In *Proceedings of the 4th International Symposium on Spatial Data Handling*, vol. 2, pages 589-598, Zurich, Switzerland, July 1990.

An Approach to Information Management in Geographical Applications [1]

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

Abstract

The design of an integrated system is described for combining spatial and nonspatial attribute data to allow fast query handling. By fast it is meant that search queries can be executed in logarithmic time. Nonspatial attribute data is managed by standard database techniques. Efficient and robust spatial data structures are used to store spatial information. In addition to the standard access methods usually provided for accessing attribute data (i.e. through the data base management system), data can also be accessed through the spatial data structures in a homogeneous way that is transparent to the user. The design makes use of an SQL-like interface. The SQL query language is extended to provide efficient access methods to spatial structures, and to match functional and efficiency requirements desired from a combination of spatial and nonspatial information. The simplicity of SQL helps to clarify the design approach. However, other interfaces can be used with minor modifications.

1 Introduction

In geographical applications, it is often desirable to attach attribute information such as elevation heights, city names, etc, to objects appearing in maps. On the other hand, in many standard database applications, it is

[1] The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

useful to add spatial attributes to describe different objects in the database such as the extent of a given river, or the boundary of a given county, etc. Many queries can be answered more efficiently when spatial and nonspatial information are combined.

As a typical situation in a spatial database, consider a set of objects (e.g., points and lines) in 2D space, and a set of features which partition the space into nonoverlapping or overlapping regions. Our aim is to design a system which integrates both spatial (e.g., points, lines, regions, etc.) and nonspatial (e.g., features, names, addresses, etc.) attributes in an efficient way. Some typical queries include the following:

Point location: Given a point in a plane, find the feature (e.g., the region, or more generally, the feature name) in the plane containing this point.

Feature location: Given a feature name, find the locations of the regions associated with this feature.

Windowing: Given a region (e.g., a rectangular window), find all points inside this region. Alternatively, find the feature names overlapping a given window (e.g., a circle).

Spatial operations: Some examples are: *arithmetic operations* (e.g., find the area/preimeter/centroid of a given region, specified by any of its attributes or by a point inside it); *set operations* (e.g., find the intersection/union of two regions, or find the negation of a region); *region expansion* also called *buffer zoning* (e.g., find the region within a certain radius from the boundary of a given region); *proximity operations* (e.g., find the nearest line to a given point, or the region to the north of a given region). For a more comprehensive list see [10].

Many prototype systems have been proposed to store spatial data along with the nonspatial data describing it in a relational database management system [1, 5, 11]. They adopt an encoding of spatial features that results in generating numeric key values. These values are then stored in record attributes in the same way as nonspatial attributes. Orenstein [5] and Abel et. al. [1] use *z-ordering* [4] to serve as the encoding method. By using this appoach, a spatial operation is viewed by many researchers as a two-step process: first, retrieve the spatial data and then operate on it. Since spatial data is stored with nonspatial data in database tuples, much time is wasted on the retrieval step. In particular both spatial and nonspatial data are retrieved although only the spatial portion in needed. This has led many researchers to try to enhance the performance of the retrieval step [1, 3, 5, 11]. In particular, Faloutsos and Roseman [3] performed experiments in order to find the coding with the least retrieval time when a range query is imposed. As an alternative, if spatial data is structurally separated from nonspatial data, while maintaining appropriate links between the two, then the spatial data retrieval bandwidth can be much higher. Thus, we can perform the spatial operations directly on these structures. Also, it gives us the freedom to choose a more suitable spatial structure besides the imposed relational structure. In other words, the goal is to enhance the performance of the entire spatial operation in the database environment, rather than just the retrieval aspect of it.

The notion of separating the structure of spatial data from that of nonspatial data to achieve better performance is the motivation behind the research of a number of groups in extensible databases (e.g., [9]). However, most these systems are still in the early stages of development and are not ready for validation in terms of efficiently supporting spatial data.

Our research places a greater emphasis on the operational side of the spatial processor while allowing high interleaving between the database management system and the spatial processor in order to optimize the query answering process. We will combine both processors (the database management system and the spatial processor) in a way that permits efficient access to both spatial and nonspatial information whenever possible. The major goal for our system is threefold. First, each data entity should be represented by the most efficient form that suits its operational purpose. For example, spatial attributes should be stored in spatial data structures and not flattened in database records. Second, the user should not be concerned with implementation details of such spatial data structures. In other words, there should not be a distinction between spatial and nonspatial attributes in the user's eyes. Third, it is desired to answer queries efficiently, and to minimize any deterioration in performance of both the spatial processor and the database management system when they operate in the same environment.

