Efficient Computation of Spatial Joins*

Oliver Günther
FAW Ulm, Postfach 2060, 7900 Ulm, Germany
guenther@faw9370.faw.uni-ulm.de

Abstract

Spatial joins are join operations that involve spatial data types and operators. Due to some basic properties of spatial data, many conventional join processing strategies suffer serious performance penalties or are not applicable at all in this case. In this paper we explore which of the join strategies known from conventional databases can be applied to spatial joins as well, and how some of these techniques can be modified to be more efficient in the context of spatial data. Furthermore, we describe a class of tree structures, called *generalization trees*, that can be applied efficiently to compute spatial joins in a hierarchical manner. Finally, we model the performance of the most promising strategies analytically and conduct a comparative study.

*Parts of this work have been carried out while the author was visiting the International Computer Science Institute and the University of California at Berkeley.

1 Introduction

Spatial databases have become a very active research topic recently, as many new database applications require spatial data types and operators. Originally, most emphasis has been put on CAD applications where the spatial data objects are mostly points or rectilinear lines and polygons, and where the operators are relatively simple (e.g. intersection of two rectangles). More recently, however, more complex applications have gained in significance. Examples include cartography and robotics, both areas that often make use of complex spatial objects, such as polyhedra or curves of complex shapes.

So far, spatial database research has concentrated on the data modeling aspects and on the design of access methods to support spatial searches. With regard to modeling, researchers have advocated both extended relational and non-relational object-oriented approaches. Regarding spatial searches, there now exists a large number of spatial access methods, and researchers have started to explore systematic ways to compare these structures both experimentally and analytically. Most methods concentrate on the computation of point and range searches, where the user specifies a search point or a search window, and the system is queried for all objects that contain or overlap the specified entity.

One important class of operators that has received comparatively little attention, however, is the class of spatial joins, where sets of objects in the database are interrelated using spatial operators. Examples include queries such as

(1) Find all Californian cities to the Northwest of Lake Tahoe or
(2) Find all houses within 10 kilometers from a lake

where *cities*, *lakes* or *houses* are database objects that are typically associated with some spatial entity (e.g. a polygon) to describe its shape, and *to the Northwest of* or *within 10 kilometers from* are spatial operators.

In the example above, query (1) is significantly easier to compute than query (2) because it relates *only one* object (*Lake Tahoe*) to the rest of the database. This type of query can be viewed as a *degenerate* spatial join: it is really just a *spatial selection*, and spatial access methods can be used efficiently to answer it. One could, for example, define a search range corresponding to *the Northwest of Lake Tahoe* and use R-trees [Gutt84] to compute the resulting search query. In query (2), however, this approach would not be feasible. This query is a *general spatial join* query, where two potentially large sets of database objects are related to each other. One possible strategy to answer this query would be to find out which objects are lakes or houses, and to examine all lake-house pairs whether the objects are within 10 kilometers from each other.

In contrast to a generic search query, where the search range is defined ad hoc by the user, a join query refers only to objects that are in the database already at the time the query is posed. This property of join queries has been utilized in traditional relational databases to speed up join processing, most notably by so-called *join indices* [Vald87]. In this paper we explore which of the traditional join strategies can be applied to spatial

joins as well. For simplicity, we assume a relational data model that is extended by spatial data types and operators. Prototypical systems with these features include POSTGRES [Ston86, Ston90] or DASDBS [Sche90]. Our results, however, are not specific for the relational data model and can be applied to object-oriented systems in a straightforward manner.

In section 2 we give a more concise definition of spatial joins and discuss how traditional join processing strategies can be applied in that context. In section 3 we discuss *generalization trees*, a class of tree structures that can be applied efficiently to compute spatial joins in a hierarchical manner. We give algorithms for both the spatial selection and the general spatial join problem. In section 4 we model the performance of the most promising strategies analytically and conduct a comparative study. Section 5 summarizes our results.

2 Joins and Their Computation

2.1 Relational Join Operations

Ullman [Ullm90] defines the $\theta-join$ of two relations R and S on columns i and j, denoted by $R \underset{i\theta j}{\bowtie} S$, as those tuples in the cross product $R \times S$ where the i-th column of R stands in relation θ to the j-th column of S. In practical applications, θ is often equality, which leads to the equijoin $R \underset{i=j}{\bowtie} S$. Tuples that join are often called *matching* tuples. Joins are usually followed by projections to drop redundant or unneeded columns.

For a typical example, consider the two relations

$customer(cno, cname, ccity)$, and
$order(custno, partno, quantity, date)$

If one wants to know the order data of all orders placed by customers in the city of *New York*, one selects those tuples from *customer* where *ccity* = "New York," yielding a relation *nycustomer*. Then one performs an equijoin $nycustomer \underset{1=1}{\bowtie} order$ to find the order data for each New York customer. Finally, one may perform some projection to strip the resulting relation from columns that are redundant (such as *custno*) or unneeded (such as *ccity*) to obtain the final result:

$nyorders(cno, cname, partno, quantity, date)$

To compute a join $R \bowtie S$ efficiently, several strategies have been proposed in the literature. The simple *nested loop* strategy checks each tuple in R against each tuple in S whether there is a match. Its performance is proportional to the product of the sizes of R and S, $|R| \cdot |S|$.

If the relations R and S can be sorted according to the tuple values in columns i and j, respectively, and if θ is a simple comparison operator, such as $=$, $>$, or $\leq$, then there

are much more efficient ways to compute the join. The *sort-merge* strategy first sorts R on column i and S on column j. Then R and S are merged and checked for matching tuples. The running time of this algorithm is proportional to $|R| \log |R| + |S| \log |S| + |J|$, where $|J|$ is the cardinality of the result of the join. Although $|J|$ may be $|R| \cdot |S|$ in the worst case, the sort-merge strategy usually represents a significant improvement over the nested loop strategy.

Another approach that takes advantage of the sortability of the columns involved is the class of *index-supported joins*. This approach can be applied if at least one of the relations involved (say R) has an index defined on the relevant column that supports the retrieval of matching tuples. Typical examples would be a relation with a B^+-tree and θ being a simple comparison operator ($=$, $<$, $\leq$, $>$, $\geq$), or a hashed relation with θ being equality. In that case one may scan the other relation (say S) and use the index on R to find the matching tuples for each tuple in S. If an index search takes time $\log |R|$, this algorithm results in a performance proportional to $|S| \log |R| + |J|$.

Finally, if the database does not encounter too many updates, it is usually worthwhile to precompute the result of frequent joins altogether and store it in a so-called *join index* [Vald87]. A join index is nothing but a two-column relation that stores the tuple IDs of matching tuples. Each join then corresponds to a simple look-up in the join index relation followed by a retrieval of the tuples involved from the disk. If one only counts join computation time, this strategy is highly effective and beats all other strategies in most cases. On the other hand, depending on the selectivity of the join the additional storage requirements may be considerable. Updates are not cheap, and in highly dynamic environments other strategies will catch up because their lower update costs outweigh the higher costs for join computation.

2.2 Spatial Joins

A join $R \underset{i\theta j}{\bowtie} S$ is called a *spatial join* if the i-th column of R or the j-th column of S are of some spatial data type, and if θ is a spatial operator. Spatial data types may include, for example, points, lines, polygons, polyhedra, or curves, and spatial operators may be, among others, distance-related operators (such as *within 10 kilometers from*) or operators referring to relative spatial location (*to the Northwest of*).

In example (2) from the introduction we discussed the query

Find all houses within 10 kilometers from a lake

which could be based on two relations

$house(hid, hprice, hlocation)$, and
$lake(lid, lname, larea)$

where *hlocation* is of data type *point* and *larea* of data type *polygon*. The query (2) then corresponds to a spatial join $house \underset{3\theta 3}{\bowtie} lake$ where θ is a point-in-polygon query: It

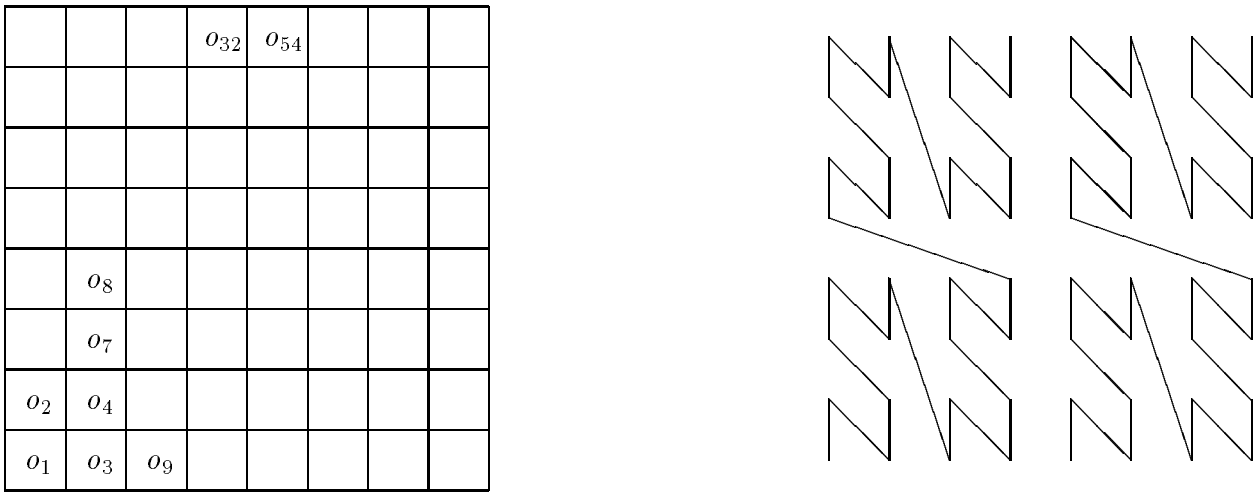

Fig. 1: A spatial grid with objects and a corresponding z-ordering

takes a point *house.hlocation* and a polygon *lake.larea* as input and is true if and only if *house.hlocation* is within the 10 kilometer buffer of *lake.larea*.

