To appear in Communications of the ACM, 1997

An Overview of the SAND Spatial Database System [1]

CLAUDIO ESPERANÇA [2]
HANAN SAMET
Computer Science Department and Center for Automation Research and
Institute for Advanced Computer Studies, University of Maryland
College Park, Maryland 20742

Abstract

An overview is given of the SAND system, an environment for developing applications involving both spatial and non-spatial data. The SAND kernel implements a relational data model extended with several geometric functions and predicates as well as quadtree and R-tree access structures. In addition, SAND provides a library which permits a modular construction of query evaluation plans. The main interface to SAND is through an embedded interpreted language. This permits the rapid prototyping of algorithms and makes SAND a useful tool both for applications and research.

[1] The support of the National Science Foundation under Grant IRI-92-16970, the Department of Energy under Contract DEFG0295ER25237, and of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under grant GEOTEC 481007/96-2 are gratefully acknowledged.

[2] Author's current address: COPPE, Programa de Engenharia de Sistemas e Computação, Universidade Federal do Rio de Janeiro, Ilha do Fundão, CT, Bloco H, Rio de Janeiro, RJ 21949-900, Brazil

1 Introduction

A principal challenge in the design of Geographic Information Systems (GIS) is the integration of spatial and non-spatial data. This field is known as spatial databases [3]. Spatial data consists of spatial objects made up of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time. Examples of spatial data include cities, rivers, roads, counties, states, crop coverages, mountain ranges, parts in a CAD system, etc. Examples of spatial properties include the extent of a given river, or the boundary of a given county, etc. Often it is also desirable to attach non-spatial attribute information such as road names and widths, city names, etc. to the spatial data. Such data is known as non-spatial data.

Since spatial and non-spatial data are so intimately connected, it is not surprising that many of the issues that need to be addressed are in fact database issues. Some deal with performance while others deal with how to interface spatial algorithms with large databases. The first issue is how to pose the query. The standard method of SQL is not particularly attractive as this requires considerable extensions whose generality and ability to handle other queries is not guaranteed.

Another issue is how to obtain the answer to the query. Traditional database management systems make use of a query optimizer. These query optimizers will generate a query plan which attempts to make use of knowledge about the underlying data and the complexity of the operations. This often makes use of selectivity factors which are related to data density. For example, suppose that we wish to find all cities within 100 miles of the Mississippi River with population greater than X. There are a number of possible query plans. The query consists of a spatial selection (100 miles within the Mississippi River) and a non-spatial selection (population greater than X).

If X is small, then it is preferable to perform the spatial selection first as it will reduce the search space. The spatial selection is the spatial analog of a range query and is known as a buffer zone or corridor in GIS parlance. Its computation is nontrivial and is greatly aided by the presence of an index on the spatial data. Next, we would use the index on the population attribute to obtain the desired cities. On the other hand, if X is large, then there will not be many cities that satisfy the query. In this case, it is not worthwhile to execute a spatial range query and instead we perform the non-spatial selection first and then check the location of each city to see if it is within 100 miles of the Mississippi River.

It should be clear that generating such plans automatically is nontrivial. In fact, this is the ultimate goal of the SAND (denoting *Spatial and Non-spatial Data*) project [1]. In this paper we report on the current state of this effort which is ongoing. Our aim here is to give an overview of the SAND environment for dealing with spatial and non-spatial data. The SAND kernel implements a relational data model extended with several geometric functions and predicates as well as a number of spatial access structures including quadtrees and R-trees. In addition, SAND provides a library which permits a modular construction of query evaluation plans. The main interface to SAND is through an embedded interpreted language. This permits the rapid prototyping of algorithms and makes SAND a useful tool both for applications and research.

The rest of this paper is organized as follows. Section 2 reviews some properties of spatial data such as extent and multidimensionality which influence the manner in which it is indexed and incorporated into a GIS. Section 3 indicates the system requirements. Section 4 describes the SAND kernel which provides the basic functionality needed to store and process spatial and non-spatial data. Section 5 presents the SAND interpreter which is the interface to the SAND kernel. Section 6 shows how a particular spatial query termed *ranking* is supported by the SAND interpreter. Section 7 gives an example of a graphical user interface to the facilities of SAND, while concluding remarks and directions for future work are discussed in Section 8.