The rest of the paper is organized as follows. First, we show how the user interacts with (i.e., views) the system by informally providing the extended data definition and manipulation languages. We demonstrate the use of both languages through examples. Next, we present the system design and show how it achieves the efficiency goals. This is followed by a comparison of our design with alternative approaches. We also discuss related research that addresses a broader class of problems. Finally, we also touch on directions for future work and research.

2 User Interface

In this section, we demonstrate the data definition and manipulation facility of the system from a user's perspective. We show how spatial attributes and spatial operations that are essential in geographic information systems can be integrated into the relational model. This makes the data model easier to use for storing geographic data, and more efficient for handling spatial and nonspatial queries.

2.1 Data Definition

The user views the system as an extended relational database management system. In addition to the usual attributes in relations, a user can define spatial attributes in a homogeneous way.

The system supports four types of spatial entities: points, line segments, polygons, and regions. The data types for them are `POINT`, `LINE_SEGMENT`, `POLYGON`, and `REGION`, respectively. The system can be extended easily to support other spatial entities in the same way (provided that the spatial processor has the capabilities to handle it).

Example 1:
Consider the following SQL schema definitions.

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

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

As an illustration, in the first definition, each tuple in the `roads` relation represents a line segment. It also defines the `road_coords` attribute as a spatial attribute of type `LINE_SEGMENT`. All other three attributes in the `roads` relation are ordinary nonspatial attributes.

2.2 Data Manipulation

In addition to the standard SQL commands, some spatial functions must be augmented in order to handle spatial map processing. In this section, we demonstrate the user's view of the extended data manipulation language by giving some examples. We will make use of the data definitions provided in Example 1.

2.2.1 Standard database operations (e.g. select, project, join, etc.)

Example 2:
Find all roads (including their coordinates) with trafficability factor greater than 9.

```
  Select all
  from roads
  where road_trafficability > 9;
```

This query results in the generation of a new relation as well as the creation of a line map. This map reflects the records selected in the operation (i.e. it will contain the line segments with trafficability factor > 9).

2.2.2 Qualification by set operations

Given one or more spatial attributes, set operations can be applied to them. This yields an output map as well as a new relation. The map contains the spatial entities resulting from the set operation, and the relation contains the nonspatial attributes selected from the tuples that describe the new spatial entities.

Example 3:
Find the names of the roads passing through the 'Univ. of Maryland' region.

```
Select road_name
from roads regions
where region_name = "Univ. of Maryland"
and intersect(region_location, road_coords)
```

The `region_location(s)` of the tuple(s) whose `region_name` is 'Univ. of Maryland' will be intersected with the `road_coords` attribute of the road map.

2.2.3 Qualification by region expansion (buffer zoning)

Example 4:
What are the important regions within 5 miles of the 'Univ. of Maryland' (i.e., surrounding it)?

```
Select r1.region_name r1.region_utilization r1.region_location
from r1 regions, r2 regions
where r1.region_importance > 8
and r2.region_name = "Univ. of Maryland"
and intersect(within(r2.region_location, 5), r1.region_location)
```

This query finds the region names, utilization, and locations of 'important' regions within 5 miles of the 'Univ. of Maryland' region. Since the `region_location` spatial attribute is selected, the result is a map containing the selected regions.

In addition to the above qualifications, many other spatial conditional expressions can be added to the query language as long as they are supported by the spatial processor. Some of these conditional expressions are given below. Notice the use of `region_attr` and `line_attr` to specify region and line attributes, respectively. On the other hand, `spatial_attr` is more general and includes `region_attr` and `line_attr`.

```
area(region_attr) > val
objectat(spatial_attr, location)
nearest_to(spatial_attr) = a
length(line_attr) > val
in_window(spatial_attr, x1, y1, x2, y2)
```

3 System components

To allow efficient information retrieval from both parts of the system (spatial and nonspatial), we use three major components: a relational database management system (RDBMS), a spatial map processor (SP), and a high-level query interpreter (HQI). Although this approach is not novel, and has been suggested by a number of researchers (e.g., [7]), it is how the two parts are combined to meet the required efficiency goals that makes the selection interesting.

We store all attribute information (nonspatial data) about map entities in the relational database management system. Map storage and management (spatial data) is handled through the spatial processor. The HQI maps a user query into an application plan which contains subqueries to both the RDBMS and the SP.

In the following sections, we discuss each system component in greater detail, and we show how to link them.