Of course, the above definition of a spatial join is still rather informal because there is no exact definition of a spatial data type and a spatial operator. In any case, the term *spatial* usually refers to objects and operators in a space of dimension 2 or higher. That implies, however, that they have the following key feature that makes the computation of spatial operators, including spatial joins and searches, significantly more difficult than the computation of their non-spatial counterparts:

There is no total ordering among spatial objects that preserves spatial proximity.

In other words, there is no mapping from two- or higher-dimensional space into one-dimensional space such that any two objects that are spatially close in the higher-dimensional space are also close to each other in the one-dimensional sorted sequence. For example, let us consider a common way of spatial sorting, the so-called *Peano curves* (also called *z-ordering*) [Oren86]. A two-dimensional example is given in Figure 1, where the space is divided into quadratic cells by means of a grid, and objects occupy one grid cell each. It is quite obvious that one can always find two objects, such as o_{32} and o_{54}, that are spatially close but far apart from each other in the Peano sequence.

This basic property of spatial data affects most of the traditional join strategies described above. *Sort-merge* often does not work at all because for many θ-operators there is no sort that makes sure that one catches all matching tuples during the following merge. For an example, consider again Figure 1 with θ meaning *adjacent*. Let the relation R contain the squares o_1, o_3 and o_4, and let S contain squares o_2, o_7, o_8 and o_9. With sort-merge, one first sorts R into the sequence (o_1, o_3, o_4) and S into the sequence (o_2, o_7, o_8, o_9). With a typical merge strategy, one obtains in sequence the matching pairs

(o_1, o_2), (o_4, o_2) and (o_4, o_7). The matching pair (o_3, o_9) remains undetected. Similar examples can be constructed for any other spatial ordering.

One notable exception from this rule is the θ-operator *overlaps*, for which a sort-merge strategy can be used rather efficiently. One possible implementation based on z-ordering has been described in detail by Orenstein [Oren86]. Note that any overlap is likely to be reported more than once. In the z-ordering implementation, for example, any overlap between two objects is reported once for each grid cell that the objects have in common.

Fig. 2: An R-Tree

The situation does not look quite as bad for the other join strategies. Of course, the *nested loop* strategy works although its lack of efficiency becomes even more obvious in the case of spatial data, where operators are usually much harder to compute than simple comparison operators on real numbers. *Join indices* can also be used although they lose some efficiency as well when applied to spatial data. First, updates become even more expensive because the computations involved are more complicated. Second, the efficient implementation of join indices, as described by Valduriez [Vald87], relies on an ordering along the join attributes, which can not be maintained in the spatial case. Nevertheless, as we will see in section 4, join indices remain a reasonably attractive option for the processing of spatial joins.

Another promising strategy is the *index-supported join*, based on the availability of spatial indices on one or more of the relations involved. Rotem [Rote91] has demonstrated the potential of this approach for the case of the grid file [Niev84], a spatial access method based on address computation. In this paper, we explore the feasibility and efficiency of tree-structured access methods with regard to spatial joins. More general, we do not restrict our discussion to a particular spatial database index, but to a whole class of tree structures that utilize *PART-OF* (also called *subset* or *containment*) *hierarchies* which are

characteristic of many spatial applications. The class investigated is called *generalization trees*, and it includes common spatial access methods such as the R-tree [Gutt84] as well as application-specific hierarchies that are typical for cartographic applications.

3 Generalization Trees

3.1 Description

A *generalization tree* is a tree structure where each node corresponds to a spatial object. Except for the root object, each object is completely contained in the object corresponding to its parent node. Different spatial objects, even objects corresponding to nodes at the same tree height, may overlap. Finally, the objects at any one height do not have to cover the total space, i.e., so-called *dead space* is allowed.

This very general definition includes several common spatial access methods, such as Guttman's R-trees [Gutt84], a structure that is based on a hierarchy of nested rectangles (Figure 2). In this case, the spatial objects corresponding to interior nodes are usually just technical entities that are of no interest to the user.

The definition of generalization trees also includes, however, application-specific hierarchies of detail that are typical for many cartographic applications. Consider, for example, the generalization tree in Figure 3, where the map is divided up into several disjoint regions corresponding to the countries on it. Each country is divided up further into smaller geographic entities that do not necessarily have to be disjoint (although they often are in this context). All nodes in this type of generalization tree represent objects that are relevant to the application.

Any such generalization tree structure can now be used to compute spatial joins in a hierarchical manner. In the remainder of this paper we assume that each generalization tree serves as a secondary index on a spatial column of exactly one relation. This is typically the case for generalization trees that are abstract indices, such as R-trees. The containment relationships between tree nodes then refer to the abstract spatial objects given by each particular tree structure. Note that they do not rely on spatial containment relationships between the actual application objects stored in the relation. This is an important point because in many applications, there may be only few containment relationships between objects in the same relation. In the relations *house* and *lake* given above, for example, there would most likely be none.

On the other hand, it should be noted that many generalization trees that are defined on application objects are likely to refer to objects from more than one relation. The tree given in Figure 3, for example, refers to objects that may belong to different relations, such as *country, state* and *city*. In this case, the algorithms presented in this paper are still applicable. In practice, however, the performance may well suffer because the data required for a join computation is potentially much less clustered than in the one-relation case.

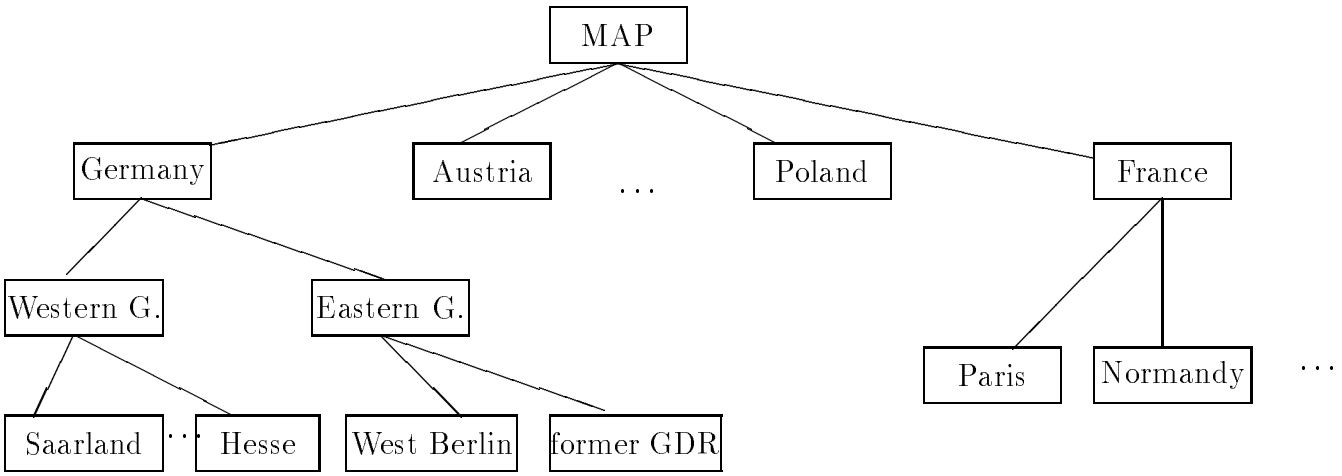

Fig. 3: A generalization tree representing a cartographic hierarchy

As in any algorithm based on the principle of *hierarchy of detail*, the idea is to examine higher levels of the tree first to see which branches may contain objects that are of interest to the join to be computed. For that purpose, it is useful to define an operator Θ, such that for two objects o_1' and o_2', $o_1' \Theta o_2'$ is true if o_1' and o_2' may have subobjects o_1 and o_2, respectively, such that $o_1 \theta o_2$. In that case it is necessary to go down the corresponding subtrees and to investigate the situation at a finer granularity. For the sake of this definition, each object is in particular considered its own subobject, hence $o_1 \theta o_2$ implies $o_1 \Theta o_2$, but not vice versa.

