HJ to denote a spatial hash join algorithm that applies on non-indexed data (e.g., [17]).

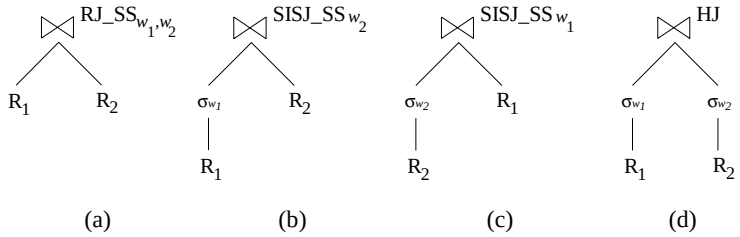


Figure 20: The four plans of $\sigma_{w_1}(R_1) \bowtie \sigma_{w_2}(R_2)$

Theoretically, all plans reach a stage where the selections have been executed and the join follows. RJ_SS follows entry pairs that intersect the windows and each other until a level, where the selections do not prune any pair (i.e., all pairs are included in the selection windows). SISJ_SS executes the selection until a specific level of the indexed input and then hashes the other input (where the selection has already taken place) in buckets. The same applies for HJ, which hashes both selected results into buckets before the join. If the selections produce few results that fit in memory, all algorithms are expected to be equivalent, since the join will take place in memory with trivial cost and the selection has almost the same cost for each algorithm. Assume now that the results of the selections do not fit in memory. In this case, when we reach the stage that the selections have been processed, RJ_SS is expected to be more efficient than the other plans because it takes advantage of the indexes to perform the join *without writing intermediate results to disk*. This advantage of RJ_SS over the other plans is equivalent to the advantage of RJ over SISJ and HJ at joining large spatial datasets without selections (see [20] for a qualitative comparison of the three methods). Regarding the relative performance of SISJ_SS and HJ, when the selection results do not fit in memory, SISJ_SS is expected to be more efficient than HJ, because it uses the R-tree entries to guide the hash process and reads the indexed input at most once. Thus the plan of Figure 21a is always superior to the one of Figure 21b and:

Lemma 2: A plan where a spatial selection on a dataset is executed *before* a join is *at least as expensive* as a plan where the selection is executed within the join using a composite operator. □

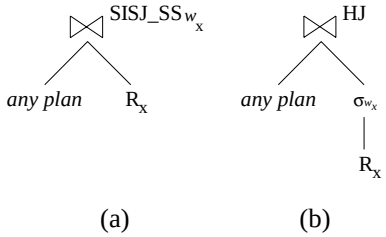


Figure 21: A case where SISJ_SS is always more efficient than HJ

Thus, during the optimization of complex spatial queries, the potential plans contain only ST_SS, SISJ_SS, and HJ in their nodes as opposed to ST, SISJ and HJ for multiway spatial joins [21]. The plain operators ST and SISJ are not considered because the inclusion of the implied windows w', assigns a selection window to every dataset. Of course ST and SISJ can still be used for other queries that do not include spatial selections, but only joins and non-spatial selections. Therefore the query optimization search space for complex spatial queries is the same as the search space for multiway spatial joins. Moreover, we expect the optimization process to produce plans similar to those for multiway joins, i.e. dense datasets will be joined best using large ST_SS plans, and sparse ones using combinations of ST_SS, SISJ_SS and HJ (see [21] for a detailed study).

6 Conclusions

In this paper we have extensively studied the problem of processing complex spatial queries that involve multiple spatial joins and selections. This work completes the study of [21] for multiway spatial joins. Our first contribution is the definition of selectivity formulae for complex spatial queries. We presented formulae that estimate the output of such queries and evaluated them through experimentation. The results prove the accuracy of the formula, the relative error being 8% for binary joins and 38% for queries of four inputs. These numbers are comparable with previous work on selectivity of spatial selections [31] or joins [32], as well as, with error propagation experiments in the context of relational queries [13]. The proposed models are essential for the optimization of queries that involve several spatial logical operators, possibly in addition to some non-spatial ones. We have extended our method for skewed, real-life data using 2D-histograms. In this case, the accuracy is not that high due to the persistence of skew in the cells of the statistical grid, but still the histogram-based method does better than straightforward application of formulae that assume uniformity.

The value of this study is not limited to spatial query processing, since operator dependencies may exist in other applications as well. Consider a relational query that consists of a join between R and S on their common $R.x = S.x$ attribute, and (range) selections on other attributes of R and S (e.g., $R.y < a$, $S.z \in [b,c]$). A dependency between $R.y$ and $S.z$ affects the query results. For instance, in the 30-R benchmark [30] there are multi-table constraints like O.Orderdate ≤ L.Shipdate (i.e., an order takes place before the corresponding line items are shipped). The selectivity of queries that apply selections on these attributes and then join the corresponding tables can be estimated in a way similar to that presented in this paper (i.e., by restricting the selections, taking under consideration the constraints, and then estimating the join selectivity on the restricted area). Another type of related complex queries involve distance joins of high-dimensional point sets [7]. Our methodology can be applied if high-dimensional selections exist in conjunction with the joins, since the domain of the query operators is the same.