2 Spatial Data

Non-spatial data is traditionally handled by standard database methods. The most common are relational and object-oriented. These methods invariably represent each tuple or object, depending on the technology, as a point (termed a *representative point*) in a higher dimensional space where the rank of the space is equal to the number of attributes that make up the relation or object specification. This has often led to a tendency to use the same techniques to deal with spatial data. These techniques are based on parameterizing the spatial data (e.g., a line segment is a pair of points corresponding to the coordinate values of its endpoints) and representing it as points in a higher dimensional space (termed the *transformation approach*). These parameters are retained as the spatial attributes of the data and are added to the non-spatial attributes to form a record or object in the database. An index is then usually built on the points.

The transformation approach is not always adequate for dealing with the wide array of spatial queries. It is fine if we just want to perform a simple retrieval of the data. For example, in the case of a database of line segments such as that found in a road network, it is good for queries about the line segment objects themselves such as which lines pass through the same point or have the same endpoints, etc. Its drawback is that it ignores the extent of the data (i.e., the space spanned by it). This means that the representative points of line segments whose lengths differ widely but are close to each other in the two-dimensional space in which they lie will not be close to each other in the higher-dimensional space into which they are mapped. This results in a large search space. Of course, this drawback can be overcome by projecting the lines back to the original space from which they were drawn, but, in such a case, we may then well ask what was the point of using the transformation in the first place?

Thus we see that when queries involve the space occupied by the data (and hence other records by virtue of their proximity), which occurs often, the situation is not so straightforward. In such a case, we need to be able to retrieve records based on some spatial properties which are not stored explicitly in the database. For example, in a roads database, we do not want to force the user to specify explicitly which roads intersect which other roads or regions. The problem is that the potential volume of such information may be very large and the cost of preprocessing it high, while the cost of computing it on the fly may be quite reasonable, especially if the spatial data is stored in an appropriate manner. Thus it is preferable to store the data implicitly so that a wide class of spatial queries can be handled. In particular, we need not know the types of queries a priori.

Being able to respond to spatial queries in a flexible manner places a premium on the appropriate representation of the spatial data. In order to be able to deal with proximity as well as range queries, the data must be sorted. Of course, all database management systems sort the data. The issue is which keys do they sort on, and the reference point. In the case of spatial data, the sort should be based on all of the spatial keys, which means that, unlike conventional database management systems, the sort is based on the space occupied by the data.

This is in contrast to sorting on the basis of distance from a particular location as would be the case if we would sort all cities on the basis of their distance from Chicago. In this case, the sort is helpful when we search for cities within 100 miles of Chicago but is useless when we want to search for cities within 100 miles of Las Vegas. Thus we would have to re-sort the cities. The reason that this approach did not work is that the locations of the cities are in two dimensions rather than one. In particular, we cannot say that the distance from x to Las Vegas is equal to the result of adding or subtracting some constant equal to the distance from Las Vegas to Chicago which is what is done in one dimension. Thus we see the need to sort the data implicitly (i.e., with respect to the space occupied by the data) instead of explicitly. Such techniques are known as *spatial indexing* methods [8, 9].

These methods decompose the space from which the data is drawn into buckets. There are four principal approaches. One approach buckets the data based on the concept of a minimum bounding (or enclosing) rectangle. In this case, objects are grouped (hopefully by proximity) into hierarchies, and then stored in another structure such as a B-tree. The R-tree (e.g., [4]) is an example of this approach.

The drawback of the R-tree (and any representation based on an object hierarchy) is that it does not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle associated with rectangle, yet it may overlap (fully or partially) several other bounding rectangles. In the worst case, this means that when we wish to determine which object is associated with a particular point (e.g., the containing rectangle in a rectangle database, or an intersecting line in a line segment database) in the two-dimensional space from which the objects are drawn, we may have to search the entire database.

The other approaches are based on a decomposition of space into disjoint cells, which are mapped into buckets. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules and by the way in which the cells are aggregated. The price paid for the disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies.

The first method based on disjointness partitions the objects into arbitrary disjoint subobjects and then groups the subobjects in another structure such as a B-tree. The R$^+$-tree [11] is an example of such a representation. Methods such as the R$^+$-tree (as well as the R-tree) have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay which are some of the principal operations in a GIS). The problem is that although these methods are good at distinguishing between occupied and unoccupied space in a a particular image, they are unable to correlate occupied space in two distinct images, and likewise for unoccupied space in the two images.

In contrast, the remaining two methods, while also yielding a disjoint decomposition, have a greater degree of data-independence. They are based on a regular decomposition. The space can be decomposed either into blocks of uniform size (e.g., the uniform grid) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach such as one for point data that recursively decomposes the space into four congruent blocks until each block contains no more than b points as is shown in Figure 1 for $b = 1$). In the former case, all the blocks are of the same size, while in the latter case, the widths of the blocks are restricted to be powers of two, and their positions are also restricted.

