Geographical Analysis ISSN 0016-7363

Extending the SAND Spatial Database System for the Visualization of Three-Dimensional Scientific Data

Hanan Samet,[1] Robert E. Webber[2]

[1]Computer Science Department, Center for Automation Research, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, [2]Computer Science Department, The University of Western Ontario, London, Ontario, Canada

The three-dimensional extension of the SAND (Spatial and Nonspatial Data) spatial database system is described as is its use for data found in scientific visualization applications. The focus is on surface data. Some of the principal operations supported by SAND involve locating spatial objects in the order of their distance from other spatial objects in an incremental manner so that the number of objects that are needed is not known a priori. These techniques are shown to be useful in enabling users to visualize the results of certain proximity queries without having to execute algorithms to completion as is the case when performing a nearest-neighbor query where a Voronoi diagram (i.e., Thiessen polygon) would be computed as a preprocessing step before any attempt to respond to the query could be made. This is achieved by making use of operations such as the spatial join and the distance semijoin. Examples of the utility of such operations is demonstrated in the context of posing meteorological queries to a spatial database with a visualization component.

Introduction

The visualization of scientific data is an important area of research (e.g., Ertl, Joy, and Varshney 2001; Moorehead, Gross, and Joy 2002). In most of the work, the data is treated as a collection with special purpose programs written to display it. It is often the case that the data has a spatial component as it covers some area in space. The data also has a nonspatial component, which is also known as its nonspatial attribute. Some examples of nonspatial data include temperature and pressure readings where the temperature and pressure readings are taken at a variety of locations. Nonspatial data is usually handled by commercial off-the-shelf database

Correspondence: Dr. Hanan Samet, Computer Science Department, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742
e-mail: hjs@cs.umd.edu

Submitted: January 01, 2004. Revised version accepted: March 10, 2005.

systems (e.g., ORACLE, INFORMIX, SYBASE, etc.). Recently, there has been considerable interest in using conventional database technology with spatial data.

The most common example of an application that mixes spatial and nonspatial data is a geographic information system (GIS). Traditionally, the main motivation for the development of a GIS has been a desire to look at the data from a graphical point of view. Clearly, a GIS can be viewed as an example of a low-level visualization system. A key challenge in the design of GIS is the integration of spatial and nonspatial data. This field is known as spatial databases (e.g., Rigaux, Scholl, and Voisard 2002; Shekhar and Chawla 2003). Most of the work in spatial databases has dealt with the two-dimensional plane. There are a number of GIS in use. They usually make a distinction between the treatment of spatial and nonspatial data in that they are not too well-integrated with the result that the spatial and nonspatial components of a query are handled separately instead of being tightly coupled. We have developed a prototype system known as SAND (Spatial and Nonspatial Data) (Aref and Samet 1991; Esperança and Samet 1996, 2002; Samet et al. 2003), which tries to couple the handling of the spatial and nonspatial components.

The SAND system was designed and built to deal primarily with two-dimensional data. In this article we report on our experience with extending SAND to deal with three-dimensional data such as that found in scientific visualization work. In particular, we show how some conventional database operations can be used to yield new ways of looking at scientific data. The result is a powerful technique for performing operations such as clustering and windowing. The rest of this article is organized as follows: the following section gives an overview of the capabilities of the SAND system. This is continued by a section that reviews the database structure of SAND and outlines the necessary extensions that needed to be made to support handling three-dimensional surface data. Succeeding three sections describe the ranking, spatial join, and distance semijoin operations provided by the SAND system. A section containing some examples of the use of the three-dimensional extension to SAND comes next. The last section provides concluding remarks.

