# Approximate Spatio-Temporal Retrieval

Dimitris Papadias[1], Nikos Mamoulis[2] and Vasilis Delis[3]

[1]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
Dimitris@cs.ust.hk

[2]CWI
Kruislaan 413, PO Box 94079
1090 GB Amsterdam, The Netherlands
Nikos.Mamoulis@cwi.nl

[3]Computer Engineering and Informatics Department
and Computer Technology Institute
University of Patras, Greece
delis@cti.gr

**Abstract**: This paper proposes a framework for the handling of spatio-temporal queries with inexact matches, using the concept of *relation similarity*. We initially describe a binary string encoding for 1D relations that permits the automatic derivation of similarity measures. We then extend this model to various granularity levels and many dimensions, and show that reasoning on spatio-temporal structure is significantly facilitated in the new framework. Finally, we provide algorithms and optimization methods for four types of queries: (i) object retrieval based on some spatio-temporal relations with respect to a reference object, (ii) spatial joins, i.e., retrieval of object pairs that satisfy some input relation, (iii) structural queries, that retrieve configurations matching a particular spatio-temporal structure and (iv) special cases of motion queries. Considering the current large availability of multi-dimensional data and the increasing need for flexible query answering mechanisms, our techniques can be used as the core of spatio-temporal query processors.

Correspondence should be addressed to Dimitris Papadias.

Room 3503
Department of Computer Science
University of Science and Technology
Clearwater Bay, HONG KONG
Tel: ++852-2358-6971, Fax: ++852-2358-1477

This paper is the extended and revised version of work presented at the 24th VLDB Conference [PMD98] and at the 6th ACM Conference on Multimedia [DMP98].

## 1. INTRODUCTION

The general theme of this work is the design and implementation of efficient retrieval mechanisms for spatio-temporal databases. Adopting a relational view, such databases are collections of entities which have either spatial attributes (e.g. geographic databases) or temporal attributes (e.g. medical databases) or combinations thereof (e.g. multimedia databases). Spatial attributes can be viewed as 0D, 1D, 2D or 3D positions[1] in a "space", either the physical one (e.g. map objects) or an artificial one such as a computer screen (e.g. multimedia objects). Temporal attributes capture the temporal existence of entities and in the general case can be represented as time points or time intervals. Continuing the relational analogue, such attributes should be allowed in the expression of user queries related through appropriate operators such as *contains, north-east, near*, *during, after*, etc.

Retrieval mechanisms able to handle queries of the above type could also be beneficial even for unstructured or semi-structured collections of spatio-temporal entities. The prohibitively large volumes and the heterogeneity of the widespread multimedia information (like maps, satellite imagery, multimedia presentations, images, etc.) render purely textual searching ineffective and raise the need for "intelligent" query processing, focusing on content [NY96]. As a result, there has already been significant progress on image and video content retrieval research ([F+94], [OS95], [SC96], [SK97], [M98], [C+98]). Most of these techniques, however, address retrieval of *visual* content, i.e. properties like colour, shape, texture, etc. A rather neglected type of content is *spatio-temporal structure*, which refers to the spatial and temporal arrangement of a collection of objects. Used in conjunction with visual content retrieval, spatio-temporal retrieval could allow for the processing of powerful similarity queries like "find all multimedia presentations depicting a *sunset image*, which is *followed* by a *slide show* on its *left*, *synchronised* with a *narration*".

Handling spatio-temporal queries requires a breakthrough in structure description, as well as, retrieval mechanisms. This is even more stressed considering a salient characteristic of such queries, the requirement for inexact matches and approximation scores to rank the query output, which can be attributed to two main reasons: First, if a user asks for a specific spatio-temporal configuration he/she may be interested in receiving matching configurations, possibly at a defined tolerance (degree of approximation). Second, spatio-temporal predicates like *far, northeast, during,* etc. do not always have crisp definitions, so users may typically accept answers similar although not identical to their query, as these could correspond to conceptually valid representations. The need for approximation constitutes a serious impediment if traditional query processing techniques are to be employed.

In this work we deal effectively with the above issues. In particular, we (i) propose a powerful framework for representing and reasoning on spatio-temporal relations at various resolution levels and arbitrary dimensions, (ii) demonstrate how the framework can be employed in database systems that use multi-dimensional data structures, (iii) develop algorithms for several types of approximate retrieval (iv) evaluate the efficiency of our methods with extensive experiments involving real data.

[1] In GIS literature a *position* is often defined as a tuple <location, size, shape, orientation>.

The rest of the paper is organized as follows: Section 2 describes a binary string encoding for the representation of spatio-temporal structure in multiple resolutions and dimensions. Section 3 illustrates how the framework is used for retrieval of objects that have certain structural characteristics within a repository of spatio-temporal information. Section 4 deals with spatial joins, i.e., retrieval of object pairs that satisfy some spatio-temporal constraint. Section 5 studies the most complex, and probably most interesting problem, that of configuration similarity between spatio-temporal scenes (object collections). Section 6 illustrates the application of our framework to motion queries, viewed as temporal sequences of spatial events. Finally, Section 7 concludes the paper with a discussion about further continuation of this work.

## 2. A FRAMEWORK FOR SIMILARITY

Several types of relations such as topological (e.g., *inside*, *overlaps*), direction (*north*, *southeast*) and distance (e.g., near, far) have been applied to express spatial queries. Even for a single type of constraint, definitions may vary [H94]; for instance direction relations can be defined by angles between object centroids (e.g., *north* may correspond to an angle of 90 degrees) or by projections (an object is north of another if all its points are higher than any point of the second one). For the temporal domain there is a widely acceptable set of 13 mutually exclusive relations (proposed by Allen [83]) which describe the relative locations between 1D (time) intervals. Sample configurations of intervals that satisfy each of these relations are shown in Figure 2.1. Relation $R_1$, for instance, corresponds to the situation where all points of the upper interval are before the lower one. Multi-dimensional extensions of Allen's relations have also been applied for spatial queries.

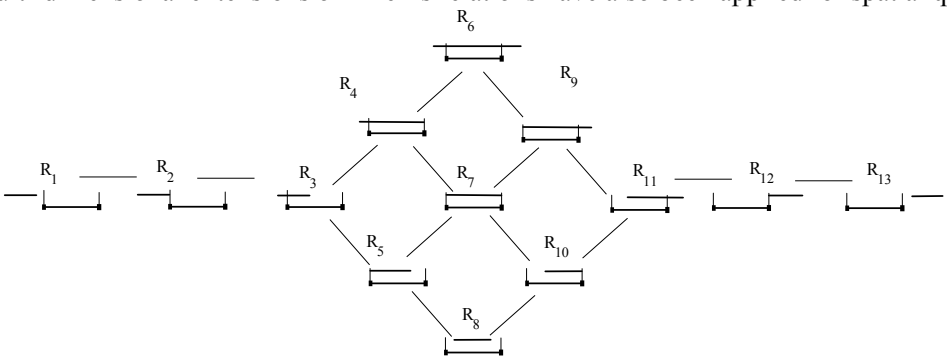



**Figure 2.1** Conceptual neighborhood for relations between intervals in 1D space

The concept of *conceptual neighborhood* [F92] provides the means for defining similarity measures for a set of relations. A conceptual neighborhood is represented as a graph whose nodes denote relations that are linked through an edge, if they can be directly transformed to each other by continuous interval deformations. In such a graph, similar relations are closer to each other than non-similar ones. Depending on the allowed deformation (e.g., movement, enlargement), several graphs may be obtained. The one in Figure 2.1, corresponds to what Freksa called *A-neighbors* (three fixed endpoints, while the fourth is allowed to move). Starting from relation $R_1$ and extending the upper interval to the right, we derive relation $R_2$. With a similar extension we can produce the transition from $R_2$ to $R_3$ and so on. $R_1$ and $R_3$ are called 1st degree neighbors of

$R_2$. The *distance d* between two relations is equal to the length of the shortest path relating them in the neighborhood graph.

Related work on conceptual neighborhoods has been carried out for direction [NNS96], topological relations [EA92] and for classes of both topological and direction relations [BE96]. According to all these approaches there is a pre-defined set of relations for which conceptual neighborhoods are pre-computed and encoded in look-up tables. Subsequent queries use the look-up tables to retrieve similar matches.

However, assuming a predefined set of relations and similarity measures is a serious restriction for most applications. Different users may impose different kinds of spatio-temporal constraints, or even if the constraints are similar they may refer to different granularities. An effective system for spatio-temporal retrieval should provide flexibility in the definition of constraints and the means for the automatic calculation of similarity measures depending on the query. In this section we describe a framework for spatio-temporal relations that supports dynamic (i.e., not pre-defined) constraints and similarity measures. The proposed framework is easily adjustable to different user needs and may have a wide range of applications in spatio-temporal query processing.

### 2.1 A Binary String Encoding of Relations

Let [a,b] a closed and continuous 1D (time) interval with endpoints a and b, $-\infty<a<b<\infty$. We identify 5 distinct *regions of interest* (which can be points or open intervals) with respect to [a,b]: 1.$(-\infty,a)$ 2.[a,a] 3.(a,b) 4.[b,b] 5.$(b,+\infty)$ (see Figure 2.2a). The relationship between a *primary* interval [z,y], and [a,b] can be uniquely determined by considering the 5 empty or non-empty intersections of [z,y] with each of the 5 aforementioned regions, modelled by the 5 binary variables *t, u, v, w, x,* respectively, with the obvious semantics ("0" corresponds to an empty intersection while "1" corresponds to a non-empty one). Therefore, we can define relations in 1D to be binary 5-tuples ($R_{tuvwx}$ : $t, u, v, w, x \in \{0,1\}$). For example, $R_{00011}$ ($t$=0, $u$=0, $v$=0, $w$=1, $x$=1) corresponds to the relation of Figure 2.2b ($R_{12}$ in Figure 2.1). Not any 5-tuple of "1"s and "0"s represents a valid spatial relation between 1D intervals. If we deal with continuous intervals with non-zero duration, the underlying constraints are:

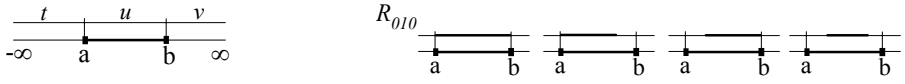
- at least one "1" must exist. If it is unique, it should not correspond to u or w (because in this case [z,y] collapses to a single point).
- all the "1"s must be consecutive (otherwise we refer to disconnected intervals).
- the intervals of interest must be a consecutive partition of $(-\infty,+\infty)$.


(a) Interval and regions of interest (b) Example of relation $R_{00011}$

**Figure 2.2** Encoding of spatio-temporal relations

The binary encoding can be extended in order to handle relations at varying resolution levels. We will initially illustrate its applicability to a coarse resolution level where only a few relations can be distinguished. In the

example of Figure 2.3a, the 1D regions of interest are (-∞,a), [a,b] and (b,+∞), respectively. The corresponding relations are of the form $R_{tuv}$, $t,u,v \in \{0,1\}$. This allows for the definition of only 6 relations since information content concerning the endpoints of [a,b] is reduced: $R_{100}$ (*before*), $R_{010}$ (*during*), $R_{001}$ (*after*), $R_{110}$ (*before_overlap*), $R_{011}$ (*after_overlap*), $R_{111}$ (*includes*). Figure 2.3b illustrates four configurations that correspond to $R_{010}$ and cannot be distinguished in this resolution.




(a) Interval and regions of interest (b) Examples of relation $R_{010}$

**Figure 2.3** Encoding at a coarse resolution level

Increasing the resolution of relations can be achieved simply by increasing the number of regions of interest. For instance, we can capture distance by refining *disjoint* relations, i.e., by splitting (-∞,a), (b,+∞) to several intervals. Figure 2.4 illustrates a simple partitioning that uses 9 bits and allows the distinction between *far* and *near* relations (*near* defined as being in a distance up to $\delta$ and *far* otherwise). An arbitrary number of distance refinements can be defined (for example a distance grid), according to the application needs. We call such consecutive partitionings of space *resolution schemes*.

The feasible relations at a particular resolution scheme are called *primitive* relations. In general, the fewer the binary variables, the coarser the resolution, and vice versa. If $b$ is the number of bits, the number of primitive relations in 1D is $b(b+1)/2 - k$, where $k$ is the number of point variables, i.e. intervals of the form [a,a]. If we fix the starting point at some bit then we can put the ending point at the same or some subsequent bit. There are $b$ choices if we fix the first point to the leftmost bit, *b-1* if we fix it to the second from the left, and so on. The total number is $b(b+1)/2$ from which we subtract the $k$ single-point intervals. For *b=9, k=4* we get the 41 relations of Figure 2.4, while for *b=5, k=2*, there exist 13 (Allen's) relations.

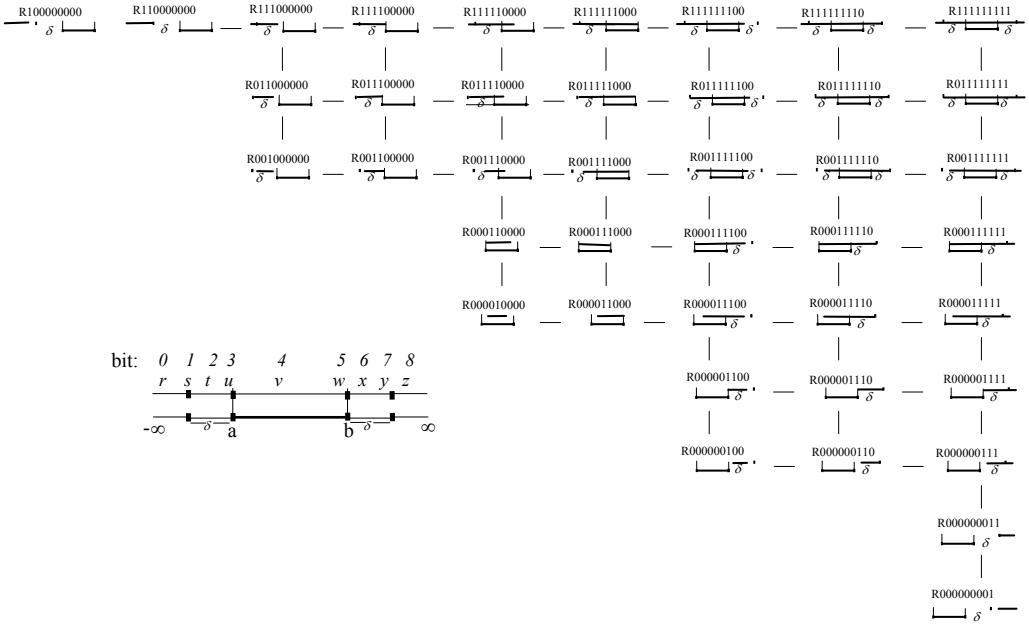



**Figure 2.4** 1D Conceptual neighborhood including distances (41 relations)

The new notation permits the automatic calculation of relation distances and, consequently, of similarity measures. Consider the neighborhood graph of Figure 2.4 where edges are arranged horizontally and vertically. The semantics of traversing the graph in either direction are captured by the following "pump and prune" rule of thumb: Given a relation $R_x$, there are 4 potential neighboring relations, denoted *right*($R_x$), *left*($R_x$), *up*($R_x$), *down*($R_x$), respectively, with the obvious topological arrangement in the graph. *Right*($R_x$) can be derived from $R_x$ by "pumping" an "1" from the right, i.e., finding the first "0" after the rightmost "1" and replacing it by a "1". *Left*($R_x$) can be derived from $R_x$, by "pruning" an '1" from the right, i.e. replacing the rightmost "1" by a "0". Similarly, *up*($R_x$) can be derived from $R_x$ by pumping an "1" from the left while *down*($R_x$) can be derived by pruning the leftmost "1". Notice that not all neighboring relations are always legal: the relation *up*($R_{110000000}$), for example, is not defined because the leftmost digit is a "1".