Clearly, the semantics of a Θ-operator is highly dependent on the semantics of the corresponding θ-operator. Table 1 gives Θ-operators for several common θ-operators. Two cases are illustrated in some more detail in Figures 4 and 5. Several θ-operators involve the notion of a spatial object's *centerpoint*. Here, the centerpoint may be taken to mean the object's center of gravity; in cartographic applications it is often defined explicitly by the user.

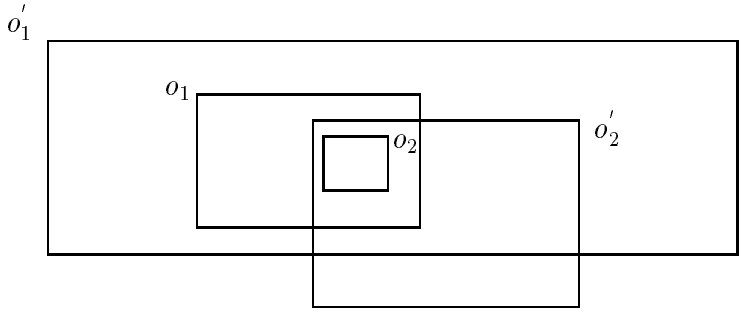

Fig. 4: o_1' overlaps o_2', and o_1 includes o_2

$o_1 \theta o_2$	$o_1' \Theta o_2'$
o_1 within distance d from o_2 (measured between centerpoints)	o_1' within distance d from o_2' (measured between closest points)
o_1 overlaps o_2	o_1' overlaps o_2'
o_1 includes o_2	o_1' overlaps o_2' (see Figure 4)
o_1 contained in o_2	o_1' overlaps o_2'
o_1 to the Northwest of o_2 (measured between centerpoints)	o_1' overlaps the NW quadrant formed by the right vertical and the lower horizontal tangent on o_2' (see Figure 5)
o_1 reachable from o_2 in x minutes	o_1' overlaps the x-minute buffer of o_2'

Table 1: θ- and corresponding Θ-operators.

3.2 Computation of Spatial Selections

Using generalization trees and Θ-operators, it is now straightforward to formulate algorithms for the computation of spatial joins. For the degenerate case of a spatial selection we obtain a simple top-down tree traversal. Let $GT_{R.A}$ denote the generalization tree that is defined as a secondary index on the relevant spatial column A of relation R. To obtain all tuples in R where the selection criterion $o\theta R.A$ is true for a given spatial object o, one first checks whether $o\Theta root(GT_{R.A})$. If yes, one does two things. First, one checks whether $o\theta root(GT_{R.A})$. If that is also true, the tuple corresponding to $root(GT_{R.A})$ is added to the solution set of matching tuples. Second, one searches the subtrees under $root(GT_{R.A})$ recursively. Note that in this formulation of the algorithm we allow for the

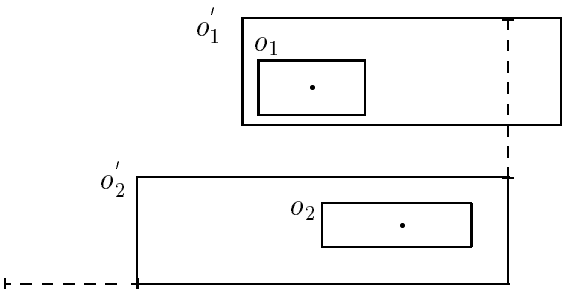

Fig. 5: o_1 to the Northwest of o_2

possibility that interior nodes correspond to application objects, i.e., objects that are of relevance to the user and may therefore qualify for the solution.

A more formal definition of the algorithm follows. Note that the root of a tree is considered at height 0.

Algorithm SELECT.

Input: A spatial data object o and a relation R that has a spatial attribute A with a generalization tree $GT_{R.A}$ as secondary index.

Output: The subset of those tuples in R where $o\theta R.A$.

SELECT1. [Initialization.] For $i := 1$ to $height(GT_{R.A})$ let $QualNodes[i]$ be the empty list. Let $QualNodes[0]$ be $[root(GT_{R.A})]$. Set $j := 0$.

SELECT2. [Tree Search.] Check all elements a in $QualNodes[j]$ whether $o\Theta a$. If yes, append all children of a to $QualNodes[j+1]$ and check whether $o\theta a$. If yes, add the tuple corresponding to a to the solution set of matching tuples. If $j < height(GT_{R.A})$, increase j by 1 and repeat step SELECT2, otherwise stop.

This algorithm basically corresponds to a breadth-first search of the tree. Of course, a depth-first search algorithm would also have been possible. The efficiency of depth-first vs. breadth-first depends on the physical clustering properties of the underlying generalization tree; see also section 4. Note that algorithm SELECT works regardless whether o is stored in relation R or not.

3.3 Computation of General Spatial Joins

Whereas spatial selection queries are basically answered by a simple spatial search algorithm, the general spatial join problem is somewhat more complicated. An efficient way to perform this computation is based on the repeated application of the algorithm SELECT. Let $GT_{R.A}$ and $GT_{S.B}$ denote the generalization trees defined on the relevant spatial columns A and B of relations R and S, respectively. In order to compute the spatial join $R \underset{A\theta B}{\bowtie} S$, one first performs selections for the two roots to find all objects a' in $GT_{R.A}$ where $a'\theta root(GT_{S.B})$ and all objects b' in $GT_{S.B}$ where $root(GT_{R.A})\theta b'$. Note that once again we allow for the possibility that interior nodes correspond to application objects and may therefore qualify for the solution.

In the course of these two spatial selections one also records for which *direct* descendants (children) a'' of $root(GT_{R.A})$ the condition $a''\Theta root(GT_{S.B})$ is true, and for which direct descendants b'' of $root(GT_{S.B})$ the condition $root(GT_{S.A})\Theta b''$ is true. For each qualifying a'' and each qualifying b'' one appends an entry (a'', b'') to the list $QualPairs[1]$. From there one proceeds down the two trees in a similar manner until all matching tuples have been found. A more concise description of the algorithm follows.

Algorithm JOIN.

Input: Two relations R and S with spatial attributes A and B, respectively, and generalization trees $GT_{R.A}$ and $GT_{S.B}$ as secondary indices.

Output: $R \underset{A\theta B}{\bowtie} S$.

JOIN1. [Initialization.] For $i := 1$ to $\min(height(GT_{R.A}), height(GT_{S.B}))$ let $QualPairs[i]$ be the empty list. Let $QualPairs[0]$ be $[(root(GT_{R.A}), root(GT_{S.B}))]$. Set $j := 0$.

JOIN2. [Tree Search.] Check all pairs (a, b) in $QualPairs[j]$ whether $a\Theta b$. If and only if that is the case, perform steps JOIN3 and JOIN4 for the pair (a, b). Once all pairs have been processed, check whether $j < \min(height(GT_{R.A}), height(GT_{S.B}))$. If yes, increase j by 1 and repeat step JOIN2, otherwise stop.

JOIN3. [Check for θ-Match.] Check whether $a\theta b$. If yes, join the corresponding tuples and add the resulting tuple to the solution.

JOIN4. [Spatial Selections.] Apply algorithm SELECT twice to find all descendants b' of b where $a\theta b'$, and all descendants a' of a where $a'\theta b$. For each *direct* descendant a'' of a where $a''\Theta b$ and each direct descendant b'' of b where $a\Theta b''$ add an entry (a'', b'') to $QualPairs[j + 1]$.

4 Performance Comparison

4.1 Modeling Assumptions

In this section we define cost formulas for the following three strategies to compute spatial joins:

I Nested Loop
II Generalization Trees
III Join Indices, as described in [Vald87]

In order to model the performance of these different strategies we make a few simplifying assumptions.

S1. All generalization trees are balanced k-ary trees of height n.
S2. Each generalization tree node corresponds to an object that is relevant to the user.
S3. For any two objects o_1 and o_2, $o_1 \Theta o_2$ implies $o_1 \theta o_2$. Because $o_1 \theta o_2$ always implies $o_1 \Theta o_2$, we have $o_1 \Theta o_2 \Leftrightarrow o_1 \theta o_2$.
S4. Join indices are implemented using B^+-trees.

Moreover, one has to make assumptions on the probability $\rho(o_1, o_2)$ that $o_1 \Theta o_2$ is true for two given objects o_1 and o_2. From there, one can deduce the probability σ_i that $o_1 \Theta o_2$ is true for two sibling nodes o_1 and o_2 at height i, and the probability π_{ij} that $o_1 \Theta o_2$ is true for any two objects o_1 and o_2 that are at heights i and j in their respective generalization trees.