The uniform grid is ideal for uniformly distributed data, while quadtree-based approaches are suited for arbitrarily distributed data. In the case of uniformly distributed data, quadtree-based

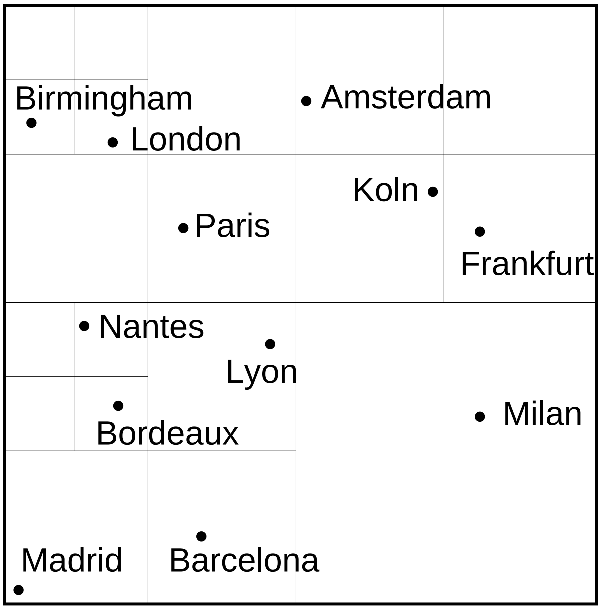

Figure 1: A quadtree block decomposition for point data based on decomposing the space into four congruent blocks until each block contains no more than one point.

approaches degenerate to a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations. Thus they are ideal for tasks which require the composition of different operations and data sets. In general, since spatial data is not usually uniformly distributed, the quadtree-based regular decomposition approach is more flexible.

3 System Requirements

In the first stages of SAND's development, we tried to outline the range of capabilities it was supposed to offer. This was a surprisingly hard task since we wanted SAND to be genuinely useful without offering the multitude of features found in commercial systems. A simple but realistic example query will help us illustrate some of the issues involved.

Assume we have a data set containing the names and locations of all the major cities of the world. Our example query consists of retrieving the names and locations of all cities which are no further from Paris than Madrid. A naive way of constructing the answer to this query is to first locate "Paris" and "Madrid" in the data set, compute the distance between them, say r, draw a circle of radius r around "Paris" and then select all cities lying within that circle.

In a typical GIS environment, the user would translate this query into an equivalent expression using the system's native query language. A query optimizer would analyze this expression and generate an evaluation plan. The plan would then be executed generating the answer data which would be stored in a disk file and/or presented on a graphical display.

Since we wanted to build a system for answering this and similar queries, a minimal list of requirements would include:

1. A database engine capable of storing and retrieving both conventional data (e.g., numbers and strings) and spatial data (e.g., points, lines, polygons).
2. Access methods suitable for indexing and sorting conventional and spatial data.
3. A software library for computing spatial operations, such as the distance between two points.
4. A graphical interface toolkit for displaying spatial and non-spatial data.
5. A way of combining the above elements into query evaluation plans.

Additionally, we would like the system to present a graphical user interface which, for simple queries such as that of the example, would not require the user to learn a query language. SAND was designed with this minimal set of features in mind.

Another key issue was how to integrate these components into a unified system and still be able to incorporate new functionality as needed. We opted to use *Tcl*, an interpreted embedded language that acts as a "glue" to hold together all the subsystems. Thus, query evaluation plans are *Tcl* programs (or *scripts*) which are written on the fly by SAND in response to a query defined by the user. As a demonstration of the level of integration achieved between the various SAND components, we wrote an application called the *SAND Browser*, which can be used to answer a variety of typical GIS queries, including that of the example. We return to this issue in Section 7, after reviewing the overall architecture of SAND.

4 The SAND Kernel

SAND adopts an extended relational data model. As in traditional relational database systems (RDBMS), SAND organizes data into a collection of tables, where each table consists of a set of tuples (i.e., records), and each tuple is an aggregation of simple data types called attributes. In SAND, different table types allow tuples to be accessed in a number of ways (e.g., by number, or by contents), and these access methods are visible to the user who can use them directly. This is in contrast to a typical RDBMS which tries to hide such details from the user, providing him instead with the means to describe the data to be retrieved, and a query optimizer which can put the available (but hidden) access methods to the best use.