Since movement in the neighborhood graph is restricted to horizontal and vertical directions, the distance between two nodes is the sum of their vertical and horizontal distances. Equivalently, the distance between any two relations can be calculated by counting how many elementary movements we have to perform on an interval in order for the two relations to become identical. The larger the number of simple movements, the less similar the relations. The binary string representation enables automatic calculation of distances using the pseudo-code of Figure 2.5, which counts the minimal number of “0”s that have to be replaced with "1"s in order to make the two strings identical (*leftmost_1*($R$) returns the position of the leftmost bit that contains 1). For example $d(R_{000110000}, R_{010000000}) = 5$ and $d(R_{000110000}, R_{110000000}) = 6$ (the underlined 0s are the ones counted during the calculation of distance). The distance between a relation $R$ and a relation set $\{R_1,\ldots,R_i\}$ equals the minimum distance between $R$ and any of $R_1,\ldots,R_i$ (e.g., $d(R_{000110000}, \{R_{010000000}, R_{110000000}\})= 5$).

```
INT distance(relation R1, relation R2)
R = R1 OR R2;  /*bitwise OR */
d = 0;
FOR i:= leftmost_1(R) to rightmost_1(R) DO
        IF R1[i]=0 THEN d++;
        IF R2[i]=0 THEN d++;
RETURN (d);
```

**Figure 2.5** Distance calculation

This method does not need look-up information for computing similarity, and at the same time is very efficient since it is based on simple binary operations. Thus, users are not restricted to a predefined resolution but are free to employ different sets of constraints depending on their needs. The framework permits the uniform representation of several types of spatio-temporal relations (e.g., topological, directional, distance) and, as we show in the next section, the encoding and distance calculation can be extended accordingly to multi-dimensional spaces.

### 2.2 Multi-dimensional Extensions

A $D$-dimensional relation is defined as a $D$-tuple of 1D *projections*. We denote with $R{\downarrow}p$ the *projection of R* on $p$, e.g. $R_{000001100\text{-}100000000}{\downarrow}x = R_{000001100}$. For axis $x$ we assume a west-east direction, while for $y$ north-south

(according to the co-ordinate system used for the computer screenshots). In order to derive a neighboring relation we have to replace one of the constituent 1D projections with its neighbors. As a result, computing *D*-relation distances is reduced to the already solved problem of computing 1D distances. In this paper we calculate the distance between two multi-dimensional relations by summing up the distances on each dimension (other metrics, such as Euclidean [NNS96], can also be applied). Figure 2.6 shows the 2D neighborhood for the distance-enhanced resolution scheme of Figure 2.4. In this "fractal" graph, 41 conceptual neighborhoods corresponding to one dimension are linked, forming a higher level conceptual neighborhood for the other dimension (each node in the big neighborhood graph is a small neighborhood graph). As illustrated in the magnified portion of the graph, each line corresponds to a complete set of 41 connections between 2D relationships.

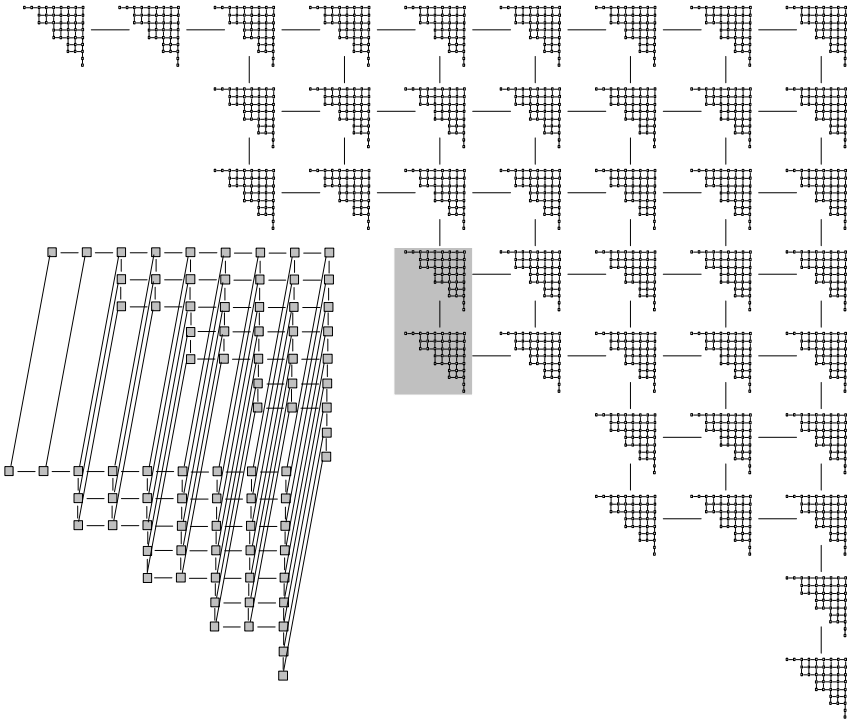
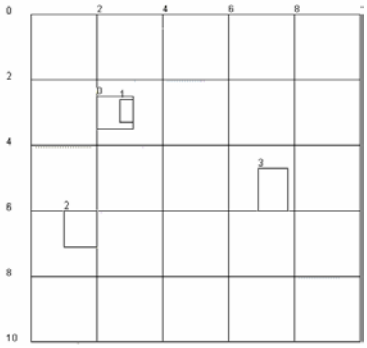
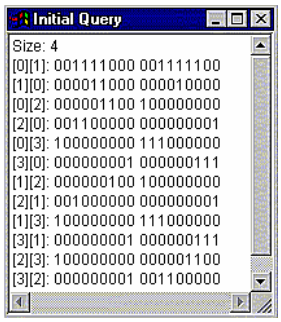


**Figure 2.6** 2D neighborhood graph for the distance-enhanced scheme

The framework can be easily applied for approximate retrieval of spatio-temporal queries. As a 2-dimensional example consider that a user is looking for all configurations of four objects that match the query of Figure 2.7(a) (this type of queries will be discussed in Section 5). The prototype configuration is drawn using a *query-by-sketch* language where the distance of the grid is set to δ. δ can be tuned to match application and user needs; for instance, a user may specify δ as 5% of the global extent per axis, while another may specify multiple δ′s of possibly different lengths. The same retrieval mechanisms are applied for both cases since the underlying data are stored using absolute co-ordinates from which the relations between stored objects are computed on-the-fly depending on the resolution scheme for a particular query. For simplicity, in the following examples we use the distance-enhanced resolution of Figures 2.4 and 2.6.




(a) Query




(b) Binary constraints

**Figure 2.7** An example 2D application

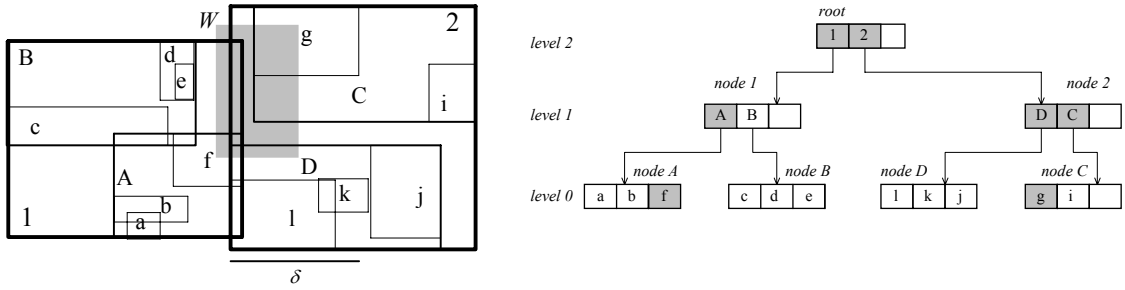
Figure 2.7(b) illustrates the set of binary constraints between all pairs of objects for the query of Figure 2.7(a). For instance, given that the direction of *y* axis is from north to south, the relation between query objects 0 and 2 is $R_{000001100\text{-}100000000}$ (the first object (0) is the *primary* object, while the second one (2) is the *reference*). If there does not exist a configuration of four objects identical to the input in some stored image, then the system should retrieve the ones that match the query constraints closely. The output should have an associated "score" to indicate its similarity to the query, which by adoption, is inversely proportional to the degree of neighborhood. Similarly the framework can be extended to capture 2D objects + 1D time (e.g., motion queries in Section 6) or 3D objects + 1D time. Depending on the application needs some dimensions (e.g., time) may be tuned at different resolution without affecting the applicability of the proposed methods.

In the rest of the paper we show how the framework can be employed for various types of spatio-temporal retrieval. We assume databases indexed by R-trees which store the minimum bounding rectangles (MBRs) of the actual objects (an assumption which is true for many commercial systems). Since MBRs are projection-based approximations, the above projection-based definitions of relations and similarity measures are particularly suitable for implementation in real systems.

## 3. OBJECT RETRIEVAL

The predominant access method for multi-dimensional data is R-trees [G84] and their variations, which are currently used in many commercial DBMSs, like Illustra, Postgress, Mapinfo etc. The R-tree data structure is a height-balanced tree that consists of intermediate and leaf nodes (R-trees are direct extensions of B-trees in many dimensions). The MBRs of the actual data objects are stored in the leaf nodes and intermediate nodes are built by grouping rectangles at the lower level. Figure 3.1 illustrates an image containing objects *a, b,.., l* and the corresponding R-tree. MBRs *a*, *b* and *f* are grouped together in a node A, which is pointed by intermediate node 1. In the rest of the paper, we make the distinction between an R-tree node N[i] and its entries $N_k$, which correspond to MBRs included in N[i]. $N_k$.ref points to the corresponding node N[k] at the next (lower) level. A leaf entry is an object MBR $r_k$. For instance, at level 1, the entries of node 1 are A, B,

which point to nodes at level 0. $N_k$,$l$ ($r_k$,$l$) and $N_k$,$u$ ($r_k$,$u$) represent the lower left point and the upper right point of $N_k$ ($r_k$), respectively.




(a) Image (b) R-tree

**Figure 3.1** A set of objects and the corresponding R-tree

### 3.1 Exact Retrieval - Window Queries

Traditionally R-trees have been used for *window queries,* which ask for a set of objects that intersect a window W (the reference object). The processing of a window query (e.g., light grey window in Figure 3.1(a)) in R-trees involves the following procedures: Starting from the top node, exclude the nodes that are disjoint with W, and recursively search the remaining ones (grey nodes in the tree of Figure 3.1(b)). Among the entries of the leaf nodes retrieved, select the ones that overlap W. Notice that even though the MBR of entry D intersects W, there is no solution MBR inside node D.

The fact that R-trees permit overlap among node entries at the same level, sometimes leads to redundant search in the tree structure. The $R^+$-tree [SRF87] and the $R^*$-tree [BKSS90] were proposed to address the problem of performance degradation caused by the overlapping regions and excessive dead-space. The $R^+$-tree achieves zero overlap among intermediate node entries by allowing partitioning of the leaf objects, whereas, the $R^*$-tree permits overlap among nodes, but tries to minimise it by organising rectangles into nodes using a more complex insertion algorithm than the original R-tree.

When the MBRs of two objects are *disjoint* we can conclude that the objects that they represent are also *disjoint*. If the MBRs however share common points, no conclusion can be drawn about the spatial relation between the objects. For this reason, spatial queries involve the following two step strategy [O86]:

1. *Filter step*: The tree is used to rapidly eliminate objects that could not possibly satisfy the query. The result of this step is a set of candidates which includes all the results and possibly some false hits.
2. *Refinement step*: The actual representation of each candidate (e.g., a set of points describing a polygonal shape) is retrieved from the database and tested for the satisfaction of the query using computational geometry techniques[2].

[2]The refinement step is performed by plane-sweep algorithms which identify whether two arbitrary polygons intersect in O($n$log$n$) time, where $n$ is the total number of edges in both polygons. Thus, it is more expensive than the filter step since finding whether two MBRs intersect requires only two comparisons per dimension. Notice that there is a trade-off between the two steps in the sense that by using finer approximations (e.g., convex hulls instead of MBRs) one can decrease the number of candidates (and the cost of the refinement step) at the expense of the filter step (which becomes more complicated) and storage (finer approximations require more than two points per object) [BKSS94]. Since MBRs are the most commonly used approximation, we follow this approach.

The two-step processing method has been extended to handle several types of queries: [PTSE95] applied R-trees for the retrieval of topological relations, [PT97] of direction relations, and [RVK95] of nearest neighbor queries. All the above methods deal with exact retrieval of objects that satisfy some spatial predicate (e.g. *inside, north, near*), with respect to some reference object. However, due to the fuzzy nature of some spatial predicates, the solution set is not always uniquely defined. For example, consider the following query: "find all objects *northeast* of *a* in Figure 3.1(a)". Depending on the user and the application, the answer may vary: object *g* is definitely *northeast* of *a*, but also *k*, or even *b* may be considered as a solution. This uncertainty raises the need for approximate retrieval, which would retrieve similar, in addition to exact, matches.

### 3.2 Approximate Object Retrieval

The problem of approximate object retrieval can be stated as follows: Given a reference object r, a spatio-temporal constraint *C* (which can be a primitive relation or a disjunction), and a maximum distance τ, "find all (primary) objects V, whose relation *R* with r is such that $d(C,R(V,r))\leq\tau$". That is, in addition to the reference object and the desired relation, the user inputs the maximum allowed distance τ from the input constraint.

Initially, we will deal with exact object retrieval, using the aforementioned framework, and limit our discussion to the case where τ=0 and *C* is a primitive relation. The goal, given such a query, is to identify a *minimal* query window to guide search. Consider, for example, that r = *a* and $C = R_{000001100}$ (which could be interpreted as *right-near*). The objects V to be retrieved should intersect the regions [*a.u*, *a.u*] and (*a.u*, *a.u*+δ), shown in Figure 3.2.