The interdependence between spatial selections and joins allows their simultaneous processing by composite operators that take advantage of existing indexes to guide search. Our second contribution is the extension of ST and SISJ to ST_SS and SISJ_SS, respectively, which process spatial selections and joins concurrently. Starting from straightforward versions of these composite algorithms, we optimize their efficiency using implicit relationships between selection windows and node extents that are joined as well as catalog information about the maximum

extents of the rectangles in the datasets. Finally, we show that the composite operators are more efficient than combinations of simple ones and prove that complex spatial query optimization is equivalent to the optimization of multiway spatial joins. This result is very important because it suggests that the optimization techniques proposed in [21] can directly be applied for complex spatial queries.

Acknowledgements

This work was supported from grants HKU 7149/03E and HKUST 6180/03E from Hong Kong RGC.

References

[1] A. Aboulnaga and J.F. Naughton, Accurate Estimation of the Cost of Spatial Selections, International Conference on Data Engineering, 123-134, Feb.-Mar., 2000.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter, Scalable Sweeping-Based Spatial Join, VLDB Conference, 570-581, Aug., 1998.

[3] S. Achaya, V. Poosala, and S. Ramaswamy, Selectivity Estimation in Spatial Databases, ACM SIGMOD Intl. Conference on Management of Data, 13-24, June, 1999.

[4] A. Belussi and C. Faloutsos, Estimating the Selectivity of Spatial Queries Using the Correlation Fractal Dimension, VLDB Conference, 299-310, Sep., 1995.

[5] T. Brinkhoff, H.P. Kriegel, and B. Seeger, Efficient Processing of Spatial Joins Using R-trees, ACM SIGMOD Intl. Conference on Management of Data, 237-246, May, 1993.

[6] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: an Efficient and Robust Access Method for Points and Rectangles, ACM SIGMOD Intl. Conference on Management of Data, 322, 331, May, 1990.

[7] C. Faloutsos, B. Seeger, A. Traina, and C. Traina, Spatial Join Selectivity Using Power Laws, ACM SIGMOD Intl. Conference on Management of Data, 177-188, May, 2000.

[8] V. Gaede and O. Günther, Multidimensional Access Methods, ACM Computing Surveys, 30(2): 123-169, 1998.

[9] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, ACM SIGMOD Intl. Conference on Management of Data, 47-57, June, 1984.

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

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

[12] R. Haralick and G. Elliott, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, 14: 263-313, 1980.

[13] Y. Ioannidis and S. Christodoulakis, On the Propagation of Errors in the Size of Join Results, ACM SIGMOD Intl. Conference on Management of Data, 268-277, May, 1991.

[14] Y. Ioannidis and Y. Kang, Randomized Algorithms for Optimizing Large Join Queries, ACM SIGMOD Intl. Conference on Management of Data, 312-321, May, 1990.

[15] I. Kamel and C. Faloutsos, On Packing R-trees, ACM International Conference on Information and Knowledge Management, 490-499, November, 1993.

[16] N. Koudas and K. Sevcik, Size Separation Spatial Join, ACM SIGMOD Intl. Conference on Management of Data, 324-335, May, 1997.

[17] M.L. Lo and C.V Ravishankar, Spatial Hash-Joins, ACM SIGMOD Intl. Conference on Management of Data, 247-258, June, 1996.

[18] M.L. Lo and C.V. Ravishankar, The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins, IEEE Transactions on Knowledge and Data Engineering, 10(1): 136-151, 1998.

[19] A. Mackworth, Consistency in Networks of Relations, Artificial Intelligence, 8, 1977.

[20] N. Mamoulis and D. Papadias, Integration of Spatial Join Algorithms for Processing Multiple Inputs, ACM SIGMOD Intl. Conference on Management of Data, 1-12, June, 1999.

[21] N. Mamoulis and D. Papadias, Multiway Spatial Joins, ACM Transactions on Database Systems (TODS), 26(4): 424-275, 2001.

[22] H. Park, G. Cha, C. Chung, Multiway Spatial Joins using R-trees: Methodology and Performance Evaluation, Symposium on Large Spatial Databases (SSD), 229-250, July, 1999.

[23] J.M. Patel and D.J. DeWitt, Partition Based Spatial-Merge Join, ACM SIGMOD Intl. Conference on Management of Data, 259-270, June, 1996.

[24] D. Papadias, N. Mamoulis, and V. Delis, Approximate spatio-temporal retrieval, ACM Transactions on Information Systems (TOIS), 19(1):53-96, January 2001.

[25] D. Papadias, N. Mamoulis, and Y. Theodoridis, Processing and Optimization of Multiway Spatial Joins Using R-trees, ACM Symposium on Principles of Database Systems (PODS), 44-55, July, 1999.

[26] V. Poosala, Histogram-based estimation techniques in databases. PhD Thesis, University of Wisconsin-Madison, 1997.

[27] F. Preparata and M. Shamos, Computational Geometry, Springer, 1985.

[28] B. Pagel, H. Six, H. Toben, and P. Widmayer, Towards an Analysis of Range Query Performance in Spatial Data Structures, ACM Symposium on Principles of Database Systems (PODS), 214-221, May, 1993.

[29] A. Silberschatz, H.F. Korth, S. Sudarshan, Database System Concepts. McGraw-Hill, 4th ed., 2002.

[30] Transaction Processing Performance Council, 30 Benchmark R (Decision Support), Rev. 1.0.1, http://www.30.org/, 1993 – 1998.

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

[32] Y. Theodoridis, E. Stefanakis, and T. Sellis, Cost Models for Join Queries in Spatial Databases, International Conference on Data Engineering, 476-483, February, 1998.

[33] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, London and San Diego, 1993.