Spatial Data Models and Query Processing[*]

Hanan Samet
Walid G. Aref
Computer Science Department and
Institute of Advanced Computer Studies and
Center for Automation Research
University of Maryland
College Park, MD 20742

January 3,1994

Abstract

An overview is presented of the issues in building spatial databases. The focus is on data models and query processing. Query optimization in a spatial environment is also briefly discussed.

Keywords and phrases: spatial databases, data models, spatial query processing, spatial query optimization, relational databases.

[*]This work was supported in part by the National Science Foundation under Grant IRI–9017393. To appear in *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim, ed., Addison Wesley/ACM Press, Reading, MA, 1994.

1 Introduction

Not so long ago the term *database management system (DBMS)* was a euphemism for distinguishing commercial applications (e.g., banking, insurance, etc.) from scientific applications (e.g., number crunching). Today the distinction is rapidly disappearing as users try to come to grips with an information explosion that increasingly involves the world around them. Some new application areas include geographic information systems (GIS), engineering information systems, CAD/CAM, remote sensing, environmental modeling, and image databases. The common thread behind all of these applications is that they make use of spatial data.

Spatial data is a term used to describe data that pertains to the space occupied by objects in a databases. This data is geometric and is varied. It consists of points, lines, rectangles, polygons, surfaces, volumes, as well as time, and data of even higher dimension. Spatial data is usually found in conjunction with what is known as *attribute* or *nonspatial* data (e.g., the name of a river, the type of soil found in a region, the current speed during a time interval, etc.).

Spatial data can be discrete or continuous. When it is discrete (e.g., points in a multidimensional space, or specific instances of time), then it can be modeled using traditional techniques from relational DBMSs. In particular, the coordinate values of the point or the time instant can be treated as additional attributes in a tuple. In contrast, such data as lines, regions, time intervals, etc. is continuous. By *continuous* we mean that the data spans a region in space or time. In other words, the attribute value holds at more than just one point or instance of time.

In this chapter, we focus on the modeling of spatial data and its integration into a DBMS. The result is termed a *spatial database*. Since the application domain is so wide, we restrict ourselves to the requirements and examples for a geographic information system. This chapter is organized as follows. Section 2 gives a brief overview of the type of queries that a spatial database must be able to handle. Section 3 presents a number of different methods of interacting with a spatial database, although the emphasis is on SQL as it is the most commonly used method. Section 4 discusses the integration of spatial and nonspatial data to build a spatial database system. The examples are primarily in the context of relational databases as this is where most of the work has been done. Section 5 describes some of the issues that must be addressed when building a query optimizer for an environment that contains spatial and nonspatial data. Note that we do not elaborate on temporal data, although we do briefly mention the close relationship between temporal and spatial data especially when they are both present (known as a *spatiotemporal database* [Al-Taha et al. 1993]). We also do not dwell on the representation of spatial data. This is the subject of the chapter on spatial data structures.

2 Typical Queries

There are several levels at which queries to a spatial database such as a geographic information system (GIS) can be described. At the highest level, the most common queries are to display the data, to find a pattern in the data, and to predict the behavior of the data at another location or instance of time. These queries are so general that often their execution does not require interacting with a spatial database. Frequently, they can be answered directly. On the other hand, other queries are at such a low level that they don't require use of a DBMS either (e.g., digitization, conversion between different data formats, map projections, enhancement, etc.).

The remainder of the queries fall in an area where a DBMS is useful. These queries can be viewed as dealing with a hierarchy of data [Tomlin 1990]. At the highest level, we have a library of maps (more commonly referred to as *layers*) all of which are in registration (i.e., they have a common origin) and the goal is to perform a sequence of operations on them. Each layer is partitioned into zones (regions) where the zones are sets of locations with a common attribute value. For example, for a given map, we can have a land-use layer, a road network layer, and a pollution layer. In the land-use layer, land is divided into land-use zones (e.g., wet-land, river, desert, city, park, and agriculturaal zones). The road network layer contains the roads that pass through the portion of space that is covered by the map. The pollution layer contains regions with different degrees of pollution. Other possible map layers correspond to vegetation, fire stations, roads, rivers, elevations, etc. Each layer can be viewed as a relation in a relational DBMS (or a class of objects in an object-oriented DBMS). The attributes in the different layers reflect some property of the map. For example, the land-use layer can have the attributes: zip-code, soil type, land-usage, and a spatial attribute of some geometric data type that represents the shape or boundary of each land-use region.

The different queries can be classified in terms of this hierarchy [Tomlin 1990]. Local queries involve locations that are coincident on various layers (e.g., what combination of features are found at location x?). Zonal queries are in terms of groups of locations that have the same attribute value on the same layer (e.g., where does wheat grow?). Focal queries deal with neighborhoods of locations on the same layer. The extent of the neighborhood is usually limited by either distance, direction, or possibly time (e.g., find all wheat growing regions within 10 miles of the boundaries of rice-growing regions). These queries are analogs of range queries in a conventional database management system. The difference is that the shape of the range depends on the extent of a spatial feature (i.e., the area spanned by it) rather than being a hyper-rectangle as in a conventional DBMS (e.g., find all people between the ages of 30 and 35 and who weigh between 150 and 180 pounds).

Using these classifications we can describe in greater depth some of the analytic capabilities that a spatial database must have in order to be able to respond to the queries that are expected to be posed[1]. Local queries consist of retrieval, classification and possibly recoding, generalization (i.e., reducing detail), and measurement (e.g., area, perimeter).

Another important query is known as *polygon overlay* or simply *overlay*. In this case, two layers involving different nonspatial attributes are overlaid and a function is applied to the corresponding attribute values at each location. For example, we can overlay the land-use and pollution layers in order to generate a new layer of land-use/pollution regions. In this case, a wet-land region can get decomposed into several regions, each with a different degree of pollution. Recalling our analogy between layers and relations (or objects), the effect is much like a join operation where the common spatial attribute is the space spanned by the two layers. Thus the result is like a Cartesian product of the two layers. This operation is a special case of what we term a *spatial join*, although the spatial join is far more general

[1]These queries are typical of a GIS environment and where necessary we give a brief definition in parentheses.

since it can involve relations more general than layers as long as the operand relations have spatial attributes that span the same underlying space.