Tables are handled in much the same way as regular disk files, i.e., they have to be opened so that input and output to disk storage can take place. Once a table is open, a memory buffer is created for the purpose of temporarily holding one tuple which is to be read from or written to the table. This is called the *tuple buffer* (refer to Figure 2).

Usually, all data pertaining to a given table is stored in a single file, except when any of the attributes is a *container* attribute. A container attribute is stored in a separate disk file and its value is not immediately fetched when a tuple is read from the table. Instead, a summary of that value and an identifier with its disk location are stored with the regular attributes. Attributes which refer to complex spatial features (e.g. polygons) are frequently stored using containers and the summary information consists of bounding structures that in many cases are sufficient

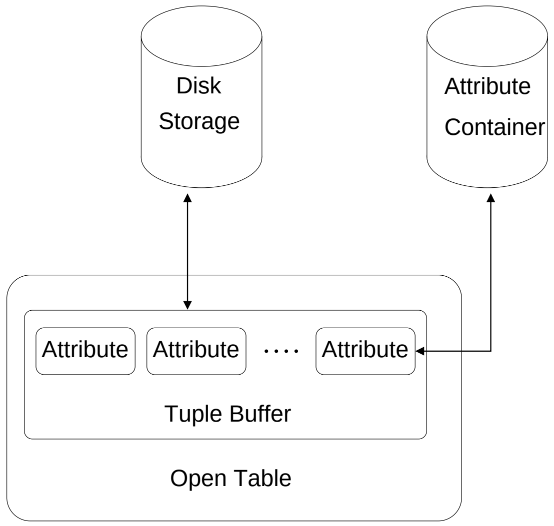

Figure 2: Example of an open table in SAND and its storage model.

to determine whether or not the corresponding value needs to be processed. Figure 2 illustrates the storage organization of an open table with one container attribute.

4.1 Table Types

All open tables in SAND respond to a minimal common set of operators, (e.g., **open**, **close**, get **first** tuple, get **next** tuple), while specialized table types offer additional capabilities. SAND currently defines three table types: relations, linear indices and spatial indices. Using object-oriented terminology, we may think of "Table" as a base class and the various table types as derived classes (see Figure 3). Each table type supports an additional set of operators which is appropriate to its function in a database environment. The function of most of these operators is to alter the order in which tuples are retrieved and, by extension, the behavior of **first** and **next**.

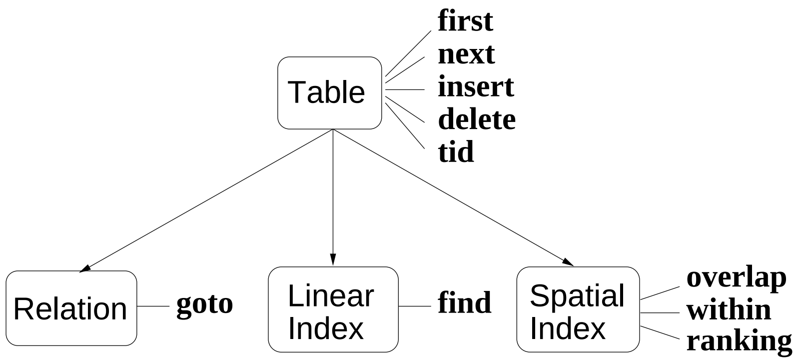

Figure 3: The class hierarchy of tables supported by SAND (the arrows denote class derivation)

Relations in SAND are tables which support direct access by tuple identifier (*tid*). Ordinarily, tuples are retrieved in order of increasing *tid*, but operation **goto** *tid* can be used to jump to the tuple associated with the given *tid* (if it exists).

Linear indices are implemented as B-trees. Tuples in a linear index are always scanned in an order determined by their contents. The ordering between two tuples is decided by applying operator **compare** to their attribute values. Linear indices support the **find** operator, which retrieves from the disk storage the tuple most closely resembling a tuple value given as an argument. The **find** operator can also be used to perform range searches, i.e., all tuples corresponding to keys lying between two given values can be retrieved by **find**ing the tuple corresponding to the lowest bound and then using **compare** and **next** until the key exceeds the highest bound.

Spatial indices are implemented as PMR-quadtrees[8, 9]. The default scanning order of a spatial index is not determined by any particular constraint, but is designed to permit the full contents of the table to be examined as quickly as possible. Spatial indices support a variety of spatial search operators, such as **overlap** for searching tuples that intersect a given feature, or **within** for retrieving tuples in the proximity of a given feature. Spatial indices also support **ranking**, a special kind of access method whereby tuples are retrieved in order of distance to a given feature or set of features.

