A Query Language for a Spatial Information System

Linda G. Shapiro
Swapan N. Engineer
Virginia Polytechnic Institute and State University
USA
Commission IV

I. Introduction

Our spatial information system (Shapiro and Haralick, 1980, Vaidya, et.al., 1982) is an entity-oriented database system, designed for flexible access to spatial information for use in cartographic applications. The building block of the system is the spatial data structure which is formally defined as a set of relations whose components may be atoms (nondecomposeable values) or may themselves be spatial data structures. Spatial data structures represent such entities as countries, states, cities, rivers, lakes, railroads, and so on. Relations represent spatial relationships such as adjacency, containment, or distance between two entities.

Because the spatial data structures are both hierarchical and relational in nature, standard query languages are not applicable. Instead, we have designed a powerful new query language SQL, that is similar to SEQUEL (Astrahan and Chamberlin, 1975), but designed to handle queries involving entities and relationships at any level of the hierarchic structure. The standard relational operators (projection, selection, join) and the set operations (union, intersection, difference) have been extended both syntactically and semantically for meaningful use with spatial data structures. An "intelligent" parser has been implemented that can translate user queries to a sequence of primitive actions leading to a response. The parser uses syntactic knowledge of the structure of each entity type in the system in the translation process. Thus the user can formulate very simple queries that work with fairly complex structures.

In this paper, we will briefly describe the spatial information system, define the query language SQL, and give examples of its use.

II. The Spatial Information System

In our spatial information system, each geographical entity is represented by a spatial data structure which is defined in the following paragraphs. The terminology is taken from Shapiro and Haralick (1980).

An atom is a unit of data that cannot be broken down any further. Integers and character strings are common examples of atoms. An attribute-value table is a set of pairs $A/V = \{(a,v) \mid a \text{ is an attribute and } v \text{ is the value associated with } a\}$. The value v can be an atom or a complex structure. For example, in an attribute-value table for a county we may have as the value of an attribute called STATE, the name of the state in which the county belongs, or the structure for the state itself.

We now formally define a spatial data structure as a set D = {R1,...,RK} of relations. Each relation Rk has a dimension Nk and a sequence of domain sets S(1,k),...,S(Nk,k). That is, for each k = 1,...,K, Rk belongs to (S(1,k)x ... x S(N_k,k). The elements of the domain sets may be atoms or spatial data structures.

Since the spatial data structure is defined in terms of relations whose components may themselves be spatial data structures, we call it a recursive data structure. This recursive nature of a spatial data structure indicates that often it will be possible to describe operations on the structure by simple recursive algorithms.

A spatial data structure represents a spatial entity (however, we can represent any other entity with the same degree of flexibility). The entity might be as simple as a point or as complex as a whole map. An entity has global properties, component parts, and related spatial entities. Each spatial data structure has one distinguished relation containing the global properties of the entity that the structure represents. The distinguished relation is an attribute value table. When a spatial entity is made up of parts, we may need to know how the parts are organized. Or, we may wish to store a list of other spatial entities that are in a particular relation to the one we are describing. Such a list is just a unary relation, and the interrelationships among the parts are n-ary relations.

From the definition of the spatial data structure it is clear that it can be used to create an arbitrary hierarchy and that spatial data structures and relations alternate in any hierarchical path. This has a very important implication; with a database organized in this manner, we can do everything that we could with a hierarchical database like IBM's IMS and a relational database like IBM's System R and some other things that may be very difficult or unnatural to do with either of them. Also we can combine the efficiency of the hierarchical model with the ease of using a relational model. Efficient utilization of space is sometimes easier to achieve by using hierarchies than relations. To represent one to many relationships, if we use the relational model, we have to duplicate some data unnecessarily. A combination of the hierarchical and the relational model will, therefore, help us optimize space. A comparison of data manipulation languages for a purely relational database and purely hierarchical databases is like comparing LISP with an ASSEMBLER (Ullmann, 1980), since the hierarchical model has very primitive data manipulation languages and requires the user to navigate through the system. The relational model, on the other hand, normally has very high level and concise query languages.

In the system we have developed, we have tried to retain the best features of the hierarchical model and the relational model.

III. Prototypes

Formally a prototype for a relation defines the domains for each of its attributes. A prototype for a spatial data structure defines the types (prototypes) of its member relations. In the literature the word prototype is interchangeably used with a conceptual model or a schema or a view.