Zonal queries are often used to implement a special case of polygon overlay. A typical zonal query involves three layers. The first two layers are the operands to the query, while the third layer contains the result. The first layer serves as a mask that partitions the second layer into zones where the value of each location in the third layer is the result of the application of the designated zonal query to all locations in the second layer that coincide with each zone. An example of a zonal query is to find the average rainfall for each region where a particular crop or crops are grown. Notice the distinction from the polygon overlay in that a zonal operation does not create new zones whereas a polygon overlay does since its result is the Cartesian product of the two operand layers. Thus the result of a zonal operation is more like a spatial selection.

Focal queries include search, proximity determination (e.g., Voronoi diagrams which, given a set of points termed *sites*, partition the plane into regions such that each point in a given region is closer to a particular site than to any other site), spatial region queries (also known as *buffers* or *corridors*), interpolation, generation of triangulated irregular networks (*TINs*) for dealing with surface data, etc. Connectivity queries are also closely related to focal queries. They involve factors such as flow and visibility.

An alternative, and even simpler, way to classify queries is on the basis of whether they are location-based (e.g., what grows at location x?) or attribute-based (e.g., where does wheat grow?). This classification has a direct bearing on the way the spatial data is organized and is discussed in greater detail in the chapter on spatial data structures.

3 Spatial Query Languages

The design of a proper query language (termed *interaction method* here) for a spatial database is a nontrivial task. For the sake of this discussion, we say that the interaction takes the form of a *command* or a *probe*. We use the identity and sophistication of the user to distinguish between these two rather similar concepts. In the case of a command, the user generally knows exactly what he wants to do and there is a premium on the avoidance of ambiguity. Thus the command can be given using a predefined syntax that simplifies its processing. The objective is to be concise and exact. On the other hand, in the case of a probe, the user does not know exactly what he wants. He is performing data exploration. Frequently, he has little or no knowledge of the nature of the data that is stored in the database, and even less about how it is represented. However, he may have a visual concept of what he is trying to achieve. Thus the querying may use the display of previous results by means such as pointing.

Attention must also be paid to the actual interaction with the spatial database. Surprisingly, in most systems this is usually an afterthought. Natural language is one approach. However, its drawback is its inherent ambiguity. This can be overcome by using a graphical user interface (e.g., [Scholl and Voisard 1992; Vijlbrief and van Oosterom 1992]). Unfortunately, designing a graphical user interface to a GIS involves more than just replacing typed commands by menus. It is important to realize that spatial data is characterized by geometry and map display. Tabular concepts and representations, as are common in some SQL-based approaches, do not play as prominent a role. Instead, we may need to develop some uniformly accepted iconic representations for spatial concepts.

A number of interaction methods are possible. They range from SQL to queries by example. In most cases, the emphasis is on retrieving data using a standardized language. This is coupled with a desire to provide sophisticated methods to formulate more complex commands and logical combinations thereof. Some of the issues that arise are a result of the presence of abstractions of spatial data that are more complex than relations. Also, there is a need to support the graphical display of the results of queries. For example, in the case of points, it is preferable to display them rather than to list the values of their coordinates. Further utility is provided by also displaying some context with the points, or even highlighting them by use of techniques such as reverse video, blinking, or simple circling or boxing.

Interaction with spatial data should be graphical. This means that both the input and the output should be graphical. One interaction technique is for the user to draw the shape of the desired output. Selection can be achieved by pointing; however, this is ambiguous if several objects are associated with the same region. Selection can also be accomplished by wandering around an area (e.g, a pan operation) or by zooming. The use of graphical renderings is also important. For example, using the same rendering for different objects at different locations conveys a notion of similarity, while the dissimilarity of renderings emphasizes differences between the objects. Of course, a legend is also needed to convey the semantics of the different renderings.

A good inspiration for the design of the interface is to make use of classical cartographic concepts [Tufte 1983; Tufte 1990]. For example, color is often used to indicate depth and height. A legend is useful to convey information about a map. Operations between maps are usually performed by overlaying them. This means that the individual map layers should be iconized and a hierarchical mechanism developed to facilitate the expression of their interrelationship. The key is to study the daily workings of a cartographer.

Here we focus on the use of SQL as it is the most commonly used method of interacting with a relational database. In fact, at times, it is also used to interact with an object-oriented database (e.g., [Deux, O. et. al 1990]). Thus it is often proposed to use it with a spatial database as well (e.g., [Aref and Samet 1991a; Gadia 1993; Roussopoulos et al. 1988; Scholl and Voisard 1992]). Note that it has been argued (e.g., [Egenhofer 1992]) that SQL is the wrong approach. These arguments are based on the inability to refer to the results of previous operations or on how the output is to be displayed. Similarly, selection by pointing is difficult to achieve. The problem is that SQL is primarily a means to retrieve from a tabular representation while spatial applications often require retrieval from a graphical representation. This is a reformulation of our earlier criticism that spatial data cannot always be represented as a point in a higher dimensional space (which is what a tuple really is — for more details on these issues, see the section on spatial indexing in the chapter on spatial data structures).

Of course, in order to use SQL with spatial data we need to extend it. There are a number of choices. Some systems extend the SQL grammar by adding a set of spatial operators (e.g., PSQL [Roussopoulos et al. 1988]). In essence, the idea is to extend the standard `select ... from ... where ...` syntax to also include spatial operators and relations involving spatial data. Its drawback is that operators cannot be added at runtime which limits its extensibility. An alternative approach is for the query language to permit user-defined functions and operators which are made known to the DBMS at runtime (termed *registering* [Stonebraker and Rowe 1986; Haas et al. 1990]). They can be scalar (e.g., area), or multivalued in which case the result can be a relation (termed a table function [Haas et al. 1990]).

In the rest of this section we discuss how to incorporate the spatial attributes into the relation. For the present, let us associate the spatial attributes with the tuple [Aref and Samet 1991a; Roussopoulos et al. 1988]. In this case, the spatial attributes appear at the same conceptual level as the nonspatial attributes. Thus the value of a particular spatial attribute is common to all of the nonspatial attributes in the tuple.