4.2 Attribute Types

SAND implements attributes of common non-spatial types (integer and floating point numbers, fixed-length and variable-length strings) as well as two-dimensional geometric types (points, line segments, axes-aligned rectangles, polygons and regions). All attribute types support a common set of operators which characterize the functionality of a base "Attribute" class (see Figure 4). Non-spatial attribute types support the **compare** operator which is used to establish a total ordering among attribute values of the same type. No total ordering is possible among spatial values, but other kinds of relationships are supported by operators such as **distance** which computes the Euclidean distance between any two spatial values, or **intersect**, which determines whether two spatial values overlap. Other useful operators are: **bbox**, which returns the smallest axis-aligned rectangle that contains a given feature (i.e., its minimum bounding rectangle), and **draw**, which produces a pictorial representation of the feature.

Notice that while comparison between two different non-spatial attributes (e.g., a string and a number) is frequently meaningless, operators defined on pairs of spatial features had to be implemented for all possible combinations of spatial types. For example, since SAND currently supports five spatial types, there are a total of $5 + \binom{5}{2} = 15$ distinct routines to compute distances.

Some spatial features support additional operators. For instance, the *region* type supports the **expand** and **coarsen** operators which can be used to perform morphological operations.

5 The SAND Interpreter

The SAND kernel provides the basic functionality needed for storing and processing spatial and non-spatial data. In order to access the functionality of this kernel in a flexible way, we opted to provide an interface to it by means of an interpreted language. We chose *Tcl* [7] for that role, mainly because it offers the benefits of an interpreted language but still allows code written in a

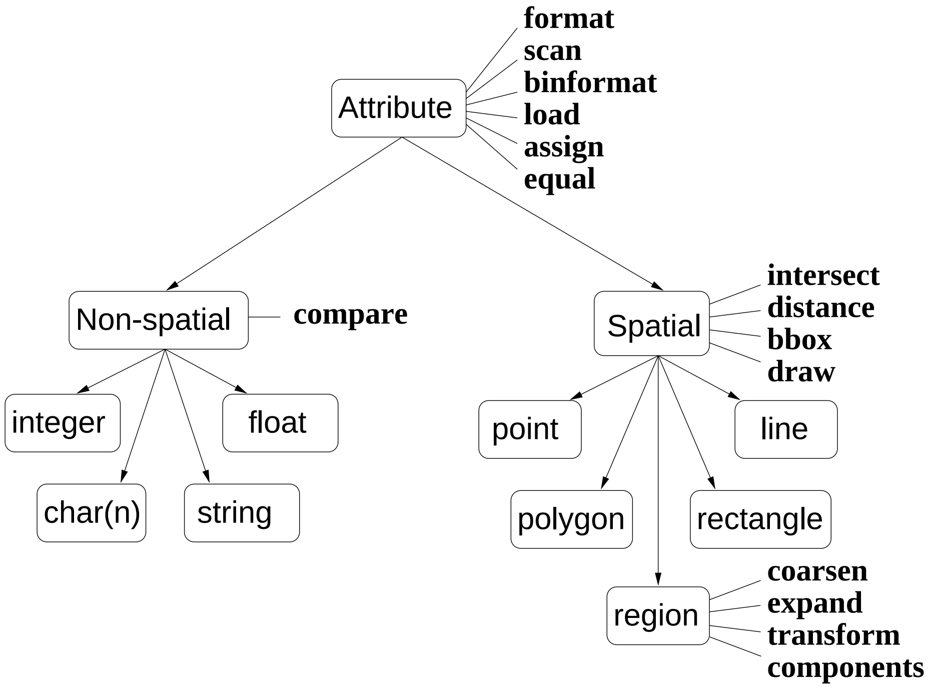

Figure 4: Attributes types implemented in SAND and some of the operations defined on them. The arrows denote class derivation.

high-level compiled language (in our case, C++) to be incorporated via a very simple interface mechanism. Another advantage offered by Tcl is that it provides a seamless interface with *Tk*[7], a toolkit for developing graphical user interfaces.

While typical DBMSs and GISs are designed to be accessed at various levels (e.g., by the database administrator via a data definition language and by the user via a structured query language), most aspects of SAND can be controlled and programmed by means of commands to the SAND interpreter. Some commands are defined by the kernel, some by the application developer, and some by the underlying *Tcl* interpreter. This enables the system to be extended by writing new methods or query strategies which are stored in a library of SAND scripts. In special cases where the kernel has to be modified, say by adding a new Table or Attribute type, the added functionality has to be incorporated to the interpreter by defining new commands. In fact, the interpreter can be viewed as the unifying element of the whole SAND system (see Figure 5, which is a block diagram of the SAND system).

