SPATIAL DATABASES

A *spatial database* contains information about some space in two or more dimensions; this information is managed by a spatial database system. A database system (database management system—or DBMS) in general organizes the data needed within some enterprise or institution in such a form that they can be easily and efficiently accessed by various applications. To this end, it offers to a user a high-level data model, which is some abstract concept of how data are stored, and a query language based on that model. In this way, the real, and far more complex, storage organization of data tuned for efficient access and processing is hidden from the user. For example, a popular data model is the *relational model,* which allows the user to imagine that data are kept in tables. In this model, one could keep information about cities as follows:

cities	name	population	country
	Paris	2,130,000	France
	Athens	885,000	Greece
	New York	8,370,000	USA
	. . .		

The query language for this model allows a user, for example, to select certain rows and columns from this table, yielding another table. The query "Give me the names and population numbers of all cities in France with less than 100,000 inhabitants" could be formulated as

```
select name, population
from cities
where country = "France" and
 population < 100,000
```

After the keyword "select" follow the names of the columns to be shown, after the keyword "from" the names of the tables to be considered (just one here), and after the keyword "where" a condition on the rows. Hence, a database system allows a user to enter ad hoc queries interactively as well as offers data access for application programs. The system analyzes a query and attempts to determine an efficient execution strategy, or *query plan,* which is then evaluated.

A spatial database system is a database system with additional capabilities for managing spatial, or geometric, information. The space to be described can be two dimensional (2D), for example, parts of the surface of the earth; in that case, the application would be geographic information systems. Another 2D space could be the design of an integrated circuit (e.g., VLSI [very large-scale integration] design). Three-dimensional spaces are, for example, the universe (use in astronomy), a part of the human body, such as a model of the brain (medicine), or a model of a protein molecule (chemistry, biology). Spatial database systems offer basic database functionality to deal with spaces and the objects contained in these spaces; usually an application system, such as a geographic information system, has to be implemented on top of the spatial DBMS to arrive at a system suitable for an end user.

There are two major ways that one can perceive, or describe, a space. On the one hand, one can view it as being composed of a collection of objects arranged in space; these objects have a clear identity, location, and extent. On the other hand, one can say for every point in the space what is there. The first view is object oriented, the second image oriented. The second view is in many cases obtained automatically by devices producing images of a space, such as satellite cameras or computer tomographs (taking images of a human body). The techniques used in database systems in the two cases are rather different. Usually the object-centered view is associated with the term *spatial database system* whereas systems that can handle images as such are called *image database systems.* Image DBMSs often provide techniques for feature extraction—that is, methods for recognizing objects in images; this provides a bridge to move from image to spatial databases. In the following we consider only spatial database systems in the restricted sense.

Another distinction has to be made between spatial DBMS and computer-aided design (CAD) database systems. The technology in spatial DBMS is designed for efficient manipulation of large collections of relatively simple geometric objects (for example, for handling 100,000 polygons). CAD databases, on the other hand, contain fewer, but highly structured descriptions of design objects (e.g., cars or planes). These are better treated with object-oriented DBMS techniques.

Spatial database systems extend traditional database systems by supporting the treatment of geometries of objects at the modeling and the implementation level. A fundamental concept is the introduction of *spatial data types.* This builds on the concept of data types in programming languages. The values manipulated in computer programs are classified into groups of similar values called *types;* the type determines which operations can be applied to values. For example, there is a type *integer* to describe whole numbers (. . ., −1, 0, 1, 2, 3, . . .) and a type *string* to describe character strings (e.g., "France"). Numbers can be added, and strings be concatenated, but it makes no sense to add two strings. Hence operations associated with type *integer* are, for example, +, −, and operations for type *string* are *concat,* or *length* (determine the number of characters). By organizing values into types, a compiler for a program can check that operations are correctly applied to values. Also, for each type, a specific storage organization (data structure) can be fixed, and operations be implemented to use that representation.