In this paper we study the following four probability distributions. The parameter p ($0 \leq p \leq 1$) indicates the selectivity of the join. A low p indicates that only few tuples qualify for a match, i.e., the join is very selective. Figure 7 shows the probabilities $\rho(o_1, o_2)$ for o_1 being the leftmost leaf node.

UNIFORM. The probability that any two objects o_1 and o_2 match is constant, i.e., $\rho(o_1, o_2) = p$. This also means that $\sigma_i = \pi_{ij} = p$ for all sensible choices of i and j. This distribution is a reasonable approximation for such θ-operators as *to the Northwest of* that do not depend at all on spatial proximity.

NO-LOC. $\rho(o_1, o_2) = p^{max(min(i_1, i_2), 1)}$, where i_1 and i_2 are the heights of o_1 and o_2, respectively. Hence, $\sigma_i = p^{max(1,i)}$ and $\pi_{ij} = p^{max(min(i,j),1)}$. This distribution also makes no mention of locality but it is more suitable to model θ-operators where matches between large objects (i.e., objects higher up in the tree) are more likely. A typical example would be *between 50 and 100 kilometers from*.

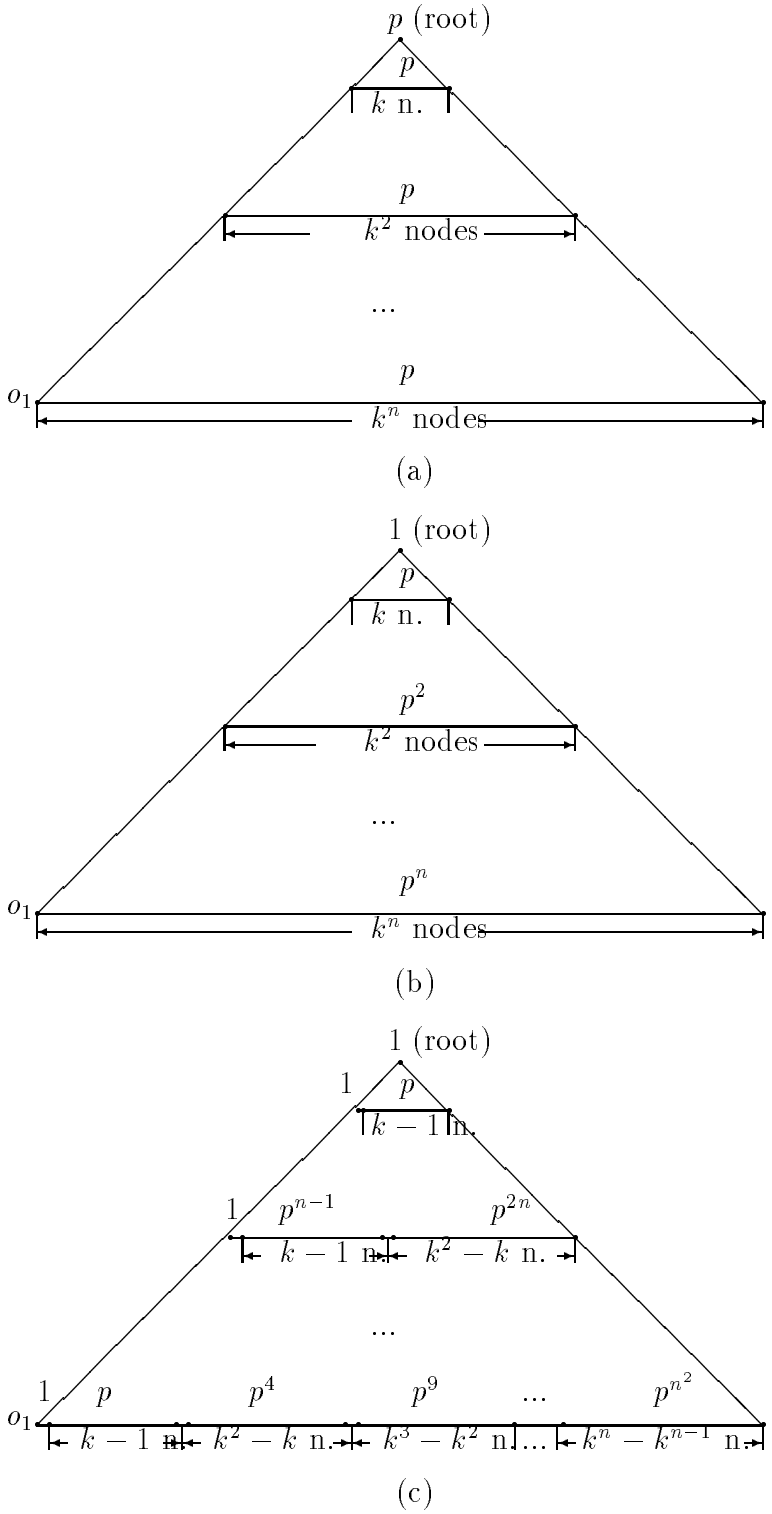

Fig. 7: The probabilities $\rho(o_1, o_2)$ for o_1 being the leftmost leafnode and for the distributions (a) UNIFORM, (b) NO-LOC and (c) HI-LOC.

HI-LOC. Let o' be the lowest common ancestor of o_1 and o_2, and let d_1 (d_2) be the difference in tree height between o' and o_1 (o_2). Then $\rho(o_1, o_2)$ is defined as $p^{d_1 \cdot d_2}$. This means in particular that any object matches any of its ancestors and descendants for certain. $\sigma_i = p$. By averaging over the nodes at the same tree height we obtain $\pi_{ij} = \frac{1+\sum_{r=1}^{min(i,j)} p^{r \cdot (n-min(i,j)+r)}(k^r - k^{r-1})}{k^{min(i,j)}}$. This distribution can only be applied if o_1 and o_2 are in the same generalization tree, i.e., for the spatial selection if the selector object o is in relation R, and for the general spatial join if a relation R is joined with itself. It represents a locality property of the θ-operator, such that objects that are spatially closer are also more likely to match.

A relation with N tuples of size v is assumed to be stored on $\frac{Nv}{sl}$ disk pages, where s is the size of a disk page and l is the average space utilization of the disk pages. In the case of the generalization tree strategy, we explore two variants with respect to clustering. The first variant (termed strategy IIa) has no clustering at all; this is quite typical for application-specific tree structures that do not interact with the physical storage manager. In the second variant (strategy IIb), the tuples are clustered on their relevant spatial attribute in breadth-first order with respect to the corresponding generalization tree. This is the case, for example, for the R-tree and many other spatial access methods. In both cases we assume that the tree nodes contain the complete tuples that correspond to the spatial object represented in that node.

Finally, the parameters used in the model are the following.

Database Dependent Parameters	
n	height of a generalization tree
k	out-degree of a tree node
p	join selectivity indicator ($0 \leq p \leq 1$)
v	size of a tuple (in bytes)
l	average disk space utilization
h	height of the selector object o in its generalization tree
T	total number of tuples in the database that have a spatial attribute
System Dependent Parameters	
s	page size (in bytes)
z	number of join index entries stored on a disk page
M	main memory size (in number of pages)
System Performance Dependent Parameters	
C_Θ	time to check whether $o_1 \Theta o_2$
C_{IO}	time per I/O operation
C_U	time for a node update
Derived Variables	
N	total number of tuples per relation: $N = 1 + k + k^2 + ... + k^n$
m	average number of tuples stored on a disk page: $m = \lfloor \frac{sl}{v} \rfloor$
d	height of join index B^+-tree: $d = \lceil \log_z N \rceil$

Table 2: Model Parameters and Variables

4.2 Updates

The cost of updates can be modeled in a simple manner because it does not depend on the matching distribution. For simplicity, in this paper we only consider the update operation where a new tuple is inserted into an existing relation R with a spatial column A. Then we find for strategy I (nested loop) that no update costs are incurred, i.e.,

$$U_I = 0$$

If a generalization tree is to be maintained, one has to insert the new object into the tree structure. That means that at each tree height one has to search $k/2$ nodes in the average to find a subtree in which the new object can be inserted (or to find out that it remains at the current height). The expected cost for this computation is $\frac{k}{2} \cdot C_u$. For the computation of the I/O requirements, we have to distinguish between the unclustered and the clustered case.

