In: J.W. Schmidt, S. Ceri, and M. Missikoff (Eds.), Advances in Database Technology - EDBT '88. Proc. of the Intl. Conf. on Extending Database Technology, Venice, March 1988, 506-527.

Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems

Ralf Hartmut Güting

Fachbereich Informatik, Universität Dortmund,
D-4600 Dortmund 50, West Germany

Abstract: *The user's conceptual model of a database system for geometric data should be*

simple and precise: easy to learn and understand, with clearly defined semantics,

expressive: allow to express with ease all desired query and data manipulation tasks,

efficiently implementable.

To achieve these goals we propose to extend relational database management systems by integrating geometry at all levels: At the conceptual level, relational algebra is extended to include geometric data types and operators. At the implementation level, the wealth of algorithms and data structures for geometric problems developed in the past decade in the field of Computational Geometry is exploited. - The paper starts from a view of relational algebra as a many-sorted algebra which allows to easily embed geometric data types and operators. A concrete algebra for two-dimensional applications is developed. It can be used as a highly expressive retrieval and data manipulation language for geometric as well as standard data. Finally, geo-relational database systems and their implementation strategy are discussed.

1. Introduction

In various application areas there is a need to store, retrieve and manipulate data with geometric components, for instance in geographic information systems, VLSI- and CAD-databases, pictorial databases etc. We use the general term *geometric database system* for a database system handling any kind of data with geometric properties. In this paper we address what we believe to be the two main problems in the development of a geometric database system:

1. Design a user model of data, including geometric data, and of data manipulation. Such a model should have the following characteristics:
   - It should be *simple*, so that a user can easily learn and understand it and the consequences of actions can be foreseen.
   - The semantics of the model should be *precisely defined* so that there is no ambiguity for the user or implementor of the model.

- It should be *highly expressive* so that it is easy to perform the desired retrieval and data manipulation tasks.

2. Develop an efficient implementation strategy for the user's conceptual model.

Clearly the two problems are closely connected since it makes little sense to develop a data model for which no efficient implementation exists. - For the processing of "business" or "standard" data the relational model of data and relational database systems as its implementation have provided a very successful solution to these problems. In spite of the apparent diversity and complexity of geometric applications we believe that it is possible to proceed in a similar way for geometric database systems. Our approach consists of two parts:

1. Extend relational algebra by including geometric data types and operators. This is possible in a clean and simple way when relational algebra is viewed as a many-sorted algebra, as proposed in [GüZC87] (see the next section).
2. Implement the extended algebra by integrating geometric algorithms and data structures, as developed in Computational Geometry in the last few years, with the standard relational implementation technology.

In the past, the main approaches to the storage and processing of geometric data can be roughly classified as *dedicated geometric systems*, *non-relational DBMS front-ends*, and *relational DBMS front-ends and extensions.* Systems of the first class usually keep their data in files in either vector- or raster-based formats; they are geared towards efficiently performing a specific collection of functions or commands. A disadvantage is the lack of any high-level data definition facility. Usually the functions offered can only be used in very specific ways; there is little flexibility to perform actions not previously conceived by the system's designers [ChCK81, NaW79]. In the second class, a geometric database or information system is implemented as an application program on top of an existing DBMS. Systems in practical use are often based on non-relational DBMSs in order to achieve reasonable efficiency [Bu84, Fr81]. The disadvantage is again the lack of flexibility, in particular in an interactive environment.

Significantly, as soon as query languages are considered, applications and extensions of relational systems are prevalent (see [ChF81] for a survey), because of their flexibility. Here the main problem is efficiency. For relational systems, several degrees of integrating geometry can be observed. The most basic approach is to force geometric data into a relational format and then to perform operations on these tables with the usual facilities of the relational DBMS [BeS77]. The next step is to augment the relational query language by specific geometric operators [ChF80] which are implemented in terms of the relational operators. Up to this point, no really efficient DBMS can be expected. Recently attempts are being made [ScWa86, LiN87] to implement geometric database systems on top of "kernel" systems supporting nested relations [ScWe86]. This allows to keep the coordinates of a geometric object together within a single hierarchical tuple. However, we feel that this is still not adequate as long as the kernel has no concept of geometric objects as special entities and does not provide specific methods for processing them.

We believe that geometric algorithms and data structures have to be thoroughly integrated with the implementation of relational systems to achieve reasonable performance. It is necessary to stop relying on the standard relational representations and file structures and to introduce specific, efficient representations for geometric objects. This implies that changes have to be made at the conceptual level too, if only to prevent the system from using its standard techniques. Research in this direction has already been reported in [StRG83] where abstract data types are proposed for geometric objects and their operations which at the implementation level have to be supported by geometric file structures. Such file structures have for instance been developed in [NiHS84, Ta82, TiMT83, Gu84]. A similar approach to [StRG83] is described in [Or86] in the context of the PROBE research project [Day86].

In some sense, our model of a geometric database system can be seen as continuing this development and some ideas used are also present in [StRG83], [Or86], and [ChF80]. The choice of geometric data types and operators, as described in the next section, is influenced by [ChF80] (POINT, LINE, PGON) and [MaC80] (AREA) where *maps* (partitions of the plane) are used as the fundamental geometric structure. However, the basic approach to extend relational systems at their foundation by embedding geometric data types and operators into a many-sorted relational algebra and to implement this model using algorithms and data structures from Computational Geometry, is new and much more radical than previous approaches. In the remainder of this paper we describe geo-relational algebra as a user model and query language for a geometric database system (Section 2) and discuss the implementation issues involved in making the transition from geo-relational algebra to a geo-relational database system (Section 3). - Since this paper has a somewhat interdisciplinary content, it may be appropriate to mention standard books on database systems [Da86] and Computational Geometry [PrS85, Me84].

2. The Geo-Relational Algebra

2.1. Basic Concepts and Survey

Geo-relational algebra starts from a view of relational algebra as a *many-sorted* algebra, as (to our knowledge) first described in [GüZC87]. This means that the objects of the algebra are atomic values such as strings, numbers, or boolean values as well as relations. The operators of the algebra include arithmetic operators on numbers (e.g. addition) as well as the relational operators (e.g. selection, join).

Within such a formal framework it is easy to embed geometric data types and operators. The basic idea is the following:

- New data types are introduced for geometric objects, such as POINT, LINE, or REGION. In particular a relation may now have attributes of these new types. Furthermore specific operators are introduced for accessing objects of these data types (geometric objects). The representation of geometric objects is hidden from the user and they are in fact accessible only by the new geometric operators.

- A tuple of a relation describes an object as a combination of geometric and non-geometric attribute values.
- A relation as a whole describes a homogeneous collection of geometric objects, that is, a set of points, a set of lines, etc. (A homogeneous collection of geometric objects is the right kind of input for many geometric algorithms.)

