INFORMATIK
BERICHTE

344 - 4/2008

User Defined Topological Predicates in Database Systems

Thomas Behr, Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

User Defined Topological Predicates in Database Systems

Thomas Behr, Ralf Hartmut Güting

April 11, 2008

Abstract

Current database systems cannot only store standard data like *integer*, *string*, and *real* values, but also spatial data like *points*, *lines*, and *regions*. The importance of topological relationships between spatial objects has been recognized a long time ago. Using the well known 9 intersection model for describing such relationships, a lot of constellations can be distinguished.

For the query language of a database system it is not desirable to have such a large number of topological predicates. Particularly the query language should not be extended by a large number of topological predicates. This paper describes how a database system user can define and use her own topological relationships. We show algorithms for computing user defined topological predicates in an efficient way. Last, we compare these general versions with specialized implementations of topological predicates.

1 Introduction

Besides standard data types, current database systems are able to store different kinds of information. Among others, it is possible to store spatial data types like *points*, *lines*, and *regions*. The importance of topological relationships between spatial objects has been recognized a long time ago. In database systems, topological predicates are used to find pairs of objects which are in an interesting relationship. In the past, several models describing topological relationships have been developed. In [6], the emptiness/non-emptiness of the intersections between the interior and the boundary of the involved objects are used to describe a topological relationship. The model is called the 4 intersection model. [5] investigates the possible relationships between regions with holes using this model. But it omits complex regions with separate components. Instead of only checking for emptiness of intersections between object components, the Dimension Extended Method (DEM)[4] defines a topological relationship using the maximum dimension of any connected point set within such intersections. Also the problem of too many different topological relationships was addressed. In contrast, the Calculus Based Method (CBM) uses only a few basic topological relationships (described in [3]) together with a **boundary** operator for providing a user friedly model for querying a database system. The CBM is applied to complex spatial objects in [1]. Detailed Topological Relationships between Composite Regions (TRCR) is the subject of [2]. Here, the number of topological relationships depends on the number of components of the involved regions. Therefore the model is inappropriate for extending the query language of a database system.

In this paper, we use the 9 intersection model [7]. It is an extension of the 4 intersection model. Here not only the emptiness of the intersections between interior and boundary of a spatial object is evaluated, but additionally the intersections of all parts of an object with the exterior of the other object. It uses a boolean matrix to represent a topological relationship between two spatial objects o_1 and o_2. The matrix entries indicate the non-emptiness of the intersection between the interior (o°), the boundary ($\delta(o)$) and the exterior (o^{-1}) of the involved objects in the following way:

$$\begin{aligned} m(o_1, o_2) &= \begin{pmatrix} o_1^\circ \cup o_2^\circ \neq \emptyset & o_1^\circ \cup \delta(o_2) \neq \emptyset & o_1^\circ \cup o_2^{-1} \neq \emptyset \\ \delta(o_1) \cup o_2^\circ \neq \emptyset & \delta(o_1) \cup \delta(o_2) \neq \emptyset & \delta(o_1) \cup o_2^{-1} \neq \emptyset \\ o_1^{-1} \cup o_2^\circ \neq \emptyset & o_1^{-1} \cup \delta(o_2) \neq \emptyset & o_1^{-1} \cup o_2^{-1} \neq \emptyset \end{pmatrix} \\ &= \begin{pmatrix} ii & ib & ie \\ bi & bb & be \\ ei & eb & ee \end{pmatrix} \end{aligned}$$

where the lower matrix is used as an abbreviation of the upper one within the remainder of this paper.

For simple spatial objects, only a few topological predicates can be realized. E.g. for two non-empty simple regions (connected areas without holes), eight topological relationships can be identified. But the number of realizable topological relationships grows drastically when switching to complex spatial objects. For example, for two non-empty complex lines (a complex line may consist of several connected parts each with an arbitrary number of end points), 82 relationships can occur. A detailed analysis of realizable topological relationships applying the 9 intersection model to complex spatial objects can be found in [15].

Whereas for simple spatial objects, providing a complete set of topological predicates makes sense, the according set of predicates for complex objects would be hard to use.

On the other hand, providing a small fixed set of topological predicates for complex spatial objects would only realize the programmer's imagination about the semantics of a topological predicate. For example, it is not clear when an **overlap** predicate applied to two <u>*region*</u> values should return `true`. Sometimes, it is sufficient, if both regions have a common 2-dimensional part. In another context, it may be required that each region also has its own part (i.e. one is not contained in the other). Different people may have different understandings of such predicates.

The next problem is the granularity of topological predicates. For example, it is not clear whether a region should be **inside** another region if it has no parts in the other region's exterior, or has each part of the region in the other region's interior (see Figure 1).

Such decisions depend on the applications and the user's understanding of topological predicates. For this reason, we will introduce an approach where the user can define and use her own topological relationships.

[14] presents an implementation for computing topological predicates for complex regions. Unfortunately, in this paper realm based regions [12] are

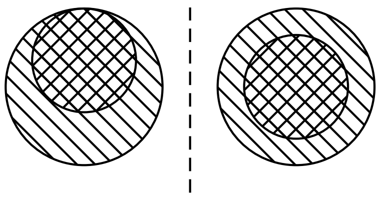
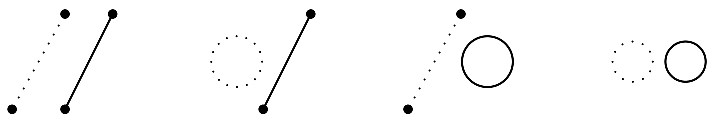


Figure 1: Different relationships or not?



$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Figure 2: Different matrices describing disjoint lines

presumed. Such regions are represented by their boundaries, these are sets of segments. In the realm based approach, these segments must not cross each other even for different objects. Furthermore, an end point of a segment must not lie within the interior of any other segment. So, segments are equal or have at most one common end point. Within a spatial database system (or a database system extended by spatial features), it is a very hard task to maintain these conditions.

Within this paper, we present algorithms computing topological relationships between complex objects in an efficient way. We describe, how we can compute a topological relationship for all combinations of spatial objects, i.e. *point*, *points*, *line*, and *region* values. We leave the assumption of realm based objects and allow segments to overlap or to cross.