In the first case, there is no clustering at all (strategy IIa) and the participating nodes are randomly distributed in the file containing the relation. One can then approximate the number of I/Os required at each tree height by the so-called Yao number [Yao77]. Here $Y(x, y, z)$ denotes the expected number of I/Os for accessing x records randomly distributed in a file of z records stored on y disk pages. It is defined as

$$Y(x, y, z) = y \cdot [1 - \prod_{i=1}^{x} \frac{z - z/y - i + 1}{z - i + 1}]$$

If one needs to access $k/2$ nodes in the average at each height, this leads to an expected number of I/O operations of $Y(\lceil \frac{k}{2} \rceil, \lceil \frac{N}{m} \rceil, N)$. If the probability to be positioned at a given height i is assumed to be proportional to the number of objects already there, viz. k^i, one obtains the following expected height for a new object: $1 \cdot \frac{k^1}{N} + 2 \cdot \frac{k^2}{N} + \ldots + n \cdot \frac{k^n}{N} = \frac{1}{N} \sum_{i=1}^{n} i k^i$

Hence, the expected cost for an insertion is

$$U_{IIa} = (\frac{k}{2} \cdot C_U + Y(\lceil \frac{k}{2} \rceil, \lceil \frac{N}{m} \rceil, N) \cdot C_{IO}) \cdot \frac{1}{N} \sum_{i=1}^{n} i k^i$$

If, on the other hand, the relation to be updated is clustered on its spatial attribute in breadth-first order with respect to the corresponding generalization tree, one obtains the following result. Because in the average m objects are stored on a disk page, the expected number of I/O operations at each height becomes $\frac{k}{2} \cdot \frac{1}{m}$. The formula for the expected storage height is the same as above, making the expected cost for an insertion

$$U_{IIb} = (\frac{k}{2} \cdot C_U + \frac{k}{2m} \cdot C_{IO}) \cdot \frac{1}{N} \sum_{i=1}^{n} i k^i$$

Finally, if a join index is maintained for joins between the relation R and another relation S (strategy III), then one has to check the new object o in R against each object o' in S whether $o \Theta o'$, which results in an update cost of $N \cdot C_U + \lceil \frac{N}{m} \rceil \cdot C_{IO}$. If join indices are maintained for all relations in the database that have spatial attributes, the update cost rises accordingly to

$$U_{III}(T) = T \cdot C_U + \lceil \frac{T}{m} \rceil \cdot C_{IO} \approx T \cdot (C_U + \frac{C_{IO}}{m})$$

4.3 Spatial Selections

In the case of spatial selections, the nested loop strategy degenerates to an exhaustive search: one has to examine all N objects in the relation R whether they match, which results in a total cost of

$$C_I = N \cdot C_\Theta + \lceil \frac{N}{m} \rceil \cdot C_{IO} \approx N \cdot (C_\Theta + \frac{C_{IO}}{m})$$

To obtain the cost of strategies that involve a generalization tree, it is necessary to compute the expected number of nodes that are involved in the join computation at each height i. In this paper, we assume that the given selector object o is also contained in a generalization tree, and that h denotes its height in that tree.[1] The probability that o matches a given object o' at height i in the generalization tree for relation R is given by π_{hi}. If there is such a match, each of the k children of o' has to be examined. Hence, the number of nodes examined at height $i+1$ is $\pi_{hi} k^{i+1}$ $(0 \leq i < n)$. Accumulating over all heights of the tree, and taking into account that o always has to be checked against the root of $R's$ generalization tree, one obtains a total expected computation cost of

$$C_{II}^{\Theta}(h) = C_\Theta (1 + \sum_{i=0}^{n-1} \pi_{hi} k^{i+1})$$

For the computation of the I/O requirements we again distinguish between the unclustered and the clustered case. In the unclustered case, each node has to be retrieved separately from disk, which at height $i+1$, leads to $Y(\lceil \pi_{hi} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)$ disk accesses. Accumulated over all heights this corresponds to an expected I/O cost of

$$C_{IIa}^{IO}(h) = C_{IO} \cdot \sum_{i=0}^{n-1} Y(\lceil \pi_{hi} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)$$

[1] This assumption is needed in order to use the HI-LOC distribution. If it is not met in a given application, one can still apply the following model by assuming a value for h that is suitably correlated with o's size and matching probability.

Note that it is assumed here that the root of the generalization tree is locked in main memory. The total expected cost of algorithm SELECT in the unclustered case is then

$$C_{IIa}(h) = C_{II}^{\Theta}(h) + C_{IIa}^{IO}(h)$$

In the clustered case, on the other hand, we find in particular that nodes with the same parent are stored clustered. This means that for each one of the $\pi_{hi}k^i$ matching nodes at height i, one needs to fetch a "record" containing k nodes at height $i+1$ from the disk. There are k^i such "records", each of which contains k tuples in turn. Hence, the number of disk pages required to store these "records" is $\lceil\frac{k^i \cdot k}{m}\rceil$. The expected number of disk accesses at height $i+1$ $(0 \leq i < n)$ can therefore be computed as $Y(\lceil\pi_{hi}k^i\rceil, \lceil\frac{k^{i+1}}{m}\rceil, k^i)$. The total expected I/O cost for algorithm SELECT, accumulated over all tree heights, therefore becomes

$$C_{IIb}^{IO}(h) = C_{IO} \cdot \sum_{i=0}^{n-1} Y(\lceil\pi_{hi}k^i\rceil, \lceil\frac{k^{i+1}}{m}\rceil, k^i)$$

which means that the total expected cost of algorithm SELECT in the clustered case becomes

$$C_{IIb}(h) = C_{II}^{\Theta}(h) + C_{IIb}^{IO}(h)$$

If a join index is available one finds that there are $\sum_{i=0}^{n}\pi_{hi}k^i$ index entries relating to object o. In order to transfer the corresponding part of the index and all the corresponding tuples into main memory, one incurs the following I/O cost. Here it is assumed that the join index is implemented using a B^+-tree of height d, whose root is locked in main memory. Because virtually no computations are necessary this is also the total expected cost for computing the spatial selection.

$$C_{III}(h) = C_{IO} \cdot [d + \lceil\frac{\sum_{i=0}^{n}\pi_{hi}k^i}{z}\rceil + Y(\lceil\sum_{i=0}^{n}\pi_{hi}k^i\rceil, \lceil\frac{N}{m}\rceil, N)]$$

4.4 General Spatial Joins

A generalization of the considerations above leads to the cost formulas for the computation of spatial joins. For the nested loop approach (strategy I), one has to check each pair (a, b) of tuples from relations R and S, respectively, whether there is a match. This leads to a total computation cost of $N^2 C_{\Theta}$. To minimize I/O requirements, we use a simplified version of a main memory utilization technique described by Valduriez [Vald87]. Here we first fill most of main memory (say, $M - 10$ pages) with the contents of one relation (say R), then scan the other relation (say S) for matching tuples. After this first pass, another large section of R is brought into main memory, S is scanned once again, and so on, until

all tuples in R have been processed. Altogether there are $\lceil \frac{N}{m \cdot (M-10)} \rceil$ passes, each causing $\lceil \frac{N}{m} \rceil$ I/O operations for the scanning of S. In addition, one requires $\lceil \frac{N}{m} \rceil$ I/O operations for bringing R into main memory. The total cost of strategy I is therefore

$$D_I = N^2 \cdot C_\Theta + \lceil \frac{N}{m \cdot (M-10)} + 1 \rceil \cdot \lceil \frac{N}{m} \rceil \cdot C_{IO}$$

If one uses the algorithm JOIN to compute a general spatial join (strategy II), the total expected cost can be approximated as follows. A pair of objects at height i, (a, b), is to be examined (i.e., it is in $QualPairs[i]$) if $a \Theta b'$ *and* $a' \Theta b$, where a' and b' are the parents of a and b, respectively. The probabilities for these matches are $\pi_{i,i-1}$ and $\pi_{i-1,i}$, respectively. Because the two events are highly correlated, we assume the probability that both conditions are true to be simply $\pi_{i,i-1}$ (instead of $\pi_{i,i-1} \cdot \pi_{i-1,i}$ if the events were stochastically independent). This way, the cost for strategy II will be somewhat overestimated. The number of matches at height i then becomes $\pi_{i,i-1} \cdot k^i \cdot k^i$ or $\pi_{i,i-1} k^{2i}$.

For each match, one performs two passes of algorithm SELECT. One pass compares the object a to all descendants of b, the other compares object b to all descendants of a. As a corollary to the cost formula for $C_{II}^{\Theta}(h)$, we obtain computation costs for the first pass of approximately $C_\Theta \cdot (1 + \sum_{j=i}^{n-1} \pi_{ij} k^{j-i+1})$ and for the second pass of approximately $C_\Theta \cdot (1 + \sum_{j=i}^{n-1} \pi_{ji} k^{j-i+1})$. The sum of these two amounts counts the comparison of a and b twice, which needs to be corrected. We therefore obtain total computation costs of approximately

$$\begin{aligned} D_{II}^{\Theta} &= C_\Theta \cdot \sum_{i=0}^{n} (\pi_{i,i-1} k^{2i})(1 + \sum_{j=i}^{n-1} \pi_{ij} k^{j-i+1} + 1 + \sum_{j=i}^{n-1} \pi_{ji} k^{j-i+1} - 1) \\ &= C_\Theta \cdot \sum_{i=0}^{n} (\pi_{i,i-1} k^{2i})(1 + \sum_{j=i}^{n-1} (\pi_{ij} + \pi_{ji}) k^{j-i+1}) \end{aligned}$$

(For technical reasons, we assume here that $\pi_{0,-1} = \pi_{-1,0} = 1$.)