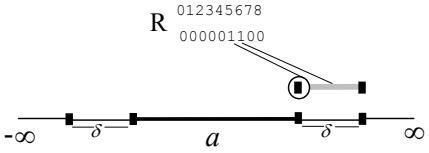


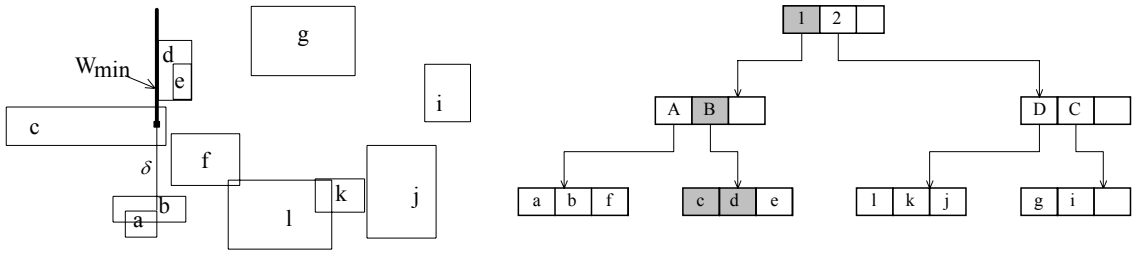

**Figure 3.2** Correspondence between 1-bits and regions

It is enough to search according to one of these regions, in order to retrieve all solution objects, plus possibly some false hits. Hence, we can answer the query in two steps:

1. Set as minimal window $W_{min}$, the smallest region defined by the 1s in *C*. The smallest regions are points which correspond to *odd* bits in *C*, e.g. for $R_{000001100}$ the minimal window is [r.*u*, r.*u*] which corresponds to the 5th bit. If there exist more than one odd bits, any of the corresponding points can be chosen as $W_{min}$. The non-existence of an odd bit implies that there is a single even bit which becomes $W_{min}$.
2. Apply a window query using $W_{min}$ and filter out the results that do not satisfy *C* with respect to *r*.

We call the bit in *C* that identifies $W_{min}$, the *minimal intersection bit, IB*(*C*). As a 2-D example consider the image of Figure 3.3(a) and let r=*a* and $C=R_{000001100\text{-}100000000}$ (the constraint between query objects 0 and 2 in Figure 2.7(a)). As shown in the previous example, $W_{min}\downarrow x$ corresponds to [*a.u*, *a.u*]. On the y-axis the only 1 is at position 8, thus $IB_y(C)=8$ and $W_{min}\downarrow y$ is set to $(a_y.u + \delta, +\infty]$. The thick line over *a* corresponds to $W_{min}$ for the 2D query. Assuming that the objects are organized in the R-tree of Figure 3.1(b), 3.3(b) shows the

search path if we apply the 2D $W_{min}$ window query. The query returns the candidates $\{c,d\}$, from which only $d$ satisfies $R_{000001100\text{-}100000000}$ with respect to $a$.




(a) Image and window queries (b) R-tree search path when applying $W_{min}$ query

**Figure 3.3** Example query (r=$a$, and $C$= $R_{000001100\text{-}100000000}$)

Now let $\tau>0$. A naive approach to solve the problem is to find all neighbor relations $\{R_i \mid d(R_i, C)\leq \tau\}$, transform them to $W_{min}$ queries, and take the union of the results. This method is expensive, since we have to apply as many tree searches as the number of $\tau$-neighbors. An improved method (*containmentWin*) computes a single window W, extending $C$ by $\tau$ 1's to the left and right, and applies a containment query. Some objects inside the window are false hits; a refinement step keeps the objects whose relation with r is similar to $C$ within the input tolerance $\tau$. However, the containment window in some cases (e.g., high tolerance) can be as large as the whole space rendering this method inefficient. Therefore, next we propose a more sophisticated algorithm (*getBits*) that avoids the disadvantages of the above techniques.

The goal of *getBits* (Figure 3.4) is to find the smallest single window $W_{min}$ that intersects *all* potential solution objects, given $\tau\geq 0$. An equivalent problem is to find a minimal set of bits, that cannot be all 0 in a $\tau$-neighbour of $C$. All solution objects will intersect at least one of the regions defined by this set of bits, thus their union is the corresponding $W_{min}$. For every projection $p$, *getBits* returns a pair of bits ($b_L$, $b_R$) that determine the bounds of $W_{min}\downarrow p$ (the $p$-projection of $W_{min}$) with respect to the $p$-projection of r ($r\downarrow p$).

Let $len_1(C)$ be the number of 1s in $C$, and $b_C$ be the central 1 in the sequence of $len_1(C)$ bits. For instance, $len_1(R_{000111110})= 5$ and $b_C = 5$. The algorithm tests four cases:

i. $\tau < \lfloor len_1(C)/2 \rfloor$. In this case, there can be no $\tau$-neighbour of $C$ having $b_C=0$. Thus we can set $b_L = b_R = b_C$, and the minimal intersection window will correspond to the area defined by $b_C$ and $r$. For instance, for $R_{000111110}$ and $\tau=1$, we choose $b_L = b_R = 5$, and $W_{min}$ = [r.$u$, r.$u$].

ii. $\tau=\lfloor len_1(C)/2 \rfloor$. Here, if $len_1(C)$ is odd, we can use the central bit $b_C$ as above (it will be 1 in all $\tau$-neighbors of $C$). However, if $len_1(C)$ is even, all 1s in $C$, can be 0 in some $\tau$-neighbour of $C$; e.g., for $R_{000111100}$ and $\tau=2$, the 2-neighbor $R_{000110000}$ has $b_5=b_6=0$ and $R_{000001100}$ has $b_3=b_4=0$. In this case, we consider as $W_{min}$ the region defined by both central bits, as both cannot be zero, i.e. $b_L = b_4$, $b_R = b_5$ and $W_{min}$ =(r.$l$, r.$u$].

iii. $\lfloor len_1(C)/2 \rfloor<\tau<len_1(C)$. As in the even length case above, all 1-bits can be 0 in $\tau$-neighbours of $C$. Furthermore, $\tau$-$\lfloor len_1(C)/2 \rfloor$ determines to what extent we have to "pull" $W_{min}$ limits from the central bit $b_C$. For instance, for $R_{000111110}$, and $\tau=3$: $b_L = b_C - (\tau\text{-}\lfloor len_1(C)/2 \rfloor) = 4$, $b_R = b_C + (\tau\text{-}\lfloor len_1(C)/2 \rfloor) = 6$, and $W_{min}$ = (r.$l$, r.$u$+$\delta$).

iv. $\tau \geq len_1(C)$. In this case, all 1s in $C$ can become 0 in a $\tau$-neighbour of $C$. Positions $b_L$ and $b_R$ are decided by extending both leftmost and rightmost 1s of $C$ by $\tau$-$len_1(C)$. E.g., for $R_{000111110}$, and $\tau$=6, $b_L = leftmost_1(C) - (\tau\text{-}len_1(C)) = 2$, $b_R = rightmost_1(C) + (\tau\text{-}len_1(C)) = 8$, and $W_{min} = (r.l\text{-}\delta, +\infty)$.

In the pseudo-code of Figure 3.4 cases ii and iii above have been combined in one. *ObjectRetrieval* (Figure 3.5) calls *getBits* to calculate the limits of $W_{min}$ and performs a window query using $W_{min}$. Then it filters the result according to the input constraint and the tolerance $\tau$.

```
(2-tuple) getBits (Constraint C, int τ)
  b_C = ⌊(leftmost_1(C) + rightmost_1(C))/2⌋; /* position of (leftmost) central 1 in C */
  IF τ<⌊len_1(C)/2⌋ THEN /* case i */
        IF even(len_1(C)) AND even(b_C) /* even number of 1's and b_C is even */
            THEN b_L = b_R = b_C+1; /* minimal intersection window is set to the odd bit right of b_C */
            ELSE  b_L = b_R = b_C; /* minimal intersection window is set to b_C */
  ELSE IF (⌊len_1(C)/2⌋ ≤ τ < len_1(C)) THEN /* cases ii,iii */
        b_L = b_C - (τ-⌊len_1(C)/2⌋ );
        IF even(len_1(C))
            THEN b_R = b_C+(τ-⌊len_1(C)/2⌋)+1;
            ELSE b_R = b_C+(τ-⌊len_1(C)/2⌋);
  ELSE  /*case iv*/
        b_L = max(0, leftmost_1(C) -(τ-len_1(C)));
        b_R = min(8, rightmost_1(C) +(τ-len_1(C)));
  RETURN (b_L, b_R);
```

**Figure 3.4** *getBits* function

```
objectRetrieval(R-tree R, rectangle r, Constraint C, int τ)
  FOR each projection p DO
        (b_L, b_R) = getBits(C↓p, τ);
        W_min↓p = Calculate_Window(b_L, b_R, r↓p); /* calculate minimal intersection window using b_L, b_R */
  rectangleSet rset = R.windowQuery(W_min); /* apply window query using W_min */
  FOR each V ∈ rset DO
        IF d(C, R(V,r))≤ τ THEN output V;
```

**Figure 3.5** *objectRetrieval*

As an example, consider again that r=$a$, $C$= $R_{000001100\text{-}100000000}$, while $\tau$=2. The grey window $W_{min}$ in Figure 3.6 corresponds to the query window calculated by the algorithm. For both dimensions the fourth case applies ($\tau \geq len_1(C)$). The tuple ($b_L$, $b_R$) is (5,6) and (0,1) for the $x$- and $y$-dimension, respectively, e.g., for the $y$ axis the first two bits cannot be 0 at the same time in any 2nd degree neighbour of $R_{100000000}$. Thus, the corresponding $x$- and $y$- projections of $W_{min}$ are $[a.u, a.u+\delta)$ and $(-\infty, a.l-\delta)$. The window query retrieves $\{c,d,e,g\}$. From these rectangles, $c$ and $g$ do not constitute solutions ($d(C, R(c,a)) = 7$ and $d(C,R(g,a)) = 3$), whereas $d$ and $e$ are solutions with distances 0 and 1, respectively. Constraints involving disjunctions of primitive relations, can be processed using the above method after finding the leftmost and rightmost 1's in all constituent relations.

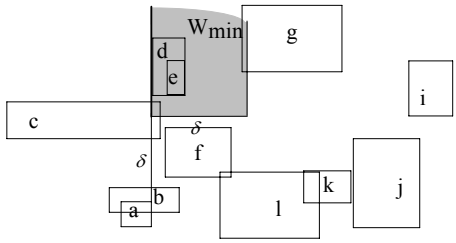



**Figure 3.6** Example of approximate retrieval (r=*a*, *C*= $R_{000001100\text{-}100000000}$ and τ=2)

We evaluated the efficiency of *getBits* by comparing it with the other two methods that can be applied for approximate object retrieval using our framework. For the following experiment we used the LB data-file [T94] which contains 53,145 rectangles representing road segments of Long Beach county. The maximum distance of the rectangles on each axis is 10000, and the data density 0.15. We inserted these rectangles to an $R^*$-tree of 4K page size. For each value of tolerance τ from 0 to 6 we generated a set of 50 queries, where r had a random position on the map and random length between 20 and 200 at each projection. Figure 3.7 shows the average number of page accesses caused by each method (no buffer scheme was used) as a function of τ. Clearly, *getBits* outperforms the other alternatives, in all cases. The number of window queries explodes with τ, and so does the complexity of the naive method. Moreover, the containment window used by the second method increases significantly with τ, and the whole dataset is traversed for large values of τ. On the other hand, *getBits* facilitates efficiency by minimizing the size of the single search window.

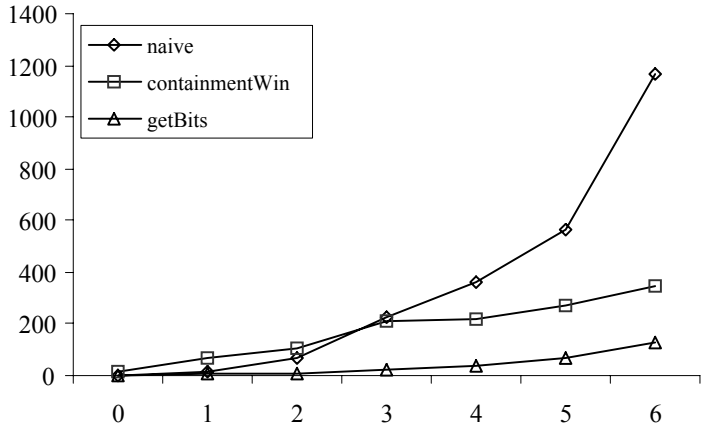



**Figure 3.7** Performance comparison of three retrieval methods

Summarising, this section constitutes the first approach on retrieval under uncertainty using the framework of Section 2. In particular we develop a projection-based technique, extensible to arbitrary dimensions, that determines some minimum intersection windows based on the input constraint and the tolerance τ. Retrieval is then performed in a traditional window query manner using the minimum windows and the output is ranked according to its distance from the input constraint. In the next section we discuss another important type of queries for spatio-temporal databases: spatial joins.

## 4. SPATIAL JOINS

The spatial join operation selects from two object sets, the pairs that satisfy some spatial predicate, usually *intersect* (e.g., "find all cities that are *crossed by* a river"). Previous work on pair-wise spatial joins, can be classified in two categories. The first one includes approaches which assume that the relations to be joined are indexed on the spatial attributes, an assumption which is true for most modern spatial databases because spatial indexing facilitates fast execution of selection queries. The dominant technique in this category [BKS93] presupposes the existence of R-trees for both relations and synchronously traverses the trees to report the join results. An alternative approach [R91] maintains a special index for spatial joins analogous to the relational join index [V87].

Methods in the second category do not presume spatial indexes for both relations. Some partition the space either regularly [PD96],[KS97], or irregularly [LR96], and distribute the relation objects into buckets defined by these partitions. The spatial join is then performed in a relational hash join fashion. Although, the above methods work well for uniform distributed inputs, they cannot guarantee good worst case performance. Another method [A+98], first applies external sorting to both files and then uses an adaptable plane sweep algorithm, considering that in most cases the "horizon" of the sweep line will fit in main memory. Finally, [LR94] join two inputs for which there exists only one R-tree index, by building a second R-tree using the existing one as a skeleton and then applying the method of [BKS93] to join them.

The methods of the second category are applicable when there do not exist indexes for both sets to be joined or when there is another operation (e.g., a selection) before join. However, when both sets are indexed by R-trees they have a disadvantage compared to the methods in the first category which utilise R-trees for efficient query processing. In this section we extend traditional R-tree based techniques for multi-relation approximation joins, by proposing algorithms and appropriate optimization methods.

### 4.1 Intersection Joins Using R-trees

The most influential algorithm for efficiently computing pair-wise, intersection joins using R-trees is presented in [BKS93]. *SpatialJoin* is based on the R-tree *enclosure property*: if two intermediate nodes do not intersect, there can be no MBRs below them that intersect. The algorithm (Figure 4.1) first joins the high level nodes and then follows the links in order to find qualifying pairs below them.

```
spatialJoin(Rtree_Node N[i], N[j])
FOR all N_l ∈ N[j] DO
  FOR all N_k ∈ N[i] with N_k ∩ N_l ≠ ∅ DO
      IF N[i] is a leaf page THEN  output (N_k, N_l)
      ELSE  /* intermediate nodes */
            ReadPage(N_k.ref); ReadPage(N_l.ref);
            spatialJoin(N[k], N[l])
```

**Figure 4.1** R-tree *spatialJoin*

Suppose that we want to join the level-1 subtrees 1 and 2 of the R-tree in Figure 3.1(b). *SpatialJoin*(1,2) will be recursively called for A and D at the next level, and finally will output the solution (*f,l*). Although the

version of Figure 4.1, assumes that the nodes to be joined are of equal height, the extension to different heights is straightforward [BKS93].