The remainder of this paper is structured as follows. In Section 2, we describe a concept for defining and using topological relationships. In Section 3 we apply that concept to an existing database system. Section 4 shows algorithms for the efficient evaluation of topological predicates. We compare the running times of the proposed general approach with existing specialized implementations for some topological predicates in Section 5. Finally, Section 6 concludes the paper.

2 Concept

For describing a topological relationship between two spatial objects, we use the 9 intersection model. Because the number of topological relationships may be very large depending on the spatial data types, it makes no sense to handle each of them as a separate topological predicate. Moreover, there are different matrices describing the same topological relationship (see Figure 2).

On the other hand, the granularity of topological relationships depends on the application.

We solve these problems by introducing some new data types together with a set of operations. The types are *int9m*, *cluster*, and *predicategroup*.

In the following, we denote the 9 intersection matrix for the objects o_1 and o_2 by $m(o_1, o_2)$. An object of type *int9m* corresponds to a 9 intersection matrix and describes an *exact* topological relationship between two spatial objects.

A *cluster* is a named set of objects of type *int9m*. It will be used to define topological relationships by the user. A topological predicate described by cluster c complies with two spatial objects o_1 and o_2 if $m(o_1, o_2)$ is contained in c. For example, we could define a **disjoint** predicate for two non empty objects by a cluster containing all matrices with a `false` entry at the positions *ii*, *ib*, *bi*, and *bb* and `true` entries at positions *ie* and *ei*. We can combine different topological relationships to a single one just by forming the union of matrices contained in the two corresponding clusters. So we overcome the problem of granularity.

A *predicategroup* defines a system of mutually exclusive topological relationships. If several users will use different realizations of topological predicates, a lot of clusters have to be defined. *predicategroups* are helpful to control this large set of clusters.

3 Using the Concept in SECONDO

The concept described above is implemented as part of the SECONDO extensible database system. SECONDO is extensible by so-called algebra modules using the concept of the second order signature [8]. An algebra module provides new data types together with operators working on them. The data types described in Section 2 are part of the `TopRelAlgebra`, where also some supporting operators are implemented. The `TopOpsAlgebra` connects the `TopRelAlgebra` and the `SpatialAlgebra`. The `SpatialAlgebra` provides the spatial data types *point*, *points*, *line*, and *region*. In the `TopOpsAlgebra` the operators **toprel** and **toppred**, but no new types are provided.

A value of type *point* corresponds to a single point in the Euclidean space. Its interior consists of this point and its boundary is empty. An object of type *points* is a finite set of points. The boundary of a *points* value also is empty. An instance of type *line* is a finite set of segments. A line may consist of several non-connected components, each with any finite number of end points forming the boundary of the line. Formally, we can define the set of end points of a line L as:

$$\delta(L) = \{e : card(\{s = (e_1, e_2) \in L | e = e_1 \vee e = e_2\}) = 1\} - \bigcup_{s \in L} s^\circ$$

where s is a segment described by its two end points e_1 and e_2.

A *region* is a regular closed point set. The boundary is approximated by polylines. There are no missing parts within the boundary.

In the following let $t_1, t_2 \in \{$ *point*, *points*, *line*, *region*$\}$. The **toprel** operator computes the 9 intersection matrix for two spatial objects. Its signature is $t_1 \times t_2 \rightarrow$ *int9m*. The signature of the **toppred** operator is $t_1 \times t_2 \times$ *cluster* $\rightarrow$ *bool*. It checks if $m(o_1, o_2)$ is contained in the given cluster.

The **toppred** operator has a similar functionality as the **SDO_RELATE** operator of the Oracle Spatial database system [13]. Whereas the **SDO_RELA-**

Name	**Signature**
union	$\underline{cluster} \times \underline{cluster} \rightarrow \underline{cluster}$
intersection	$\underline{cluster} \times \underline{cluster} \rightarrow \underline{cluster}$
+, -	$\underline{cluster} \times \underline{int9m} \rightarrow \underline{cluster}$
name_of	$\underline{cluster} \rightarrow \underline{string}$
contains	$\underline{cluster} \times \underline{int9m} \rightarrow \underline{bool}$
disjoint	$\underline{cluster} \times \underline{cluster} \rightarrow \underline{bool}$
transpose	$\underline{cluster} \times \underline{string} \rightarrow \underline{cluster}$

Table 1: Operations on Clusters

TE operator receives one of a small set of topological predicates (or a disjunction of them), our operator is more flexible because of the use of arbitrary clusters.

3.1 Defining and Manipulating Matrices

Normally, single matrices are not defined by the user. But they may be helpful to define and manipulate clusters. In SECONDO, an object can be created by a `const` construct having the format `[const` *type* `value` *valuelist*`]`. For an object of type <u>*int9m*</u> there are several possibilities for *valuelist*. For a matrix m it can be a list containing the boolean entries of m in the following order: (*ii ib ie bi bb be ei eb ee*). As an abbreviation, the concatenation of the entries in the matrix can also be interpreted as a binary number, so that each matrix can be described by a unique number in range [0, 511].

3.2 Defining and Manipulating Clusters

A cluster is a named set of matrices describing a topological relationship. It can be defined directly by writing up the set of contained matrices: `[const cluster value ("name" (`m_1 m_2 `...))]`, where m_i is a matrix description from Section 3.1. Because it is inconvenient for the user to select the contained matrices manually, there is an alternative way for describing a set of matrices by giving the set of matrices as a boolean expression. The basic conditions are *ii*, ..., *ee*. Each such condition corresponds to a `true` value at the corresponding matrix position. They can be combined using **and**(`&`), **or**(`|`), **not**(`!`), $\rightarrow$ ($\Rightarrow$), and $\leftrightarrow$($\Leftrightarrow$) operations and brackets in the usual way.

For example, defining a cluster describing a **touches** relationship can be done in the following way. Such a relationship should hold, if the involved objects have a common boundary but no further common parts. The expression: `[const cluster value ("touches" "bb & !( ii | ib | bi)")]` defines the corresponding cluster.