3.1 One Spatial Extension to SQL

3.1.1 Spatial Data Definition

Using the above method of incorporating spatial attributes, we now show how SQL would be used in a spatial database by examining a typical situation. Consider a set of objects (e.g., points and lines) in two-dimensional space, and a set of features which partition the space into nonoverlapping or overlapping regions. We use the following two schemas. The first schema is for a relation containing line segments called `roads`, and the second is for a relation containing areas called `regions`. The `roads` relation has one spatial attribute `road_coords` of type `LINE_SEGMENT`. The `regions` relation has one spatial attribute `region_location` of type `REGION`. Of course, other spatial data types are available such as `POINT`, `POLYGON`, `BOX`, etc.

```
create table roads
      (road_id NUMBER,
       road_name CHAR(30),
       road_type CHAR(30),
       road_coords LINE_SEGMENT); /* spatial attribute */

create table regions
      (region_id NUMBER,
       region_name CHAR(30),
       region_zip_code NUMBER,
       region_utilization CHAR(30),
       region_importance NUMBER,
       region_location REGION);  /* spatial attribute */
```

3.1.2 Spatial Data Manipulation

The standard relational operations such as projection, selection, and join are available in the nonspatial domain and are also adapted to the spatial domain. There are many ways of characterizing an SQL predicate as being spatial or relational (equivalent to nonspatial). Here we characterize a predicate as spatial if the condition (i.e., in the `where` clause) involves at least one of the spatial attributes or a spatial operation (e.g., intersection); otherwise, the predicate is characterized as relational. This is the case even if the predicate results in a map as is the case whenever all the attributes are selected and at least one of the attributes is spatial. This can be seen by the following example which selects all regions whose importance is greater than 5. In particular, the result is a new relation which means a new region map as well. This new relation (and map) contains all the tuples selected in the operation (i.e. it will contain the regions with importance values > 5).

```
select all
       from regions
       where region_importance>5;
```

Spatial and relational projections are distinguished in the same way as spatial and relational selections. The difference is that in the case of projection only a subset of the attributes are selected. In fact, if none of the selected attributes are spatial, then only a relation results (and not a map).

Spatial conditions typically consist of comparisons involving the result of the application of functions of spatial attributes. Some examples are given below:

```
  area(region_attr)>val
  perimeter(spatial_attr)
  centroid(region_attr)
  object_at(spatial_attr,location)
  in_circle(spatial_attr,location,radius)
  nearest_to(spatial_attr)=a
  length(line_attr)>val
  in_window(spatial_attr,x1,y1,x2,y2)
```

3.1.3 The window operation

As an example of the use of a spatial condition, consider the following command which combines a spatial selection (i.e., a window operation which is the same as a rectangular range query) with a nonspatial selection. The result is a map containing just the part of the freeways contained in the space spanned by the window. In addition, we have a new relation whose tuples correspond to the freeways that lie in the window.

```
select all
       from roads
       where in_window(road_coords,w) and
             road_type=freeway;
```

The window operation is a special case of a set-theoretic operation in that we are taking the intersection of the space spanned by a regular map and an otherwise empty map containing the space spanned by the window. The attribute values of the regular map in the selected area are retained as a result of this operation. Using our classification of operations it is a local operation as it involves corresponding locations in two layers. Given one or more spatial attributes, set-theoretic operations can be applied to them as well. For example, suppose we wish to find the names of all the roads that pass through College Park. It is executed by intersecting the `region_location(s)` of the tuple(s) whose `region_name` is `College Park` with the `road_coords` attribute of the road map.

```
select road_name
       from roads regions
       where region_name=College Park
             and intersect(region_location,road_coords);
```

3.1.4 Spatial Join

When the intersection involves more than one spatial attribute, the operation is a spatial join rather than a spatial selection. The reason is that we are now combining two relations. In particular, we know that a θ-join is a join that involves comparisons between attributes that are θ-comparable, where θ is a comparator. If the comparator is based on spatial attributes, then the command is said to be a *spatial join*, while if the comparator is based on a nonspatial attribute, then the operation is called a *nonspatial* or *relational join*.

To understand the meaning of the spatial join better, suppose that in our example we selected all the attributes instead of just the road names (i.e., we ignore the projection implied by the fact that we only selected the nonspatial attribute corresponding to the names of the roads). The result of the join is a new relation consisting of a merge of all tuples whose corresponding spatial attributes are in College Park. This new relation contains all the attributes of the two participating relations including the two spatial attributes (i.e., `region_location` and `road_coords`) and their corresponding maps. Some other examples of spatial join operations include determining all spatial features that are adjacent to other spatial features, within a certain distance of other spatial features, contained in other spatial features, etc. Interestingly, the spatial join may involve the same relations. For example, to find all regions adjacent to Universities we would use the following command.

```
select all
       from l regions, k regions
       where adjacent_to(l.region_location,k.region_location)
             and l.region_utilization=university;
```

In this case, the resulting relation will have two spatial attributes corresponding to `region_location` and two maps – that is, one map for the university regions and one map for the neighboring regions.

It is interesting to note that in the literature, the term *spatial join* has a number of meanings. For example, in [Orenstein and Manola 1988], Orenstein uses the term spatial join to mean, in the context of this chapter, the spatial join intersection operation. Ooi [Ooi 1988] does not distinguish between spatial join and spatial selection. They use the term spatial join to mean both join and selection. They perform selections by fixing one of the arguments of the spatial join. However, no join action, in the conventional database sense, takes place in this case. In fact, a window operation can be viewed as a selection operation rather than a spatial join with a constant object (the window) that has no further nonspatial attributes. Güting [Güting 1989] defines spatial joins and spatial selections in essentially the same way as done here.

3.2 An Alternative Spatial Extension

The approach that we have described above associates the spatial attributes with the tuple. In particular, the spatial attributes appear at the same conceptual level as the nonspatial attributes – that is, the value of a particular spatial attribute is common to all of the nonspatial attributes in the tuple. An alternative approach associates the spatial attributes as a sub-hierarchy of the nonspatial attributes [Gadia 1993]. This is equivalent to distributing the spatial attribute across all of the nonspatial attributes. The drawback is that we have now established a hierarchy where the spatial attributes are subservient to the nonspatial attributes.