Two local optimization techniques are used to improve the CPU speed of the above algorithm. The first, *search space restriction*, reduces the quadratic number of pairs to be evaluated when two nodes N[i], N[j] are joined. If an entry $N_k \in N[i]$ does not intersect the MBR of N[j] (that is the MBR of all entries contained in N[j]), then there can be no entry $N_l \in N[j]$, such that $N_k$ and $N_l$ overlap. In the above example, entry B of node 1, does not intersect node 2, so it cannot intersect any entry inside 2. Using this observation, space restriction performs two linear scans in the entries of both nodes before starting the *spatialJoin* procedure, and prunes out from each node the entries that do not intersect the MBR of the other node. The second technique, based on the *plane sweep* paradigm [PS88], applies sorting in one dimension in order to reduce the overhead of computing overlapping pairs between the nodes to be joined.

In addition, [BKS93] employ a technique that uses *pinning* (or *page fixing*), a well known I/O buffer management method, to force page fetching according to the optimal order. In [HJR97], *spatialJoin* was extended by introducing an on-the-fly indexing mechanism to optimize the execution order of matchings at intermediate levels. [BKS94] study the multi-step processing of spatial joins using several approximations, while [BKS96] employ parallel execution.

### 4.2 Multi-relation Approximate Joins

The general problem of multi-relation approximate spatial joins is: Given two sets of objects (potentially indexed by two R-trees $R_i$, $R_j$), a spatial constraint $C_{ij}$, and a maximum distance $\tau$: find all pairs of objects $(V_i,V_j)$, $V_i \in R_i$, $V_j \in R_j$, such that $d(C_{ij}, R(V_i, V_j)) \leq \tau$. *SpatialJoin* is not directly applicable for the processing of this general type of spatial joins, because intermediate nodes that may contain solutions do not necessarily overlap. We study two alternative techniques to process joins using the framework of Section 2.

A first approach to process multi-relational spatial joins is to apply the *indexed nested loop join* algorithm [ME92], which is originally used for relational joins. In this adapted version of INLJ, all MBRs r in the *outer* object set $R_j$ are scanned sequentially, and for each r an object retrieval query is applied to find all objects $V_i$ in the *inner* object set $R_i$, such that $d(C_{ij}, R(V_i, r)) \leq \tau$. INLJ uses the *objectRetrieval* algorithm presented in section 3.

```
INLJ(Rtree Ri, Rj, Constraint Cij, int τ)
FOR each leaf MBR r ∈ Rj DO
        objectRetrieval(Ri, r, Cij, τ);
```

**Figure 4.2** R-tree indexed nested-loop join

The second algorithm extends the techniques proposed in [BKS93] to handle multiple relations. In order to use an arbitrary constraint as the join condition in *spatialJoin*, we need a mapping from relations, to *bounding conditions* between intermediate node entries that should be recursively joined. Figure 4.3 shows the bounding condition $BC_{ij}$ for $N_i$ given $N_j$. This condition is based solely on the positions of the leftmost and

rightmost 1's in $C_{ij}$. In particular, the leftmost 1, determines the position of $N_i.l$ with respect to $N_j.u$, while the rightmost 1 of $N_i.u$ with respect to $N_j.l$. Entries that do not satisfy these conditions can be excluded during search.

| | |
|---|---|
| $R_{1XXXXXXXX}$ | $N_i.l < N_j.u - \delta$ |
| $R_{01XXXXXXX}$ | $N_i.l \leq N_j.u - \delta$ |
| $R_{001XXXXXX}$ | $N_i.l < N_j.u$ |
| $R_{0001XXXXX}$ | $N_i.l \leq N_j.u$ |
| $R_{00001XXXX}$ | $N_i.l \leq N_j.u$ |
| $R_{000001XXX}$ | $N_i.l \leq N_j.u$ |
| $R_{0000001XX}$ | $N_i.l < N_j.u + \delta$ |
| $R_{00000001X}$ | $N_i.l \leq N_j.u + \delta$ |
| $R_{000000001}$ | $N_i.l$ unlimited |

(a) leftmost bit

| | |
|---|---|
| $R_{XXXXXXXX1}$ | $N_i.u > N_j.l + \delta$ |
| $R_{XXXXXXX10}$ | $N_i.u \geq N_j.l + \delta$ |
| $R_{XXXXXX100}$ | $N_i.u > N_j.l$ |
| $R_{XXXXX1000}$ | $N_i.u \geq N_j.l$ |
| $R_{XXXX10000}$ | $N_i.u \geq N_j.l$ |
| $R_{XXX100000}$ | $N_i.u \geq N_j.l$ |
| $R_{XX1000000}$ | $N_i.u > N_j.l - \delta$ |
| $R_{X10000000}$ | $N_i.u \geq N_j.l - \delta$ |
| $R_{100000000}$ | $N_i.u$ unlimited |

(b) rightmost bit

**Figure 4.3** Bounding condition $BC_{ij}$ for $N_i$

Assume, for instance, the query "find all objects $V_i$ and $V_j$ related by $R_{000000001}$" (i.e., $V_i$ is to the *right* and *far* of $V_j$). An entry $N_i$ is bounded with respect to $N_j$ by the following condition: ($N_i.u > N_j.l + \delta$). This bounding condition corresponds to the first row of Figure 4.3(b); the position of the leftmost bit (last row of Figure 4.3(a)) does not constrain $N_i.l$. Figure 4.4 illustrates an example: if $N_j$ is the intermediate node entry containing an object $V_j$, then the upper point of candidate entries for $N_i$ ($N_i.u$) should lie in the grey area. Entries like $N'_i$, not satisfying this constraint, cannot contain objects $V_i$. For approximate retrieval, bounding conditions are easily adapted to include $\tau$.

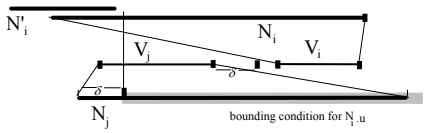



**Figure 4.4** Example of bounding condition for intermediate nodes

Using the above transformation, *spatialJoin* can be extended to handle multiple relations. Figure 4.5 illustrates the pseudo-code for *multi-relation spatial join* (*MSJ*). In this case, the desired relation $C_{ij}$, as well as $\tau$, are passed as parameters. Each $BC_{ij}$ is computed using $C_{ij}$ and Figure 4.3 (inverse conditions are also computed, but omitted here for simplicity). Leaf nodes constitute solutions, if they are related by a relation whose distance from $C_{ij}$ is $\leq \tau$. Intermediate nodes are recursively searched if they satisfy $BC_{ij}$. Initially MSJ is called with parameters the roots of the trees $R_i$ and $R_j$ to be joined.

```
MSJ(Rtree_Node N[i], N[j], Constraint C_ij, int τ)
FOR all N_l ∈ N[j] DO
  FOR all N_k ∈ N[i] DO
      IF N[i] is a leaf page THEN
          IF d(C_ij,R(r_k, r_l)) ≤ τ THEN output (r_k, r_l, d)
      ELSE /* intermediate nodes */
          IF BC_ij(N_k, N_l , C_ij, τ) THEN
              ReadPage(N_k.ref); ReadPage(N_l.ref);
              MSJ(N[k], N[l], C_ij, τ)
```

**Figure 4.5** Multi-relation spatial join

### 4.3 Optimization Methods

In order to enhance the performance of MSJ we have implemented a multi-relation version of the *space restriction heuristic*. The following *spaceRestriction* routine takes the entries N[i] one by one and tests them against N[j], eliminating the ones that do not satisfy the corresponding bounding conditions.

```
spaceRestriction(Rtree_Node N[i], N[j], Constraint C_ij, int τ)
IF N[i] is a leaf page THEN
      FOR all r_k ∈ N[i] DO
        IF NOT (LBC_ij(r_k, N[j], C_ij, τ))
            THEN exclude r_k from N[i];
ELSE   /* N[i] is a intermediate node */
      FOR all N_k ∈ N[i] DO
        IF NOT (BC_ij(N_k, N[j], C_ij, τ))
            THEN exclude N_k from N[i];
```

**Figure 4.6** Multi-relation space restriction

The bounding conditions of Figure 4.3 are used when N[i] is at an intermediate level. On the other hand, when N[i] is a leaf node (its entries are object MBRs) a more restrictive bounding condition can be applied. Consider that in Figure 4.7 we want to join objects in N[i] with all objects in N[j] with respect to $R_{000000001}$ (in Figure 4.4 we showed that N[0] satisfies the corresponding BC). Once we know the locations of each MBR in N[i] we can determine that some objects, such as $r'_i$, can be excluded - $r'_i$ cannot be related by $R_{000000001}$ with any MBR in N[j] because $r'_i.l < N[j].l+\delta$. If only the bounding conditions of Figure 4.3 were used, $r'_i$ would pass the space restriction test.

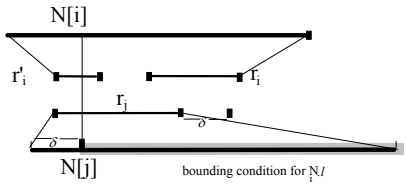



**Figure 4.7** Example of leaf bounding conditions

Figure 4.8 illustrates the complete set of *leaf bounding conditions* $LBC_{ij}$ between object MBRs and intermediate nodes. The bounding condition for the previous example is at the bottom row of the first table (the corresponding condition was *unlimited* in Figure 4.3).

| | |
|---|---|
| $R_{1XXXXXXXX}$ | $r_i.l < N[j].u - \delta$ |
| $R_{01XXXXXXX}$ | $r_i.l < N[j].u - \delta$, $r_i.l \geq N[j].l - \delta$ |
| $R_{001XXXXXX}$ | $r_i.l < N[j].u$, $r_i.l > N[j].l - \delta$ |
| $R_{0001XXXXX}$ | $r_i.l < N[j].u$, $r_i.l > N[j].l$ |
| $R_{00001XXXX}$ | $r_i.l < N[j].u$, $r_i.l > N[j].l$ |
| $R_{000001XXX}$ | $r_i.l < N[j].u$, $r_i.l > N[j].l$ |
| $R_{0000001XX}$ | $r_i.l < N[j].u + \delta$, $r_i.l > N[j].l$ |
| $R_{00000001X}$ | $r_i.l \leq N[j].u + \delta$, $r_i.l > N[j].l + \delta$ |
| $R_{000000001}$ | $r_i.l > N[j].l + \delta$ |

(a) leftmost bit

| | |
|---|---|
| $R_{XXXXXXXX1}$ | $r_i.u > N[j].l + \delta$ |
| $R_{XXXXXXX10}$ | $r_i.u > N[j].l + \delta$, $r_i.u \leq N[j].u + \delta$ |
| $R_{XXXXXX100}$ | $r_i.u > N[j].l$, $r_i.u < N[j].u + \delta$ |
| $R_{XXXXX1000}$ | $r_i.u > N[j].l$, $r_i.u < N[j].u$ |
| $R_{XXXX10000}$ | $r_i.u > N[j].l$, $r_i.u < N[j].u$ |
| $R_{XXX100000}$ | $r_i.u > N[j].l$, $r_i.u < N[j].u$ |
| $R_{XX1000000}$ | $r_i.u > N[j].l - \delta$, $r_i.u < N[j].u$ |
| $R_{X10000000}$ | $r_i.u \geq N[j].l - \delta$, $r_i.u < N[j].u - \delta$ |
| $R_{100000000}$ | $r_i.u < N[j].u - \delta$ |

(b) rightmost bit

**Figure 4.8** LBC that MBR $N_k$ must satisfy to pass space restriction

In addition, we apply a multi-relation plane sweep (MPS), which can deal with the whole set of relations of the current resolution scheme. MPS finds intersections of rectangles belonging to nodes N[i], N[j] in two steps:

1. first transforms the *x*-projection of each rectangle $N_l \in N[j]$ to a new one $N'_l$, according to $C{\downarrow}x$. This transformation is done so that: if $N'_l{\downarrow}x$ does not intersect on the *x*-axis with some entry $N_k{\downarrow}x \in N[i]$, then the original rectangles $N_l$ and $N_k$ cannot not be related by $C_{ij}$.
2. then it applies spatial sorting and plane-sweep to find all pairs ($N_k{\downarrow}x$, $N'_l{\downarrow}x$) that intersect. For each such pair it checks whether the corresponding pair ($N_k$,$N_l$) satisfies $C_{ij}$ and $C_{ji}$.

In order to perform the transformation, MPS chooses a bit, whose value is 1 on $C_{ij}{\downarrow}x$. This bit is the *minimal intersection bit* IB($C_{ij}{\downarrow}x$), described in section 3. For instance, consider that $C_{ij}{\downarrow}x = R_{000000011}$. We transform the reference interval $N_l \in N[j]$ to $N'_l$ as shown in Figure 4.9. If $N'_l$ (which is a single point) does not intersect some $N_k$ then the original intervals cannot satisfy $R_{000000011}$.

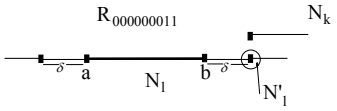



**Figure 4.9** An example transformation

The transformation differs for intermediate and leaf-level entries. The *x*-projection of each leaf level-entry $N_l$ is transformed to the area defined by IB($C_{ij}{\downarrow}x$) and $N_l$ (see Figure 4.10a). For intermediate node entries the transformation relies on different bounding criteria presented in Figure 4.10b. The first column of the tables correspond to IB($C_{ij}{\downarrow}x$) and the second to the transformed rectangle.

| | |
|---|---|
| 0 | $N'_l.l$=-∞, $N'_l.u$=$N_l.l$-δ |
| 1 | $N'_l.l$=$N'_l.u$=$N_l.l$-δ |
| 2 | $N'_l.l$=$N_l.l$-δ, $N'_l.u$=$N_l.l$ |
| 3 | $N'_l.l$=$N'_l.u$=$N_l.l$ |
| 4 | $N'_l.l$=$N_l.l$, $N'_l.u$=$N_l.u$ |
| 5 | $N'_l.l$=$N'_l.u$=$N_l.u$ |
| 6 | $N'_l.l$=$N_l.u$, $N'_l.u$=$N_l.u$+δ |
| 7 | $N'_l.l$=$N'_l.u$=$N_l.u$+δ |
| 8 | $N'_l.l$=$N_l.u$+δ, $N'_l.u$=+∞ |

(a) leaf nodes

| | |
|---|---|
| 0 | $N'_l.l$=-∞, $N'_l.u$=$N_l.u$ |
| 1 | $N'_l.l$=$N_l.l$-δ, $N'_l.u$=$N_l.u$-δ |
| 2 | $N'_l.l$=$N_l.l$-δ, $N'_l.u$=$N_l.u$ |
| 3,4,5 | $N'_l.l$=$N_l.l$, $N'_l.u$=$N_l.u$ |
| 6 | $N'_l.l$=$N_l.l$, $N'_l.u$=$N_l.u$+δ |
| 7 | $N'_l.l$=$N_l.l$+δ, $N'_l.u$=$N_l.u$+δ |
| 8 | $N'_l.l$=$N_l.l$, $N'_l.u$=+∞ |

(b) Intermediate nodes

**Figure 4.10** Transformation of x-axis projections

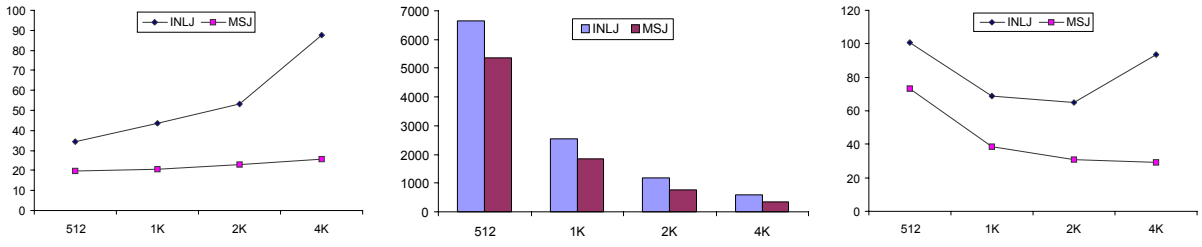
### 4.4 Experimental Evaluation

We have implemented and tested INLJ and MSJ in order to compare their performance. For the following experiments we used the LB data-file [T94] (described in Section 3.2). Figure 4.11 illustrates the data-set and two sample results of multi-relational self joins in magnified portions of the map. The left result corresponds to a join condition $R_{000111100\text{-}000111100}$ (light grey, dark grey rectangle), while the right one to $R_{000011100\text{-}000111000}$.