3.1 The Relational DBMS

Relational databases are becoming the major mechanism for storing large collections of data. In our design, we chose the relational model instead of other models since it is highly flexible, has improving performance, and other models (e.g., network and hierarchical) can be reduced to the relational model [2]. Consider the SQL schema definition for table `regions` given in Example 1. This definition is translated by the HQI into two coupled definitions. In this section we discuss the RDBMS aspect of the definition. The SP definition is deferred to the next section. In the RDBMS definition of the `regions` schema, the `region_location` spatial attribute is represented by an attribute of type `NUMBER` that contains an index to a candidate point internal to the map region corresponding to the tuple. This is based on the assumption that regions are represented by their internal area. However, it is easy to extend this approach to other region representations. This pointer establishes a link from the relation to the map. It guarantees that given a tuple_id (or any other attribute) that helps select a subset of tuples, we can access the corresponding spatial object (e.g., the corresponding region in the coupled map). We call this pointer a **forward link**, as opposed to a **backward link** that is directed from the map to the relation. Forward and backward links facilitate many queries as shown below.

Example 5:
Find the names of the roads passing through the 'Univ. of Maryland' region.

```
Select road_name
from roads regions
where region_name = "Univ. of Maryland"
and intersect(region_location, road_coords)
```

This query is processed as follows:

1. The RDBMS searches the `regions` relation for the tuple whose region name is 'Univ. of Maryland'. The pointer(s) to the spatial region location (actually the forward link(s)) of 'Univ. of Maryland' stored in the `region_location` attribute are projected from the selected tuples and passed to the SP.

2. The SP performs an intersection operation between the `roads` map (representing the `road_coords` spatial attribute), and the subset of the `regions` map corresponding to the 'Univ. of Maryland' region (built by the SP given the pointers passed by the RDBMS as described above). The result of the intersection is a subset of the `roads` map.

3. Using the backward links from the map to the relation, the pointer(s) to the tuple(s) corresponding to the selected line(s) are passed back to the RDBMS.

4. Given the tuples in the `roads` relation selected by the intersection operation through the SP, the RDBMS performs a projection to get the road names of these tuples (i.e., the road names of the line segments that intersect the 'Univ. of Maryland' region).

The above example utilizes backward and forward links. The implementation of backward links from the spatial objects to the corresponding tuples is described in the next section. It is clear that each of the above operations is performed in the system that performs it best. The search for `region_name` = 'Univ. of Maryland' is performed in the RDBMS while the intersection is performed in the SP.

3.2 The Spatial Processor (SP)

The SP is responsible for storing and maintaining the spatial objects as well as performing spatial operations such as set operations, windowing, etc. It is also responsible for maintaining the backward links from the spatial objects to the corresponding tuples in the RDBMS.

In addition to the data structure for storing spatial objects in a map, each spatial object representation contains an additional field called the `class` field. The class field serves as an index to a tuple in the corresponding relation coupled with the map. This establishes the **backward link** from the SP to the RDBMS. Therefore, given a pointer to a spatial object, we can access its corresponding tuple containing all of the object's nonspatial information. If the spatial object is represented by more than one entry in the spatial data structure, then each such entry has to point to the same tuple by storing the tuple's index in its class field. Note that if, for example, a schema definition has two spatial attributes, then two different maps are created, one for each spatial attribute. Each map keeps its own set of backward links into the relation tuples.

Now let us look at how the SP responds to the `regions` relation schema definition of the previous section. As mentioned earlier, regions are stored in maps that are represented by spatial data structures (e.g., counties, crop regions, etc.). The SP creates a region map to be coupled with the `region_location` (defined as a `REGION`-type attribute), and stores the region objects pointed at by the `region_location` attribute values. By coupled, it is meant that forward and backward links are established between the spatial attribute in the relation and the spatial objects in the spatial data structure.

Recall that the `REGION`-type attribute stores a pointer to a candidate point inside the spatial region. Therefore, for the design to be complete, given a point inside a region, the SP must retrieve or extract the whole connected region from the spatial map efficiently (which is achievable by using a connected component labeling algorithm [6, 8]). Being able to retrieve the whole region quickly, will not degrade the queries directed from the RDBMS to the SP. The result is superior to storing the whole region in the RDBMS since there is no overhead in the retrieval process.

Note that storing the regions by using spatial data structures is the key to performing spatial operations efficiently as it makes use of the underlying spatial index mechanism that is inherent to the spatial data structure. Some of these spatial operations are region expansion, windowing, computing the perimeter of a region, etc.