In a database system, types are used to describe collections of objects, single objects, and properties of objects, called *attributes.* Actually, the types offered, or the facilities for constructing types, are a major distinguishing feature between various kinds of database systems (e.g., relational, or object-oriented, systems). In the relational model, there is a type for tables (called *relation*), a type for rows in a table (*tuple*), and, at least in the classical relational model, a small collection of predefined "atomic" data types for attributes (e.g., *integer, float, string,* and *boolean*). The attribute types used in our example table are captured in a description called a *relation schema:*

```
cities(name: STRING, population: INTEGER,
 country: STRING)
```

The idea of spatial data types is to describe the geometries of objects by means of a set of atomic types, such as *point, line, polygon,* together with operations on these types, such as *inside*(p, q)—check whether point p is located within polygon q, or *intersection*(q, r)—determine the joint area of polygons q and r (this is a set of polygons in general). The geometry of an object can then be treated as an attribute of a spatial data type, so we can now have relations for cities and states:

```
cities(name: STRING, population: INTEGER,
 center: POINT)
states(sname: STRING, language: STRING,
 region: POLYGON)
```

Assuming an operation *distance* applicable to two points is available, we can then formulate a query "Find all cities within 200 km from Paris" as

```
select Y.name
from cities X, cities Y
where X.name = "Paris" and
 distance(X.center, Y.center) < 200
```

The implementation of a spatial database system needs to support spatial data types by offering data structures for the types and code (procedures) for the operations. Furthermore, the techniques for processing sets of objects (tuples) in the database system also have to be extended. One basic problem in set-oriented processing is finding all tuples of a relation whose value for a certain attribute fulfills some condition, called a *selection.* For example, the condition can be `name = "Paris"`. This is supported by an *index,* a data structure that allows one to search for the character string "Paris" and find with very little effort the positions of all tuple representations in storage having the value "Paris" for attribute "name." An index allows one to evaluate a selection without reading the complete representation of the relation. To make this possible for conditions on spatial data types—for example, to find all cities within a given polygon without reading the whole cities relation—one needs specialized index structures. Finally, the planning component of a DBMS, the so-called *optimizer* that tries to find a good evaluation strategy for a given query, must also be extended so that it actually uses the new index structures and other specialized techniques for spatial data processing.

In the following sections we first consider the design of spatial data types and their integration into a DBMS data model and query language in more detail. We then look at implementation issues such as system architecture of a spatial DBMS, representation of spatial data type values, implementation of operations, and spatial index structures. A deeper discussion of these issues can be found in Ref. 1.

MODELING AND QUERYING

Basic Abstractions

For modeling spatial objects, different basic abstractions play a role depending on the dimension of space and the application to be supported. For example, in VLSI design a rectangle is a fundamental entity since chips are defined in terms of large sets of rectangles. We consider in the sequel the modeling needs of geographic information systems that are typical for many applications of spatial DBMSs. Here the basic abstractions for modeling single objects are as follows:

- *Point.* Describes an entity for which only the location in space, but not the extent, is relevant. Examples are cities on a large-scale map, or post offices on a city map.
- *Line.* Describes a facility for moving through space, or connections in space. Examples are roads, rivers, phone lines. A line is viewed as a description of a curve in space; it is usually represented (approximated) by a sequence of straight line segments, a *polyline.*
- *Region.* Describes an entity for which the extent is of interest (e.g., a forest, a lake, an administrative region such as a district). A region may consist of several disjoint pieces each of which may have holes. This is usually represented by a set of polygons with holes.

Besides single objects, one needs to capture collections of spatially related objects. The most important examples are the following:

- *Partition.* Describes a decomposition of some area into disjoint regions (for example, a partition of a country into states, or a land use thematic map).
- *Network.* A network is a spatially embedded graph whose nodes have associated points in the plane and whose edges have associated polylines connecting these points. It can be used to describe highways, rivers, electricity networks, etc.

Figure 1 shows these basic abstractions. Modeling geographic information is discussed in more detail in Ref. 2.

Spatial Data Types

The basic abstractions for single objects as well as their possible relationships can be modeled by designing an appropriate system of spatial data types (including operations). A system of one or more types with operations is also called an *algebra.* An important design criterion is closure under operations. For example, if we form the intersection of two polygons, we obtain in general a set of polygons; hence a type offering just single simple polygons as possible values is not closed. Similarly, the intersection of two polylines can be a set of points.