**Figure 4.11** LB dataset and sample retrieval results for multi-relational spatial self-joins

From LB we built several R*-trees of different block sizes, i.e. 512 bytes, 1K, 2K, and 4K. The LRU buffer size of the R*-trees was set to 128. An artificial set of 20 constraints was constructed. Then the self-join of the data set was computed using these constraints and the two algorithms. In order to avoid trivial queries, we excluded constraints with *far-disjoint* ($R_{100000000}$ and $R_{000000001}$). The distance limit δ, was set to 100. In all queries τ was set to 0. The implementation language was C++, and the experiments were run on a SUN UltraSparc2 (200MHz) workstation with 256 MB of RAM.

Figure 4.12 illustrates the performance of the algorithms for various R-tree page sizes. MSJ outperforms INLJ by means of both CPU-time (a) and number of I/O page accesses (b). The high cost of INLJ is due to the linear scanning of the outer file. The CPU-cost of both algorithms increases with page size. For INLJ this is due to the degeneration of the tree (the higher the level of the tree, the faster the search in terms of CPU-time). For MSJ, the slight increase of cost is due to the increase of the number of pairs that have to be joined for 2-specific nodes (this cost is in the worst case $C^2$, where C is the capacity of the node).




(a) CPU-time (sec) (b) I/O page accesses (c) Overall cost (sec)

**Figure 4.12** Performance of INLJ and MSJ for various block sizes

On the other hand the decrease of I/O page accesses pays off this computation cost and the overall efficiency of MSJ increases with the page size, as shown in Figure 4.12(c). Here, the overall cost of the algorithms has

been calculated by charging 10ms for each page access (a typical value [HJR97]). Observe that a page size equal to 2KB is the best for INLJ, while the performance of MSJ tends to stabilize for large page sizes.

Despite its inferior performance, INLJ can be very useful when there exists a multidimensional index only for the one of the two data sets to be joined. In this case, MSJ cannot be applied and INLJ remains the only alternative. Furthermore, as we show in the next section, algorithms based on a similar idea can be very efficient for structural query processing.

## 5. CONFIGURATION QUERIES

This section examines an alternative form of spatio-temporal information processing, namely, queries involving the retrieval of n-tuples ($n>2$) of objects that satisfy some *structure*. This type of retrieval pre-supposes that pre-processing techniques have been applied to extract information about the objects in a spatial scene and their locations. As an example consider the query of Figure 2.7a that asks for all configurations of four objects that match the input drawing. Alternatively the query could be expressed by an extended SQL language: *select* $V_0$, $V_1$, $V_2$, $V_3$, *from* Map, *where* $R_{001111000\text{-}001111100}(V_0,V_1)$ and $R_{000001100\text{-}100000000}(V_0,V_2)$ ..etc. Linguistic terms may be used instead of bitstrings e.g., *covers*$(V_0,V_1)$ instead of $R_{001111000\text{-}001111000}$. Although the particular query specifies relations between all pairs of variables, in some cases queries may be *incomplete* (some constraints may be left unspecified) or *indefinite* (constraints may be disjunctions of relations). Furthermore, in real applications some additional unary constraints may appear; these may specify object properties at the feature (e.g., $V_0$ is red) or semantic level (e.g., $V_0$ is a building). Although such constraints are easy to handle (provided that the corresponding properties have been extracted), for generality we omit them here and deal only with binary spatio-temporal ones.

Formally, a configuration (or otherwise, structural) query can be described as a binary constraint satisfaction problem [N89] (CSP) which consists of:

- A set of $n$ variables, $V_0,V_1,\ldots,V_{n-1}$ that appear in the query.
- For each variable $V_i$ a finite domain $D_i=\{r_0,\ldots, r_{N-1}\}$ of $N$ object MBRs. We assume that all domains are identical, i.e., each variable can be instantiated to any object.
- For each pair of variables $V_i,V_j$ a binary spatio-temporal constraint $C_{ij}$.

Figure 5.1 illustrates a solution where variable $V_0$ is instantiated to object 143, $V_1$ to object 207 and so on (the length of the grid is δ). A binary instantiation $\{V_i\leftarrow r_k, V_j\leftarrow r_l\}$ is *consistent,* if $R(r_k,r_l)\subseteq C_{ij}$. For instance, the constraint between $V_0$ and $V_3$ is $R_{100000000\text{-}111000000}$, which is also the relation between their corresponding instantiations (143,42) in Figure 5.1; therefore, $\{V_0\leftarrow 143, V_3\leftarrow 42\}$ is consistent. On the other hand, although the constraint between $V_0$ and $V_1$ is $R_{001111000\text{-}001111100}$, the relation between objects 143 and 207 is $R_{001111000\text{-}001111\underline{0}00}$; therefore the particular solution is approximate. The total distance of a *solution* $\{V_0\leftarrow r_p, \ldots, V_{n-1}\leftarrow r_r\}$ is the sum of all binary distances:

$$\sum_{\forall ij,\, 0\le i,j<n} d(C_{ij}, R(r_k,r_l)) \text{ where } \{V_i\leftarrow r_k,\, V_j\leftarrow r_l\}$$

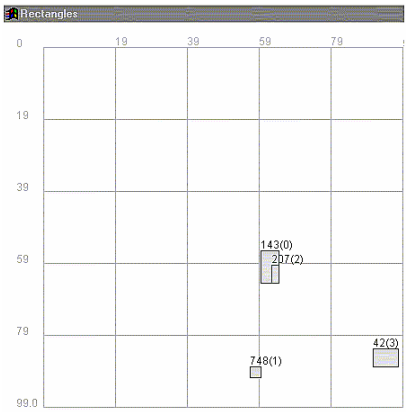



**Figure 5.1** A solution (to the query of Figure 2.7)

The maximum allowed total and pair-wise distances, T and τ, are submitted with a query in order to adjust the trade-off between the level of approximation and the cost of query processing. For instance, if T=6 and τ=2, only solutions that produce total relation distance ≤ 6 and pair-wise distance ≤ 2 will be retrieved. Obviously as T and τ increase, so does the number of solutions, but also the cost of query processing.

### 5.1 Forward Checking

Since configuration queries can be viewed as CSP problems, they can be processed by a variety of CSP algorithms. One of the most effective ones, is *forward checking* (FC) [HE80] which has been shown to perform very well in a wide range of problems involving "crisp" constraints [BG95]. FC must be modified for configuration queries in order to handle *soft* constraint processing using *T* and τ.

A "branch and bound" version of FC for the current problem works as follows: when a variable $V_i$ is assigned a value $r_k$, the domain of each *future* (un-instantiated) variable $V_j$ is pruned according to $r_k$ and the constraint $C_{ij}$, for all j>i. That is, all MBRs $r_l$ that produce a distance $d(C_{ij},R(r_k,r_l))> \tau$ are removed from the domain of $V_j$. The same happens for MBRs that produce global distance > T, taking into account the constraints between $V_j$ and all instantiated variables[3]. Consequently, when we reach instantiation level i (variables up to $V_i$ have been instantiated), the values of variables $V_0,\ldots,V_i$ will constitute a *partial solution*, and the domains of future variables will contain only values that may lead to a (complete) solution given the instantiations so far.

The procedure of pruning the domains of the future variables is called *check forward*. If, after a check forward the whole domain of a future variable is eliminated, the algorithm un-assigns the current variable's value, and restores the values of future variables, which were eliminated due to the current instantiation. When the domain of the current variable is exhausted the algorithm *backtracks* to the previous one and assigns a new value to it. FC outputs a solution whenever the last variable is given a value, and terminates when it backtracks from the first variable.

[3] The inverse constraints $C_{ji}$ are also considered but, for the sake of simplicity, we omit these tests in the rest of the paper.

In order to keep track of the allowable values for each variable at every instantiation level, FC uses a *n*x*n*x*N domain table*, where *n* is the number of variables and *N* the domain size. Each element of *domain*[*i*][*j*] is an array of *N* values that $V_j$ can take at different levels. Before FC starts, *domain*[0][*j*] is initialized to D for all variables. When $V_0$ is assigned an object $r_p$, *domain*[1][*j*] is computed for each remaining $V_j$, by including only objects $r_l \in$ *domain*[0][*j*] such that $d(C_{0j},R(r_p,r_l)) \leq \tau$. In general if $r_k$ is the current instantiation of $V_i$, *domain*[*i*+1][*j*] is the subset of *domain*[*i*][*j*] which is valid w.r.t. $C_{ij}$ and $r_k$. In this way, at each instantiation level the *domain*[*i*][*j*] of $V_j$ continuously shrinks; when we reach level j, $V_j$ gets instantiated from *domain*[*j*][*j*] which contains only MBRs compatible with the instantiations of all previous variables. If a value of $V_i$ results in the domain of some $V_j$ to become empty, a new value is chosen and *domain*[*i*+1][*j*] is re-initialized to *domain*[*i*][*j*]. The pseudo-code of a non-recursive version of FC which can be applied for configuration query processing is given in Figure 5.2.

```
FC-DVO(Query q, int τ, T )
FOR j = 0 TO n-1 DO domain[0,j] = D /*initialize all domains to D */
i = 0; /* index to the current variable */
WHILE (TRUE) {
    new_value := chooseNextValue(domain[i][i]);
    IF new_value = NULL THEN /* empty domain */
        IF i=0 THEN RETURN;
        ELSE i:=i-1; CONTINUE;  /*Backtrack*/
    ELSE  /* not empty domain */
        instantiations[i] := new_value; /*store instantiation*/
        IF i = n-1 THEN  /*last variable instantiated*/
            output_solution(instantiations);
        ELSE /* intermediate variable instantiated */
            IF checkForward(i) THEN /* successful instantiation*/
               DVO(i+1,n-1); /*var. with the smallest domain as next*/
               i := i+1; /* successful instantiation: go forward */
 }
BOOLEAN checkForward(int i)
 FOR j = i+1 TO n-1 DO /*for all uninstantiated variables*/
        domain[i+1][j]= domain[i][j];
        FOR all MBRs r_l ∈ domain[i+1][j] DO
            IF d(C_ij,R(instantiations[i],r_l)) > τ OR  T exceeded THEN /* value r_l is illegal for variable V_j */
                domain[i+1][j]= domain[i+1][j]-{r_l}; /*r_l is removed from the domain of V_j */
        IF domain[i+1][j]=∅ THEN RETURN FALSE; /* the domain of V_j becomes empty */
 RETURN TRUE;
```

**Figure 5.2** Branch and bound forward checking with dynamic variable ordering

Dynamic variable ordering (DVO) [BvR95] is a technique employed by several CSP algorithms to improve efficiency. The key idea behind FC-DVO is to reorder the future variables according to their domain size after "checking forward" at the current instantiation level. The variable with the minimum domain size becomes the next variable to be tested. In this way the number of search paths is minimized, because the variable with the smallest domain is the most likely to be pruned out; the algorithm will backtrack faster in the case that there is

no valid assignment after the current partial solution. DVO is responsible for changing the order of $V_1$ and $V_2$ in Figure 5.1.

FC-DVO has two drawbacks for the current application. First it is inapplicable for large spatial databases, because the 3D *domain table* cannot fit in main memory. The second drawback is the fact that it does not utilize the spatial indices which may exist for spatial relations. The incorporation of R-trees and appropriate query processing techniques can solve both these problems.

### 5.2 Window-Reduction Algorithms

Window reduction techniques combine the concepts of indexed nested loop join (section 4.2) and forward checking. The algorithms after instantiating a variable will use its value as a query window to restrict the possible values of subsequent variables. For instance, after assigning $V_0 \leftarrow 143$ (Figure 5.1), object 143 becomes the query window for values that will constitute the domain of $V_1$, avoiding unnecessary consistency checks. The pseudo-code of window reduction (WR) algorithm is presented in Figure 5.3.

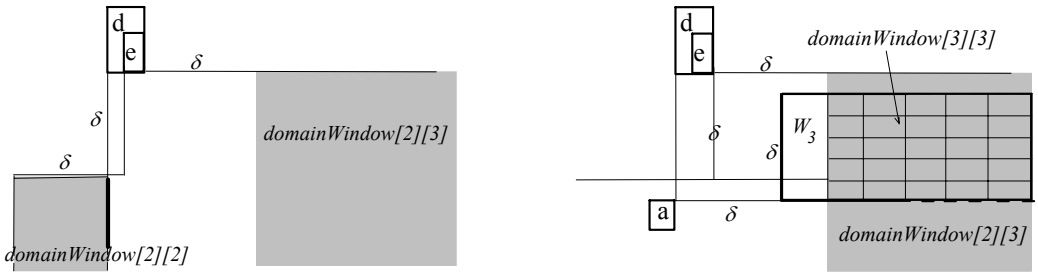
```
WR(Query q, int τ, T )
FOR j=0 TO n-1 DO domainWindow[0][j] = U;  /*Universal Space*/
i=0; /* index to the current variable */
WHILE (TRUE) {
    new_value := getNextValue(domainWindow[i][i]);
    IF new_value = NULL THEN /* empty domain */
        IF i=0 THEN RETURN;
        ELSE i:=i-1; CONTINUE;  /*Backtrack*/
    ELSE /* not empty domain */
        IF d(C_ij,R(instantiations[i],r_l)) > τ OR  T exceeded THEN
            CONTINUE /* invalid value inside domain window */
        ELSE    /* valid instantiation */
            instantiations[i] := new_value; /*store instantiation*/
            IF i = n-1 THEN  output_solution(instantiations); /* last variable instantiated */
            ELSE /* intermediate variable instantiated */
                IF windowReduction(i) THEN /* successful instantiation*/
                    Window_DVO(i+1,n-1); /*var. with smallest window next*/
                    i := i+1; /* successful instantiation: go forward */
}

BOOLEAN windowReduction(int i)
  FOR j = i+1 TO n-1 DO /*for all uninstantiated variables*/
      W_j = computeWindow(instantiations[i],C_ji,τ);
      domainWindow[i+1][j]= domainWindow[i][j]∩W_j;
      IF domain[i+1][j]=∅ THEN RETURN FALSE;
  RETURN TRUE;
```

**Figure 5.3** Window-reduction algorithm

In order to avoid the 3D *domain set* used by FC, WR maintains a *n*x*n domain window* that encloses all potential objects for each variable (and possibly some false hits). When $V_i$ takes a new value $r_k$, a new

window $W_j$ is computed for every un-instantiated variable $V_j$ taking into account $r_k$ and $C_{ji}$. The intersection of $W_j$ with (existing) *domainWindow*[*i*][*j*] is stored at *domainWindow*[*i*+1][*j*]. Figure 5.4(a) illustrates the domain windows for $V_2$ and $V_3$, assuming that the first two variables of the example query have been instantiated to *d* and *e* respectively. When $V_2$ is instantiated to *a* (Figure 5.4b), the constraint $C_{32}$ specifies that valid instantiations for $V_3$ should lie in $W_3$. The new *domainWindow*[3][3] for $V_3$ is the intersection of *domainWindow*[2][3] and $W_3$, i.e., it corresponds to the only area that may contain values consistent with both $\{V_0 \leftarrow d, V_1 \leftarrow e\}$ and $V_2 \leftarrow a$. The domain windows are computed in a way similar to *objectRetrieval*.