The interactive environment provided by the SAND interpreter can be best illustrated by a sample session. Let us return to the example query described in Section 3 and assume the existence of a relation called `city` with attributes `cityname` (a string) and `citypos` (a point). A user could start the interaction with SAND by opening `city` and printing the first few tuples[3]:

```
% sand open city
table0
```

[3] In the session transcript, the user input is typeset with a `fixed-width font` while the interpreter's output is typeset in *italics*.

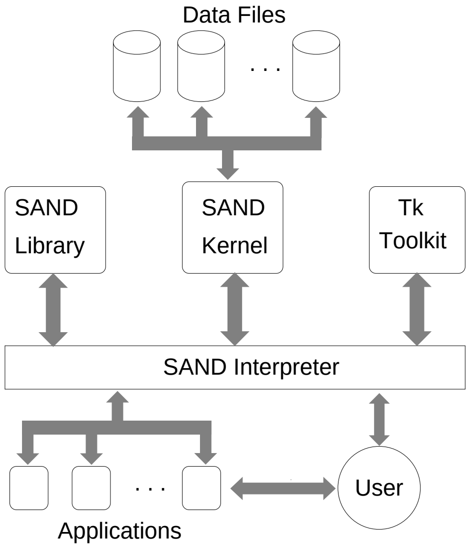

Figure 5: A block diagram of the SAND system.

% `table0 first; table0 get`
Oslo {point 10.687 59.8877}
% `table0 next; table0 get`
Minsk {point 27.4149 53.4485}

The meaning of this session is as follows: `sand open city` is a kernel command that requests that relation `city` be open for input and output. The result of this command, i.e., *table0* is a new command that encapsulates the functionality of the recently opened table. Thus, `table0 first` retrieves the first tuple into the tuple buffer and `table0 get` returns its value which is printed. Therefore, the first tuple corresponds to the city whose name is *Oslo* located at a point with coordinates *10.687* and *59.8877*. Similarly, the command sequence `table0 next; table0 get` retrieves and prints the next tuple, which is *Minsk* located at a point with coordinates *27.4149* and *53.4485*.

Since the first step in answering the example query clearly consists of finding "Paris", we could have used one of *Tcl* 's native control structures to repeat `table0 next` until the desired tuple was found. In fact, assuming that no other access structures exist for this data set, such a strategy could very well be optimal. In a more realistic scenario, however, `city` will be indexed by name using, say, a linear index called `cityname`. Normally, the user should not be required to have knowledge of such issues but still be able to take advantage of indices or other access structures that might be available. This functionality is provided in SAND by library scripts which analyze the query parameters provided by the user and generate new commands which implement an efficient (if not the best) access plan. For example, script `sandSelectPlan` takes a relation name and a selection condition as arguments and returns a new command which can be used to scan those tuples of the relation that satisfy the condition. Here is how it would be used to locate the tuple corresponding to "Paris".

% `sandSelectPlan city {cityname=="Paris"}`

```
plan0
% plan0 first; plan0 get
Paris {point 2.16623 49.2}
```

Notice that the plan command *plan0* returned by `sandSelectPlan` has a functionality that is very similar to the open table command *table0* returned by he `sand open` command. Whereas *table0* can be used to scan all tuples of `city`, *plan0* can be used to scan all tuples having attribute `cityname` equal to "Paris". This similarity is intentional, so that there is little difference between using primitive kernel commands or library scripts.

The remaining steps to answer the example query, i.e., finding the location of "Madrid", computing its distance to "Paris" and using this value to limit a search based on distance to "Paris" can all be expressed by commands derived from the SAND kernel, the SAND library, or from the repertoire of intrinsic *Tcl* commands.

6 Spatial Queries

The processing of spatial queries is supported by interpreter commands associated with spatial attributes and spatial indices. For instance, a command called `distance` can be used to compute the distance between any two spatial values, and a command called `forall` can be used on spatial indices in order to locate tuples that satisfy several spatial conditions.