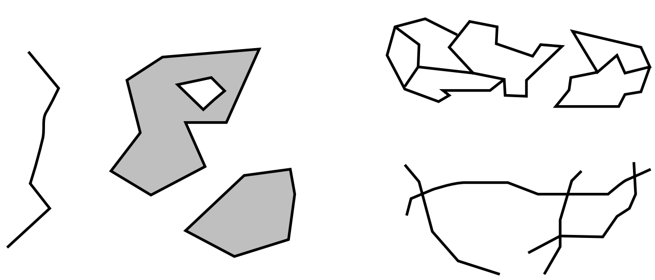

Figure 1. The five basic abstractions for spatial objects (*point, line, region*) and for collections of spatially related objects (*partition, network*).

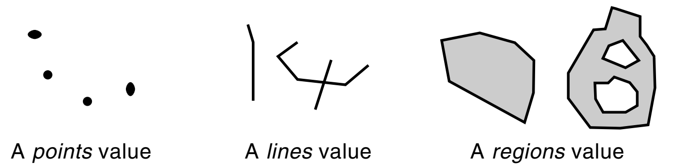

Figure 2. The spatial data types *points, lines,* and *regions* of the ROSE albegra.

An example of a spatial algebra taking this into account is the ROSE algebra (3). It offers three types called points, lines, and regions. A *points* value is a set of points, a *lines* value can be an arbitrary graph structure over line segments, and a *regions* value is essentially a set of polygons with holes, as in the preceding third basic abstraction. The three types are shown in Fig. 2.

Operations on these types can be classified into four groups. For each group, we show a few example operations by giving the *signature,* which describes the argument and result types for the operation. Some operations are applicable to several types. This is described by introducing type variables ranging over sets of types—namely, EXT = {*lines, regions*} and GEO = {*points, lines, regions*}. There are also some operations manipulating sets of database objects (such as cities or states). We denote by OBJ the set of object types in the database (e.g., a city type) and use *set* as a type constructor. Hence if *obj* is a type variable ranging over the types in OBJ, then *set*(*obj*) describes the type of a set of objects of type *obj.*

Spatial Predicates

$\forall$ ext$_1$, ext$_2$ in EXT.		
points $\times$ regions	$\rightarrow$bool	inside
regions $\times$ regions	$\rightarrow$bool	adjacent
ext$_1$ $\times$ ext$_2$	$\rightarrow$bool	intersects

Hence the *intersects* operation is applicable to any combination of *lines* and *regions* values. All spatial predicates allow one to check whether a certain, usually topological, relationship between the two values holds.

Operations Computing New Spatial Data Type Values

$\forall$ geo in GEO.	$\rightarrow$lines	intersection
lines $\times$ regions	$\rightarrow$geo	plus, minus
geo $\times$ geo		

Here *plus* and *minus* compute for any two values of one of the three types the union or difference of the underlying point sets.

Spatial Operations Returning Numbers

$\forall$ geo$_1$, geo$_2$ in GEO.	$\rightarrow$real	area, perimeter
regions	$\rightarrow$real	dist
geo$_1$ $\times$ geo$_2$		

These can be used for numeric measurements.

Spatial Operations on Sets of Objects

$\forall$ obj in OBJ. $\forall$ geo in GEO.
set(obj) $\times$ (obj $\rightarrow$ geo) $\rightarrow$ geo sum

Here *sum* is a spatial aggregate function. It takes a set of objects (of some type *obj* provided by the DBMS) and an attribute of some type *geo* and returns the union of the geometric values for all objects. For example, if the set of objects is a relation describing the U.S. states, then this can be used to compute a *regions* value describing the entire area of the United States.

There exist a few designs of spatial algebras; another example is discussed in Ref. 4. A basic question is whether a collection of operations is complete in any precise sense. At least for topological relationships there have been systematic studies; a fundamental paper in that area is Ref. 5. Another issue is whether data types can be defined in terms of exact Euclidean geometry or must take into account the finite precision available in computer number representations (6).