Clusters can be manipulated using the operations from Table 1. The operation **union** (**intersection**) creates the union (intersection) of the matrices of both arguments. The name is taken from the first argument. The operations +, **-**, **name_of**, **contains**, and **disjoint** should be self-explanatory. The **transpose** operator defines a symmetrical relationship by replacing all contained matrices by their transposed counterparts. The string parameter is the new name of the result. For example, if there is a cluster `cl_inside` defining

Name	Signature
stdpgroup	$\rightarrow$ *predicategroup*
clustername_of	*predicategroup* $\times$ *int9m* $\rightarrow$ *string*
cluster_of	*predicategroup* $\times$ *int9m* $\rightarrow$ *cluster*
sizeof	*predicategroup* $\rightarrow$ *int*
getcluster	*predicategroup* $\times$ *string* $\rightarrow$ *cluster*

Table 2: Operations on Predicate Groups

an **inside** relationship, we can create a **contains** cluster using the command: `let cl_contains = transpose(cl_inside, "contains")`. In general, $m(o_1, o_2)$ = **transpose**$(m(o_2, o_1))$ holds. Hence also **toppred**$(o_1,\ o_2,\ cl)$ = **toppred**$(o_2,\ o_1,$ **transpose**$(\ cl))$ holds.

3.3 Defining Predicate Groups

A *predicategroup* describes a system of mutually exclusive topological relationships. It is realized by a set of disjoint clusters. There are several operators creating predicate groups from clusters.

Among others, the operator **stdpgroup** is provided creating a constant predicate group realizing our imagination of topological predicates. The result contains eight topological predicates, namely `covers`, `coveredBy`, `contains`, `inside`, `equal`, `disjoint`, `meet`, and `overlap`.

The operator **cluster_of** returns the cluster containing the 9 intersection matrix given as second argument. **clustername_of** is an abbreviation for **name_of**(**cluster_of**$(g,\ m)$). It is provided just for more comfort. Using the **getcluster** operator, we can extract a cluster from a group by its name.

3.4 Using Defined Objects

In this section we illustrate how to use the types and operators from the last section by some examples.

Queries are formulated using the query language of SECONDO's optimizer [9]. The query language is very similar to SQL but differs in some details. Further applications of the SECONDO system are described in [10].

For the following examples, we use the database "Germany" which is described in detail in Appendix C.1.

Example 1: Find out which counties are the neighbours of the county with name "Magdeburg".

For better readability, we first extract the cluster describing adjacent spatial objects from the standard predicate group:

```
let 'cl_meet = getcluster(stdpgroup(),"meet")'.
```

In a second step, we define a function object.

```
let 'myAdjacent = fun(r1 : region, r2: region)
                  toppred(r1,r2,cl_meet)'.
```

Note that the expression "`toppred(r1,r2,cl_meet)`" could also be replaced by "`cl_meet contains toprel(r1,r2)`". The reasons for providing an additional operator are better readability of the queries as well as a speed up by implementing of an early exit (see Section 4.4). The function object now can be used like a usual predicate. Of course, these steps must be done only once for the definition of an topological predicate. Later, the predicate can just be used without further changes.

We have to tell the optimizer to use the newly defined predicate as a prefix operator of SECONDO. After that, we can use it.

```
assert(secondoOp(myAdjacent, prefix, 2)).

select kname from kreis where myAdjacent(gebiet,magdeburg).
```
[1]

Example 2: Which are the counties that contain or touch the river "Rhine"? This question can be answered by:

```
let 'cl_intersects =
           [const cluster
               value ("intersects" "ii | ib | bi | bb")]".

select kname
from [fluss, kreis]
where [fname ="Rhein" ,
        toppred(fverlauf, gebiet, cl_intersects)].
```

Example 3: Which parts of highways are disjoint or completely inside of "Magdeburg"?

```
let 'cl_di = getcluster(stdpgroup(),"disjoint") union
             getcluster(stdpgroup(),"inside")'.

select [aname,anr]
from autobahn
where toppred(averlauf, magdeburg, cl_di).
```

By simply defining a new cluster, we can combine two completely different topological relationships. So, we only have to compute a single topological predicate. If a fixed set of topological relationships is provided, we would have to evaluate two different topological predicates and connect their result within a conjunction. Because the computation of a topological predicate may be expensive, our approach saves running time in such cases.

But also more unconventional topological relationships can be defined. For example, if we want to find out whether a river exists which ends within the county Magdeburg but is not completely inside of that county.

In the relation "Fluss", rivers are divided into several parts. Hence we use the relation "Fluss2" where rivers are concatenated.

First, we define the cluster realizing the predicate from our question. Then, we execute the corresponding query.

[1]For technical reasons, in SECONDO's SQL-like query language, names of objects (relations) and attributes are written in lower case.

```
let 'cl_ends_in = [const cluster value
                       ("endsIn" "bi & (ie | be)")]'.

select count(*)
from   fluss2
where  toppred(fverlauf2, magdeburg, cl_ends_in)
first 1.
```

4 Implementation

4.1 Data Types

For the representation of a 9 intersection matrix, we use a bit vector. The bit vector is realized using the C++ data type `short int`. A boolean flag indicates whether a matrix has a defined value or is undefined.

Because a 9 intersection matrix consists of 9 boolean values, there are 512 possible matrices. So, a cluster can contain at most 512 matrices. This number is small enough to represent it as a bit vector. The fixed size of the bit vector has some advantages when storing the object persistently. Inserting, deleting and check for containedness of a single matrix can be done in constant time. For the set operations union, intersection and so on, we have to scan the whole bit vector. Due to the restricted size, this can also be done efficently. For two different spatial types t_1 and t_2 the complex algorithms realizing the **toprel** and the **toppred** operators only are implemented in one direction. For example, the algorithm processing the types *line* and *region* is implemented with signature (*line* $\times$ *region* $\times \ldots$) but not with signature (*region* $\times$ *line* $\times \ldots$). Instead, the **toprel** operator transposes the result after calling the algorithm and the **toppred** operator transposes the argument cluster. Hence, transposing of a cluster is a frequently needed operation. To accelerate it, we also store the bit vector representing the transposed cluster and swap them for transposing.