In particular, spatial indices support *ranking* with respect to a given spatial object. Ranking can be viewed as a spatial analog to sorting. For example, it is not possible to order a collection of line segments in the same sense that a collection of numbers can be sorted in ascending or descending order. However, it is possible to *rank* line segments in ascending or descending order of distance from a given point. Moreover, ranking, in SAND is performed incrementally[5]. This means that once the parameters for a ranking operation are given, the first tuple is returned as soon as possible. In other words, when the spatial index receives command `first`, a searching algorithm is executed just until the point where the first tuple in the ranking order can be established. Consecutive tuples are obtained in a similar fashion. The key point of our algorithm is that having computed the first k elements, to get the $(k+1)^{st}$ element we need not sort the $k+1$ elements. In other words, we resume the search for the $(k+1)^{st}$ element from the point at which we obtained the k^{th} element. This is a better solution than sorting the entire data set especially when we may only need a few of the closest elements rather than all of the elements in which case sorting the set would have been a good idea. This characteristic is extremely important in an interactive environment.

Consider again the example query of Section 3. A solution to this query can be obtained interactively by ranking cities in ascending order of distance with respect to the location of "Paris". The first tuple will necessarily be "Paris" itself, followed by neighboring cities at increasing distances up to the moment where "Madrid" is returned. At this instant, the ranking operation can be interrupted and we will have collected the tuples that answer to the query.

Query evaluation plans involving spatial conditions, as those involving non-spatial data, are relatively simple in SAND. In fact, many different plans can be constructed that will correctly evaluate any given query. However, query optimization, i.e., a procedure for automatically generating the most efficient plan for a given query is a considerably harder problem when spatial relationships are involved. Despite this fact, the plans produced by the SAND library are usually adequate for simple spatial queries.

7 The SAND Browser

The interactive use of spatial databases requires some means for inputting and outputting spatial features other than through a conventional text-based interface. Additionally, query languages are not really practical in an interactive environment, especially when considering untrained or infrequent users. For this reason, spatial databases either offer a full-featured graphical user interface (e.g., GEO++[12]) or they supplement an existing text-based query interface with a presentation language (e.g., GEOQL[6], Spatial SQL[2]).

The SAND Browser is our first attempt to provide a graphical user interface to the facilities of SAND. It permits the visualization of the data contained in a SAND relation by specifying two types of controls: the scan order in which tuples are to be retrieved and an arbitrary selection predicate.

The class of queries currently implemented in the SAND Browser is restricted to selections and spatial semijoins. The user specifies queries by choosing the desired selection conditions from a variety of menus and dialog boxes. Spatial values can either be drawn on the appropriate display pane or typed in by filling forms. Query results can either be displayed interactively using the *First* and *Next* buttons or saved in relations for use in subsequent SAND queries.

The SAND Browser also benefits greatly from the incremental query processing capabilities of SAND, in the sense that a user rarely has to wait too long to get visual feedback provoked by an action. Prompt visual feedback is an important issue in the design of graphical user interfaces.

Figure 6 illustrates a sample session with the SAND Browser in which the answer to the example query of Section 3 is computed. The first step consists of obtaining the location of "Paris". This is done by specifying that the relation should be scanned ordered by name starting from "Paris" (Figure 6a). Once the location of "Paris" is known (Figure 6b), we start an incremental search where tuples are reported in order of distance from "Paris" (Figure 6c). By pressing the **Next** button, all cities which answer the query are displayed in succession until "Madrid" is found (Figure 6d), at which time we stop.

8 Concluding Remarks

SAND is an on-going project. We are conducting a thorough reorganization of its components in order to enhance its extension mechanisms. In particular, the Table and Attribute classes are being redesigned so that the incorporation of a new attribute or table type will require no modification of the interpreter command interface.

Another goal we aim to achieve is to provide SAND with a full-featured query optimizer. At this point, some optimizer functions are handled by a library which generates evaluation scripts for multi-way join queries. The strategies for generating such scripts are still very primitive. For instance, spatial indices are always used in queries involving spatial selection. Also, the potential of adaptive bounding structures as cost-estimation tools deserves further study.

In its current state, however, SAND has already proved to be of value. All experiments reported in this document were implemented within the framework of SAND. Additionally, SAND has been used for the prototype of an image database system [10].

References

[1] W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91*, pages 299–318, Berlin, 1991. Springer-Verlag.

[2] M. J. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86–95, February 1994.

[3] R. H. Güting. An introduction to spatial database systems. *Special Issue on Spatial Database Systems of the VLDB Journal*, 3(4), October 1994.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.

[5] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases — Fourth International Symposium, SSD'95*, pages 83–95, Portland, ME, August 1995. (also Springer Verlag Lecture Notes in Computer Science 951).

[6] B. C. Ooi. *Efficient Query Processing in Geographic Information Systems*, volume 471 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[7] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, April 1994.