As an example of this alternative, consider the `regions` relation. Let us add the attribute `region_soil`. In this case, the spatial attribute `region_location` no longer appears with the nonspatial attributes in the schema. Instead `region_location` is declared at the top level with the name of the relation. Let the new relation be called `new_regions`. This means that `region_location` is to be distributed across the nonspatial attributes and `region_location` has a value for every nonspatial attribute in each tuple. In fact, the nonspatial attributes in each tuple may now have several values if the corresponding spatial attribute requires decomposition. This is the case when the `region_name`, `region_zip_code`, and `region_soil` values are not the same for the entire region represented by the tuple.

```
create table new_regions
      (spatial_attribute:  region_location REGION)
      (region_id NUMBER,
       region_name CHAR(30),
       region_zip_code NUMBER,
       region_utilization CHAR(30),
       region_importance NUMBER,
       region_soil CHAR(30));
```

For example, suppose that we have a tuple t with `region_utilization = university`, `region_id = 55`, and `region_importance = 5`. This area has more than one name (e.g., `R1` is `Beltsville`, `R2` is `College Park`, and `R3` is `Adelphi`), more than one zip code (e.g., `R4` is `20740` and `R5` is `20742`), and more than one soil type (e.g., `R6` is `sand` and `R7` is `clay`). Figure 1 shows the resulting decompositions of the region. Tuple t now has multiple-valued attributes. This must be expressed in the tuple. One way to incorporate this approach in an instance (i.e., tuple) of the `new_regions` schema is given below.

```
tuple in table new_regions
     (region_id:  55,
      region_name:
```

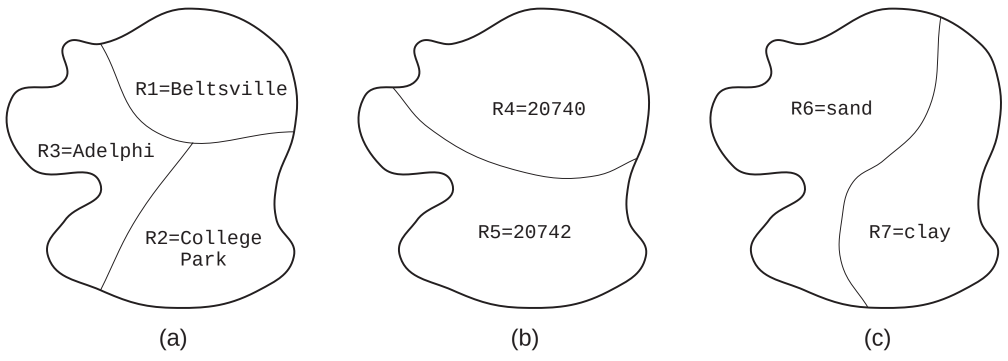

Figure 1: Three partitions of the same region: (a) by region name; (b) by zip code; and (c) by soil type.

```
    {R1:  Beltsville,
     R2:  College Park,
     R3:  Adelphi},
  region_zip_code:
    {R4:  20740,
     R5:  20742},
  region_utilization:  university,
  region_importance:  5,
  region_soil:
    {R6:  sand,
     R7:  clay});
```

Using this approach, we can also separate the spatial conditions from the nonspatial conditions. This is the approach taken by Gadia [Gadia 1993] who adds a `restricted to` clause to the standard SQL command. In this case, the command takes the form `select` ... `restricted to` ... `from` ... `where` .... The condition in the `where` clause only includes the nonspatial attributes, while the condition in the `restricted to` clause deals with the spatial data.

For example, suppose that we wish to find the name of the regions covering a university whose soil type is `sand` and are in the `20742` zip code. The query is formulated as follows:

```
select region_name from new_regions
       restricted to region_soil=sand ∩ region_zip_code=20742
       where region_utilization=university;
```

This query is executed by first selecting all the tuples whose `region_utilization` value is `university` and then examining their corresponding spatial attributes. In particular, we restrict our view to the locations that are of soil type `sand` and in zip code `20742`. Now, we return the names of the regions that overlap these locations. This can be achieved by intersecting the result of the `restricted` clause with the corresponding spatial attribute value of `region_name`. Notice that what we have done is define the semantics of a projection operation in such an environment.

Gadia [Gadia 1993] proposes to use this approach to deal with spatiotemporal data. The temporal attribute is treated in the same way as the spatial attribute. The difference is that the temporal attribute is restricted to points or intervals by the very nature of the time dimension. In contrast, there is no such restriction on the spatial dimension. It is also interesting to note that SQL commands can be nested within the `restricted` clause.

4 Integration of Spatial and Nonspatial Data

The key issue in building a spatial DBMS is deciding how to integrate the representation of spatial and nonspatial data. In Section 3, a number of methods for interacting with such a database were proposed. This was done in a manner that was largely independent of the underlying architecture of the system. In this section we discuss a number of spatial DBMS architectures. Since the field is still developing, most of these architectures are research prototypes rather than commercial systems.

Some researchers use the classifications dedicated, dual, layered, and integrated to distinguish between the different architectures [Vijlbrief and van Oosterom 1992]. In the following, we first give an overview of the issues that arise when using these architectures. This is followed by a more detailed comparative discussion of features of some research prototypes.

4.1 Dedicated Systems

Many prototype systems are suggested that support spatial objects. The principal shortcoming of these systems is their development path. One common path is as a dedicated system with the purpose of supporting applications in a specific domain (e.g., CAD databases) without a full understanding of database issues such as the absence of high-level data definition facilities (e.g., [Shaffer et al. 1990; Tomlin 1990]). Also, these systems are not easily extendible in the sense that it is difficult to modify them to perform actions not previously envisioned by the system's designers. An alternative path is a general database tool that supports a wide variety of applications often without a complete understanding of the requirements of such applications. In both cases, the mentioned shortcomings lead to a reduction in the efficiency of the data processing capabilities of the system.

4.2 Dual Architectures

Dual architectures are based on distinguishing between the spatial and nonspatial data by using different data models for them. Examples are ARC/INFO [Peuquet and Marble 1990], SICAD [Schilcher 1985], etc. Communication between the systems is via common identifiers. The shortcomings are traditional database issues such as synchronization, locking, integrity (e.g., the results of actions in the spatial data model might not be reflected in the nonspatial data model, etc.).

A dual architecture implies the existence of two storage managers. This can be avoided by storing spatial data in a purely relational data model. This means that spatial data must be transformed (e.g., by use of methods such as representative points) or decomposed into constituent pieces (e.g., a region boundary into a sequence of line segments or a region interior into a set of blocks or pixels). This approach implies a hierarchy and in fact is the basis of the layered architecture. In this case, we have a GIS at the top layer, followed by a spatial support layer, followed by a relational DBMS at the bottom layer. Examples of this approach include SIRO-DBMS [Abel 1989] and GEOVIEW [Waugh and Healey 1987]. Of course, the relative order of some of the elements in the hierarchy can be changed. For example, in [Gadia 1993] spatial and temporal data are placed below the relational layer (see the discussion at the end of Section 3).