A predicate group is realized by a `DBArray`, a data type realizing a persistent vector. The `DBArray` is provided by the SECONDO system. The single entries are stored sorted by their names. An additional cluster contains all matrices which are not defined explicitly by the user.

4.2 Used Spatial Data Types

To understand the algorithms computing the 9 intersection matrix for two spatial objects, we have to clarify the data structures of the spatial types. A *point* is just a pair of real coordinates together with a flag for the `defined` state. A value of type *points* represents a set of points. It is realized by a `DBArray` where the single points are stored in (x,y)-lexicographical order.

A *line* represents a set of segments. Each contained segment is stored as two halfsegments. A halfsegment is a segment with a signalized point called the dominating point. The halfsegments are stored sorted where the first criterion for sorting is the dominating point. Thereby each segment is reached twice during a single scan of all halfsegments. First, we hit the left end point of the segment and later the right one. The second criterion is the slope of the segment. So for halfsegments having the same dominating point, the one having a smaller

y value at the right of the dominating point will be the smaller halfsegment. By this representation, we avoid the expensive creation of an explicit event structure for plane sweep algorithms. The disavantage is the doubled stored data size. *line* values are not restricted to have a fixed number of end points and can consist of several components. Unfortunately, our line representation also allows crossing or overlapping segments which must be handled by some special cases in the algorithms.

For a *region*, we store its boundary as halfsegments. For the *region* type, halfsegments are extended by a flag `insideAbove` indicating where the inner part of the region is. Within a representation of a region, both, overlapping and crossing segments, are not allowed.

4.3 Algorithms

In this section we describe how the topological relationship between two spatial objects can be computed. We will only present the main part of the algorithms. In the actual implementation, in a preliminary step, checks for emptiness and disjoint minimum bounding boxes are performed (see also Section 4.5).

4.3.1 *point* × *point*

This combination is very easy to handle, because two single points can only be equal or not. To compute the result, we have just to check it and to set the appropriate entry of the resulting matrix to `true`. So, the result can be computed in constant time.

4.3.2 *point* × *points*

Because the points are stored sorted, we perform a binary search, to find out, whether the point is part of the *points* value. Then, we set the corresponding matrix entries to `true`. Because of the binary search, the result is computed in $O(\log(n))$ time, where n is the number of points contained in the *points* value.

4.3.3 *points* × *points*

Again, we benefit from the sorted storage of the entries for a *points* value. To compute the result, we do a parallel scan and set the matrix entries to the correct values. Because only entries at the matrix positions `ii`, `ie`, and `ei` can be found during this scan, we can stop the process when these entries are all set to `true`. Here, the running time is $O(m + n)$ time, where n and m are the sizes of the arguments.

4.3.4 *point* × *region*

Here, we use a usual plumb line algorithm to find out, where the point is located with respect to the region. The running time is proportional to the number of halfsegments contained in the region's representation.

4.3.5 *points* × *line*

For the combinations *point* × *line* and *points* × *line*, the same algorithm is used. The complete algorithm can be found in Appendix A.

The event structure of the algorithm consists of three parts, the halfsegment array of the *line* value, the point array of the *points* value, and a priority queue q containing halfsegments coming from split operations of original halfsegments (see below). A function `selectNext` is used to get the smallest entry from the three parts. It returns $first$, if the next segment comes from the *points* value, *second* if the next segment is part of the line, or *none* if all parts are empty. The value of the next element is returned in the parameter s. If a point is the result, s contains a degenerated segment.

As a status structure, we use an ordinary AVL-Tree. It only stores segments of the line. The segments are ordered by their y coordinate with respect to the current x coordinate. We disallow overlapping or crossing segments within the tree. This is achieved by splitting such segments. If a segment was splitted, only its left part is stored in the tree. The remaining part is moved into a priority queue for later processing. As for halfsegments, the slopes of segments are used as second criterion for sorting them within the tree. Overlapping segments are assumed to be equal. If we come to a right end point it may be that the corresponding segment is not stored in the tree because we were required to split it. But it will overlap a part which comes from split operations. For this reason, we have to check the exact equality (both end points are equal) of the segment in the tree and the currently processed segment.

The dominating points of the halfsegments are candidates for the line's boundary. We use the variable *count* to count how often a point was hit by such a dominating point. If the number of hits is one, this point is part of the boundary of the line. A number greater than one indicates a point within the interior of this line.

When the next event is launched by the *points* value, we check, if the current point is equal to the last dominating point. If so, we set the appropriate entry depending on the value of the counter within the matrix to `true`. If not, we look whether the point is located on a segment stored in the tree. If we can find such a segment, the point is located in the interior of the line, otherwise in its exterior. If the next event comes from the line, we perform required splits and insert the segment into the tree. Additionally, we also update the information about the last dominating point (*ldp*) and change the matrix content if needed.

4.3.6 *line* × *line*

The algorithm used here is very similar to the *points* × *line* combination. For this reason, we just describe the differences instead of writing up the complete algorithm. Segments are extended by an attribute describing their owner. Possible values are `first`, `second`, and `both`. For the last dominating point, we now use a separate counter for each line. The update of the counters and the associate handling of the lines' boundaries is done in the obviously way. The right parts resulting from split operations are stored in two separate priority queues (one for each line). When a right end point is processed, we remove the corresponding segment from the sweep status structure and split the neighbours if required. We ignore segments not having an equivalent in the tree.

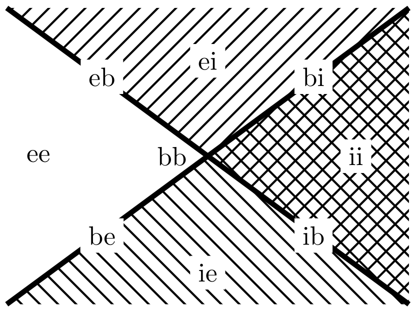


Figure 3: All intersections for crossing segments

Such segments have been splitted in some steps before but the corresponding event caused by its right end point was not removed from the event structure.