Based on these principles one can design different algebras (that is, collections of data types and operators) for specific applications. In the sequel we describe an algebra which contains rather fundamental data types and operators for the manipulation of two-dimensional geometric data. The choice of data types is perhaps somewhat biased towards geographic applications. For VLSI-design databases, for instance, one would surely add a type RECTANGLE (with specific operators) and perhaps other types. Our specific algebra has the following object classes (or data types):

NUM	numbers	REL	relations
STR	strings		
BOOL	boolean values		
POINT	points in the plane, in a cartesian coordinate system		
LINE	lines (a line is a sequence of line segments)		
REG	regions in the plane		

The geometric object classes are discussed in more detail below. For example, a relation to represent cities might look as follows:

cities	cname	center	cpop
	STR	POINT	NUM

For each city its name, its geographic location with respect to a fixed coordinate system, and its population is given. It is assumed that this relation is to be used in a macro-geographic context such that it is sufficient to describe a city as a point.

A complete list of operators of this algebra is given in Table 2-1. At this point one should only note that each line describes a class of operators with the same functionality and that there are five groups of classes of operators. The first group comprises all the usual operations of a relational database system plus some additional operators. The remaining four groups contain geometric operators. Below we shall discuss each group in detail and illustrate its operators by examples.

1.	NUM $\times$ NUM	$\rightarrow$ NUM	+, -, *, /
2.	BOOL $\times$ BOOL	$\rightarrow$ BOOL	**and**, **or**
3.	BOOL	$\rightarrow$ BOOL	**not**
4.	REL $\times$ REL	$\rightarrow$ REL	$\cup$, $\cap$, $-$, $\times$, $\vert\times\vert$
5.	REL	$\rightarrow$ REL	σ, π, λ
6.	NUM $\times$ NUM	$\rightarrow$ BOOL	$=$, $\neq$, $<$, $\leq$, $\geq$, $>$
	STR $\times$ STR	$\rightarrow$ BOOL	
	BOOL $\times$ BOOL	$\rightarrow$ BOOL	
7.	REL	$\rightarrow$ NUM	**count**
8.	NUM*	$\rightarrow$ NUM	**sum**, **avg**, **min**, **max**
9.	REL	$\rightarrow$ ATOM	**extract**

10.	POINT $\times$ POINT	$\rightarrow$ BOOL	$=$, $\neq$
	LINE $\times$ LINE	$\rightarrow$ BOOL	
	REG $\times$ REG	$\rightarrow$ BOOL	
11.	GEO $\times$ REG	$\rightarrow$ BOOL	**inside**, **outside**
12.	EXT $\times$ EXT	$\rightarrow$ BOOL	**intersects**
13.	AREA $\times$ AREA	$\rightarrow$ BOOL	**is_neighbour_of**

14.	LINE* $\times$ LINE*	$\rightarrow$ POINT*	**intersection**
	LINE* $\times$ REG*	$\rightarrow$ LINE*	
	PGON* $\times$ REG*	$\rightarrow$ PGON*	
15.	AREA* $\times$ AREA*	$\rightarrow$ AREA*	**overlay**
16.	EXT*	$\rightarrow$ POINT*	**vertices**
17.	POINT* $\times$ REG	$\rightarrow$ AREA*	**voronoi**
18.	POINT* $\times$ POINT	$\rightarrow$ REL	**closest**

19.	POINT*	$\rightarrow$ PGON	**convex_hull**
20.	POINT*	$\rightarrow$ POINT	**center**
	EXT	$\rightarrow$ POINT	

21.	POINT $\times$ POINT	$\rightarrow$ NUM	**dist**
22.	GEO $\times$ GEO	$\rightarrow$ NUM	**mindist**, **maxdist**
23.	POINT*	$\rightarrow$ NUM	**diameter**
24.	LINE	$\rightarrow$ NUM	**length**
25.	REG	$\rightarrow$ NUM	**perimeter**, **area**

Table 2-1: Operators of the geo-relational algebra

The functionality of the operators is basically described in terms of the classes of objects mentioned above. Some additional notations are used to enhance clarity and readability. The notation ATTR* where ATTR represents one of the atomic object classes (admissible as attribute types) is used to refer to a relation of which an attribute of type ATTR is specified. Hence the ATTR* notation determines a column of a relation or, intuitively, a set of objects of class ATTR. This notation is very useful because geometric operators are often applied to a set of geometric objects, given as a column of a relation. In a more formal context, however, one should replace every occurrence of ATTR* by REL. - The notation ATTR* used on the right hand side of an arrow has a slightly different meaning and will be explained below, as well as the generalizations and specializations of object classes denoted by ATOM, GEO, EXT, PGON, and AREA. - Note that Table 2-1 does not yet specify the syntax for writing down algebra expressions nor the parameters that certain operators take. For instance, the projection operator (π) takes the projection attributes as parameters. The distinction between operands and parameters is that operands are objects of the algebra, parameters additional information for the operator. Syntax and parameters are described below.

More formally, the algebra is a collection of sets and functions between these sets; it is described by an *S-sorted signature* Σ where S is a set of sorts (names for the sets) and Σ a family of sets $\Sigma_{w,s}$ of operator symbols (names for the functions), where $w \in S^*$ and $s \in S$ describe the functionality of operators in $\Sigma_{w,s}$ [GoTW78]. The set of sorts of geo-relational algebra is S = {REL, BOOL, NUM, STR, POINT, LINE, PGON, AREA} and the signature is basically given by Table 2-1. So the algebra consists of a set for each sort in S and a function for each element in Σ. In this paper, due to the limited space, we are only able to sketch the database model, mostly describing it informally. A complete formal model is given in [Gü87] defining the sets and functions of the algebra precisely. (Actually there geo-relational algebra is viewed as one specific extension of a general many-sorted algebra model of an extensible database system and we describe here a slightly simplified version of that model.) [Gü87] in turn develops further the model of [GüZC87].