4.3 Integrated Architectures

An integrated architecture is the most general. It involves users extending the DBMS with their own abstract data types so as to provide better support for spatial applications. This extension frequently involves making use of an extensible database management system (e.g., [Carey et al. 1988; Güting 1989; Haas et al. 1990; Schek and Waterfeld 1986; Stonebraker and Rowe 1986]) as well as object-oriented DBMSs (e.g., [Deux, O. et. al 1990]). Such systems attempt to provide a generalized DBMS that facilitates the support of unconventional applications such as spatial databases. These systems add some new constructs to offer additional modeling power. Included among the new constructs is support for abstract data types (e.g., [Carey et al. 1988]), procedural fields (e.g., [Stonebraker and Rowe 1986]), complex objects (e.g., [Carey et al. 1988; Kim et al. 1987]), set-valued attributes (e.g., [Zaniolo 1983]), etc.

Users of integrated architectures are motivated in part by a belief that each of the data types (i.e., spatial and nonspatial) should be represented by an appropriate data structure that suits its operational needs. This has been done primarily by extending the relational model (e.g., see [Stonebraker 1986]). At times, the extensions are such that the result is a layered architecture. In such cases, the system can be described as belonging to both architectures. Nevertheless, we describe it in conjunction with integrated systems as we feel that being an extension of the relational model is the systems' most important characteristic.

4.4 Prototype Systems

SIRO-DBMS [Abel 1989], GEOVIEW [Waugh and Healey 1987], SAND [Aref and Samet 1991a], Gral [Güting 1989], Probe [Orenstein and Manola 1988], GEOQL [Ooi 1988], Geo-Kernel [Schek and Waterfeld 1986], and GEO++ [Vijlbrief and van Oosterom 1992] are examples of systems based on extending the relational model. Some systems (e.g., [Abel 1989; Orenstein and Manola 1988; Waugh and Healey 1987]) implement a spatial database on top of a relational DBMS with minor changes to the relational system. Changes are in the form of shells outside the DBMS (as in [Abel 1989]). In most of these systems spatial data is flattened into the relational format which means that spatial data is treated as if it is regular attribute data.

GEO++ [Vijlbrief and van Oosterom 1992] is built on top of the POSTGRES [Stonebraker and Rowe 1986] extensible DBMS. It makes use of the primitive data types point, line segment, path, and box provided by POSTGRES to define other data types as well as operators.

Probe [Dayal and Smith 1986; Orenstein and Manola 1986; Orenstein and Manola 1988] contains a general geometric object class: a point set, but no specific types (e.g., point, line, etc.). It treats space and time as generic types with the same status as integers, floating point numbers, strings, etc. Probe does not have a spatial query language. However, Probe does provide a general method for indexing the space by linearizing the spatial index to one-dimension through the use of bit-interleaving techniques and storing the resulting values into an attribute that is indexed by a B-tree. The user can add spatial data types based on his application needs as a specialization of the point set type, and index the underlying space using this general spatial attribute. One disadvantage is that Probe limits itself to space-filling curve representations of spatial data when there are other interesting spatial data structures that can be used.

Geo-Kernel [Schek and Waterfeld 1986; Wolf 1989] implements a geometric data model on top of the DASDBS kernel system [Schek and Waterfeld 1986; Wolf 1989] that supports Non-First-Normal-Form (NF^2) relations [Schek and Scholl 1989]. The implementation focusses on efficient processing of window queries with possible feature selection from the window. Also, in contrast to the assumptions made in other systems (e.g., SAND [Aref and Samet 1991a]), Geo-Kernel models sets of objects (e.g., a set of point, a set of lines, or a set of regions) as atomic objects. A set of objects is the atomic unit of processing for each geometric operator.

Other systems (e.g., Gral [Güting 1989] and GEOQL [Ooi 1988]) extend the relational model a step further to achieve efficient spatial data processing. Spatial data is stored in separate spatial data structures. Spatial operations are executed on top of these structures. They build on the early work and ideas in [Stonebraker 1986].

Gral [Güting 1989] permits user-defined data types and operations. It provides an integrated data model and query language for geometric applications. The user interface is algebraic and procedural. It uses a many-sorted algebra both as an algebraic query language and as an executable language to describe query plans (although with lower-level primitives and operators).

4.5 Linking Spatial and Non-spatial Data

One important issue that arises in the design of systems based on an integrated architecture is how to link the spatial data description (stored in some data structure) of an object with the rest of the object's nonspatial description. Many systems are biased towards either the spatial or the nonspatial aspect of the system (e.g., GEOQL [Ooi 1988]). GEOQL extends SQL to support spatial applications. The underlying architecture is composed of an SQL backend, a spatial processor, and an extended optimizer. In GEOQL, each relation is assumed to have only one spatial attribute. Multiple spatial representation of an object is difficult under this model (e.g., modeling a city once as a point in a point map and once as a region by defining its bounding perimeter). GEOQL is biased towards the relational component in several aspects. In particular, even though spatial data structures are maintained, spatial operations cannot be composed directly without building intermediate database relations. This limits the efficiency of spatial query processing.

In SAND [Aref and Samet 1991a] spatial and nonspatial data are linked bidirectionally. A spatial data structure is associated with each spatial attribute in the schema and is used to store all data instances of that spatial attribute over the set of homogeneous objects (e.g., line data in a road network). The spatial data structure that is chosen depends on the attribute's spatial data type (e.g., point, line, region, etc.). The spatial data structure serves as an index for spatial objects and an environment for the execution of spatially-related operations (e.g., image rotation and scaling, polygon intersection, area computation, proximity queries).

The data instances of the set of nonspatial attributes are stored in database relations. Each tuple in the relation corresponds to one object. Two logical links are maintained between the spatial and nonspatial data instances of an object: *forward* and *backward* links (see Figure 2). The linked instances and the links form what is termed a *spatial relation*. Forward links are used to retrieve the spatial information of an object given the object's nonspatial information. Backward links are used to retrieve the nonspatial information of an object given the object's spatial information. Since the nonspatial information of an object is stored in a tuple, the backward link can be the tuple-id. On the other hand, since spatial data structures contain all the spatial information necessary to identify an object, the forward link can be a representative value of one part of the object that uniquely selects the object. For example, in the case of nonoverlapping region data, an example of a forward link is a candidate point inside the region.