Spatial Data Types in DBMS Data Model and Query Language

All DBMS data models offer facilities for describing sets of objects, where objects have a number of properties, or *attributes.* Hence spatial data types can be integrated into such models by making them available as attribute types, as described for the relational model in the introductory section.

Spatial data types cover the modeling of the three basic abstractions for single objects. Modeling spatially related collections of objects, such as partitions or networks, may need extensions to the DBMS data model. A partition can be viewed as a set of objects with an attribute of type *regions* with an additional integrity constraint that no two regions in the set overlap. For a network, the graph structure must be described. Basically, one needs object classes for nodes with a *points* attribute (giving the location of the node), object classes for edges with references to source and target nodes as well as a *lines* attribute (to describe the geometry for the edge), and possibly object classes for path entities with references to the nodes and edges forming the path. For describing a highway network, one would model exits and junctions as node objects, connecting pieces of highway as edge objects, and highways themselves as path entities. A model offering such concepts is described in Ref. 7.

Querying can be considered at the level of formal models (e.g., query algebras or calculi) or of "syntactically sugared" query languages such as SQL and OQL. We first consider the integration of spatial data types into algebras (e.g., relational algebra). Essentially one needs to describe operations on sets of objects with spatial attributes. These can be classified as spatial selection, spatial join, spatial function application, and other set operations.

Spatial selection is in fact nothing new; it is just selection based on a spatial predicate. Similarly, spatial join is a join of two relations (object classes, in general) based on a comparison of spatial attributes. Here are two examples (for compatibility with the ROSE algebra we now assume a "cities" relation with attribute "center" of type *points* and "states" with attribute "region" of type *regions*):

```
cities select[center inside Germany]
cities states join[center inside region]
```

Here "Germany" is assumed to be a constant of type *regions.* Spatial selection and join are often based just on the predicates of a spatial algebra. However, the other operations, computing new spatial data type values or numbers, can be used as well in selection or join conditions or to compute new attributes for the result (this is meant by spatial function application). For example, the following query determines for each state its size and returns those with more than 1 million square kilometers:

```
states extend[size: area(region)]
 select[size > 1000000]
```

Here the *extend* operator appends to each tuple a new attribute with name "size" and value computed by the expression *area*(region).

A spatial algebra may provide other operations on sets of database objects (for example, *overlay* for two sets of objects with *regions* attributes, describing partitions). At the level of a query algebra, it is no problem to integrate such operations.

In keyword-oriented query languages such as SQL, it is no problem to formulate spatial selection and spatial join. Spatial function application is also possible in the "where" clause or the "select" clause. Integrating other operations such as *overlay,* however, is a problem since this needs syntactic extensions to the language. Problems with extending SQL are discussed in Ref. 8.

For interactive querying, the user interface has to be extended or specifically designed (9). There must be a facility for graphical display of geometries. It must be possible to overlay the results of several queries, because often the purpose of querying is to construct a map or picture of the space of interest tailored to fulfill some specific information need. Graphical input of geometries must also be possible (e.g., to provide constants used in queries). For example, one can sketch the boundary of some area within a map outline on the screen and then ask in a query for the total population inside that area.

IMPLEMENTATION

System Architecture

We consider a spatial database system to be a regular database system with extensions to accomodate spatial data types. The extensions needed are as follows:

- Representations (data structures) for the types
- Procedures for the operations
- Spatial access methods (index structures) to support spatial selection and spatial join
- Spatial join methods
- Statistical models for sets of objects with spatial attributes (for query optimization)
- Cost functions based on these models for all operations
- Optimization rules, or other extensions of the optimizer, to describe the mapping of spatial operations from query language to query plans
- Extensions of the user interface for graphical representation and input

Clearly, these extensions affect all levels of a system architecture. This is no problem if one starts building a new database system from scratch. It is a problem, however, if one would like to reuse an existing system. Fortunately, there has been a branch of database research since about the mid-1980s called *extensible database systems.* The goal was to build systems in such a way that extensions at all these levels are possible. This research was driven by the need to support new kinds of applications that depend on specialized data types. In fact, support for spatial applications such as geographic information systems has been a major motivation. An extensible system architecture basically leaves the choice of atomic data types open and offers well-defined software interfaces for extensions of the kind mentioned. Research prototypes of extensible systems include POSTGRES (10) and Starburst (11). Recently, the technique became commercially available in systems where one can buy *data blades,* which are really implementation packages of related atomic data types, or algebras. In the following, we briefly discuss representation of spatial data types, implementation of operations, and spatial indexing.