2.2. The Standard Operators (Classes 1 - 9)

1.	NUM × NUM	→ NUM	+, -, *, /	(_ # _)
2.	BOOL × BOOL	→ BOOL	**and**, **or**	(_ # _)
3.	BOOL	→ BOOL	**not**	# (_)
4.	REL × REL	→ REL	$\cup$, $\cap$, $-$, $\times$, $\lvert\times\rvert$	_ _ #
5.	REL	→ REL	σ, π, λ	_ #
6.	NUM × NUM	→ BOOL	=, ≠, <, ≤, ≥, >	(_ # _)
	STR × STR	→ BOOL		
	BOOL × BOOL	→ BOOL		
7.	REL	→ NUM	**count**	(_ #)
8.	NUM*	→ NUM	**sum**, **avg**, **min**, **max**	(_ #)
9.	REL	→ ATOM	**extract**	(_ #)

The operators of this group allow to perform most of the usual retrieval and data manipulation tasks possible in a relational database system. Classes 4 and 5 contain the standard relational operators union, intersection, set difference, cartesian product, join, selection, and projection. However, in this algebra the σ (selection) and the |×| (join) operator can take an arbitrary algebra expression of result type BOOL as a parameter. Hence it is possible to include arithmetic calculations or aggregate functions in selection or join conditions, to nest further applications of the selection operator into a condition and so forth. These capabilities are also crucial for the extension to geometric objects and operators, as will be shown below.

The λ operator is a powerful facility introduced in [GüZC87]. It allows to dynamically extend an existing relation by a new attribute. For each tuple, a value for the new attribute is computed as the result of evaluating an algebra expression given as a parameter of the λ operator. More generally, the λ operator can take any number of parameter algebra expressions; for each of them it creates a new attribute or replaces the value of an existing attribute.

The **extract** operator is the only new operator in this group with respect to [GüZC87]. It allows to extract an atomic value from a relation in order to make it available for further operations. The **extract** operator takes two parameters: The first parameter is a selection condition, that is, an algebra expression with boolean result, which has to identify a single tuple of the operand relation. The second parameter is an attribute name. If the parameter expression selects more than one tuple the algebra expression containing this **extract** operator is considered erroneous; its evaluation stops and a message is sent to the user.

The listing of operators of classes 1 - 9 has been augmented by a description of the required syntax for writing algebra expressions which allows to use the algebra directly as a query language. The symbols "#" and "_" represent the operator and an operand, respectively. Parentheses have to be put as indicated. Basically the syntax is defined according to the following rules:

- Use postfix notation whenever an operator takes a relation as an operand.
- Put parentheses if the resulting object is atomic.
- Apart from that use the standard notation (infix or prefix) for the respective operator.

Finally, to avoid superfluous parentheses, the outermost pair of parentheses may be stripped from an expression. - Note that the syntax description does not specify the parameters that some operators can take. Parameters appear in square brackets behind the operator symbol. Which parameters an operator takes is in this paper explained only informally together with the operator, but defined formally in [GüZC87] and [Gü87].

We illustrate the use of operators of this group as well as the syntax by a few standard examples. Let *items* be a relation describing the items in a warehouse:

items	item#	qty	desc	s_price	v_price
	NUM	NUM	STR	NUM	NUM

For each item an identifying number, the quantity in the warehouse, a textual description, the supplier price, and the vendor price are given. Here are some example queries:

"Return the names of items of which more than 100 are in stock, and whose vendor price exceeds $ 1000."

items σ [(qty > 100) **and** (v_price > 1000)] π [desc]

The following example illustrates the possibility to put arbitrary calculations into selection conditions: "Find items whose vendor price exceeds the supplier price by more than 20%."

items σ [(s_price/v_price) < 0.8] π [desc]

The next example shows the use of the λ operator and of aggregate functions: "Compute the total value of items in the warehouse."

(items λ [qty * v_price {amount}] **sum** [amount]) {total}

The λ operator extends each tuple of the items relation by a new component whose value is given by the product of the qty and v_price components of this tuple. The new attribute is named "amount" (braces are used universally for introducing names). Hence after application of the λ operator the schema of the resulting relation is:

items	item#	qty	desc	s_price	v_price	amount
	NUM	NUM	STR	NUM	NUM	NUM

The type of the new attribute is, of course, given by the result type of the parameter expression. Finally, let us give an example of a join operation in this framework. Let there be a relation describing articles sold by departments

depts	dept_name	article#
	STR	NUM

assuming the article numbers match the item numbers of the items relation. Then the question "Which departments sell pencils" can be formulated as:

items σ [desc = 'pencil'] depts |×| [article# = item#] π [dept_name].

It should be emphasized that our geometric extension does not start from the classical relational algebra but from the *many-sorted* algebra developed in [GüZC87] which we consider by itself to be a fundamental and significant extension of the classical model. It allows to include domain operations (such as arithmetic operators) and aggregate functions into the formal model, provides a much more general and powerful concept of selection and join, and generally allows *computed* values in result relations. Furthermore [GüZC87] provides a formal treatment of nested and ordered relations (sequences of tuples) and nested algebra expressions.

2.3. Geometric Object Types

We now discuss the geometric objects of the algebra in some more detail. Any geometric object is modeled by *a set of points in the Euclidean plane*. Except for the special case of a POINT object, this set is infinite although it can always be finitely specified. To distinguish between the representa- tion of an object and the object itself we introduce a function *points* which associates with a symbol or representation the set of points represented. Let us start by some geometric preliminaries.

The Euclidean plane is given by the set $P = \mathbf{R} \times \mathbf{R}$ of pairs of real numbers with the usual metric. A *point* is simply an element of P. Of course, for a point $p = (x, y)$ we have *points* (p) = $\{(x, y)\}$. Given two distinct points p_1 and p_2 in P, the set $\{\alpha p_1 + (1-\alpha)p_2 \mid \alpha \in \mathbf{R}\}$ defines a *line*. If we additionally restrict α to lie between 0 and 1, we obtain the set $\{\alpha p_1 + (1-\alpha)p_2 \mid \alpha \in \mathbf{R},\ 0 \leq \alpha \leq 1\}$ which describes a *line segment* , namely the part of the line lying between p_1 and p_2, which are called the *end points* of the segment. A *chain of segments* is a finite sequence of line segments such that any two adjacent segments share an end point and no end point belongs to more than two segments. The set of points associated with a chain is of course the union of the point sets of all segments in the chain. A chain is called *simple* if no point other than an end point is shared by any two segments.

A chain is called *closed* if also the first and last element of the sequence share an end point and in this case is also called a *polygon*. Note that a closed, simple chain divides the plane into two disjoint regions, the *interior* and the *exterior*. A *simple polygon* is thus given by a closed, simple chain and we define its set of points to be the points of the chain together with the points of the interior.

The object types of the algebra can now be defined as follows: A value of type POINT is a point in the sense of the definition above. A value of type LINE is a non-closed, simple chain of segments. A value of type PGON or a value of type AREA is a simple polygon.

In applications, an object of type POINT can be used to model an object whose position, rather than extension, in some plane space is relevant, for instance a city in a geographic application. A LINE object approximates a curve in this space, such as a river or highway. A PGON or AREA object models a simple region (approximating its boundary) without holes, such as a lake or a country. The distinction between the two types PGON and AREA will be explained shortly.