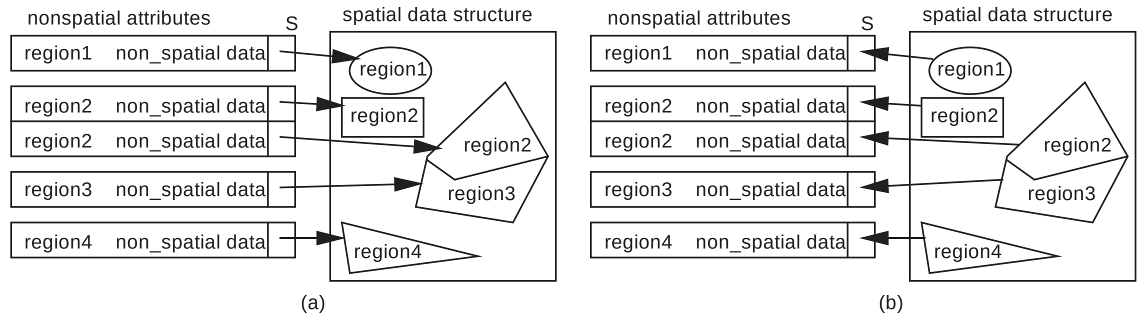

Figure 2: (a) Forward links and (b) backward links for the regions relation.

Maintaining forward and backward links between the spatial and nonspatial parts of a set of objects facilitates browsing in the two parts and permits efficient query processing. Flexibility in the interaction between spatial and nonspatial attributes enable operations (whether spatial or nonspatial) to be performed in their most natural environment.

The need to support bidirectional links between database objects has been realized as early as hierarchical and network database management systems (e.g., IMS [McGee 1977] and CODASYL DBTG [CODASYL 1971]). They are also proposed in [Lorie and Meier 1984]. In [Lorie and Meier 1984] arbitrary links (via references and identifiers which could be used to implement bidirectional links) are adapted to geographical databases where a geographical object is represented by a tuple having some unique identifier and a long field to store the geographic representation of the object in addition to other nonspatial attributes. Links are used to express explicit relationships between geographical objects (e.g., parent-child, or reference relationships).

Bidirectional links are also used in Geo-Kernel [Schek and Waterfeld 1986; Wolf 1989] in order to allow each of the spatial and nonspatial parts of an object to access the other part. Spatial objects are partitioned into cells. Cells are clustered according to their two-dimensional neighborhood. The address of the nonspatial description of an object is stored with each cell of this object. Since Geo-Kernel uses nested relations [Schek and Scholl 1989], an attribute of type relation (i.e., a sub-relation) is used in order to store all the addresses of the cells comprising a spatial object. This resembles a forward link in a spatial relation in SAND [Aref and Samet 1991a]. However, in contrast to sub-relations, in a spatial relation only one spatial identifier is stored as a forward link and it is the responsibility of the spatial data structure (or its encapsulating spatial process or ADT) to extract the rest of the spatial description of an object. This greatly reduces the storage overhead from the relational side as well as reduces the cost that must be borne in systems such as Geo-Kernel to maintain the set of addresses stored in a sub-relation that refer to different cells of the same object.

The idea of introducing abstract data types (ADTs) as new attribute domains into a relational database system and supporting user-defined index structures appears in [Stonebraker 1986]. Following this approach, spatial data structures are viewed in many extensible database systems (e.g., [Güting 1989; Roussopoulos et al. 1988; Stonebraker and Rowe 1986]) as indexing structures. As a result, only backward links (i.e., from the index structure to the actual tuple) are needed in these architectures. In contrast, it is important to note that in SAND's spatial relations [Aref and Samet 1991a], user-defined data structures serve not only as spatial indexes for speeding up operations but also as containers for the full description of spatial data. In the latter, the bidirectional links between corresponding components of the ADT can be used for efficient access. Spatial relations can be viewed as a merge of the ADT work in [Stonebraker 1986] and the complex object support in [Lorie and Meier 1984]. In a spatial relation, a spatial attribute is an ADT that is implemented by user-defined data structures. Functions may be applied to an instance of one spatial attribute in a given tuple (e.g., computing the area or perimeter of a region where a particular crop is grown), or to the whole relation (e.g., find all the crops that are grown in a particular query).

Gral [Güting 1989] avoids forward links (and hence does not use containers) by employing spatial data structures as indexes that approximate spatial objects (e.g., by using bounding boxes). In addition, the full description of the spatial object (e.g., the coordinate values of the vertices of a polygon) is stored with the object's corresponding tuple in an on-line format (i.e., formatted in a memory-mapped image such that when the tuple is loaded into main memory, the spatial description of the object is ready for use without any additional format conversions). In this case, when a query is posed, the spatial data structure is used as a filtering step that produces a set of candidate objects, then the full description of these objects is used as a refinement step to produce the final answer to the query. The performance of these alternative data architectures with respect to spatial query processing and optimization is an interesting research problem. One drawback of this approach, though, is that every time a tuple participates in a join, the full description of the spatial object needs to be duplicated as well (in many cases, the size of this spatial description of an object is large; for example, this is the case with polygon objects).

4.6 Protoypes based on Object-Oriented Systems

The object-oriented model can also be used as a basis of a spatial database. In the object-oriented approach, information is highly structured by the introduction of classes and inheritance concepts. Data encapsulation and overloading facilitate data manipulation and make the physical and logical representations of the data independent of each other. In the case of a spatial database, concepts such as classes, inheritance, encapsulation, type and method extensions are handy. The different spatial data types can be implemented as classes with inheritance used to define subclasses (e.g., the class for a polygon with holes inherits from the more general polygon class). Complex objects can be defined from simpler ones (e.g., a road is composed of several line segments).

Scholl and Voisard [Scholl and Voisard 1992] describe a GIS built on top of the O2 object-oriented DBMS [Deux, O. et. al 1990]. However, their implementation makes little use of object-oriented techniques. Instead, they implement an extended relational system on top of O2 where by *extended* we mean supporting relations with abstract data types to encapsulate geometric data types. Spatial data is modeled at two levels: map and geometric. A spatial database consists of a set of maps, where a map is a relation that has at least one spatial attribute. Map operations are implemented as methods on map objects. The geometric level corresponds to the spatial attributes which are represented by geometric abstract data types (e.g., points, lines, regions, etc.).

5 Query Processing and Optimization

Query optimization for spatial databases is a relatively undeveloped field. It is heavily dependent on the application. Frequently, there is no special treatment for spatial data in the sense that the optimization strategies for the underlying DBMS are used with few special provisions being made for the difference in the data. A cost model and framework for comparing different optimization strategies is needed.

One issue is how to take into account the effect of the different representations and spatial access methods. For example, for a given set of spatial queries, representations based on computing a minimum bounding rectangle can be compared with representations based on storing an exact description, on storing increasing levels of details, and transformations into points in a higher dimensional space.