One of the primary goals of a database system is to systematize access to data elements and to achieve data independance (Date 1981). When applications are data dependent, it is very difficult to change the storage structure or the access methods without any side effects. A prototype (or a number of prototypes) defines the logical structure of a database system without any consideration for storage/access methods (Wiederhold, 1977).

Let us define the logical structure of a database for a country. First we will give a high level description of prototypes that is computer/programing language independent. We have the entities COUNTRY, STATE, COUNTY, CITY etc. Each entity has some properties and relations that identify it. The spatial data structure (SDS) described above is ideal to represent an entity logically.

Figures 1 to 4 show the prototypes for different SDSs and relations in the system. In Figure 1 we have shown the prototype for a country. We treat a country as an entity with four relations; av_country, state_adjacency, exports, and imports. The relation av_country is the attribute-value table for the country. The attributes for the country are the name of the country, the area of the country, the population of the country, the number of states in the country, the name of the continent to which the country belongs, the national language of the country, and the name of the capital of the country. The relation state_adjacency is a binary relation in which each component is a spatial data structure representing a state. Exports and imports are unary relations and store the names of the commodities the country exports, and imports, respectively. Figures 2 to 4 have similar explanations.

Operations on prototypes include creation, deletion, display, equality test, concatenation, copying, and saving. Prototypes are created when requested by the user or when required for a newly created SDS or relation. Prototypes for temporary or no-longer-used SDS's and relations are deleted. When two prototypes are determined to be equivalent, one can be deleted. Concatenation of prototypes is required for the result of certain join operations and copying is a utility used with several creation operations such as union. Saving a prototype makes it permanent.