[8] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[9] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[10] H. Samet and A. Soffer. Integrating images into a relational database system. Technical Report CS-TR-3371, University of Maryland, College Park, MD, October 1994.

[11] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[12] T. Vijlbrief and P. van Oosterom. The GEO++ system: An extensible GIS. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 40–50, Charleston, SC, August 1992.

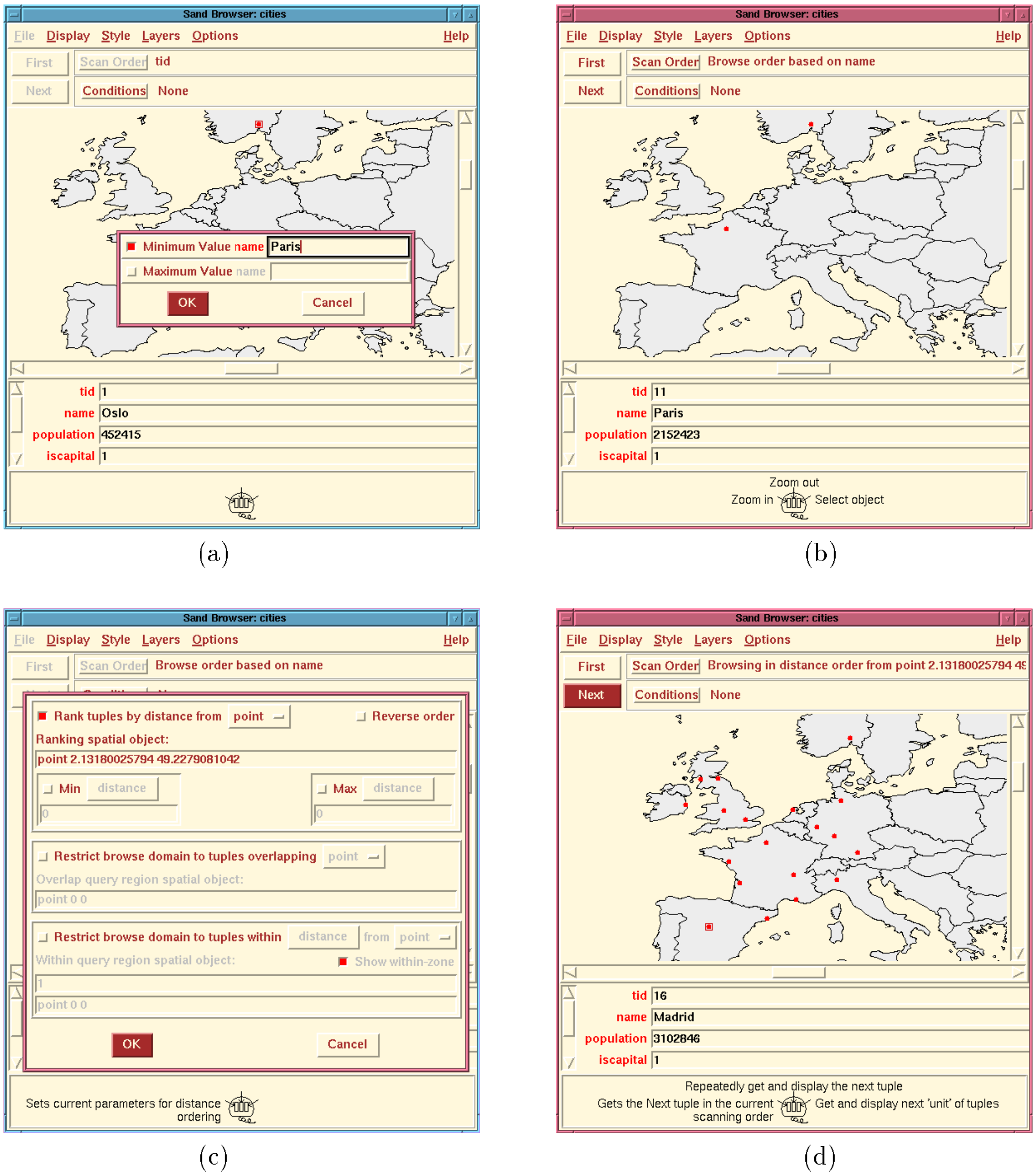

Figure 6: Steps taken when using the SAND Browser to answer the query "Which cities are no further from Paris than Madrid". (a) Specification of a scan sorted by `cityname` starting with "Paris"; (b) the location of "Paris"; (c) specification of an incremental search of cities according to their distances from "Paris"; (d) the resulting group when "Madrid" is encountered.