Representation of Spatial Data Types and Implementation of Operations

A data structure to represent, for example, *regions* values, has to be compatible on the one hand with a generic DBMS view for the representation of attribute data types. On the other hand, the data structure should be chosen in such a way that the operations on *regions* can be implemented efficiently. On the DBMS side, a requirement is that an arbitrary size of the representation of a value must be supported, since there is no a priori bound on the number of vertices of the polygons describing a region. However, it is also desirable that the representation of an object, or tuple, should fit into a single page of secondary storage since otherwise buffer management becomes much more difficult. Hence the DBMS needs to provide mechanisms to accomodate such arbitrary growth of the data type representation, preferably in a way that it is hidden from other levels of the system. For example, an attribute value is moved automatically from the compact tuple representation to its own sequence of pages when it grows.

To support efficient implementation of operations, a good idea is to store the plane sweep sequence. *Plane sweep* is a technique used in geometric algorithms (12). In two dimensions, the idea is conceptually to move a vertical line from left to right through the plane and to "observe" the intersections of the line with the geometries to be processed. For example, plane sweep would be used to compute the *intersection* operation of two *regions* values. Plane sweep processes the vertices in *xy*-lexicographic order (that is, in the order of *x*-coordinates, for equal *x* in the order of *y*-coordinates). To save sorting for each operation, one can just store this order in the representation. Other important ideas are the use of approximations, such as bounding boxes (the smallest enclosing rectangle), and precomputing unary function values. Approximations are used with a filter-and-refine strategy. For example, to find all points within a certain polygon, one first searches with the bounding box of that polygon ("filter"). It is much cheaper to compare points with a rectangle than with a polygon with possibly hundreds or thousands of vertices. In a second "refine" step, only points qualifying in the "filter" step are checked for actual containment. Therefore, it is useful to store the bounding box in the value representation. Precomputing unary function values means, for example, that the perimeter and area of a *regions* value are computed once when the value is initially constructed and then stored with the value. When the operation is later used in a query, the value is only looked up.

Spatial Indexing

Indexing allows one to find objects through the values of their attributes. The data types in standard database systems, such as *integer* and *string* form one-dimensional space (that is, are totally ordered). They can be indexed by a tree structure organizing a one-dimensional space, the B-tree. For indexing two- or higher-dimensional spaces, specialized structures are needed. An index structure is, in general, a set of pages of secondary storage. A basic strategy to design such structures is hierarchical decomposition of space. For example, to organize a set of points in two dimensions, we first select an *x*-coordinate splitting the set into two parts of roughly equal size. For each subset, a *y*-coordinate is selected that again splits the subset into parts of equal size. Each split introduces a node in a tree. Splitting stops when the current subset of points is small enough to fit into a single page (Fig. 3).

Given such a tree, one can find all points in a query rectangle *q* by deciding in each node whether *q* overlaps the left subspace, or the right subspace, or both, and then recursively searching the relevant subtrees. In a leaf, one just checks for each point whether it is in *q*.

Since spatial data type values can be complex, usually they are not stored directly in an index. Instead, an approximation, most often the bounding box, is stored. Therefore, spatial index structures are usually designed to store either sets of points or sets of rectangles (in 2D or multidimensional rectangles otherwise). Storing rectangles is more difficult than storing points, since in general space cannot be partitioned in such a way that each rectangle falls into only one cell of the partition (corresponding to one page). Two possible strategies are (1) to allow the regions associated with two subtrees to overlap, or (2) to keep the regions disjoint but split the rectangles to be stored (hence a rectangle will now be stored in several leaves). These strategies lead to the R-tree (13) and the R+-tree (14) structures, which are among the most popular spatial access methods.

There has been a lot of research on spatial index structures, and many structures have been proposed. We can only touch on the issue here. For a deeper discussion, see Ref. 15.