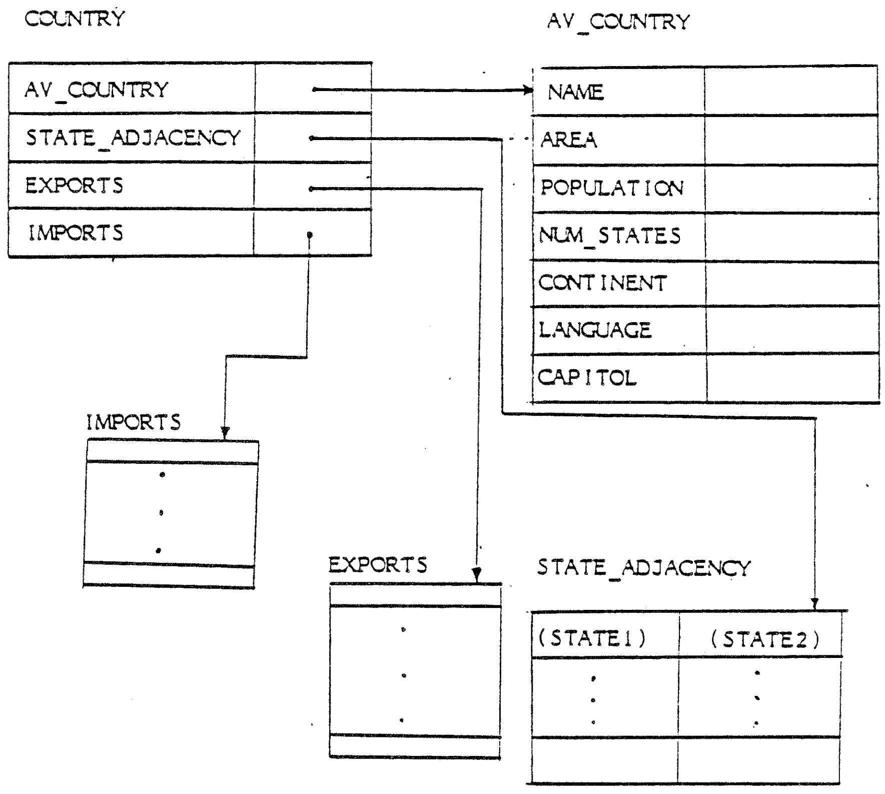

Figure 1: Prototype for a country

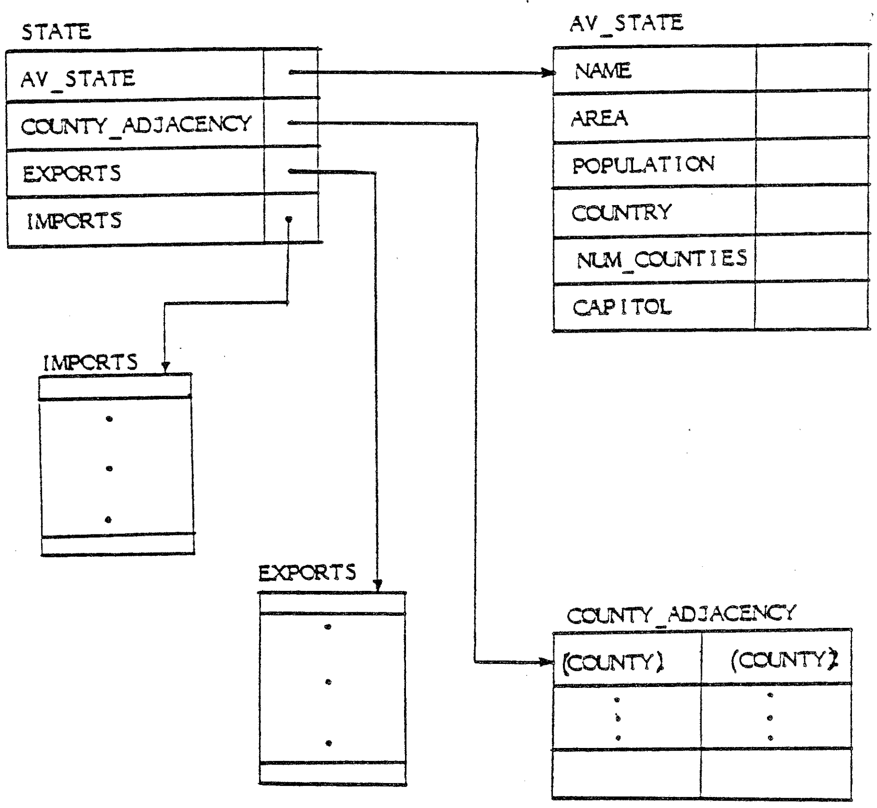

Figure 2: Prototype for a state

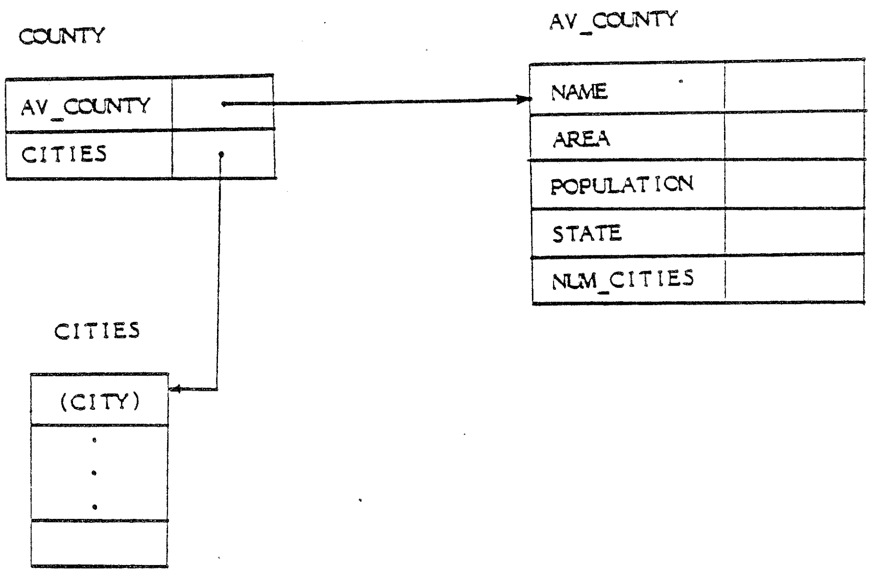

Figure 3: Prototype for a county

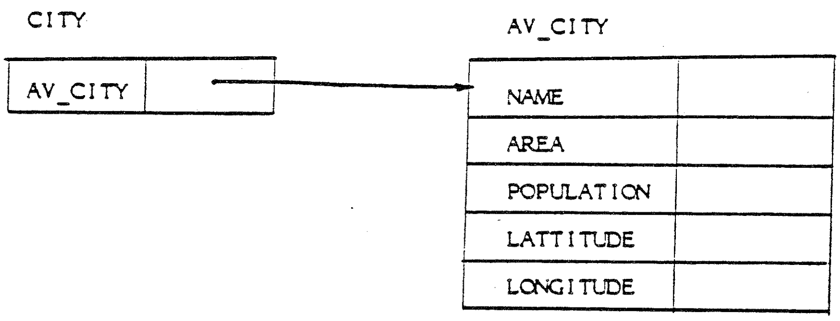

Figure 4: Prototype for a city

IV. The Spatial Query Language

The operations in the Spatial Query Language (SQL) consist of simple projection, selection, simple join, equi-join, general join, union, intersection, difference, and display. The operations are meaningful on relations and on entire spatial data structures. The exact syntax and semantics of the operations are complex and are given in Engineer (1983). Here we give some introductory examples with explanations.

Simple Projection

```
1) Select name (city), area (state,city)
   Into   name_area
   From   Virginia.
```

Assuming that Virginia is a name of a state, this query generates a new binary relation name area that contains the name of each city in Virginia together with its area.

We can specify query 1) in several different ways, all of which will cause the system to follow exactly the same steps and produce exactly the same output. One of the ways in which we can restate query 1) is:

```
1') Select name (city), area
    Into   name_area
    From   Virginia.
```

Action: The operand of the "From" (called ID) is either a name of an SDS or a relation and the attribute list after "Select" (called ALIST) is a list of attributes whose values are required. As the definition of the spatial data structure has recursion built into it, the semantics of select, project and join are no longer exactly the same as their conventional definitions in a relational database. In particular, it is no longer required that the attributes specified in the ALIST be in ID.

If ID is an SDS, member relations are searched for the attributes in ALIST. If a particular attribute is not found in any of the member relations prototypes of SDS's that are components of member relations are searched and so on. A combination of depth first search and breadth first search is employed to generate a path for each attribute in ALIST from ID. Similarly if ID is a relation, a path for each attribute in ALIST is generated by searching prototypes recursively.

<u>Selection</u>

```
2)  Select name (city), area, name (state)
    Into   dense_cities
    From   usa
    Where  (population(city) / area(city) >=
            population (state) / (15 * area(state)) or
           (population(city) > 10000.00).