For some applications it would certainly be desirable to admit more complex types of regions, in particular regions with holes. For instance, a lake having an island must now be modeled as if the island's area were part of the lake; the island must be specified as a separate region overlapping the lake. However, we feel that a too high price would need to be paid for generally allowing regions to contain holes. First, these objects are conceptually much more complex which is important when it comes to specifying (and visualizing) the effect of operations. Second, the internal representation is more complex and it is much more difficult to treat these objects algorithmically. So we have restricted the algebra to hole-free regions; this does not preclude the possibility of later extending the algebra by a type for regions with holes.

The distinction between types PGON and AREA concerns the set of polygons occurring as a column of a relation: If the attribute is of type PGON then polygons within the set may intersect each other. If it is of type AREA then essentially all polygons within this set (column) are disjoint. More precisely, we require that two polygons of type AREA belonging to the same attribute are either equal or *their interiors* are disjoint (which means they may share a boundary edge or vertex). Fig. 2-2 illustrates the geometric data objects that can occur as a column of a relation.

Type AREA is motivated by the fact that in applications often subdivisions of some plane space into disjoint regions occur, for instance the area of a continent may be divided into countries, or some land area may be split into regions according to the type of industrial or agricultural use etc. For planar subdivisions special types of operators are applicable, for instance asking for the neighbours of a region, which do not make sense for general collections of polygons.

The restriction imposed on type AREA can be viewed as an integrity constraint on relations. We are forced to allow two polygons in different tuples to be equal because certain relational operators, for instance join, create duplicates. Conceptually we can still think of a subdivision of the plane even when some areas are present more than once. Also the data structures and algorithms for representing and processing subdivisions can handle these duplicates without problems.

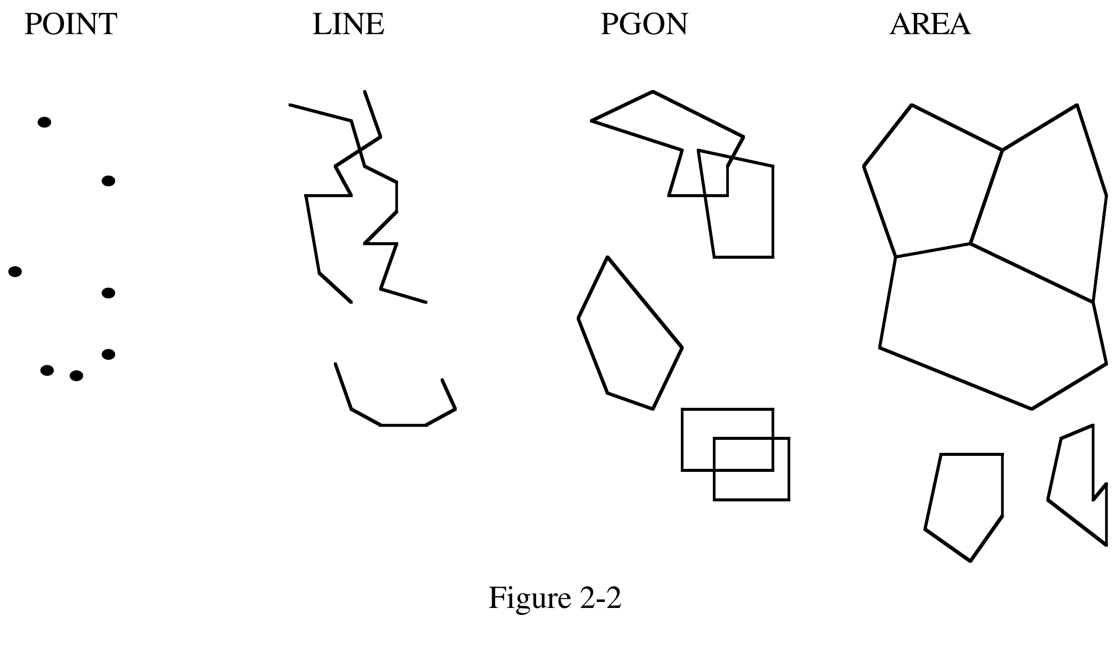


Figure 2-2

Generalizations of the four basic geometric types are REG, EXT, and GEO, referring to any type of region, any extended object (not a point), and any geometric data object, respectively, for instance REG =

{PGON, AREA}, EXT = {LINE, PGON, AREA}, etc. A set identifier occurring in the signature listing is to be interpreted as a shorthand for one corresponding line (class of operators) for each element of the set.

2.4. Geometric Predicates

10.	POINT × POINT	→ BOOL	=, ≠	(_ # _)
	LINE × LINE	→ BOOL		
	REG × REG	→ BOOL		
11.	GEO × REG	→ BOOL	**inside**, **outside**	(_ # _)
12.	EXT × EXT	→ BOOL	**intersects**	(_ # _)
13.	AREA × AREA	→ BOOL	**is_neighbour_of**	(_ # _)

The operators in this group allow to compare two geometric objects in different ways; such a comparison can be part of the parameter expression of a selection or join operation. For instance, it is possible to test whether any geometric object (point, line, or region) is completely inside a region (PGON or AREA). - Here are some examples how the functions associated with operator symbols can formally be defined. Let x, y be the operands.

(x = y)	:=	points(x) = points(y)
(x ≠ y)	:=	points(x) ≠ points(y)
(x **inside** y)	:=	points(x) ⊆ points(y)
(x **outside** y)	:=	points(x) ∩ points(y) = Ø
(x **intersects** y)	:=	points(x) ∩ points(y) ≠ Ø

Note that the description in terms of the associated point sets frees the definition from unnecessary details about the representation of objects. - Not all operators are quite as simple to define as these geometric predicates. Full definitions of the remaining operators are given in [Gü87].

We illustrate the use of these operators by examples. Suppose besides a relation describing cities of the US there is also a relation describing the states:

cities	cname	center	cpop	states	sname	region	spop
	STR	POINT	NUM		STR	AREA	NUM

We may then ask for a list of cities associating with each city the state it is in:

cities states |×| [center **inside** region] π [cname, sname]

As usual the result of the join is a subset of the cartesian product relation such that for each result tuple the condition "center **inside** region" holds. By projection we obtain the desired list. We introduce a further relation for rivers (and one for highways needed later):

rivers	rname	route	highways	hname	way
	STR	LINE		STR	LINE

We may then determine which rivers are totally or partially in California:

states σ [sname = 'California'] rivers |×| [route **intersects** region]

Of course it is also possible to change the condition to "route **inside** region" to find the rivers completely inside California. We may also ask for the neighbour states of California:

states σ [sname = 'California'] states |×| [1.region **is_neighbour_of** 2.region]

Here it is necessary to distinguish between the region attributes of the first and second operand by using a numerical prefix. Sometimes it is easier to formulate a query in a more procedural style by determining intermediate results step by step. For this purpose the query language has been augmented by the possibility to assign a name to the result of an algebra expression and to refer to this result in subsequent expressions. The current example query can alternatively be posed as follows:

(states **extract** [sname = 'California', region]) {CA};
states σ [region **is_neighbour_of** CA]

The result of the first expression is the polygonal area describing the state of California which is called "CA". The effect of the semicolon is to remove the current operand. The second expression can now determine the neighbour states of California by a selection instead of a join. - Note that two areas are considered to be neighbours if they have a common edge.