With regard to the I/O requirements, we first note that only those nodes in $GT_{R.A}$ ($GT_{S.B}$) will participate in the search whose parent has a successful Θ-match at least with the root of $GT_{S.B}$ ($GT_{R.A}$). We can therefore approximate the number of participating nodes (including the root nodes) by $1 + \sum_{i=0}^{n-1} \pi_{i0} k^{i+1}$ and $1 + \sum_{i=0}^{n-1} \pi_{0i} k^{i+1}$ for $GT_{R.A}$ and $GT_{S.B}$, respectively.

To speed up the algorithm, we use a main memory utilization strategy similar to the one described above. We first fill most of main memory (say, $M - 10$ pages) with the top nodes of one generalization tree (say, $GT_{R.A}$), then scan the other one ($GT_{S.B}$) for matching tuples. After this first pass, the next lower levels of $GT_{R.A}$ are brought into main memory, $GT_{S.B}$ is scanned again, and so on, until all of $GT_{R.A}$ has been processed. Altogether, there will be $\lceil \frac{1 + \sum_{i=0}^{n-1} \pi_{i0} k^{i+1}}{m \cdot (M-10)} \rceil$ passes. In analogy to the cost formulas for C_{IIa}^{IO}

and C_{IIb}^{IO}, we find that each pass causes about $\sum_{i=0}^{n-1} Y(\lceil \pi_{0i} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)$ disk accesses in the unclustered case (strategy IIa) and $\sum_{i=0}^{n-1} Y(\lceil \pi_{0i} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i)$ disk accesses in the clustered case (strategy IIb). Note that by this approximation we somewhat overestimate the number of I/O operations because except for the first pass, one probably would not need to bring in all nodes of $GT_{S.B}$ that match *the root* of $GT_{R.A}$. Remember that in later passes one only deals with lower levels of $GT_{R.A}$, and the number of nodes in $GT_{S.B}$ matching lower nodes in $GT_{R.A}$ is usually lower than the number of nodes matching $GT_{R.A}$'s root. To keep the model simple, however, this effect has not been taken into account.

Finally one requires $\sum_{i=0}^{n-1} Y(\lceil \pi_{i0} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)$ disk accesses in the unclustered case and $\sum_{i=0}^{n-1} Y(\lceil \pi_{i0} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i)$ disk accesses in the clustered case to page those nodes of the generalization tree $GT_{R.A}$ into main memory that match the root of $GT_{S.B}$.

The total I/O costs for the unclustered case can therefore be approximated by

$$D_{IIa}^{IO} = C_{IO} \cdot [\lceil \frac{1 + \sum_{i=0}^{n-1} \pi_{i0} k^{i+1}}{m \cdot (M-10)} \rceil \cdot \sum_{i=0}^{n-1} Y(\lceil \pi_{0i} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N) + \sum_{i=0}^{n-1} Y(\lceil \pi_{i0} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)]$$

and for the clustered case by

$$D_{IIb}^{IO} = C_{IO} \cdot [\lceil \frac{1 + \sum_{i=0}^{n-1} \pi_{i0} k^{i+1}}{m \cdot (M-10)} \rceil \cdot \sum_{i=0}^{n-1} Y(\lceil \pi_{0i} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i) + \sum_{i=0}^{n-1} Y(\lceil \pi_{i0} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i)]$$

and we obtain

$$D_{IIa} = D_{II}^{\Theta} + D_{IIa}^{IO}$$

$$D_{IIb} = D_{II}^{\Theta} + D_{IIb}^{IO}$$

If a join index is used, one just has to read in the join index and the qualifying tuples from disk. No additional computations are required. Overall the expected number of qualifying tuples is $\sum_{i=0}^{n} \sum_{j=0}^{n} \pi_{ij} k^i k^j$. Paging in the join index therefore requires $\lceil \frac{1}{z} \cdot \sum_{i=0}^{n} \sum_{j=0}^{n} \pi_{ij} k^i k^j \rceil$ disk accesses. In order to retrieve the corresponding tuples we once again use the main memory utilization technique described above. There are about $\sum_{i=0}^{n} \pi_{i0} k^i$ tuples in R participating in the search, and it takes $\lceil \frac{\sum_{i=0}^{n} \pi_{i0} k^i}{m \cdot (M-10)} \rceil$ passes to cycle through all of them. During each pass one needs to scan those tuples in S that match any one tuple of R that is currently in main memory. By computing a weighted average, we obtain an expected probability that any two tuples from R and S may match of $\frac{\sum_{i=0}^{n} \sum_{j=0}^{n} \pi_{ij} k^i k^j}{N^2}$. The probability that a tuple in S matches any of the

$m \cdot (M-10)$ tuples of R that is currently in main memory can therefore be approximated by $1-(1-\frac{\sum_{i=0}^{n}\sum_{j=0}^{n}\pi_{ij}k^ik^j}{N^2})^{m\cdot(M-10)}$.

The total cost for this strategy therefore amounts to

$$\begin{aligned} D_{III} &= C_{IO}[\lceil\frac{\sum_{i=0}^{n}\sum_{j=0}^{n}\pi_{ij}k^ik^j}{z}\rceil \\ &\quad +\lceil\frac{\sum_{i=0}^{n}\pi_{i0}k^i}{m\cdot(M-10)}\rceil\cdot Y(\lceil(1-(1-\frac{\sum_{i=0}^{n}\sum_{j=0}^{n}\pi_{ij}k^ik^j}{N^2})^{m\cdot(M-10)})\cdot N\rceil,\lceil\frac{N}{m}\rceil,N)] \end{aligned}$$

4.5 Performance Comparison

In order to evaluate the performance of those strategies we performed several simulations with the above cost formulas and the following parameter values.

n	6
k	10
v	300
l	0.75
h	6
s	2,000
z	100
M	4,000
C_Θ	1
C_{IO}	1,000
C_U	1
N	1,111,111
m	5
d	4

Table 3: Parameter Values

Several things should be noted. First, the generalization trees and therefore the relations have been chosen relatively large ($N \approx 10^6$). Especially if a query optimizer performs one or more selections before computing the actual join, the size of the participating relations will typically be much smaller than that. Second, tuple size has also been chosen relatively large. However, if geometric objects are actually stored in the columns, those tuple sizes can be easily reached. Third, for the spatial selection we restricted our evaluation to the case where the selector object o was stored in a leaf of its generalization tree ($h = n$).

Computing the cost formulas for this set of parameters lead to some interesting results. Fig. 8-10 show the results for the spatial selection operation. Note that both x- and y-axis have a logarithmic scale. As expected, in terms of update costs, the join index (U_{III}) is quite expensive in comparison to the update costs for the unclustered (U_{IIa}) and clustered (U_{IIb}) generalization tree. Space requirements have not been computed explicitly but it can easily be shown that they are also considerably higher for the join index than for the generalization tree.

For the UNIFORM distribution (Fig. 8), the search performance of the join index (C_{III}) is almost identical to the search performance of the unclustered generalization tree (C_{IIa}). If a clustered generalization tree (C_{IIb}) is available, search costs may be cut by up to an order of magnitude. Clustered generalization trees are clearly the method of choice, as they yield the best search performance combined with low update costs. The nested loop or exhaustive search strategy (C_I) is never really competitive.

In the case of the NO-LOC distribution (Fig. 9) the performance of the join index (C_{III}) is somewhere between the performance of the unclustered (C_{IIa}) and the clustered (C_{IIb}) generalization tree for higher join selectivities. Once the join selectivity p drops below about 0.08, however, the cost of paging in the join index is becoming more significant. As a result, the overall performance of the join index drops below the performance of the generalization tree. At such a low join selectivity the difference between the clustered and the unclustered version of the generalization tree becomes marginal.

For the HI-LOC distribution (Fig. 10) we again obtain a slightly different picture. Here the performance of the join index (C_{III}) is consistently between the performance of the unclustered (C_{IIa}) and the clustered (C_{IIb}) generalization tree. Once again, the performance of the nested loop strategy (C_I) is never competitive.

For the general join computation (Fig. 11-13) the situation looks somewhat different. Regardless of the distribution, join indices (D_{III}) provide the best join performance if the join selectivity is sufficiently small. In the case of the UNIFORM distribution, the crossover point is at a join selectivity of about 10^{-9}, for NO-LOC it is at about 10^{-8}, and for HI-LOC there is a tie between all three strategies for any reasonable join selectivity. The difference between the unclustered (D_{IIa}) and clustered (D_{IIb}) generalization tree is usually negligible with the exception of medium join selectivities in the NO-LOC distribution. In terms of update costs we obtain basically the same picture as in the case of the spatial selection. Update costs for join indices are almost prohibitively high. In summary, we find that join indices are only efficient if update ratios are very low and if join selectivities are comparatively low. Otherwise, the generalization tree is the superior approach. Again, the nested loop strategy (D_I) is not competitive.