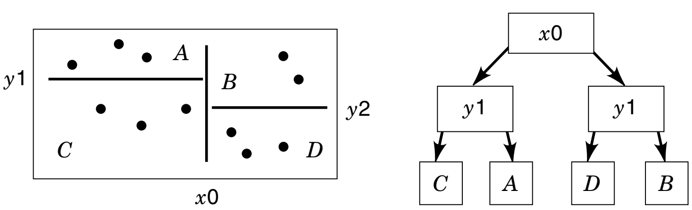

Figure 3. On the left, a set of points is shown within a two-dimensional space. Selecting an *x*-coordinate $x0$ first partitions the space into a left and a right subspace. Each subspace is then further divided by *y*-coordinates $y1$ and $y2$, respectively, into subspaces *C*, *A*, *D*, and *B*. This subdivision is reflected in the tree shown on the right. Here each box represents a node. The leaves are buckets containing the data points in the respective subspaces *C*, *A*, *D*, and *B*.

BIBLIOGRAPHY

1. R. H. Güting, An introduction to spatial database systems, *VLDB Journal,* **3**: 357–399, 1994.
2. T. R. Smith et al., Requirements and principles for the implementation and construction of large-scale geographic information systems, *Int. J. Geograph. Information Systems,* **1**: 13–31, 1987.
3. R. H. Güting and M. Schneider, Realm-based spatial data types: The ROSE algebra, *VLDB Journal,* **4**: 100–143, 1995.
4. M. Scholl and A. Voisard, Thematic map modeling, *Proc. First Int. Symp. on Large Spatial Databases,* Santa Barbara, 1989, 167–190.
5. M. J. Egenhofer, A formal definition of binary topological relationships, *Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms,* Paris, 1989, 457–472.
6. A. Frank and W. Kuhn, Cell graphs: A provable correct method for the storage of geometry, *Proc. 2nd Int. Symposium on Spatial Data Handling,* Seattle, 1986, 411–436.
7. R. H. Güting, GraphDB: Modeling and querying graphs in databases, *Proc. of the 20th Int. Conf. on Very Large Data Bases,* Santiago, Chile, 1994, 297–308.
8. M. J. Egenhofer, Why not SQL!, *Intl. J. Geographic Information Systems,* **6**: 71–85, 1997.
9. M. J. Egenhofer, Spatial SQL: A query and presentation language, *IEEE Trans. Knowl. Data Eng.,* **6**: 86–95, 1994.
10. M. Stonebraker, L. A. Rowe, and M. Hirohama, The implementation of POSTGRES, *IEEE Trans. Knowl. Data Eng.,* **2**: 125–142, 1990.
11. L. M. Haas et al., Starburst mid-flight: As the dust clears, *IEEE Trans. Knowl. Data Eng.,* **2**: 143–160, 1990.
12. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction,* New York: Springer-Verlag, 1985.
13. R. Guttmann, R-trees: A dynamic index structure for spatial searching, *Proc. ACM SIGMOD Conf.,* 1984, 47–57.
14. T. Sellis, N. Rossopoulos, and C. Faloutsos, The R^+-tree: A dynamic index for multi-dimensional objects, *Proc. 13th Int. Conf. on Very Large Data Bases,* Brighton, 1987, 507–518.
15. T. Ohler and P. Widmayer, A Brief Tutorial Introduction to Data Structures for Geometric Databases, In J. Paredaens and L. A. Tenenbaum (eds.), *Advances in Database Systems: Implementations and Applications,* CISM Courses and Lectures No. 347, Vienna: Springer-Verlag, 1994, 329–351.

RALF HARTMUT GÜTING
FernUniversität Hagen

SPATIAL FILTERING. See IMAGE TEXTURE.

SPATIAL INFORMATION SYSTEMS. See GEOGRAPHIC INFORMATION SYSTEMS; SPATIAL DATABASES.

SPATIAL LIGHT MODULATORS. See OPTOELECTRONICS IN VLSI TECHNOLOGY.

SPATIAL POWER COMBINING. See QUASI-OPTICAL CIRCUITS.