2.5. Geometric Relation Transformers

14.	LINE* × LINE*	→ POINT*	**intersection**	_ _ #
	LINE* × REG*	→ LINE*		
	PGON* × REG*	→ PGON*		
15.	AREA* × AREA*	→ AREA*	**overlay**	_ _ #
16.	EXT*	→ POINT*	**vertices**	_ #
17.	POINT* × REG	→ AREA*	**voronoi**	_ _ #
18.	POINT* × POINT	→ REL	**closest**	_ _ #

The operators in this group take one or more relations (viewed as sets of geometric objects) as operands and produce a result relation. They also, except for the **closest** operator, construct new geometric objects and embed them into the result relation. For instance, the **intersection** operator can be applied to two sets of lines and it constructs all intersection points between lines of the first operand and lines of the second operand, as illustrated in Fig. 2-3.

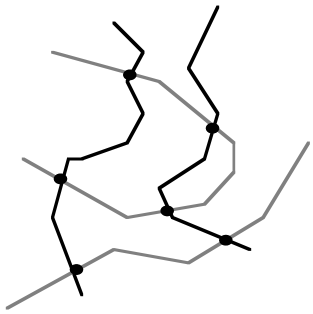


Figure 2-3

It is important to note that a single line of the first operand can intersect a single line of the second operand in several points. The result relation contains one tuple for each intersection point. This tuple consists of all attribute values of the first operand tuple, all attribute values of the second operand tuple and one additional attribute value of type POINT for the constructed intersection point. This is in fact the meaning of the ATTR* notation on the right hand side of an arrow: the result relation will have a new attribute of type ATTR. - The principle explained here for the **intersection** operator applied to sets of lines is the same for the remaining functionalities of the **intersection** operator and the other operators in this group. So for instance the intersection of lines with regions will result in the construction of new lines and each result tuple will contain all information of the participating line and region plus the constructed intersection line.

The **overlay** operator allows to combine two partitions of the plane into disjoint regions (sometimes called *maps*). The result relation contains one tuple for each new region obtained as the intersection of a region of the first operand with a region of the second operand. Note that our definition of AREA attributes does not require that the plane is covered completely by the regions occurring as attribute values. In this case it is possible that a region of the first operand does not intersect any region of the second operand; it will therefore not be part of any result tuple. The **overlay** operator is illustrated in Fig. 2-4.

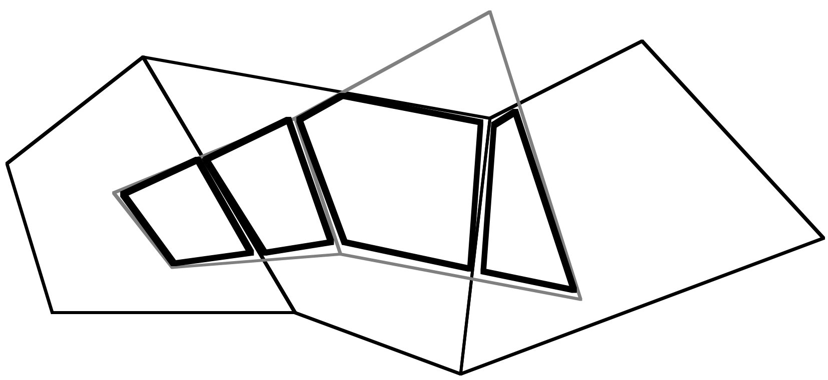


Figure 2-4

The **vertices** operator returns the vertex points of extended objects, for instance, all corner points of a polygon. In the result relation a tuple of the operand relation with a triangular region, say, will appear three times, extended by the different point values of the three vertices. - The Voronoi diagram is a well-known structure in (computational) geometry. For a given set S of points in the plane the Voronoi diagram associates with each point p from S the region consisting of those points of the plane that are closer to p than to any other point in S. Usually some regions of a Voronoi diagram are infinite, namely those belonging to points on the convex hull of the point set. In our framework infinite regions are not allowed. Hence our **voronoi** operator takes as a second operand a polygonal region and constructs the Voronoi diagram for the points inside that region (Fig. 2-5).

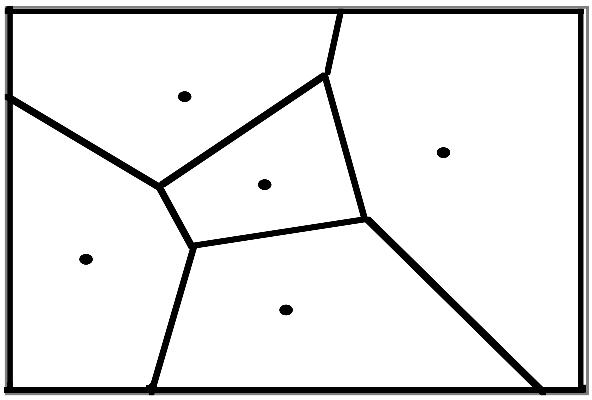


Figure 2-5

Finally the **closest** operator takes a relation, viewed as a set of points, as an operand and returns that or those tuples of this relation whose point component is closest to the point given as a second operand (see Fig. 2-6).

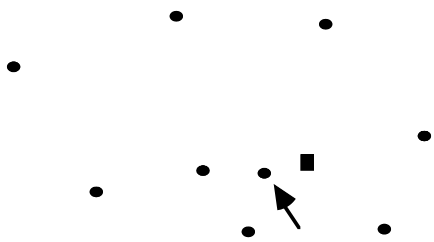


Figure 2-6

We illustrate the use of operators in this group by a few examples. Suppose we want to partition the state of California into "areas of attraction" of big cities (more than 500 000 inhabitants). We can achieve an effect like this by constructing the Voronoi diagram for these cities. We assume that the state area of California has already been determined as shown before and is known as "CA".

cities σ [(center **inside** CA) **and** (cpop > 500000)] CA **voronoi** [center, {attr_area}]

The parameters of the **voronoi** operator are the name of the point attribute of the operand relation and a new name for the Voronoi regions to be constructed. Hence the result relation has the schema:

cities	cname	center	cpop	attr_area
	STR	POINT	NUM	AREA

Here is a somewhat more involved example on the use of the **intersection** operator: "Which city with more than 100000 inhabitants is closest to the point where interstate highway 90 crosses the Mississippi river?"