3)  Select x, y, z
    From   a
    Where  not ((x >= y * x) and (z ¬< - (x / y))).
```

<u>Action</u>: The action taken is the same as above, except that predicates in the list following "Where" (called PLIST) are first tested and only when PLIST evaluates to true, does the corresponding tuple of attributes in ALIST become part of the output relation. A '*' can be specified as ALIST only if ID is a relation. If '*' is specified as ALIST, each entire tuple that satisfies PLIST becomes part of the output relation.

```
4)  Cross x, y
    Into  z.

5)  Join x y.

6)  Join x & y
    Into z.
```

<u>Action</u>: If ID1 and ID2 (the two names following "cross" or "join") are relations, the cross (cartesian) product of ID1 and ID2 is computed. The results are saved in the operand of "Into" if it is specified. ID1 and ID2 can be arbitrary relations with some/all/no attributes in common.

If ID1 and ID2 are SDSs, it is required that both have the same prototype. A new SDS is formed (if "Into" ID3 is specified) which contains exactly the same number of relations as ID1 and ID2. The kth relation of the new SDS is a cross product of the kth relation of ID1 and the kth relation of ID2. The names of member relations for the new SDS are dynamically generated.

<u>Equi-join</u>

```
7)  Join x y
    Over a,b,c,d.

8)  Join x & y
    Into z
    Over a,b,c,d.
```

If ID1 and ID2 are relations, it is required that they have all the attributes specified in the list following "Over" (called JLIST) in common. Each tuple of ID1 is tested against every tuple of ID2. Let t1 denote a tuple from ID1 and t2 that from ID2. For a pair (t1, t2), if all the attributes specified in JLIST have the same values in t1 and t2, a new tuple t3 is constructed consisting of

1. the values of the attributes in t1, but not in JLIST,
2. the values of the attributes in JLIST, and
3. the values of the attributes in t2, but not in JLIST.

If "Into" ID3 is specified, t3 is added to ID3. It is possible that this query may result in an empty relation or response.

If ID1 and ID2 are SDSs, it is required that they have the same prototype.

General-join

```
9)  Join x y
    Into z
    Where a(x) = b(y).

10) Join x & y
    Into z
    Where (a(x) >= 10 * b(y)) and
          (c(y) < d(x)/f(y)).
```

General-join is quite powerful compared to equi-join in that it allows us to specify arbitrary conditions under which two relations are to be joined. General_join, however is limited in that it is not defined for spatial data structures.