On the other hand, if we were to store the whole region in the database [5] (e.g., by storing each pixel inside the region in a separate record, or by storing each quadtree block in a separate record if the region is represented by a quadtree), then we would lose the spatial processing power. In other words, there would be a tremendous degradation in performance because of a need to maintain and operate spatially on a large number of database records representing the whole region (along with the nonspatial attributes describing it), while having to do sequential search to access all of them when performing some spatial operations.

3.3 The High-level Query Interpreter (HQI)

The HQI is responsible for translating extended data definition and manipulation statements issued by the user into an **application plan**. An application plan consists of lower level subcommands to either the RDBMS or the SP. As an example, consider the following window query.

```
Select all
from roads
where in_window(road_coords, x1, y1, x2, y2)
and road_name ≠ "Route 1"
```

This query selects all roads (except 'Route 1') overlapping with the window $[x_1 : x_2] \times [y_1 : y_2]$. Notice that all attributes have been selected for output (via the `all` command) which means that the spatial attribute `road_coords` is among the selected ones. Therefore, an output map is also generated containing all the selected line segments.

In writing application plans, we identify the commands by the name of the processor which processes them (e.g., SP, RDBMS, and HQI*). The HQI* commands are for bookkeeping purposes. For the above query, the HQI produces the following application plan:

HQI*: get the map name associated with the attribute `road_coords` in the `roads` relation.

SP: create a temporary map (for output of the selected line segments).

RDBMS: create a temporary relation (for output of the selected tuples).

SP: apply in_window_iterator to each line (each time it is called, the in_window_iterator generates a new line that overlaps with the window).

- SP: insert the generated line into the output map.
- SP: get `road_id`; (stored in the class field in the current line),
- RDBMS: extract this line from the `roads` relation given its `road_id`, and add it into the output relation.

It should be clear that the above plan is not optimized. By using a special optimizer, a more efficient plan can be generated. Pipelining back and forth between the SP and the RDBMS (e.g., by using an iterator construct) is one technique to avoid the unnecessary traversals and buffering between the two processors.

The HQI design can be enhanced in a number of ways. Applying standard optimization techniques to spatial operations is an obvious direction. The subject of plan generation and optimization in the context of our design is a subject for future research.

4 Conclusions

In this paper, we have shown how spatial and non-spatial attributes can be combined in an efficient way. Our measure of efficiency was to allow operations (whether spatial or non spatial) to be performed in their most natural environment. Flexibility in the interaction between spatial and nonspatial attributes was achieved through the use of forward and backward links. We believe that these are the necessary and sufficient additions to both the SP and the RDBMS to achieve the efficiency goal for answering spatial and nonspatial queries in a combined environment. This architecture can be used to benchmark other proposed general purpose extensible systems. The objective is to see how these systems restrict the implementation of this architecture, since any restriction will adversely affect efficiency.

One-to-one onto and many-to-one links are supported between the SP and the RDBMS ('many' in the SP side, and 'one' in the RDBMS side). For point data one-to-one onto links are used. For line data, depending on the data structure used to represent it, one-to-one or many-to-one links may be used. For region data, a design compromise took place since a region can be represented by more than one spatial entity (as in the case of representing regions by their internal area, or by their bounding line segments in a polygon representation).

Directions for future work include studying the automatic generation of application plans. Optimizing these application plans and developing measures for them is another direction.

References

[1] D. J. Abel. Relational data management facilities for spatial information systems. In *Proceedings Third International Symposium on Spatial Data Handling*, pages 9–18, Sydney, Australia, August 1988.

[2] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, fourth edition, 1986.

[3] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. Technical Report Computer Science TR-2242, University of Maryland, College Park, MD 20742, May 1989.

[4] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[5] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM SIGMOD Conference*, pages 326–336, Washington, DC, May 1986.

[6] A. Rosenfeld and A. C. Kak. *Digital Picture Processing.* Academic Press, New York, second edition, 1982.

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

[8] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28(3):487–501, July 1981.

[9] H. Schek and W. Waterfeld. A database kernel system for geoscientific applications. In *Proceedings, Second Intl. Symp. on Spatial Data Handling*, pages 273–288, Seattle, WA, July 1986.

[10] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. Technical Report Computer Science TR-1885.1, University of Maryland, College Park, MD 20742, July 1987.

[11] T. C. Waugh and R. G. Healey. The GEOVIEW design: A relational data base approach to geographical data handling. *Intl. J. Geographical Information Systems*, 1(2):101–118, 1987.