(highways σ [hname = 'I90'] rivers σ [rname = 'Mississippi']
intersection [way, route, {crossing}] **extract** [**true**, crossing]) {crosspoint};

cities σ [cpop > 100000] crosspoint **closest** [center]

In a first step the intersection point of Interstate 90 with the Mississippi river is determined and called "crosspoint". This is done by selecting the right tuples from the highways and rivers relations and computing the intersections of their respective attributes of type LINE. The query assumes that we know there is only one intersection point (otherwise the verbal question would not make sense). The **intersection** operator will then produce a result relation with a single tuple containing the attribute values of the highway and the river, respectively, and additionally their intersection point in a new attribute called "crossing". The **extract** operator in this case just checks that its operand relation contains indeed only a single tuple and extracts the "crossing" attribute value from this tuple which is then called "crosspoint". - The second part of the query is then a simple application of the **closest** operator to the relation containing cities with more than 100 000 inhabitants.

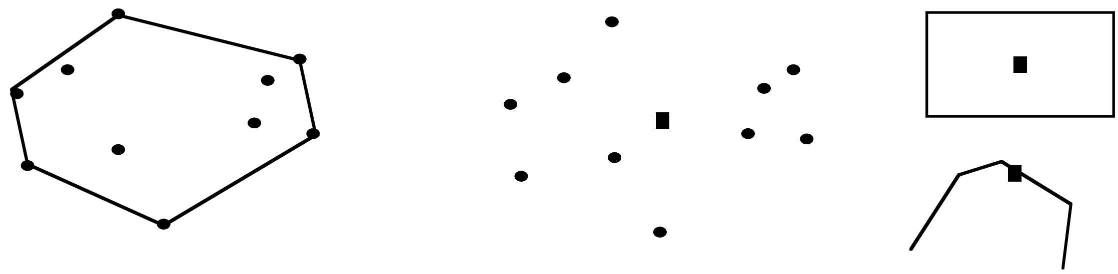
2.6. Operators Returning Atomic Geometric Objects

19.	POINT*	→ PGON	**convex_hull**	(_ #)
20.	POINT*	→ POINT	**center**	(_ #)
	EXT	→ POINT		# (_)

The first operator constructs the convex hull of a set of points, defined as the smallest convex polygon enclosing all points. The second determines the center of either a set of points or an extended atomic geometric object. Syntactically correct expressions are, for example:

(cities **convex_hull** [center])
(cities **center** [center])
center (states **extract** [sname = 'California', region])

The geometric meaning of these operators is illustrated in Fig. 2-7.



convex_hull **center**

Figure 2-7

2.7. Geometric Operators Returning Numbers

21.	POINT × POINT	→ NUM	**dist**	# (_ , _)
22.	GEO × GEO	→ NUM	**mindist**, **maxdist**	# (_ , _)
23.	POINT*	→ NUM	**diameter**	(_ #)
24.	LINE	→ NUM	**length**	# (_)
25.	REG	→ NUM	**perimeter**, **area**	# (_)

The meaning of these operators should be obvious from their name with the exception, perhaps, of the **diameter** operator. The diameter of a set of points is defined as the largest distance between any two points in the set. We give a final example involving the **area** operator: "Which US state has the highest population density?"

states λ [spop/**area**(region) {density}] {xstates} σ [density = (xstates **max** [density])]

For each state tuple its population density is computed and added as a new attribute, the result relation is named "xstates". From this relation the state tuple with maximal value of the density attribute is selected. Note that it is necessary to introduce the name "xstates" for the intermediate result to be able to refer to it within the selection condition.

3. Remarks on Geo-Relational Database Management Systems

Geo-relational algebra is meant to be the user model as well as the formal foundation for the implementation of a geo-relational database management system. The crucial part of the step from geo-relational algebra to a geo-relational DBMS is the inclusion of efficient geometric algorithms and geometric data and file structures. The relationship between relational algebra, geo-relational algebra, a relational DBMS and a geo-relational DBMS is illustrated in Fig. 3-1.

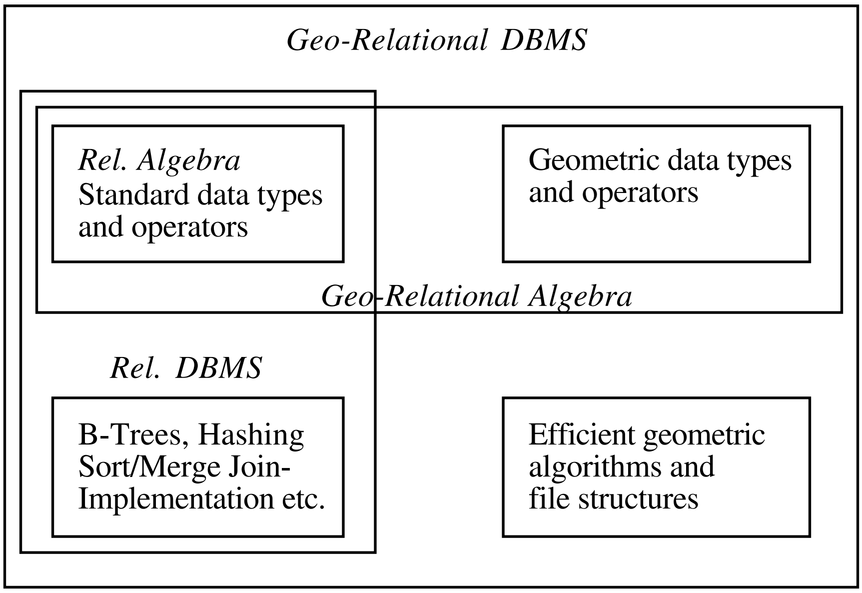


Figure 3-1

A second important issue in the transition to a geo-relational DBMS is the user interface. Clearly the values of geometric attributes cannot be displayed in the usual tabular style nor be entered through a keyboard. In the sequel we first discuss the integration of geometric algorithms into a geo-relational DBMS and then the user interface.

3.1. Implementation

Geo-relational algebra was designed from the beginning with geometric algorithms in mind. The fundamental concept that a column of a relation represents a homogeneous set of geometric objects and that geometric operators of the algebra are applied to such columns harmonizes with the fact that data structures in Computational Geometry are designed to represent such homogeneous sets and that algorithms mostly take one or more such sets as input. Furthermore, relational algebra, rather than the relational calculus, was chosen for the geometric extension, because it is thus possible to establish a direct correspondence between operators of the algebra and well-known geometric algorithms. Here are some examples:

intersection	Bentley/Ottmann, Algorithms for Reporting and Counting Geometric Intersections [BeO79].
overlay	Nievergelt/Preparata, Plane-Sweep Algorithms for Intersecting Geometric Figures [NiP82].
voronoi	Shamos/Hoey, Closest-Point Problems [ShH75]
convex_hull	Graham, An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set [Gr72].