Another equally important issue is the nature of the datasets. For example, clustering of the data may alter the selectivity of a window operation (i.e., what is the expected percentage of the tuples that will be selected for output). Such clustering can be detected by distinguishing between a rural and an urban area. For point and line data, data density is important. For region data, skewness, roundness, connectedness, and topology (e.g., number of holes) come into play. As an example of the importance of skewness, it is useful to distinguish between querying a map of rivers and a map of lakes.

Of course, a tighter grip on the issue of estimating the cost of spatial operations is also needed. This depends on two factors. The first consists of the execution time of the underlying algorithm and I/O. The second is the time needed to produce the output. The latter is facilitated by a knowledge of selectivity factors. For example, consider the following two different spatial selection operations:

1. `nearest_to(`p`)` which finds the nearest object to point p.

2. `object_at(`p`)` which finds the object stored at point p.

Assume that spatial indexes exist for each operation. In this case, the selectivities are the same for both operations – that is, each returns just one object and thus takes $O(1)$ time to produce the output. However, the execution times of the algorithms can be quite different.

In particular, depending on the underlying representation of the spatial data, `nearest_to` may require $O(n)$ I/O operations while `object_at` may require just $O(\log n)$ I/O operations.

A number of recently suggested spatial database architectures address the issue of spatial query optimization (e.g., SAND [Aref and Samet 1991b], Gral [Güting 1989], GEOQL [Ooi 1988], and Geo-Kernel [Wolf 1989]). However, they differ in the capabilities and degrees of freedom that they provide to the spatial query optimizer as a result of the manner in which they integrate spatial data with nonspatial data. In particular, the underlying architecture may limit some feasible strategies for spatial query processing.

GEOQL's spatial query optimizer [Ooi 1988] extends the well-known query decomposition technique [Wong and Youssefi 1976] to handle spatial queries as well. However, GEOQL is biased towards the relational side. In particular, every relation is supported by at most one spatial attribute that is implicit and is always associated with the relation. Each query is decomposed into disjoint subqueries that consist entirely of either spatial or nonspatial conditions. The nonspatial subqueries are executed by an SQL backend, while spatial queries are executed by a spatial processor. Additional SQL subqueries are introduced to merge multiple partial results (i.e., temporary relations). Spatial operations cannot be composed directly – that is, each operation returns a relation and no further optimization is possible. GEOQL's optimizer only estimates the cost of nonspatial operations and does not take the I/O cost of spatial operations into account (i.e., only their selectivities). Different query execution plans are attempted. Heuristics are used to prune the number of plans whose costs are to be estimated (e.g., the cost of subqueries in an AND-clause that have no overlap is not affected by their order of execution).

In Geo-Kernel [Wolf 1989] spatial information is stored in textual form as an attribute value in a relation. For example, a polygon relation can be expressed as an attribute. This is potentially quite costly from a storage point of view (i.e., spatial data is usually voluminous) as a join may cause the same spatial data to be stored with different records. Nevertheless, during query evaluation appropriate spatial data structures are used to operate on spatial data. Thus in Geo-Kernel there is a need for conversion procedures to toggle between these data structures and the textual or byte-string form for each spatial data type. Notice that in order to perform the operation *intersects* or *closest*, for example, the entire set of spatial objects in the relevant relations have to be down-loaded into the spatial data structures. This is an expensive task and its cost has to be included when considering different query evaluation plans. As a result, the query optimizer prefers to perform relational selections before spatial selections in order to lower the cost of spatial data conversion (i.e., downloading) by reducing the number of qualifying tuples.

Gral [Becker and Güting 1989; Güting 1989] uses an algebraic query language at both the query description and execution levels. It uses a rule-based optimizer to normalize and optimize at the descriptive algebra level. Some examples include exchanging the order of operations using a predefined partial order of operations (e.g., selects before joins), and finding a good order for performing joins using selectivity estimates. The optimizer also translates the query to an executable algebraic form and optimizes at that level (e.g., by combining selection operations). It only takes the selectivity of the operation into account and thus ignores the actual cost of performing the operation. For example, it prefers performing a 'closest' operation to a 'select' operation since 'closest' reduces the number of tuples although its I/O execution cost may be considerably higher.

Systems such as GEO++ [Vijlbrief and van Oosterom 1992] are implemented on top of POSTGRES [Stonebraker and Rowe 1986]. POSTGRES allows users to add application-dependent operators. The operators are characterized so that the query optimizer can decide which optimization techniques should be applied. This is done by using cost estimates. Some of the characterizations include precedence, associativity, whether or not the operator is hash joinable, commutator and negator operators, and select and join selectivities. The operator and index characteristics are stored in tables. Based on these characteristics, the optimizer maps the operators (if possible) into existing database methods (e.g., hardwired join algorithms).

SAND's query processing and optimization strategies [Aref and Samet 1991b] include the ability to reorder operations as well as merging. Merging is useful when a query contains two conditions (can be spatial or nonspatial) and it is desired to perform them independently and merge the results. Merging can be either homogeneous (i.e., the conditions are both spatial or both nonspatial) or non-homogeneous (i.e., one condition is spatial while the other is nonspatial). The former is implemented by intersecting the sets resulting from the operations. For the latter, the execution can be either spatial-driven or relational-driven. The one that is chosen depends, in part, on selectivity factors. For example, in the case of a spatial-driven merge, the spatial data structure is traversed and only the spatial objects (and their tuples) that satisfy the spatial condition and whose tuples satisfy the relational condition are retained.

Other optimizations that SAND is capable of performing include combining successive spatial operations or successive relational operations, as well as making use of pipelining and composition. At times, an operation may not be very selective in which case it is preferable to delete the unselected items from the map rather than creating a new map containing the multitude of selected items. Early projection is also useful when it is known that some of the attributes are no longer needed. Frequently, the optimization is application-dependent. For example, when part of the condition involves the computation of the nearest object, it may be preferable to rank order all the objects at once rather than computing the nearest object anew each time.

6 Concluding Remarks

An overview has been presented of some of the issues that arise in building spatial databases. As we saw, they consist, in part, of finding better ways to interact with the database, integrating the spatial and nonspatial data, and optimizing the processing of the queries. It should be clear that such databases are still in their infancy in the sense that much remains to be done before the ideas explored in research prototypes will find their way into commercial systems.

References

Abel, D. J. 1989. SIRO-DBMS: A database tool-kit for geographical information systems. *International Journal of Geographical Information Systems, 3,* 2 (April–June), 103–116.