If a left end point of a halfsegment is processed, we first search for an overlapping segment in *sss*. If the stored segment has the same owner as the current segment, we just insert a possibly exiting right part into the corresponding queue. If they are different, we replace the stored segment by the common part (with owner = *both*) and insert a remaining right part into the corresponding queue. In this case the intersection of both interiors is set to be *true* within the result matrix. If no overlapping segment can be found, we look for intersections with the neighbours, perform required splits, and insert the segment (or its left part) into the tree.

4.3.7 points × region

The processing of halfsegments is the same as in algorithm A. We do not need a counter for the dominating points because the boundary of the region is built by the complete segments instead of only some of the end points. If a point from the points value is processed, we search in the AVL-tree the segment which is on or directly below the point. If the segment contains the point, it is located on the region's boundary. If there is no segment below the point, it is in the exterior of the region. Otherwise, we can determine whether the point is in the interior or in the exterior of the region using the `insideAbove` flag of the segment.

4.3.8 region × region

Here, for each segment so called *coverage numbers* (`con_above` and `con_below` are computed. Such numbers show, how many regions (0, 1, or 2) are covering the area above and below the segment. The coverage numbers can be computed easily using the `insideAbove` flags of the segments. At the right end point of a segment, we can be sure that the coverage numbers and the owner are up to date. So, we can use the coverage numbers to derive the corresponding matrix entries. If two segments are crossing, all possible intersections are found (see Figure 3). Thus we can stop our computation.

The complete algorithm can be found in Appendix B.

4.3.9 *line* × *region*

Actually, this is the most complex algorithm, because the coverage numbers as well as the counter for the line's potential end points must be handled. But basically, it works as a combination of the algorithms before. For segments owned exclusively by the line, the coverage numbers below and above are always equal, namely the `con_above` value of its left neighbour in the sweep status structure. To check where the last processed dominating point was, we additionally store its `con_below` value.

4.4 Acceleration for Topological Predicates

Normally, in database systems we are not interested in the exact description of the topological relationship by a 9 intersection matrix. Rather, we want to find pairs of objects which have an interesting relationship (given as a cluster). This is realized by the **toppred** operator. It checks whether the exact topological relationship (a 9 intersection matrix) is contained in a cluster. So, a naive implementation would be:

```
Algorithm toppred ( S1, S2 , C )
Input: S1, S2 : two spatial objects;
       C : cluster;
Output: true, if C contains the matrix
        describing the topological relationship
        between S1 and S2
   return C.contains(toprel(S1,S2));
end toppred.
```

Unfortunately, this simple approach leads to long running times. For example, we want to check two *line* values for equality given as condition (!ib & !ie & !bi & !be & !ei & !eb). Although, we could return `false` if one of the non-allowed intersections is found, we have to scan all the segments of the line because the complete matrix must be computed. Even bounding box checks for lines will have no effect using the naive approach because the matrix entries *eb* and *be* cannot be derived from bounding box constellations. To enable an early exit, we extend our algorithms computing the matrix by an additional argument of type cluster. In a first step, this cluster is restricted to contain the matrices valid for the current combination of spatial objects. For example in the computation for two *points* values, we remove all matrices containing a `true` at any entry related to an object's boundary. A complete discussion about the possible matrices can be found in [15]. Whenever an intersection is found, we restrict our cluster to such matrices having a `true` at the corresponding position. If the cluster becomes empty, we can stop the computation and the result is `false`. On the other hand, the cluster may contain all matrices which can be derived from the partial result by setting further intersections to be `true`. In this case, we call the cluster an extension of the matrix (see Figure 4 as an example). Because during the run of the algorithm no entries are set to `false`, we can also stop the computation here and return `true`.

$$m = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad c = \left\{ \begin{matrix} \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \end{matrix} \right\}$$

Figure 4: Cluster c is an extension of matrix m

4.5 Minimum Bounding Box Checks

To avoid unnecessary expensive evaluations of complex geometries, in practice, minimum bounding box checks are used as prefilter.

For most predicates, disjoint boxes or the emptiness of an involved object render complex computations needless. In the context of database systems, additionally the expensive access to the spatial data may be avoided in such cases.

However, in our approach it is possible to define predicates where the disjointness of the bounding boxes cannot help. For example, consider a predicate describing the relationship "Some boundary parts of the second object are outside the first object". For a *line* value nothing is known about the existing end points (the line's boundary). Hence, we have to perform a check for the existence of them.

Because the clusters acting as arguments of the algorithms are not constrained, we cannot do the bounding box check without knowledge about the cluster. For this reason, we have extended our cluster representation by some flags describing, which bounding box tests will successfully avoid expensive computations. A function of the *cluster* type uses these flags to perform the possible bounding box checks in constant time, i.e. without scanning the whole cluster. The flags are computed when the cluster is created by a scan of all contained matrices. The creation of a new cluster is a rare operation in contrast to the evaluation of a predicate. So, the additional running time during the creation is acceptable. In the following, we describe the flags, their meaning, and the inferences for bounding box checks.

In the sequel, matrices are written down linearly. A '&' denotes a bitwise conjunction of the matrix entries.

PART_INTER $\forall m \in c : m\&(110\ 110\ 000) \neq (000\ 000\ 000)$

The flag `PART_INTER` is set to `true`, if each matrix within the cluster requires an intersection of the objects' boundaries or inner parts. So, disjoint bounding boxes or an empty object induce a `false` result for the predicate evaluation.

NO_EXT_INTER $\forall m \in c : m\&(001\ 001\ 110) = (000\ 000\ 000)$

This flag is `true`, if in each contained matrix no part of an object is in the exterior of the other one, and vice versa. This is possible, if the bounding boxes are equal. So different bounding boxes will produce a `false` in the predicate evaluation.

CPU	Intel® Pentium®IV at 2.93 GHz
OS	SuSe Linux 10.3
Kernel Version	2.6.18
RAM	1 GB
SWAP	8 GB

Table 3: Configuration of the PC used for the Experiments

O1_EMPTY $\forall m \in c : m\&(111\ 111\ 000) = (000\ 000\ 000)$
This flag indicates that in each matrix the intersections of the boundary and the interior of o_1 are empty. Thereby the predicate can be evaluated to `true` if o_1 is empty.