(a) $\{V_0 \leftarrow d, V_1 \leftarrow e\}$ (b) $\{V_0 \leftarrow d, V_1 \leftarrow e, V_2 \leftarrow a\}$

**Figure 5.4** Example of WR

If some domain window becomes null (empty intersection), the current instantiation is invalid and the algorithm proceeds to the next value for $V_i$. WR can be thought of as a "lazy" version of forward checking because the domain windows are calculated but no objects are retrieved until the variable gets instantiated. A drawback of this method is the fact that a possibly empty domain of $V_j$ (i.e., a window not containing any objects) cannot be detected until WR reaches instantiation level j and performs the window search. However, this disadvantage is counterbalanced by the smaller number of R-tree searches.

The next value for a variable $V_i$ is retrieved via *getNextValue*(), which uses *domainWindow*[*i*][*i*] as the query window for $V_i$. *GetNextValue*() does not perform a window query every time it is invoked, but the whole search path for each variable is maintained in memory. The overhead for this path-holding technique is pinning at most $n \cdot h$ pages - a small number for most applications. After a value is retrieved for $V_i$, the algorithm checks whether it is consistent with the previous instantiations since not all values that fall inside the domain window of $V_i$ are necessarily legal.

In addition to domain windows and path maintenance techniques, WR uses *DVO*: when the domain windows of the future variables are calculated after an instantiation, the variable with the smallest domain window becomes the next to be examined. This is led by the intuition that a small window is more likely to contain the least number of instantiations and minimize redundant consistency checks.

WR essentially searches the whole space in order to instantiate the first variable, but after doing so it performs only window queries which are cheap operations in R-trees (in this sense it is similar to INLJ). The disadvantage of blindly instantiating the first variable in the whole universe could be avoided by an algorithm that combines properties of multi-relation spatial join and window reduction. Join Window Reduction (JWR) first applies a pairwise spatial join to retrieve instantiations for the first pair of variables and then uses window

reduction to instantiate the rest of the variables. The subsequent variables are instantiated in the same way as WR:

```
JWR(Query q, int τ,T )
FOR j=1 TO n-1 DO domainWindow[1][j] = U; /*Universal Space*/
i=1; /* index to the current variable. Initially set to 1*/
WHILE (TRUE) {
  IF i=1  THEN /* values for first pair of variables (0,1)*/
        IF getNextPair(instantiations,q)=NULL THEN RETURN   /* termination */;
  ELSE    /* values of subsequent variables */
        new_value := getNextValue(domainWindow[i][i]);
        IF new_value = NULL THEN /* end of domain */
                i:=i-1; CONTINUE;  /*Backtrack*/
        IF d(Cij,R(instantiations[i],rl)) > τ OR  T exceeded
                        THEN CONTINUE /* invalid value inside domain window */
                         ELSE instantiations[i] := new_value; /*store instantiation*/
  IF i = n-1 THEN  output_solution(instantiations);
  ELSE /* intermediate variable instantiated */
        IF windowReduction(i) THEN /* successful instantiation*/
                Window_DVO(i+1,n-1); /*var. with smallest window next*/
                i := i+1; /* successful instantiation: go forward */
  }
```

**Figure 5.5** Join window-reduction algorithm

Function *getNextPair*() assigns the next pair that satisfies the relations between the first two variables using *MSJ*. For calculating the first pair of variables to be joined we use statistical information about the number of occurrences of each relation in the data files. Relations that occur rarely prune search space more effectively than frequent ones. For instance, the constraint $R_{001111000\text{-}001111100}$ between $V_0$ and $V_1$ is more restrictive than the other relations, because only a few pairs of objects satisfy it in normal data distributions.

### 5.3 Multilevel Forward Checking

Multilevel forward checking (MFC), is another variation of FC that extends MSJ to deal with n-tuples instead of pairs. MFC finds all n-combinations of intermediate nodes (at each level of the R-tree) that may contain some solution objects and follows the references to the next level, until it reaches the leaves, where it outputs solutions. As an example consider the tree of Figure 3.1. The path to solution (*d,e,a,k*) of the example query is: (1,1,1,2) at the top, (B,B,A,D) at level 1 and (*d,e,a,k*) at level 0.

The calculation of combinations of the qualifying nodes at each level (e.g., (1,1,1,1), (1,1,1,2), …., (2,2,2,2) for the top) is expensive, as their number can be as high as $C^n$, where $C$ is the capacity of an R-tree node. Although the search space is not prohibitively large (usually $n \leq 10$ and $C \leq 200$), the computational burden is due to numerous appearances of the problem during query processing. Finding the subset of node combinations which is consistent with the input query can be treated as a local CSP at each level. In particular the problem consists of:

- A set of $n$ variables, $V_0, V_1, \ldots, V_{n-1}$.

- For each variable $V_i$ a domain $D_i=\{N_0,\ldots,N_{I-1}\}$ of $I$ ($I \leq C$) potential values which correspond to *entries* in R-tree node N[i].
- For each pair of variables $V_i,V_j$ a binary constraint which: i] for intermediate nodes is a bounding condition $BC_{ij}$ derived from Figure 4.2 using the corresponding $C_{ij}$ and $\tau$, ii] for leaf nodes is a constraint $C_{ij}$ (disjunction of primitive relations).

The CSP in the case of the top level of the tree in Figure 3.1 has four variables $V_0,V_1,V_2,V_3$, which can be instantiated to entries 1 or 2 of the root. Consider the constraint $R_{000000001\text{-}001100000}$ between $V_3$ and $V_2$ (Figure 2.7). The bounding condition on the $x$ dimension for $R_{000000001}(V_3,V_2)$ is $BC_{32}$: ($N_3.u > N_2.l + \delta$) (example of Figure 4.4). The binary instantiation $\{V_2 \leftarrow 2, V_3 \leftarrow 1\}$ cannot lead to a solution at the lower levels because ($1.u < 2.l + \delta$). Therefore, all combinations (x,x,2,1) can be pruned out during search. The pseudo-code for MFC is shown in Figure 5.6.

```
MFC(Query q, Rtree_Nodes N[], int τ, T )
FOR j = 0 TO n-1 DO
    domain[0][j] = {N_l| N_l ∈ N[j]} /*N_l is an entry of N_j*/
i = 0; /* index to the current variable */
WHILE (TRUE) {
    new_value := chooseNextValue(domain[i][i]);
    IF new_value = NULL THEN /* end of domain */
        IF i=0 THEN RETURN;
        ELSE i:=i-1; CONTINUE;  /*Backtrack*/
    ELSE  instantiations[i] := new_value; /*store instantiation*/
    IF i = n-1 THEN  /*last variable instantiated*/
        IF (N[i] is a leaf page) THEN output_solution(instantiations);
        ELSE MFC(q, instantiations.ref, τ, T ) /*go to lower tree level */
    ELSE /* intermediate variable instantiated */
        IF checkForward(N[i].level,i) THEN /*valid instantiation*/
            DVO(i+1,n-1); /*var. with the smallest domain as next*/
            i := i+1; /*go to the next variable */
}
BOOLEAN checkForward(int level, int i)
FOR j = i+1 TO n-1 DO /*for all uninstantiated variables*/
   domain[i+1][j]= domain[i][j];
    FOR all objects r_l ∈ domain[i+1][j]
        IF (level = 0) THEN  /*leaf nodes*/
            IF d(C_ij,R(instantiations[i],u_l)) > τ OR T exceeded
                THEN domain[i+1][j]= domain[i+1][j]-{r_l};
        ELSE /*intermediate nodes*/
            IF NOT (BC_ij(instantiations[i],r_l))
                THEN domain[i+1][j]= domain[i+1][j]-{r_l};
    IF domain[i+1][j]=∅ THEN RETURN FALSE;
  RETURN TRUE;
```

**Figure 5.6** Multilevel FC

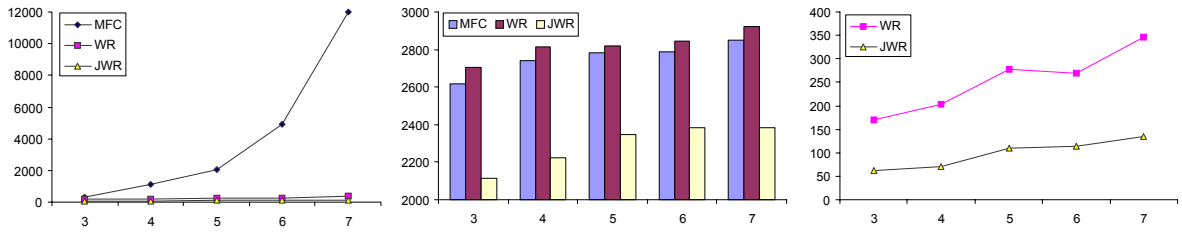
MFC applies *forward checking* to solve the CSP at each R-tree level: every time a variable $V_i$ is instantiated to an entry $N_k$, the algorithm eliminates all $N_l$ that do not satisfy $BC_{ij}(N_k,N_l)$ from the domains of each un-instantiated variable $V_j$. Initially N[] is set to an n-tuple that points to the tree root for all variables, i.e. N[i]=root, for i=0…n-1. A solution for the current tree level is found when the last variable is instantiated. The algorithm is then recursively invoked for the lower level, taking as parameter the n-tuple of the solution's references. Solutions are output if they refer to actual objects. MFC returns to the previous tree level when it backtracks from the first variable at the current level.

In the example of Figure 3.1, when the first valid combination (1,1,1,1) is found at the top, MFC will be called for the next level, trying to find a combination of nodes inside node 1 that satisfy all $BC_{ij}$ (the domain of all variables is now D={A,B}). If such a combination does not exist, as is the case here, it will backtrack to the top level and attempt to find another solution - assume (1,1,1,2). The new domains for the next call of MFC become: $D_0=D_1=D_2=\{A,B\}$ and $D_3=\{C,D\}$. A solution at this level is $\{V_0\leftarrow B, V_1\leftarrow B, V_2\leftarrow A, V_3\leftarrow D\}$. At the next call of MFC for level 0, the domains become $D_0=D_1=\{c,d,e\}$, $D_2=\{a,b,f\}$, $D_3=\{l,k,j\}$ and the solution $(d,e,a,k)$ is found.

### 5.4 Experiments

In order to compare the performance of the three algorithms (WR, JWR, MFC), we used the experimental set-up of the previous sections. We constructed 5 artificial sets of 30 queries: the number of variables in the queries of each set was fixed to 3, 4, …, 7. The distance between two variables on each axis did not exceed δ, which was set to 100.

Figure 5.7(a) shows the mean CPU-time and 5.7(b) the I/O page accesses averaged over all query-sets on the R*-tree with 1KB block size. WR and JWR clearly outperform MFC by orders of magnitude in terms of CPU-time. The performance gap widens with the query size because the domain windows in WR and JWR are continuously decreasing as new variables are instantiated. Moreover, empty window domains of the latter variables are detected early using the window reduction policy. On the other hand, the relaxed constraints between intermediate nodes do not permit MFC to prune the search space at the higher levels of the tree; thus, MFC cannot avoid the combinatorial explosion of possible instantiations as the number of variables increases. It is interesting to notice that MFC is better than WR in terms of page faults and this is due to the fact that WR instantiates the first variable in the whole space.

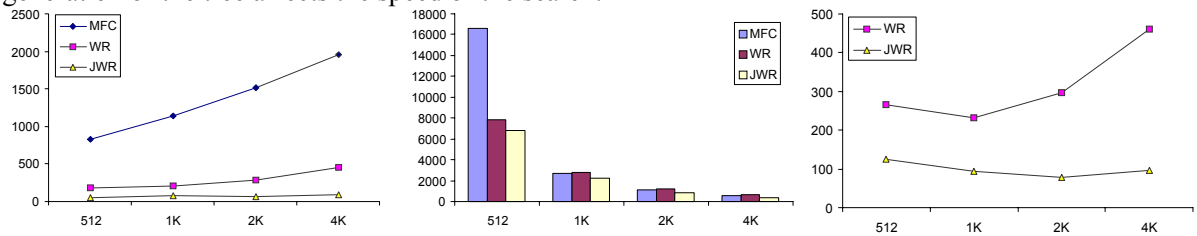



(a) CPU-time as a function of *n* (b) Page accesses as a function of *n* (c) CPU-time for WR - JWR

**Figure 5.7** Experimental evaluation 1

Figure 5.7(c) illustrates the relative CPU-time performance of WR and JWR (also for block size of 1K). JWR maintains a significant performance gain over WR. The performance gap is not affected by query size, because the only difference of the algorithms is the instantiation method for the first pair of variables.

In order to evaluate the algorithms for various block sizes we executed the 4-variable query set using R*-trees of 512, 1K, 2K, and 4K bucket sizes. CPU-time and page accesses are shown in Figure 5.8(a) and (b), respectively. Figure 5.8(c) shows the overall cost for WR and JWR, which was estimated by charging 10ms for each page access. The algorithms perform better for page size of 2K, while for larger sizes (4K) the degeneration of the tree affects the speed of the search.




(a) CPU-time as a function of block size (b) Accesses as a function of block size (c) Overall cost of WR and JWR

**Figure 5.8** Experimental evaluation 2

Another important observation from our experiments (not obvious in these diagrams) was the expected behaviour of MFC for almost all queries; the CPU-time was at the same levels depending only on the query size. On the other hand, the performance of WR and JWR was unpredictable: for instance the CPU time of WR may differ an order of magnitude for two different queries of the same size. This unstable behaviour is due to the fact that the resolution scheme may facilitate large reduction of the domain windows for some queries (e.g. *inside*), and not for others (e.g. *disjoint*).

Finally, we tested the performance of JWR over queries with non-zero degrees of inconsistency. In all experiments the $T$ was set to 10. Figure 5.9 illustrates the overall cost of JWR for the 2K page size R*-tree. Each line corresponds to a different value of local tolerance $\tau$. Because approximate retrieval is equivalent to exact retrieval using a larger window, the domain windows of JWR get larger as $\tau$ increases. Larger windows imply more potential legal values and more consistency checks.