Al-Taha, K. K., Snodgrass, R. T., and Soo., M. D. 1993. Bibliography on spatiotemporal databases. *SIGMOD Record, 22,* 1 (Mar.), 59–67.

Aref, W. G. and Samet, H. 1991a. Extending a dbms with spatial operations. In Günther, O. and Schek, H.-J., Eds., *Advances in Spatial Databases*, pp. 299–318. Lecture Notes in Computer Science 525. Springer-Verlag, Berlin.

Aref, W. G. and Samet, H. 1991b. Optimization strategies for spatial query processing. In Lohman, G., Ed., *Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB)*, (Barcelona, Spain, Sept.), pp. 81–90.

Becker, L. and Güting, R. H. 1989. Rule-based optimization and query processing in an extensible geometric database system. Technical Report 312, Dortmund University, Dortmund, West Germany, (Aug.).

Carey, M., DeWitt, D., and Vandenberg, S. 1988. A data model and query language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, (Chicago, IL, June), pp. 413–423.

CODASYL. 1971. CODASYL Data Base Task Group. *ACM*, (Apr.).

Dayal, U. and Smith, J. M. 1986. A knowledge-oriented database management system. In Brodie, M. L. and Mylopoulos, J., Eds., *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, pp. 227–257. Springer-Verlag, New York.

Deux, O. et. al. 1990. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, *2*, 1 (Mar.), 91–108.

Egenhofer, M. J. 1992. Why not SQL! *International Journal of Geographical Information Systems*, *6*, 2 (March–April), 71–85.

Gadia, S. K. 1993. Parametric databases: seamless integration of spatial, temporal, belief, and ordinary data. *SIGMOD Record*, *22*, 1 (Mar.), 15–20.

Güting, R. H. 1989. Gral: An extensible relational system for geometric applications. In Apers, P. M. G. and Wiederhold, G., Eds., *Proceedings of the Fifteenth International Conference on Very Large Databases (VLDB)*, (Amsterdam, Aug.), pp. 33–44.

Haas, L. M., Chang, W., Lohman, G. M., McPherson, J., Wilms, P. F., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M. J., and Shekita, E. 1990. Starburst mid flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, *2*, 1 (Mar.), 143–160.

Kim, W., Banerjee, J., H.Chou, Garza, J., and Woelk, D. 1987. Composite object support in an object-oriented database system. In *Proceedings of the 2nd ACM OOPSLA Conference*, (Orlando, FL, Oct.), pp. 118–125.

Lorie, R. and Meier, A. 1984. Using a relational DBMS for geographical databases. *Geo-Processing*, *2*, 243–257.

McGee, W. C. 1977. The IMS/VS system. *IBM Systems Journal*, *16*, 2 (June), 84–168.

Ooi, B. C. 1988. *Efficient Query Processing for Geographic Information Systems*. PhD thesis, Monash University, Victoria, Australia. (Also Lecture Notes in Computer Science 471, Springer-Verlag, Berlin, 1990).

Orenstein, J. A. and Manola, F. A. 1986. Spatial data modeling and query processing in PROBE. Technical Report CCA-86-05, Computer Corporation of America, Cambridge, MA, (Oct.).

Orenstein, J. A. and Manola, F. A. 1988. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering, 14*, 5 (May), 611–629.

Peuquet, D. J. and Marble, D. F. 1990. ARC/INFO: An example of a contemporary geographic information system. In Peuquet, D. J. and Marble, D. F., Eds., *Introductory Readings In Geographic Information Systems*, pp. 90–99. Taylor & Francis, London.

Roussopoulos, N., Faloutsos, C., and Sellis, T. 1988. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering, 14*, 5 (May), 639–650.

Schek, H. and Scholl, M. 1989. The two roles of nested relations in the DASDBS project. In Abiteboul, S., Fischer, P. C., and Schek, H.-J., Eds., *Nested Relations and Complex Objects in Databases*, pp. 50–68. Springer-Verlag, Berlin. (Also Lecture Notes in Computer Science 361).

Schek, H. and Waterfeld, W. 1986. A database kernel system for geoscientific applications. In *Proceedings of the 2nd International Symposium on Spatial Data Handling*, (Seattle, WA, July), pp. 273–288.

Schilcher, M. 1985. Interactive computer graphic data processing in cartography. *Computers & Graphics, 9*, 1, 57–66.

Scholl, M. and Voisard, A. 1992. Object-oriented database systems for geographic applications: an example with O2. In Gambosi, G., Scholl, M., and Six, H.-W., Eds., *Geographic Database Management Systems*, pp. 103–137. Springer-Verlag, Berlin.

Shaffer, C. A., Samet, H., and Nelson, R. C. 1990. QUILT: A geographic information system based on quadtrees. *International Journal of Geographical Information Systems, 4*, 2 (April–June), 103–131.

Stonebraker, M. 1986. Inclusion of new types in relational data base systems. In *Proceedings of the 2nd International Conference on Data Engineering*, (Los Angeles, CA, Feb.), pp. 262–269.

Stonebraker, M. and Rowe, L. 1986. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, (Washington, DC, May), pp. 340–355.

Tomlin, C. D. 1990. *Geographic Information Systems and Cartographic Modeling.* Prentice Hall, Englewood Cliffs, N.J.

Tufte, E. R. 1983. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT.

Tufte, E. R. 1990. *Envisioning Information.* Graphics Press, Cheshire, CT.

Vijlbrief, T. and van Oosterom, P. 1992. The GEO^{++} system: an extensible GIS. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 1, (Charleston, SC, Aug.), pp. 40–50.

Waugh, T. C. and Healey, R. G. 1987. The GEOVIEW design: A relational data base approach to geographical data handling. *International Journal of Geographical Information Systems, 1*, 2 (April–June), 101–118.

Wolf, A. 1989. The DASDBS GEO-Kernel: Concepts, experiences, and the second step. In Buchmann, A., Günther, O., Smith, T. R., and Wang, Y.-F., Eds., *Design and Implementation of Large Spatial Databases, Proceedings of the First Symposium SSD'89*, pp. 67–88. Santa Barbara, CA. (Also Lecture Notes in Computer Science 409, Springer-Verlag, Berlin, 1990).

WONG, E. AND YOUSSEFI, K. 1976. Decomposition - A strategy for query processing. *ACM Transactions on Database Systems*, *1*, 3 (Sept.), 223–241.

ZANIOLO, C. 1983. The database language GEM. In *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*, (San Jose, CA, May), pp. 207–218.