O2_EMPTY $\forall m \in c : m\&(110\ 110\ 111) = (000\ 000\ 000)$
This is just the symmetric case to the one above.

O1_NON_EMPTY $\forall m \in c : m\&(111\ 111\ 000) \neq (000\ 000\ 000)$
This flag is `true`, if each matrix in the cluster has at least one intersection of the non-exterior parts of o_1. If o_1 is empty, the corresponding predicate cannot be evaluated to `true`.

O2_NON_EMPTY $\forall m \in c : m\&(110\ 110\ 111) \neq (000\ 000\ 000)$
Works like O1_NON_EMPTY, but for object o_2.

O1_INNER $\forall m \in c : m\&(001\ 001\ 000) = (000\ 000\ 000)$
If this holds, each part of $o1$ cannot be in the exterior of $o2$. This implies that the bounding box of $o1$ must be contained within the bounding box of $o2$.

O2_INNER $\forall m \in c : m\&(000\ 000\ 110) = (000\ 000\ 000)$
Symmetrically to O1_INNER.

5 Experiments

The introduction of general implementations for any topological predicate will produce some overhead in contrast to specialized solutions for each predicate. I.e. a specialized **disjoint** operator can abort its computation, if the algorithm detects an intersection of any parts of the arguments. The general implementation must check if the cluster contains a matrix which has a `true` entry at the corresponding position.

In this section we measure the running times for checking two spatial objects for a set of (user defined) topological predicates and compare them with the running times for specialized predicates which were already part of the SECONDO system.

The data sets used in the experiments are described in Appendix C. The system used for the tests was a standard PC. Its configuration is shown in Table 3.

Since the goal of the experiments is to compare the efficiency of the predicates, we use a simple nested loop join comparing all pairs of objects from the

Operator	**Meaning**
=	check for equality
#	check for inequality
intersects	true, if any common point is found
inside	true, if nothing is outside of the other object
overlaps	true, if there is a common point in the interior
onborder	true, if a point is located on the border of a region
ininterior	true, if a point is located in the interior of a region

Table 4: Topological Relationships implemented in the `SpatialAlgebra`

two argument relations rather than a spatial join also available in SECONDO. All predicate implementations (in previous algebras of SECONDO as well as in the `TopOpsAlgebra` presented here) use a bounding box test as a pre-filter step.

For example in the column "Kreis × Kreis" in Table 5, we have each predicate evaluated 439 * 439 = 192,721 times. The bounding box check rejects 189,418 pairs. The proper algorithm for testing the predicate is executed 3,303 times. (See Appendix C for the numbers.) Obviously, the time for the bounding box check is very small compared to the proper predicate implementation. As a reference, the nested loop join for "Kreis × Kreis" checking only for bounding box overlap has a running time of 1.7 seconds.

5.1 Topological Predicates of the SpatialAlgebra

In the `SpatialAlgebra`, some topological predicates are defined. Their names and meanings are collected in Table 4.

Unfortunately it turned out during our experiments that most implementations of the spatial predicates were not well-engineered. The reason was the big effort in writing dedicated code for each predicate. For example, the operator descriptions were not unique, e.g. the **overlap** operator was described by "returns true if the arguments overlap each other". Only a look into the source code could help to find out the exact meaning of this. Furthermore, for most topological relationships only a naive implementation exists. For example, the = operator checks for equality of the representations. Because in SECONDO the representation of a spatial object is not unique, we cannot trust a `false` result of this predicate. The remaining operators are implemented to have quadratic running time. For this reason, the general implementation beats the specialized ones in most cases. An exception is the configuration *point* × *region* where a sophisticated implementation described in [11] is used in the `SpatialAlgebra`. Here, the general implementation uses a simple plumb line algorithm. But it is possible to tune the new implementation by using the improved version of the plumb line algorithm.

Nevertheless we have collected our results in Table 5. In the table, we can recognize reasonable running times for the operators of the `TopOpsAlgebra`. The results for the `SpatialAlgebra` are given just for fun.

	Kreis × Kreis		Fluss × Kreis		Autobahn × Kreis		Ort × Kreis	
op	SA	TOA	SA	TOA	SA	TOA	SA	TOA
=	2.24s	18.5s	-	-	-	-	-	-
#	2.4s	18.1s	-	-	-	-	-	-
intersects	7:10m	40.4s	1:05m	22.3s	28.8s	17.2s	-	-
inside	1:51m	31s	11.3s	4.1s	2.5s	2.4s	7.7s	32.5s
overlaps	13:29m	1:21m	-	-	-	-	-	-
onborder							3:29m	36.9s
ininterior							7.2s	34.2s

Table 5: Running time comparison with the `SpatialAlgebra`
SA : SpatialAlgebra, TOA: TopOpsAlgebra

p_intersects	**PlaneSweep**	**TopOps**
Kreis × Kreis	1:27 m	1:14m
Kreis × Fluss	30.1 s	22.3 s
Kreis × Autobahn	SIGSEGV**	16.5 sec
Fluss × Autobahn	3.1 sec *	1.9 sec
SeqPoly × SeqPoly_small	49.4 s	42.7 s
SeqPoly × SeqLine_small	43.3 s	46.7 s
SeqLine × SeqPoly_small	1:01 m	1:10 m
SeqLine × SeqLines_small	44.2 s	50.2 s

* wrong result

**the system crashes during running the query

Table 6: Running time comparison with the `PlaneSweepAlgebra`
(**p_intersects** predicate)

5.2 Topological Predicates of the PlaneSweepAlgebra

The `PlaneSweepAlgebra` – like the `TopOpsAlgebra` – uses plane sweep algorithms for detecting the topological relationships between spatial objects. But only two topological relationships are implemented, namely **p_intersects** (both objects have a common point in their interior), and **intersects_new** (both arguments have any common point). We define the corresponding clusters by the conditions "`ii`" for the **p_intersects** predicate and "`ii | ib | bi | bb`" for the **intersects** predicate.

Table 6 compares the running times of the **p_intersects** predicate using the specialized implementation of the `PlaneSweepAlgebra` with the running times of our general implementation. We can see that the overhead produced by our general implementation is resonable. In some cases our implementation is even better than the specialized versions. Unfortunately, the `PlaneSweepAlgebra` produces some wrong results.