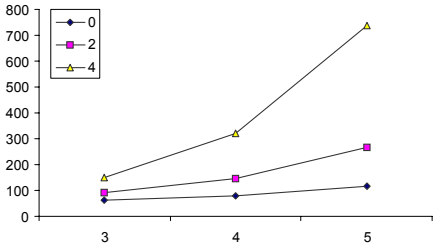



**Figure 5.9** Overall cost of JWR for partial retrieval

### 5.5 Discussion

Although the previous descriptions refer to retrieval from a single image, the extension to multiple images is straightforward: repeat the same process in each selected image. Let $n$ be the number of variables and $N_I$ be

number of objects in image $I$: in the worst case (exhaustive search), all $n$-permutations of $N_I$ objects (i.e., $N_I! / (N_I - n)!$) have to be searched in order to find solutions in $I$. In most applications where $N_I >> n$, this number is $O(N_I{}^n)$, meaning that the retrieval of configuration queries can be exponential to the query size. In order to avoid this problem, most related previous techniques (e.g., [GR95],[NNS96]) have focused on a specific instance where each image contains a spatial arrangement of the same set of known (labelled) objects. The goal is to find all images that contain a subset of the objects matching some given configuration. The problem in this case is polynomial ($O(n^2)$), since it suffices to locate all query objects in some image and then for each pair calculate its similarity to the input constraints.

Petrakis and Faloutsos [PF97] move a step further and employ spatial indexing to solve configuration queries for images that contain a constant number of labelled objects (e.g., lungs) and a small number of unlabelled ones (e.g., tumours). They map each image onto a point in multi-dimensional space, where each dimension corresponds to a relation between a specific pair of objects (the number of dimensions is quadratic to the number of objects), and engage R-trees for nearest neighbour retrieval. In order to keep the number of dimensions stable, images containing unlabelled objects are decomposed into combinations of images of fixed size. Although the above method (and feature-based methods, in general) is efficient for domains involving small images with few unlabelled objects (e.g., medical databases of X-rays) it is not applicable to large images of unlabelled objects, because of the potentially huge number of dimensions[4] (R-trees are not suitable for spaces of very high dimensionality [B$^+$98]) and the enormous number of sub-images generated by the decomposition of each image in smaller ones with a certain number of objects. In addition, the method can only be applied with a predetermined resolution scheme according to which the multi-dimensional index is built. On the other hand, our techniques do not make any assumptions about the size of images and the types of objects but solve the general problem assuming the same indexes as for window and join queries.

A number of methods are based on several variations of 2D strings, which encode the arrangement of objects on each dimension into sequential structures. *2DB* strings [LYC92] capture the object projections, effectively approximating each object by its MBR (similar to the approach taken here). *2DC* and *2DG* strings decompose objects in entities with disjoint convex hulls, allowing the representation of more detailed spatial information at the expense of storage [CJL89],[LH92]. Every database image is indexed by a 2D string; queries are also transformed to 2D strings and configuration similarity retrieval is performed by applying appropriate string matching algorithms [CSY87]. If the query contains only labelled objects, the cost of processing each image is linear, while in the general case it is exponential since matching has to be performed for multiple instantiations of the variables to different image objects. Unlike our methods, users are not allowed to define and use their own relations but only the scheme according to which 2D strings are built.

In case of non-indexed images with unlabelled objects, Papadias et al. [P$^+$99] propose retrieval heuristics for configuration similarity based on genetic algorithms, iterated improvement and simulated annealing.

[4] If all images contain $N$ objects and the resolution scheme defines $r$ relations, the number of dimensions would be $r \cdot N^2$, that is, for the distance-enhanced scheme and images of only 5 objects, the number of dimensions would be 1025.

Experimental evaluation suggests that these techniques, outperform forward checking for queries of the form "find one solution (or a small percentage of the solutions) with similarity above a target" or "find the best solution within a restricted time". Although they are efficient for retrieval involving numerous relatively small images (e.g., video clips, medical imagery), the methods cannot be applied for queries involving the retrieval of all solutions in large indexed images.

An alternative approach for processing configuration queries using underlying indexes is motivated by multiway spatial joins [MP99][5]. Consider the example query of Figure 2.7: MSJ can be applied for computing the join between $V_0$ and $V_1$ and between $V_2$ and $V_3$; the intermediate results may then be combined by some pairwise algorithm for non-indexed inputs. A problem with this approach is that algorithms for non-indexed inputs (see Section 4 for references) are developed for the *overlap* predicate and exact retrieval. Their extension to arbitrary constraints and approximate retrieval is complicated and outside the scope of this paper. Conversely, the proposed algorithms can be easily modified for multi-way spatial join processing. Papadias et al. [PMT99] describe formulae for the expected number of solutions in case of uniform datasets and overlap constraints, which are applied for optimization of joins using a combination of WR and MFC. The idea is similar to JWR, but instead of two, any number of inputs can be processed by MFC and then pipelined to WR. This technique, however, cannot be effectively employed for configuration similarity queries due to the lack of accurate estimations for the number of solutions in case of arbitrary relations. Park et al. [PCC99] proposed a modified version of MFC and several optimization techniques which is efficient for multiway spatial joins involving dense datasets and query graphs.

## 6. MOTION QUERIES

In this section we show how our techniques can be applied for the handling of motion queries. Motion can be defined as a temporal sequence of discrete phenomena called *frames*. Assuming an ordered set of frames representing any ordered collection of images of moving objects (e.g., satellite imagery), several queries may be of importance to a user, examples of which are given below:

1. Find the set of frames where a set of objects move from some initial positions to some destinations.
2. Find the set of frames where an object performs a specific movement with respect to a reference object.
3. Describe the movement of an object as a set of relation variances.
4. Which object moves (qualitatively) faster?
5. Given a set of frames, find a frame with a specific spatial arrangement.

The core of any motion query processor must include a mechanism that compares consecutive frames and decides whether they are similar enough to be regarded as "elementary" motion. Similarity between different movement patterns relies on several factors:

[5] Multiway spatial joins can be thought of as a special form of configuration queries, where the spatial constraint is *overlap* and retrieval is exact.

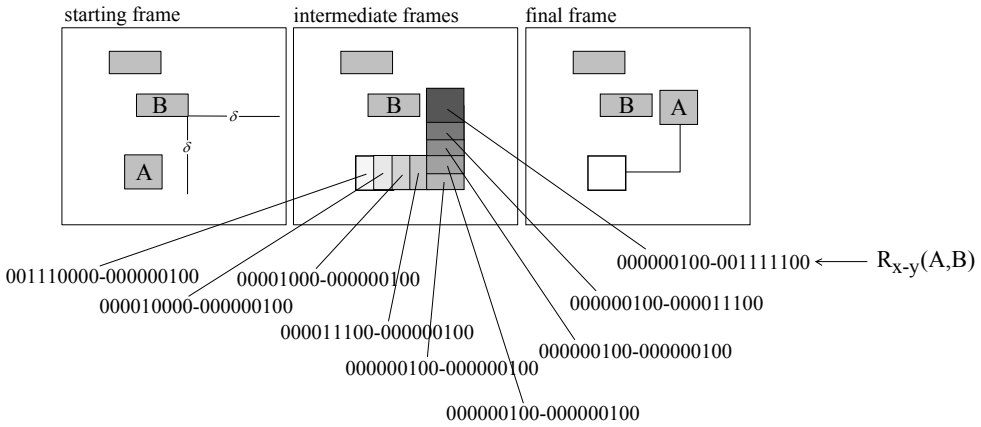



**Figure 6.1** Assessing movement by fixed reference

- the resolution scheme; e.g. a small object's movement along a large reference object will not be considered as motion unless a sufficiently refined resolution is adopted to distinguish among several *overlap* relations
- the sampling rate of frames, as it controls the perceived motion's smoothness
- the user's expectations and the application requirements

There exist various ways to elaborate on the identification of motion patterns. For instance, assume that we are interested in assessing "smooth motion" as opposed to arbitrary movement. Figure 6.1 illustrates the initial position of object A with respect to a reference object B, seven intermediate frames, and the final frame which shows A in its target position. The assessment of whether these frames constitute "smooth motion" is based on a comparison of relative objects' positions.

Let $R_i$ be the relation between A and B in frame $i$. Then a motion constraint can be defined as: $d(R_i, R_{i-1}) \leq \tau$, meaning that in order for an arbitrary movement to constitute motion the distance between the relations of A and B in two successive frames must be less or equal than a certain threshold (e.g., $\tau$=2). An obvious implicit constraint is that A is not allowed to have the same position in any two successive frames. The degree of smoothness can be indicated by several possible measures, one of which is the following:

$$S = \frac{\sum_{i=1}^{f-1} d(R_i, R_{i-1})}{f-1}$$ where $f$ is the number of frames

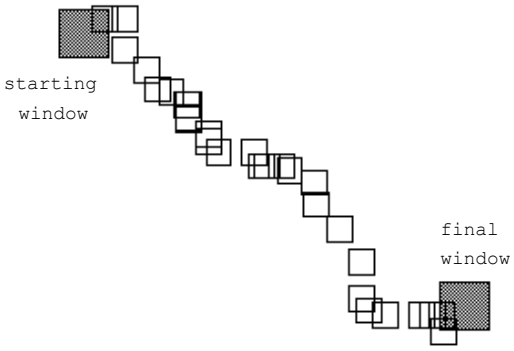
As long as the above constraints hold, the smaller the value of $S$, the smoother the movement in the corresponding set of frames. For the example in Figure 6.1 (at the distance enhanced resolution scheme), the value of $S$ is (2+0+2+2+0+0+2+2)/8 = 1.25, which could be less for a more dense sampling of frames.

In order to evaluate the cost of motion queries as a function of various input parameters, we constructed an artificial database aiming at simulating a satellite imagery application. The initial image (or frame) contains 5,000 distinct objects uniformly distributed in a square workspace with density 0.2. Between two successive images, each object moves at a maximum distance of 5% on each axis with probability 0.1 (essentially the majority of objects in two successive frames are in the same position, while a few are in neighboring ones). We have developed algorithms that use this database to process the following queries:

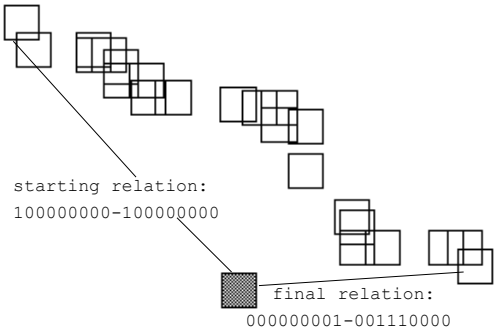
1. Given a well-defined starting window $W_1$ and ending window $W_2$, find all objects in a specific sequence of frames that have moved from $W_1$ to $W_2$. The query is processed by finding for each object $o_i$ the first frame $f_i$ and the last frame $l_i$ in the sequence where the object intersects $W_1$ and $W_2$, respectively. If $f_i$ is before $l_i$ then the object qualifies the query. In order to detect motion, we define a reference window $W_{ref}$ whose end-points are considered the average of the corresponding end points of $W_1$ and $W_2$, i.e., $W_{ref,1} = ½(W_{1,1} + W_{2,1})$, etc. We verify whether a candidate object moves, by comparing its relation to $W_{ref}$ in consecutive frames, as described above.
2. Given a (not necessarily static) reference object $o_j$, find all objects in a sequence of frames which have changed relative position with respect to $o_j$, according to a starting relation $R_{st}$ and an ending $R_{end}$. This query is processed in a similar way as the first one, by defining $W_1$ and $W_2$ using the position of $o_j$ at each frame. The reference object $o_i$ is used to detect motion.
3. Given a well-defined (static) reference window W, find the fastest object $o_i$ in a sequence of frames which has moved with respect to W, according to a starting relation $R_{st}$ and an ending $R_{end}$. Speed is defined as the number of frames between the starting and ending position of the move. This query is processed by first identifying the objects which have moved with respect to the reference window and then ranking them according to their speed. Again, the reference window W is used for motion detection.

Figure 6.2 illustrates retrieval results from the database. For the sake of presentation, we have drawn versions of the same object in a sequence of frames together. Figure 6.2(a) shows the trajectory of an object moving between two windows. Figure 6.2(b) shows the move of an object that changes relative position with respect to a static object, while in 6.2(c) the reference object is moving as well. Results of the third query type are depicted in Figure 6.2(d) and 6.2(e). In this example query, nine objects have changed relative position with the reference window W, according to the constraints. The move of the fastest one (Figure 6.2(d)) lasts 10 frames, whereas the slowest one (Figure 6.2(e)) moves for 48 frames. The move in some cases is too smooth to be visible in the corresponding figure.

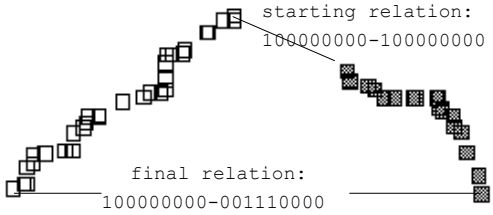
Since all types of queries are reduced to motion detection between two windows, we experimented only with queries of the first type. The results of the experiment can be used to draw conclusions for the other query types. We measured the response time (in seconds) of queries of the first type as a function of the number of frames (10, 20, …,100) and the ratio query_window_area/average_object_area (50, 100, …, 500). Figure 6.3 shows the results. Observe that query time increases linearly with the length of the sequence of frames where inside we seek for a move. It is also linear to the size of the windows $W_1$ and $W_2$ since the number of objects that lie in a window is linearly dependent to its area.



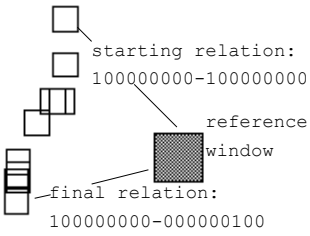

(a) motion between two windows



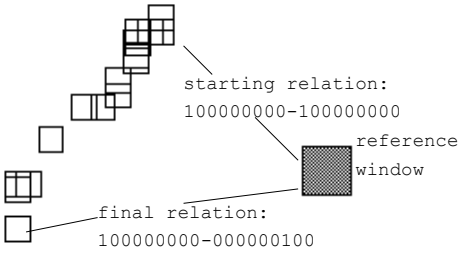

(b) motion with respect to a static object




(c) motion with respect to a moving object




(d) relative movement of a “fast” object




(e) relative movement of a “slow” object

**Figure 6.2** Sample retrieval results of motion queries

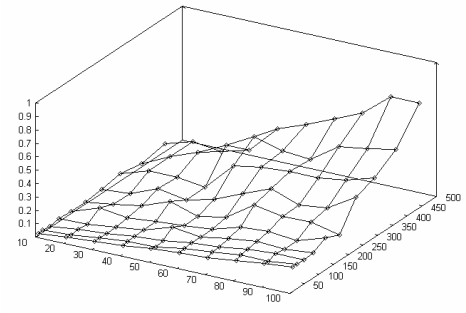



**Figure 6.3** Cost of simple motion queries

Apart from the three types of simple queries described above, more complex ones can be processed using our framework. Queries such as "Given a set of frames, find a frame with a specific spatial arrangement" are easily modelled as structural queries in multiple frames (instead of single images). Queries of the form: "Find all pairs of objects that perform a specific movement with respect to each other" can be processed as spatial joins, where the input constraints correspond to movement relations. This special type of motion query, called *motion join*, can be expressed as follows using the proposed framework. Given a sequence of frames, find all pairs of objects ($o_i$, $o_j$) having a starting relation $R_{st}$ at the first frame f, an ending relation $R_{end}$ at the last frame l and move in the intermediate frames. We have implemented three algorithms that process motion joins:

1. *OID join*. This method performs two multi-relational spatial joins (see section 4) at the two frames f and l, using the respective relations $R_{st}$ and $R_{end}$. It then sorts and merges the identifiers of the qualifying pairs in order to find the common object pairs that constitute the problem's solution.
2. *Join and verify*. This method performs a multi-relational join to one of the two frames, where the respective relation is expected to be more restrictive. We will explain later how the *constrainedness* of a relation is defined and used as a metric for this algorithm. Each qualifying pair of objects is then tested whether it satisfies the other relation in the less restrictive frame.
3. *Nested loops*. In this simple method each object pair is checked for satisfaction of both relations at both frames. Naturally, this method is expected to be more expensive than the other two, but we implemented it for the sake of comparison with them.