Of course for some of these operators there exist many published algorithms of which the cited one is only a representative.

Geometric predicates, such as **inside** or **intersects**, occur usually in selection or join conditions. For the implementation two basic principles can be observed:

- A selection involving a geometric predicate, when applied to a basis relation (not an intermediate result) should lead to a search on a specific geometric file structure.
- A join involving a geometric predicate is normally executed by a plane-sweep or a divide-and-conquer algorithm.

For instance, suppose the location of the city Baltimore has already been determined and is known as "Baltimore". A selection may then ask for the state containing Baltimore:

states σ [Baltimore **inside** region]

The geometric problem is illustrated in Fig. 3-2 (in principle, no attempt was made to draw a map of the US). If the US states are represented in a specific file structure for geometric subdivision searching (data structures for this purpose are published for instance in [Ki83] and [SaT86]) then this query can be executed efficiently. - The **inside** operator can also occur in a join condition (a previous example), also illustrated in Fig. 3-2:

cities states |×| [center **inside** region]

At least if there is no file structure for geometric subdivision searching available then a plane-sweep through the set of points and regions is probably the most efficient strategy.

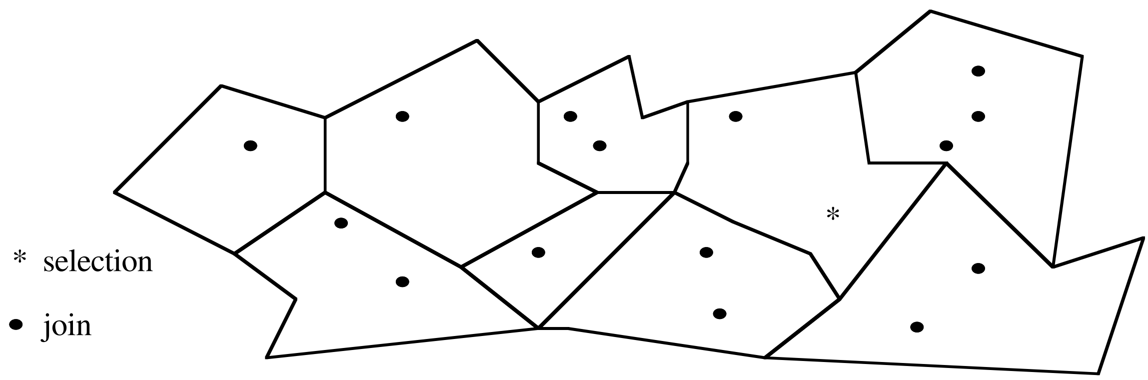


Figure 3-2

The example queries show a further important point: In general, several strategies could be used to obtain the desired result. The selection of the hopefully best algorithm can depend on many factors. Also depending on the existence of a geometric file structure to support a query (*geometric index*) certain methods may or may not be available in the specific case. For instance, if no geometric file structure is present to support the selection, it is necessary to scan all tuples of the relation. Instead of plane-sweep, the join could be performed by repeatedly searching a geometric index for one operand, if present. It might also be worthwhile to first build a geometric data structure representing one operand set and then to repeatedly search this data structure. If the cardinality of the operand sets is very different, in particular, if one operand set has only a single or a few elements, then again it may not be useful to invest the work for building a data structure for one operand set, but one should rather just scan this operand set comparing with the few elements of the other operand set. And so forth.

Hence it is clear that the "optimizer" component of a geo-relational DBMS has at least a similar importance as its counterpart in a "normal" relational system. It is also obvious that the traditional knowledge used in optimizing has to be extended by a good deal of insight about geometric algorithms.

The approach includes a challenge to the field of Computational Geometry. Namely, the standard assumption that all geometric data and data structures fit into main memory is often inappropriate in a database context. A transition must be made from efficient internal data structures for geometric searching problems to (external) geometric file structures based upon the same principles. Algorithms processing sets of geometric objects must be modified to keep at any time only a small part of their data in main memory. A few papers in Computational Geometry have already addressed this problem [SzVW83, OtW86, GüS87].

3.2. User Interface

We assume that a geo-relational DBMS is accessed through a work station with a bitmap screen and a pointing device (mouse etc.). The hardware may also include a graphics tablet or similar input device for geometric data. The result of a query will usually be displayed in two windows on the bitmap screen: The non-geometric attributes of a relation are shown in tabular format in the first window whereas the geometric attributes are drawn in the second (graphics) window. The two windows may interact with each other, for instance, selecting a row (tuple) in the table will visually emphasize the associated graphic object in the graphics window and vice-versa. Very important is the possibility to define interactively an atomic geometric object (by drawing it in the graphics window) and then to use this object in a query. For instance, it is possible to draw an arbitrary region into a display of a map and then to ask how many people live in this area:

cities σ [center **inside** *] **sum** [cpop]

("*" refers to the currently selected graphical object.) One might measure the size of this area by applying the **area** operator. Or one might draw a projected highway into the map and ask for the roads and rivers intersected by it. And so forth.

To make geometric objects drawn in the graphics window recognizable it should be possible to display as a "background" a predefined map of the area. To support entering the query a third "interaction" window might display the schemas of existing relations. Operators might be selected via menus that offer just the operators applicable to the current operand. There are many more possibilities to design a user-friendly interface.

4. Conclusions

In this paper we have described a conceptual model for a geometric database system. For a user the essential aspects of the model are:

(1) High-level geometric objects are available to model geometric aspects of real life entities and one can perform "calculations" with these objects in a manner similar to writing down simple numerical expressions. For instance, it is as easy to write down a test whether two polygons intersect as whether one number is greater than another. We believe a user does not care to see polygons represented by large collections of numbers stored in relations, nor does he/she want to program the intersection test in SQL, say, based on this representation. Our concept is to hide the internal representation of objects entirely; this representation is defined in the implementation language of the database system which is also used to implement operations efficiently based on this representation.

(2) Writing down a query is conceptually just the composition of functions initially applied to objects known to the system (such as stored relations). Since the domain and range types of functions fit together in many ways a very flexible and expressive query language results, as we hope to have demonstrated by the examples. The underlying mathematical concept is that of a many-sorted algebra.