In the experiments, we have used narrowed versions of the Sequoia Benchmark relations as a second argument. This was done because we want to focus on predicate evaluation, as explained earlier, and have again used a simple nested loop join algorithm.

intersects_new	**PlaneSweep**	**TopOps**
Kreis × Kreis	1:14 m	35.3 s
Kreis × Fluss	30.1 s *	20 s
Kreis × Autobahn	23.9 s *	15.8 s
Fluss × Autobahn	3.1 s	1.9 s
SeqPoly × SeqPoly_small	1:02m	54.8 s
SeqPoly × SeqLines_small	45.1 s	54.1 s
SeqLines × SeqPoly_small	1:00 m	1:13 m
SeqLines × SeqLines_small	45.3 s	54.9 s

*wrong results

Table 7: Running time comparison with the `PlaneSweepAlgebra` (**intersects_new** operator)

Table 7 shows the running times of the **intersects_new** operator and its alternative realized in the `TopOpsAlgebra`. Also here, the `PlaneSweepAlgebra` produces wrong results in some cases. Again the overhead of our general implementation is adequate or the running times are better than the ones for the specialized versions.

6 Conclusions

We have shown how user defined topological predicates can be realized within a database system. After sketching some problems with specialized topological relationships provided by a database system, we have presented the advantages and the implementation of our general approach. The performance of the general implementation has been shown to be similar to existing specialized implementations for some predicates within the SECONDO extensible database system.

References

[1] E. Clementini and P. di Felice. A Model for Representing Topological Relationships Between Complex Geometric Features in Spatial Databases. *Informations Sciences*, 90(1-4):121–136, 1996.

[2] E. Clementini, P. Di Felice, and G. Califano. Composite Regions in Topological Queries. *IS*, 20(7):579–594, 1995.

[3] E. Clementini and P. D. Felice. An Object Calculus for Geographic Databases. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 302–308, New York, NY, USA, 1993. ACM.

[4] E. Clementini, P. D. Felice, and P. van Oosterom. A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In *SSD: Advances in Spatial Databases*. LNCS, Springer-Verlag, 1993.

[5] M. Egenhofer, E. Clementini, and P. Di Felice. Topological Relations between Regions with Holes. *International Journal of Geographical Information Systems*, 8(2):128–142, 1994.

[6] M. J. Egenhofer. A Formal Definition of Binary Topological Relationships. In W. Litwin and H. Schek, editors, *Third International Conference on Foundations of Data Organization and Algorithms (FODO)*, volume 367 of *Lecture Notes in Computer Science*, pages 457–472. Springer-Verlag, June 1989.

[7] M. J. Egenhofer and J. R. Herring. Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases. Technical report, Department of Surveying Engineering, University of Maine, Maine, 1990.

[8] R. H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 277–286, 1993.

[9] R. H. Güting, T. Behr, V. Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann. SECONDO: An Extensible DBMS Architecture and Prototype. Technical report, FernUniversität Hagen, 2004.

[10] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching. In *ICDE*, pages 1115–1116. IEEE Computer Society, 2005.

[11] R. H. Güting and Z. Ding. A simple but effective improvement to the plumb-line algorithm. *Inf. Process. Lett.*, 91(6):251–257, 2004.

[12] R. H. Güting and M. Schneider. Realm-based spatial data types: the ROSE algebra. *The VLDB Journal*, 4(2):243–286, 1995.

[13] R. Kothuri, A. Godfrind, and E. Beinat. *Pro Oracle Spatial for Oracle Database 11g.* Springer-Verlag, 2007.

[14] M. Schneider. Implementing Topological Predicates for Complex Regions. In *Proceedings of the Int ernational Symposium on Spatial Data Handling*, pages 313–328, 2002.

[15] M. Schneider and T. Behr. Topological Relationships between Complex Spatial Objects. *ACM Trans. Database Syst.*, 31(1):39–81, 2006.

[16] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Benchmark. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 2–11. ACM Press, 1993.

A Algorithm for *points* × *line*

```
Algorithm: TopRel(P, L);
Input     : P : Point; L : Line
Output    : int9m, describing the top. rel. between P and L;
var  res      : int9m;
var  sss      : tree<segment>;
var  owner    : Ownertype;
var  q        : pqueue;
var  ldp,dp,p : Point;
var  count    : int;
var  s, member,left ,right  : segment;
var  left1 , left2 , right1 , right2  : segment;

q := empty; sss := empty; count := 0;
res := ( 0 0 0 0 0 0 1 0 1 );
ldp := undefined;
owner := selectNext(P,L,q,s);
while owner # none do
   dp := s.getDominatingPoint();
   if owner = first then  // the point
      if dp = ldp then
        if count = 1 then
           res.ib := true;
        else // count > 1
           res.ii := true;
        fi
        count := 2
      else // dp != ldp
         member := sss.find(s);
         if member.defined then
            res.ii := true;
         else
            res.ie := true;
         fi
      fi
   else   // the line
     member := sss.find(s);
     if(s.isRightDominatingPoint) then
       if(member.defined and exactEqual(member,s)) then
          left := sss.leftNeighbour(member);
          right := sss.rightNeighbour(member);
          sss.remove(member);
          if left.intersects(right) then
            p := intersection point of left and right;
            split left and right at p;
            replace left and right by their left parts;
            insert the remaining parts into q;
          fi
          if dp = ldp then
             count := count + 1;
          else
             if count = 1 then
               res.eb := true;
```

```
              fi
              ldp := dp;
           fi
        fi
     else // a left end point
        if member.defined then
          q.insert( s - member);
        else
          foreach neighbour n of member in sss do
             if( n.intersects(s))
                split n and s at their intersection point;
                replace n in sss by its left part;
                replace s by its left part;
                insert both right parts into q;
             fi
          done
          sss.insert(s);
          if dp = ldp then
             count := count + 1;
          else
             if count = 1 then
                res.eb := true;
             fi
             lpd := dp;
          fi
     fi
   fi
   owner = selectNext(P,L,q,s);
 done
 if count = 1 then
    res.eb := true;
 fi
 return res;
end TopRel;
```