Verification of move in the intermediate frames is done in the same way for all methods. Figure 6.4 illustrates the relative performance of the three methods using the database described above. The y-axis shows the retrieval cost in seconds, whereas the x-axis captures the reverse constrainedness of a query defined as follows. Let size(f,*R*) be the expected output size (i.e. number of qualifying object pairs) of a multirelational self-join in frame f using relation *R*. This quantity can be easily estimated using selectivity formulae for spatial join queries. More specifically, it is defined as the area of the window defined by *R* and a random object in the frame using the methodology of WR (see Section 5.2), divided by the area of the workspace. Relations $R_{100000000}$ and $R_{000000001}$ naturally have the largest output size (i.e. they are the loosest). The reverse constrainedness of a motion join is defined as the minimum expected output size of the joins on the two frames, divided by the largest possible expected output size on a frame. Thus queries including only relations $R_{100000000}$ and $R_{000000001}$ are expected to have large reverse constrainedness and queries that include relations with the central bit on will have small reverse constrainedness. For each constrainedness value in the experiment we ran 100 queries and took the average.

As expected, *nested loops* presents a stable behaviour; its cost is independent on the constrainedness and much higher than the cost of the other methods. From the other alternatives, *join and verify* is the winner in all cases, having an almost stable performance difference with *OID join*. Since the number of objects in each frame is relatively small, verification was performed in memory with low cost. However, for large

applications verification may require access to secondary memory, thus *join and verify* may be less efficient than *OID join* in some cases.

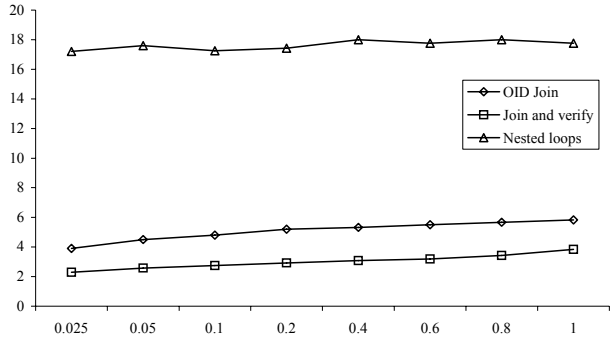



**Figure 6.4** Experimental comparison of motion join methods

Although not initially the main motivation for this work, this section touches briefly upon motion queries, treating them as special cases of spatio-temporal queries. The main goal of the section is to indicate the flexibility of our framework and its easy adaptation for various applications. We examined some obvious query cases entailing movement of objects and conducted a few experiments. However, the framework could be potentially used for more complicated motion analysis tasks, e.g. the identification of motion patterns like periodicity, global motion patterns vs. single-axis patterns, etc. This could be accomplished in conjunction with the use of string matching algorithms to identify patterns in both relation variances and relation strings.

## 7. CONCLUSION

In this work we have attempted an in-depth treatment of spatio-temporal queries. In addition to traditional spatio-temporal applications, their importance is even more stressed, considering them as a means to effectively retrieve information from large unstructured or semi-structured multimedia repositories like the WWW, where the condition is spatio-temporal structure instead of textual content. In essence, such queries represent similarity assessments among spatio-temporal configurations. Therefore, exact matches may not be sufficient for users. Instead, approximation (similarity) measures should be employed to relate each retrieved configuration with the queried one.

We have reduced this general similarity problem to elementary 1D relation similarity and, by borrowing concepts from spatio-temporal reasoning research, we have defined a formal yet practical framework for encoding 1D relations in a way that allows efficient reasoning on conceptual neighborhoods. We subsequently extended the model in a uniform way to arbitrary dimensions and multiple resolution levels with respect to the definition of relations, thus covering many potential applications. This logical representation proved effective and efficient for spatio-temporal retrieval, used in conjunction with appropriate data structures such as R-trees.

We applied the framework in three major types of spatio-temporal queries which have been the topic of active research in the database community: object retrieval, spatial joins and structural similarity, as well as to

a special class of motion queries as an indication of the flexibility of our approach. In addition to algorithms, we provided optimization methods and evaluated the performance of query processing through experiments with real data. Due to the lack of available higher-dimensional real data, we confined our experiments to 2D. This however does not undermine the validity of our approach, as similar indexing techniques are employed for 3D or 4D data (see [VTS98]), thus rendering our framework directly applicable.

Our techniques have a wide range of potential applications in various areas involving multi-dimensional data. A relative limitation of the approach is in its dependence on visual feature extraction algorithms, as our model assumes that images are pre-processed. For example, in order to assess motion in a set of satellite images, one has first to identify meaningful objects, define MBRs, index them, and subsequently apply our techniques. In some applications, objects are already identified when images are entered in the system (e.g., most VLSI and GIS applications), while in others identification can be done automatically due to domain restrictions (medical images).

Future continuation of this work is possible in both theoretical and practical directions. For example, the algebraic properties of different sets of relations that are feasible at different resolution levels could be studied and motivate the framework's extension to hierarchical relation similarity problems. From a practical point of view, a very fruitful research direction would be the coupling of our techniques with appropriate query languages for spatio-temporal domains (possibly a combination of pictorial and verbal languages). Finally, findings from the currently active research on indexing techniques for higher dimensions are expected to enrich the applicability range of our approach and improve its computational feasibility.

## REFERENCES


[A+98] Arge L., Procopiuc O., Ramaswamy S., Suel T., Vitter J. S. "Scalable Sweeping-Based Spatial Join". VLDB, 1998.

[A83] Allen, J., "Maintaining Knowledge About Temporal Intervals". CACM, 26(11), 1983.

[B+98] Berchtold S., Ertl B., Keim D. A., Kriegel H.-P., Seidl T. "Fast Nearest Neighbor Search in High-dimensional Space". ICDE, 1998.

[BB84] Ballard, D., Brown, C. "Computer Vision". Prentice Hall, 1984.

[BE96] Bruns, T.H., Egenhofer, M.J., "Similarity of Spatial Scenes". 7th Symposium on Spatial Data Handling, 1996.

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

[BKS93] Brinkhoff, T., Kriegel, H.-P., Seeger, B., "Efficient processing of spatial joins using R-trees". ACM SIGMOD, 1993.

[BKS96] Brinkhoff, T., Kriegel, H.-P., Seeger B. "Parallel Processing of Spatial Joins Using R-trees". IEEE ICDE, 1996.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles". ACM SIGMOD, 1990.

[BKSS94] Brinkhoff, T., H.-P. Kriegel, R. Schneider, and B. Seeger "Multi-step Processing of Spatial Joins". ACM SIGMOD, 1994.

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

[C+98] Chang, S.-F., Chen, W., Meng, H.J., Sundaram, H., Zhong, D. "A Fully Automatic Content-Based Video Search Engine Supporting Multi-Object Spatio-temporal Queries". IEEE Transactions on Circuits and Systems for Video Technology, Special Issue on Image and Video Processing for Interactive Multimedia, Vol. 8, No. 5, pp. 602-615, Sept. 1998.

[CJL89] Chang, S., Jungert E., Li., T. "Representation and Retrieval of Symbolic Pictures Using Generalized 2D Strings". *SPIE Visual Communications and Image Processing IV*, 1989.

[CSY87] Chang, S, Shi, Q., Yan C. "Iconic Indexing by 2-D String". IEEE PAMI 9(3), 413-428, 1987.

[DPM98] Delis, V, Papadias, D., Mamoulis, N. "Assessing Multimedia Similarity: A Framework for Structure and Motion". ACM SIGMM, 1998.

[EA92] Egenhofer, M.J., Al-Taha, K. "Reasoning about Gradual Changes of Topological Relations", International Conference GIS- From Space to Territory. Springer Verlag LNCS, 1992.

[F+94] Faloutsos C., Barber R., Flickner M., Hafner J., Niblack W., Petkovic D., Equitz W., "Efficient and Effective Querying by Image Content". Journal of Intelligent Information Systems,Vol. 3, No 3/4 , 1994.

[F92] Freksa, C., "Temporal Reasoning based on Semi Intervals", Artificial Intelligence, Vol 54, pp. 199-227, 1992.

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

[G93] Guenther, O. "Efficient computation of spatial joins". IEEE International Conference on Data Engineering, 1993.

[GR95] Gudivada, V., Raghavan, V. "Design and evaluation of algorithms for image retrieval by spatial similarity". ACM Transactions on Office Information Systems, 13(1):115-144, 1995.

[H94] Hernandez, D. "Qualitative Representation of Spatial Knowledge". Springer Verlag LNAI, 1994.

[HE80] Haralick, R.M., Elliott, G.L., "Increasing tree search efficiency for constraint satisfaction problems". Artificial Intelligence Vol 14, pp 263-313, 1980.

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

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

[LH92] Lee, S, Hsu, F. "Spatial Reasoning and Similarity Retrieval of Images using 2D C-Strings Knowledge Representation". Pattern Recognition, 25(3), 305-318, 1992.

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

[LR96] Lo M.-L., Ravishankar C.V. "Spatial Hash-Joins". ACM SIGMOD, 1996.

[LYC92] Lee, S, Yang, M, Chen, J. "Signature File as a Spatial Filter for Iconic Image Database". Journal of Visual Languages and Computing, 3, 373-397, 1992.

[M98] Maybury M. (ed.) "Intelligent Multimedia Information Retrieval". AAAI Press/MIT Press, 1998.

[ME92] Mishra, P., Eich, M. "Join Processing in Relational Databases". ACM Computing Surveys, Vol. 24, No. 1, 1992.

[MP99] Mamoulis, N, Papadias, D. "Integration of Spatial Join Algorithms for Processing Multiple Inputs". ACM SIGMOD, 1999.

[N89] Nadel, B. "Constraint Satisfaction Algorithms". Computational Intelligence, 5, pp. 188-224, 1989.

[NNS96] Nabil, M., Ngu, A., Shepherd, J., "Picture Similarity Retrieval using 2d Projection Interval Representation". IEEE TKDE, 8(4), 1996.

[NY96] Nabil A., Yelena Y., "Strategic Directions in Electronic Commerce and Digital Libraries". ACM Computing Surveys, Vol. 28, No 4, 1996.

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

[OS95] Ogle V., Stonebraker M., "Retrieval From a Relational Database of Images". IEEE Computer, Sept., 1995.

[P+99] Papadias, D., Mantzourogiannis, M., Kalnis, P., Mamoulis, N., Ahmad, I. "Content-Based Retrieval Using Heuristic Search". ACM SIGIR, 1999.

[PCC99] Park, H., Cha G., Chung C. J.M. "Multiway Spatial Joins using R-trees: Methodology and Performance Evaluation". 6th International Symposium on Spatial Databases.

[PD95] Patel J.M., DeWitt D.J. "Partition Based Spatial-Merge Join". ACM SIGMOD, 1996.

[PF97] Petrakis, E., Faloutsos, C. "Similarity Searching in Medical Image Databases". IEEE TKDE, 9 (3) 435-447, 1997.

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

[PMT99] Papadias, D., Mamoulis, N., Theodoridis, Y. "Processing and Optimization of Multiway Spatial Joins Using R-trees". ACM PODS, 1999.

[PS88] Preparata F, Shamos, M. "Computational Geometry". Springer, 1988.

[PT97] Papadias, D., Theodoridis, Y., "Spatial Relations, Minimum Bounding Rectangles, and Spatial Data Structures". International Journal of Geographic Information Systems, 11(2), pp. 111-138, 1997.

[PTSE95] Papadias, D., Theodoridis, Y., Sellis, T., Egenhofer, M. "Topological Relations in the World of Minimum Bounding Rectangles: A study with R-trees". ACM SIGMOD, 1995.

[R91] Rotem, D. "Spatial Join Indices". IEEE ICDE, 1991.

[RKV95] Roussopoulos, N., Kelley, F., Vincent, F. "Nearest Neighbor Queries". ACM SIGMOD, 1995.

[SC96] Smith J., Chang, S., "Searching for images and videos on the World-Wide Web". Technical Report CU/CTR 459-96-25, Columbia University, 1996.

[SK97] Seidl T., Kriegel H. "Efficient, User Adaptable Similarity Search in Multimedia Databases". VLDB, 1997.

[T94] TIGER/Line Files, 1994 Technical Documentation / prepared by the Bureau of the Census, Washington, DC, 1994.

[V87] Valduriez P. "Join Indices", ACM TODS 12(2): 218-246 , 1987.

[VTS98] Vazirgiannis, M., Thedoridis, Y., Sellis, T., "Spatiotemporal Composition and Indexing for Large Multimedia Applications". ACM/Springer Multimedia Journal, vol. 6(4), 1998.

**TABLE OF SYMBOLS**

| | |
|---|---|
| N[i] | node |
| $N_k$ | node Entry |
| $N_k$,$l$ | lower left point of the MBR of $N_k$ (also used for all types of rectangles) |
| $N_k$,$u$ | upper right point of the MBR of $N_k$(also used for all types of rectangles) |
| $R\downarrow p$ | projection of multi-dimensional relation $R$ on axis $p$ (also used for all types of rectangles) |
| $C$ | user constraint |
| d($R$,$C$) | distance between $C$ and relation $R$ (as defined by the conceptual neighborhood graph) |
| $V_i$ | query variable (also used to denote primary objects to be retrieved) |
| $C_{ij}$ | constraint between variables $V_i$ and $V_j$ |
| $r_k$ | object rectangle (also used to denote reference objects in window queries) |
| $V_i \leftarrow r_k$ | instantiation of variable $V_i$ to object $r_k$ |
| $d(C_{ij},R(r_k,r_l))$ | distance between $C_{ij}$ and $R(r_k,r_l)$ where $V_i \leftarrow r_k$ and $V_j \leftarrow r_l$ |
| $\tau$ | maximum $d(C_{ij},R(r_k,r_l))$ permitted by the query |
| T | maximum $\sum d(C_{ij},R(r_k,r_l))$ permitted by the query |
| *domain*[*i*][*j*] | set of consistent values for $V_j$ at instantiation level i (for FC and MFC) |
| *domainWindow*[*i*][*j*] | window containing the consistent values for $V_j$ at instantiation level i (for WR and JWR) |