The choice of geometric operators was motivated by the desire to obtain a practically useful query language and also to show how well-known geometric algorithms, such as constructing the Voronoi diagram, can be embedded into a database system. There was no intention to obtain, for instance, a minimal set. We know of no theoretical criteria for deciding which operators should be present. In practice it will surely happen that an operator appears to be missing for a particular application. The only solution to this problem is to keep the architecture of the database system extensible; that is, to allow further types and operators to be added to the system. Their implementation should be of the same quality as that of the original types and operators.

The implementation of a restricted prototype system called "Gral" (also described in [Gü87]) with such an extensible architecture has been started at the University of Dortmund. The hardware is a SUN work station and the implementation language is Modula-2. At the time of writing, simple storage management and catalog components, a rudimentary user interface, and a facility for managing user-defined types are running. Currently a parser for algebra expressions (independent of specific operators), the general mechanism for evaluating queries, the mechanism for adding operators and their implementation procedures to the system, several fundamental operators, and in particular a plane-sweep algorithm for the "point **inside** region" join have been designed and are being implemented. Of course, there is still a long way to go, for instance no index structures exist yet, optimization has not yet been touched, and almost all geometric operators still wait for implementation.

Acknowledgment. Thanks to Gisbert Dröge for many interesting discussions on geo-relational algebra. He in particular studied geometric operators and helped to shape the collection shown in Table 2-1.

References

[BeO79] Bentley, J.L., and Th. Ottmann, Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers C-28 (1979)*, 643-647.

[BeS77] Berman, R.R., and M. Stonebraker, GEO-QUEL: A System for the Manipulation and Display of Geographic Data. *Computer Graphics 11 (1977),* 186-191.

[Bu84] Buchmann, A.P., Current Trends in CAD Databases. *Computer-Aided Design 16 (1984),* 123-126.

[ChF80] Chang, N.S., and K.S. Fu, A Relational Database System for Images. In: S.K. Chang and K.S. Fu (eds.), Pictorial Information Systems, Springer, 1980, 288-321.

[ChF81] Chang, N.S., and K.S. Fu, Picture Query Languages for Pictorial Data-Base Systems. *Computer 11 (1981),* 23-33.

[ChCK81] Chock, M., A.F. Cardenas, and A. Klinger, Manipulating Data Structures in Pictorial Information Systems. *Computer 11 (November 1981),* 43-50.

[Da86] Date, C.J., An Introduction to Database Systems. Addison-Wesley, 1986.

[Day86] Dayal, U., and J.M. Smith, PROBE: A Knowledge-Oriented Database Management System. In: M.L. Brodie and J. Mylopoulos (eds.), On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Springer, 1986, 227-257.

[Fr81] Frank, A., Application of DBMS to Land Information Systems. Proc. of the 7th VLDB Conf., Cannes, 1981, 448-453.

[GoTW78] Goguen, J.A., J.W. Thatcher, and E.G. Wagner, An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: R. Yeh (ed.), Current Trends in Programming Methodology, Vol. IV, Prentice Hall 1978, 80-149.

[Gr72] Graham, R.L., An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters 1 (1972)*, 132-133.

[Gü87] Güting, R.H., Modelling Non-Standard Database Systems by Many-Sorted Algebras. Fachbereich Informatik, Universität Dortmund, manuscript in preparation, 1987.

[GüS87] Güting, R.H., and W. Schilling, A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem. *Information Sciences 42 (1987),* 95-112.

[GüZC87] Güting, R.H., R. Zicari, and D. Choy, An Algebra for Structured Office Documents. IBM Almaden Research Center, San Jose, California, Report RJ 5559, 1987.

[Gu84] Guttman, A., R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. of the ACM SIGMOD Conf. on Management of Data, Boston, 1984, 47-57.

[Ki83] Kirkpatrick, D., Optimal Search in Planar Subdivisions. *SIAM Journal of Computing 12 (1983),* 28-35.

[LiN87] Lipeck, U., and K. Neumann, Modelling and Manipulating Objects in Geoscientific Databases. In: S. Spaccapietra (ed.), Proc. of the 5th Int. Conf. on the Entity-Relationship Approach, Dijon, 1986, North-Holland, 1987, 67-86.

[MaC80] Mantey, P.E., and E.D. Carlson, Integrated Geographic Data Bases: The GADS Experience. In: A. Blaser (ed.), Data Base Techniques for Pictorial Applications, Springer, 1980, 173-198.

[Me84] Mehlhorn, K., Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry. Springer, 1984.

[NaW79] Nagy, G., and S. Wagle, Geographic Data Processing. *Computing Surveys 11 (1979),* 139-181.

[NiHS84] Nievergelt, J., H. Hinterberger, and K.C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems 9 (1984),* 38-71.

[NiP82] Nievergelt, J., and F.P. Preparata, Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM 25 (1982),* 739-747.

[Or86] Orenstein, J.A., Spatial Query Processing in an Object-Oriented Database System. Proc. of the ACM SIGMOD Conf. 1986, 326-336.

[OtW86] Ottmann, Th., and D. Wood, Space-Economical Plane-Sweep Algorithms. *Computer Vision, Graphics, and Image Processing 34 (1986)*, 35-51.

[PrS85] Preparata, F.P., and M.I. Shamos, Computational Geometry: An Introduction. Springer 1985.

[SaT86] Sarnak, N., and R.E. Tarjan, Planar Point Location Using Persistent Search Trees. *Communications of the ACM 29 (1986),* 669-679.

[ScWa86] Schek, H.J., and W. Waterfeld, A Database Kernel System for Geoscientific Applications. Proc. of the 2nd Int. Symp. on Spatial Data Handling, Seattle, WA, 1986, 273-288.

[ScWe86] Schek, H.J., and G. Weikum, DASDBS: Concepts and Architecture of a Database System for Advanced Applications. TU Darmstadt, West Germany, Report DVSI-1986-T1, 1986.

[ShH75] Shamos, M.I., and D. Hoey, Closest-Point Problems. Proc. of the 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, 151-162.

[StRG83] Stonebraker, M., B. Rubenstein, and A. Guttmann, Application of Abstract Data Types and Abstract Indices to CAD Data Bases. Proc. of the ACM/IEEE Conf. on Engineering Design Applications, San Jose, 1983, 107-113.

[SzVW83] Szymansky, T.G., and C.J. van Wyk, Space Efficient Algorithms for VLSI Artwork Analysis. Proc. of the 20th IEEE Design Automation Conference, 1983, 734-739.

[Ta82] Tamminen, M., Efficient Spatial Access to a Data Base. Proc. of the ACM SIGMOD Conf. 1982, 200-206.

[TiMT83] Tikkanen, M., M. Mantyla, and M. Tamminen, GWB/DMS: A Geometric Data Manager. Proc. of the Eurographics Conf., Zagreb, 1983, 99-111.