B region × region

```
Algorithm: TopRel(R1, R2);
Input    : R1, R2 : region;
Output   : int9m, describing the top. rel. between R1 and R2;
var  res     : int9m;
var  sss     : tree<segment>;
var  owner   : Ownertype;
var  q1,q2   : pqueue;
var  ldp,dp,p : Point;
var  count   : int;
var  s, member,left ,right  : segment;
var  left1 , left2 , right1 , right2 : segment;

q := empty; sss := empty; count := 0;
res := ( 0 0 0 0 0 0 0 0 1 );
ldp := undefined;
owner := selectNext(R1,R2,q1,q2,s);
while owner!=none do
```

```
dp := s.getDominatingPoint();
if dp = lpd then
   if dp.owner # lpd.owner then
      res.bb := true;
   fi
else
   lpd := dp;
fi
member := sss.find(s);
if s.isRightDomPoint() then
   if member.defined and member.exactEquals(s) then
      left := the left neighbour of member in sss;
      right := the right neigbour of member in sss;
      sss.remove(member);
      if left.crosses(right) then
        res.setAll(true);
        return res;
      fi
      if left.intersects(right) then
         let p be the intersection point of left and right;
         split the segments at p;
         replace left and right by the parts left of p;
         insert the right parts into q1 and q2 respectively;
      fi
      if member.owner # both then
         // check where member is located
         if member.con_above=0 or member.con_below=0 then
            // member in exterior od the other region
            if member.owner = first then
               res.be := true; res.ie := true;
            else
               res.eb := true; res.ei := true;
            fi
         fi
         if member.con_above=2 or member.con_below=2 then
            // member in interior of the other region
            if member.owner = first then
               res.bi := true; res.ii := true;
               res.ei := true;
            else
               res.ib := true; res.ii := true;
               res.ie := true;
            fi
         fi
      else // member.owner = both
         res.bb := true;
         if member.con_above=2 or member.con_below=2 then
           // combination 2 - 0
            res.ii := true;
         else // combination 1 - 1
            res.ei := true; res.ie := true;
         fi
      fi
   fi
```

```
    else // left dominating point
       if member.defined then
          res.bb := true;
          set the entries in res according to the
              owners and insideAbove flags;
          sss.remove(member);
          split member by s; // three parts, left, common, right
          if s.insideAbove then
             common.con_above := common.con_above + 1;
          else
             common.con_above := common.con_above - 1;
          fi
          sss.insert(left);
          sss.insert(common);
          insert right into the corresponding pqueue;
       else // no overlapping segment found in sss
          let left and right the neighbours of member in sss;
          if s crosses left or s crosses right then
             res.setAll(true);
             return res;
          fi
          // update coverage numbers
          if left.defined then
             s.con_below := left.con_above;
          else
             s.con_below := 0;
          fi
          if s.insideAbove = true then
             s.con_above := s.con_below + 1;
          else
             s.con_above := s.con_below - 1;
          fi
          sss.insert(s);
       fi
    fi
    owner = selectNext(R1,R2,q1,q2,s);
  done
  return res;
end TopRel.
```

C Used Data Sets

C.1 Database "Germany"

The database "Germany" contains cities, counties, rivers, and highways of Germany in the following relations:

Kreis
(KName : string, Flaeche : real, Bev : int, Bev_maennlich : int, Gebiet : region)

where `KName` is the name of the county, `Flaeche` is the area occupied by the region, `Bev` is the population in thousand, `Bev_maennlich` is the amount of the male population, and `Gebiet` is a *region* value describing the territory of the county. The relation consists of 439 entries. The contained regions have

	Kreis	Fluss	Autobahn
Kreis	3303	1682	1597
Fluss		1381	674
Autobahn			1379

Table 8: Number of intersecting bounding boxes

368 segments on average (50 minimum, 1086 maximum). The regions within this relation form a partition of Germany, i.e. there are no overlapping regions. For this reason, we only can find equal, adjacent, or disjoint regions within the relation `Kreis`.

Fluss
(FName : string, FNr : int, FVerlauf : line)

The relation contains 375 parts of rivers. The geometry is described in the attribute `FVerlauf`. The lines consist of 82 segments on average (5 minimum, 471 maximum).

Fluss2
(FName : string, FNr : int, FVerlauf : line)

This relation contains the same data as the relation "Fluss". The difference is that here rivers are not longer divided into several parts. The relation contains 122 rivers with 251 segments on average (8 mininum, 1010 maximum).

Autobahn
(AName : string, ANr : int, AVerlauf : line)

This relation contains 325 parts of highways of Germany. The size of the *line* value is 37 segments on average (1 minimum, 217 maximum).

Orte
(key : int, Ort : string, Position : point, . . .)

This relation comes from opengeodb. It contains 17409 cities (14288 located in Germany). Beside the position also further but here unused attributes are stored.

Table 8 shows how many pairs of entries within the relations have intersecting bounding boxes.

Furthermore, the database contains some "single" objects. This means objects stored outside a relation. In this paper we use the object "magdeburg" of type *region*, which was extracted from the relation `Kreis`.

C.2 Database "Sequoia"

The database "Sequoia" contains a subset of the Sequoia 2000 benchmark data [16]. Because we only are interested in the spatial attributes, we have also reduced the set of attributes. Within our paper, we only use the line and the polygon features of the benchmark. We have stored them in two relations with schemas:

	Poly	Poly_small	Lines	Lines_small
Number of Tuples	79,607	200	201,658	200
Number of segments				
min	0	4	0	1
max	5537	922	466	160
avg	46	40	18	18

Table 9: Sizes of the relations of the Sequoia database

	SeqLine	SeqPoly
SeqLines_small	924	544
SeqPoly_small	1081	1,543

Table 10: Numbers of intersecting bounding boxes

SeqPoly
(No : int, Reg : region)

SeqLines
(No : int, Line : line)

The relations and attributes have the sizes described in Table 9.

To reduce the query time, we have created two smaller relations which are randomized subsets of the relations `SeqLines` and `SeqPoly`, each with size of 200 tuples.

Table 10 shows the numbers of overlapping bounding boxes for these relations.