Set Operations

Each set operator is a binary operator, and the result of the operation has the same form as that of the operands. If ID1 and ID2 are relations, it is necessary that they share the same prototype.

Set operations involving spatial data structures do not require that SDSs participating in the operation share the same prototype. In fact even if we impose the restriction that both share the same prototype, we do not gain anything.

Display

The display command can be used to display any relation/SDS/prototype.

V. Retrieval

Each query is processed by the parser which builds a predicate tree -- an intermediate structure specifying all the predicates that must be satisfied. After parsing, the predicate tree is checked for type-validity; the final result of evaluation must be a boolean value. Next a path-tree is constructed for the attributes specified in the FLIST and PLIST. This path tree representes the paths through the data structures that lead to the required attributes which may come from the top level of the SDSs or relations involved or may be part of substructures. For example, the simple query

```
Select    name (state), name (city), population
Into      a_new_relation
From      USA
Where     ((population (state) / population (city) < 15) and
           (area (city) > 100)) or
          ((commodity (state, exports) = 'wheat').
```

generates the path tree shown in Figure 5

Retrieval is then achieved by a recursive algorithm (Engineer, 1983).

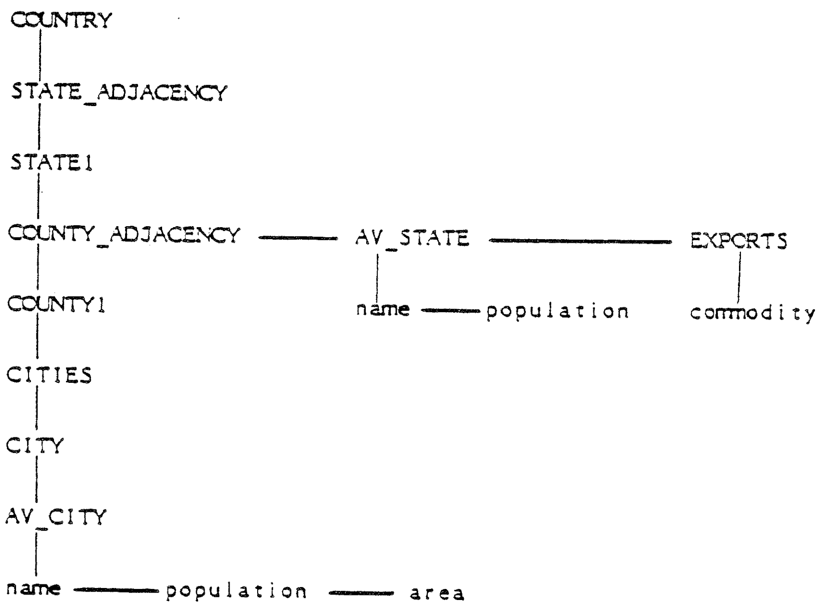

Figure 5: Path-tree for a sample query

VI. Summary

The use of hierarchical relational structures allows for a rich and natural representation of spatial data. The SQL language extends the kinds of queries allowed in relational systems to the more complex kinds of queries that can be asked about the hierarchical relational structures. The syntax and semantics of the language is, however, still very simple and straight forward. In particular, the generation of path trees during retrieval allows the user to specify as little as possible and makes the system do all the work. An information system based on these structures and this language should be a very useful system.

References

Astrahan, M. M and D. D. Chamberlin, "Implementation of a Structural English Query Language", Communication of the ACM, 18:10, 1975, pp. 580-587.

Date, C. J., An Introduction to Database Systems, Addison-Wesley, 1981.

Engineer, S. N., An Experimental Spatial Information System, M. S. Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1983.

Shapiro, L. G. and R. M. Haralick, "A Spatial Data Structure", Geo-Processing, Volume 1, 1980, pp. 313-337.

Ullmann, J. D., Principles of Database Systems, Computer Science Press, Rockville, MD 1980.

Vaidya, P. D., L. G. Shapiro, R. M. Haralick, and G. J. Minden, "Design and Architectural Implications of a Spatial Information System", IEEE Transactions on Computers, Volume C-31, No. 10, October 1982, pp. 1025-1031.

Wiederhold, Geo, Database Design, McGraw-Hill, Inc., New York, 1977.