It is important to note again that the two experiments represent two extreme cases of a spatial join. The first case (spatial selection, Fig. 8-10) is a degenerate spatial join where one of the two participating relations has only one tuple (viz., the selector object). The other case (general spatial join, Fig. 11-13) is computationally much more expensive because the participating relations are very large in comparison. We expect that in most

applications the size of the participating relations is much smaller, especially because they might not be original relations but the result of one or more selections and projections. By interpolation, we expect that the performance figures are somewhere between our results for the spatial selection and the spatial join.

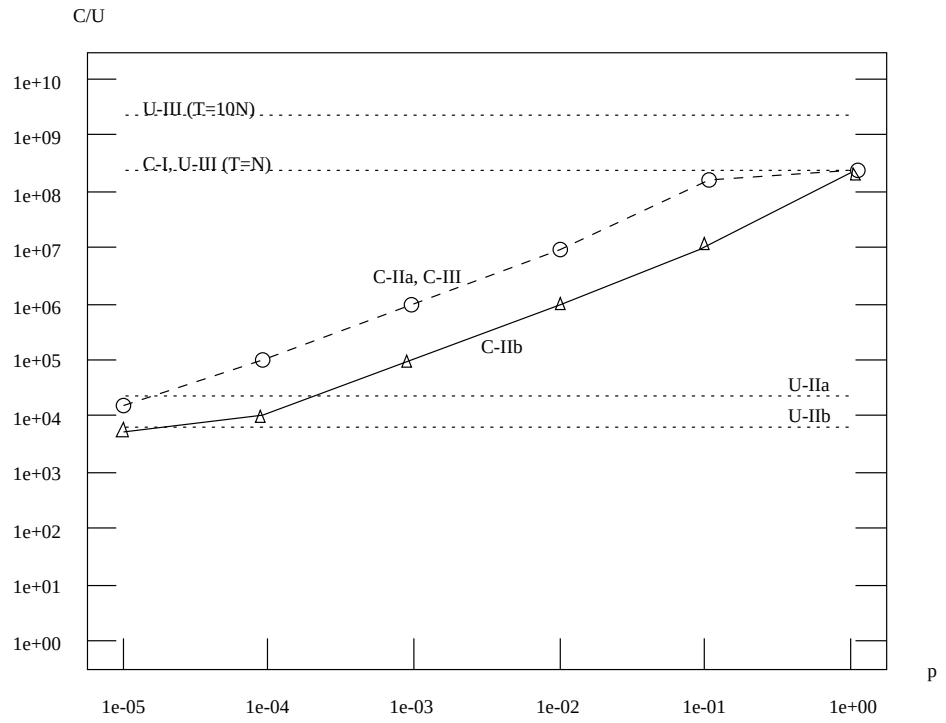

Fig. 8: SELECT, UNIFORM Distribution

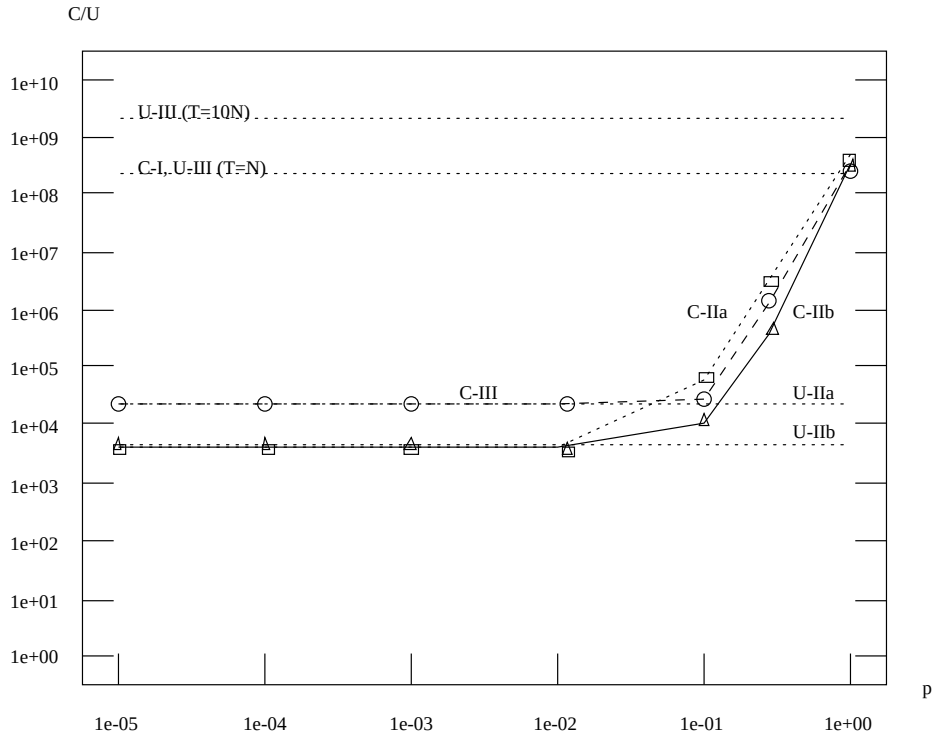

Fig. 9: SELECT, NO-LOC Distribution

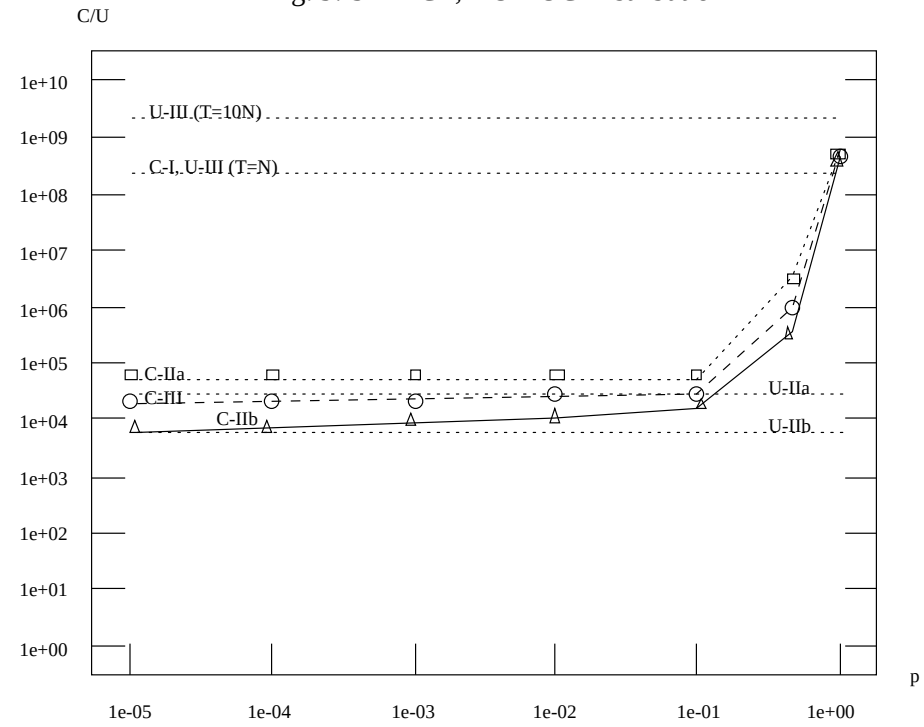

Fig. 10: SELECT, HI-LOC Distribution

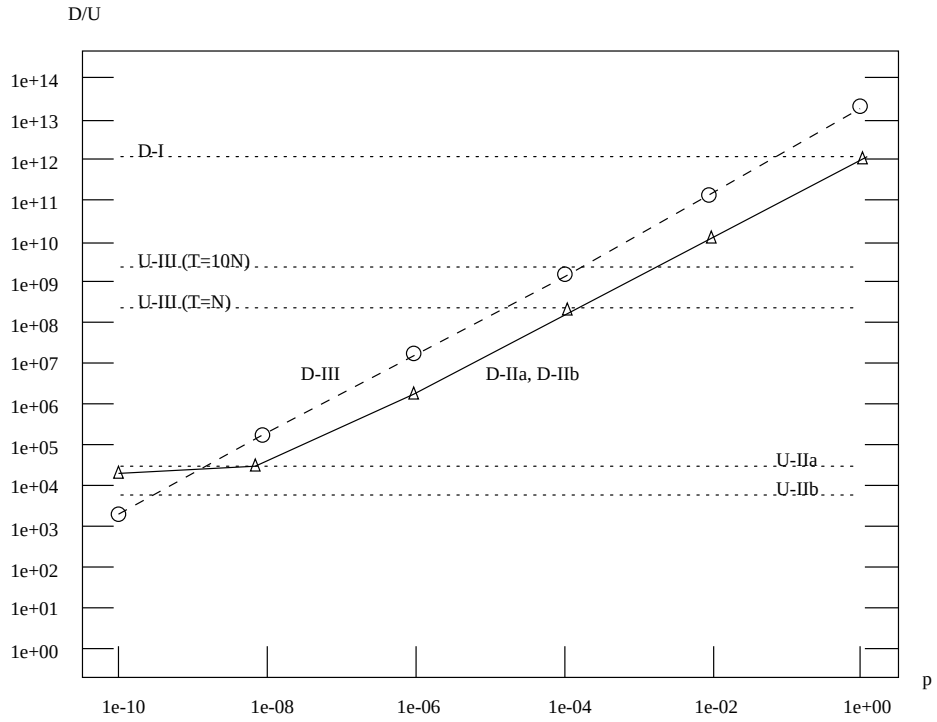

Fig. 11: JOIN, UNIFORM Distribution

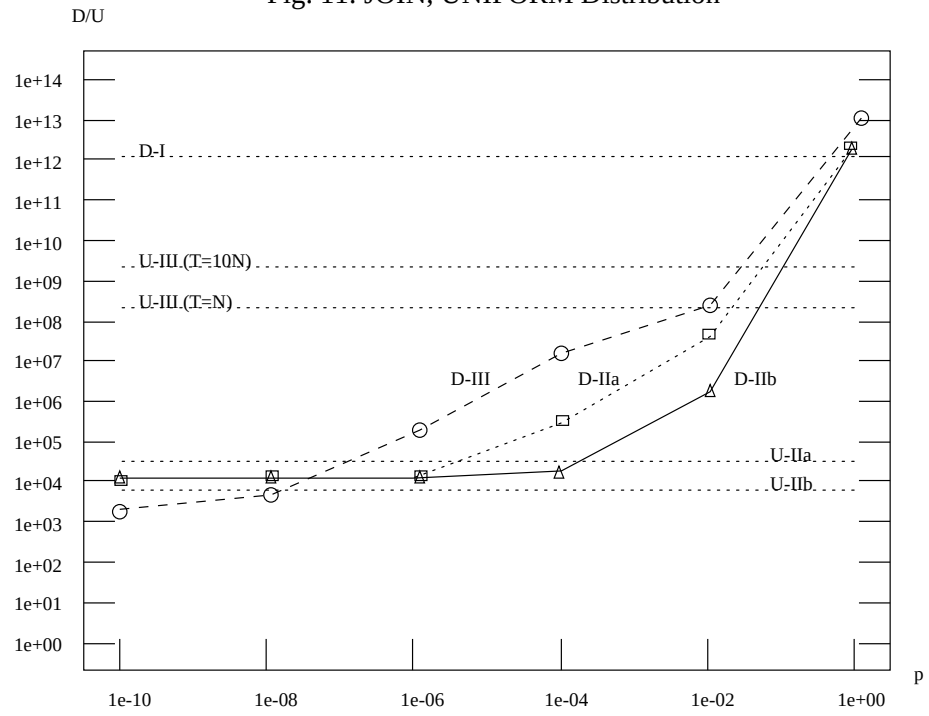

Fig. 12: JOIN, NO-LOC Distribution

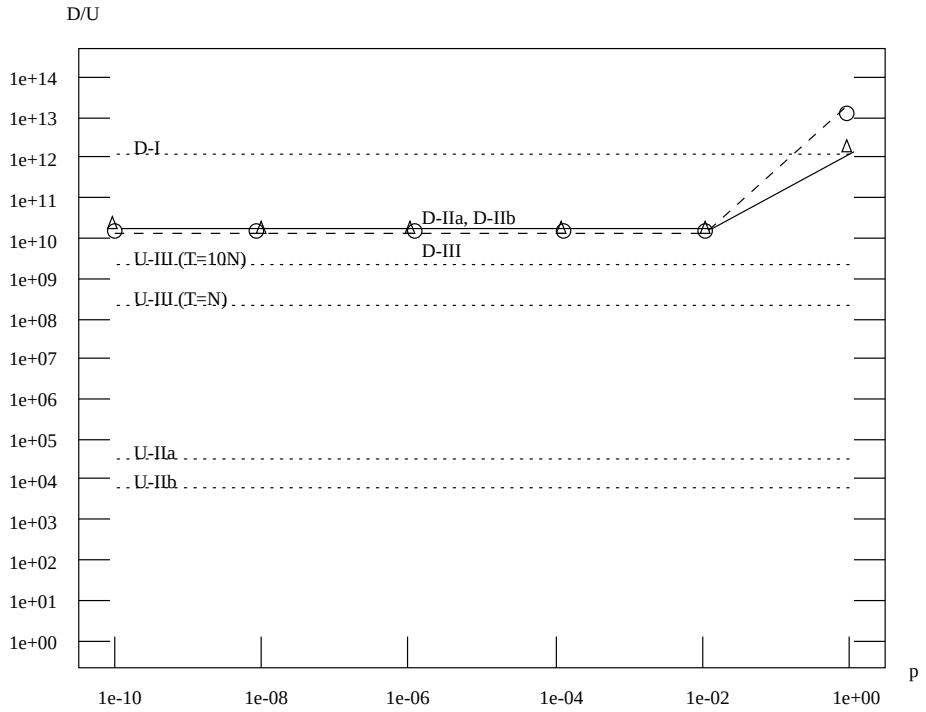

Fig. 13: JOIN, HI-LOC Distribution

5 Conclusions

In this paper we first evaluated several traditional join processing strategies how they perform in the context of spatial data. It turns out that the sort-merge strategy is usually not applicable at all because there exist no appropriate sorting orders. The nested loop strategy and join indices are both applicable, as are index-supported join strategies. For the latter case, spatial access methods play a very important role, and we described in this paper a class of tree structures, the so-called *generalization trees*, that can be used efficiently for this purpose.

We then derived a set of cost formulas for the different strategies in order to evaluate their respective performance in comparison. As expected, the nested loop strategy is never really competitive. For the *spatial selection* operation, clustered generalization trees clearly seem to be the most efficient strategy. Join indices, on the other hand, perform somewhat worse; in addition, they have higher space requirements and much higher update costs. Unclustered generalization trees usually give the worst results in terms of search performance; note however, that their update costs are essentially the same as for clustered generalization trees, which means that they are several orders of magnitude below the update requirements of the join index.

For the computation of a *general spatial join*, on the other hand, the results are mixed, depending on join selectivity. For a low join selectivity, join indices usually provide the best join performance. This result is consistent with the observation of Valduriez

[Vald87] that join indices are most efficient for very selective joins. For higher selectivities, generalization trees are usually the better option. The differences between the clustered and unclustered case have been negligible in most of our examples. Note however, that the update costs of join indices are again prohibitively high, and generalization trees remain the best overall strategy if update rates are significant.

In many applications, typical spatial join operations will be somewhere between the two cases we considered in detail. More research is required to find the exact crossover points where join indices become more efficient than generalization trees and vice versa. More detailed cost formulas and more comparative studies are required for this purpose; in particular, we intend to apply the analysis in [Vald87] to our setting. Furthermore, we want to explore the concept of so-called *local join indices* between objects that are indexed by the same generalization tree and have some ancestor in common. This extension can be viewed as a mixture between the pure generalization trees (strategy II) and pure join indices (strategy III), and we expect one of those mixed strategies to be the one that is optimal in terms of average performance.

References

[Gutt84] A. Guttman, R-Trees: A dynamic index structure for spatial searching, in *Proc. ACM SIGMOD Conference on Management of Data*, 1984.

[Niev84] J. Nievergelt, H. Hinterberger and K. C. Sevcik, The grid file: An adaptable, symmetric multikey file structure, *ACM Trans. on Database Systems*, Vol. 9, No. 1, 1984.

[Oren86] J. Orenstein, Spatial query processing in an object–oriented database system, in *Proc. ACM SIGMOD Conference on Management of Data*, 1986.

[Rote91] D. Rotem, Spatial join indices, in *Proc. Seventh International Conference on Data Engineering*, 1991.

[Sche90] H.-J. Schek, H.-B. Paul, M. H. Scholl and G. Weikum, The DASDBS project: objectives, experiences, and future prospects, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990.

[Ston86] M. Stonebraker and L. Rowe, The design of POSTGRES, in *Proc. ACM SIGMOD Conference on Management of Data*, 1986.

[Ston90] M. Stonebraker, L. Rowe and M. Hirohama, The implementation of POSTGRES, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990.

[Ullm90] J. Ullman, *Principles of database and knowledge base systems: Volume I*, Computer Science Press, Rockville, MD, 1988.

[Vald87] P. Valduriez, Join indices, *ACM Trans. on Database Systems*, Vol. 12, No. 2, 1987.

[Yao77] B. S. Yao, Approximating block accesses in database organizations, *Comm. ACM*, Vol. 20, No. 4, 1977.