Overview of the capabilities of SAND

SAND is a GIS system developed at the University of Maryland to merge database-style query languages with quadtree-style indexing strategies to handle geometric information. The original development of SAND was focussed on two-dimensional data such as country boundaries, river paths, and city locations and facilitating the response to queries involving them. GIS queries involve objects of different type, such as what is the closest power plant to the border of a particular country. Scientific visualization usually involves higher dimensional data. In this article we focus on how to handle the third dimension in SAND. This is done through the addition of a new object type, *isosurface*, to enable us to deal with data such as that found in meteorological applications. With this new type, SAND is now able to respond to queries such as finding the closest city to a 270 K (roughly $-3\,^{\circ}$C)

temperature reading at a particular minimum elevation or elevation range value (which, in conjunction with other measurements might indicate the potential for snow). SAND does not incorporate the concept of time except for the ability to handle time as an additional dimension. Of course, data can be stored for different time periods by using time as an attribute.

A major aspect of SAND is the notion of finding not just the closest city to a particular location or spatial object, but, instead, incrementally producing a list of all cities ordered by their distance from the particular location or spatial object. Thus, for example, if we are looking for a major manufacturing city near the Canadian border, we could launch a query for cities ordered by distance from the Canadian border and then inspect the list in order until a city occurred that ''seemed like'' a major manufacturing center. This allows the user to be involved in the data analysis. An alternative approach would be for the user to have to quantify what is meant by ''major manufacturing center'' and spend time ''debugging'' the formal definition of such a subjective term, which would probably take longer than just scanning the list and intuitively classifying the cities in terms of their manufacturing capabilities. Another way of thinking about this is that SAND aids in the exploration of the data set. Presumably such capabilities would be useful to scientists exploring meteorological data as well.

In addition to aiding exploration of the data set by queries on spatial attributes, SAND also permits queries regarding nonspatial attributes. Taking our manufacturing center example a bit further, one might have stored with each city the amount of freight shipped from the city in a year. Rather than just asking for cities near the Canadian border, the user could ask for cities near the Canadian border that ship more than a certain amount of freight each year and thus have a smaller and more relevant list of cities to inspect.

In SAND, countries, rivers, and cities are not predefined types. Instead, the predefined types are their corresponding spatial objects such as polygons, sequences of line segments, and points. A collection of cities then becomes a collection of points where the user knows that that particular collection of points corresponds to actual cities. Thus, at a more abstract level, we can view the original version of SAND as a database engine for working with two-dimensional geometry. The extensions necessary to handle isosurface data essentially mean extending SAND to handle three-dimensional geometry (i.e., points, line sequences, and polygons located in a three-dimensional space).

Database structure of SAND and needed extensions to support three dimensions

SAND adopts an extended relational data model. It organizes data into a collection of tables (i.e., relations), 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. Tables are handled in much the same way as regular disk files. SAND offers three table types: relations, linear indices, and spatial indices. Each table type supports operators, which usually serve to alter the order in which tuples are retrieved.

1. Relations in SAND are tables, which support direct access by tuple identifier (*tid*).
2. 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 determined by applying operator **compare** to their attribute values. Linear indices support the **find** operator, which given the value v of a search key, retrieves from the disk storage the tuple whose key value is equal to v or is as close as possible to v (e.g., the minimum value greater than equal to v). The **find** operator can also be used to perform range searches—that is, given the range of key values $[b,e]$, determine all tuples whose key values lie in this range. This can be achieved by **find**ing the tuple with the minimum key value $\geq b$ and then using the **compare** and **next** operators to successively retrieve keys in increasing order until finding a key whose value is $> e$.
3. Spatial indices are implemented as PMR-quadtrees Samet (1990a,b, 2005). The default scanning order of a spatial index is not determined by any particular constraint, but instead 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 as well as **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 the order of the distance of their associated object(s) from a given feature or set of features.

SAND implements attributes of common nonspatial 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. Nonspatial 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.

Notice that while comparison between two different nonspatial attributes (e.g., a string and a number) is frequently meaningless, this is not the case for different spatial attributes. In particular, we often need to be able to determine the distance between, for example, points and lines, lines and polygons, etc. It is precisely this situation where we had to devote considerable effort to enable SAND to deal with three-dimensional data.

As we pointed out above, our three-dimensional data is in the form of a surface. However, the data usually comes in the form of data samples on a grid. Thus the surface is usually interpolated to form a polygonal surface. It is frequently convenient to represent three-dimensional surfaces as networks of triangles. The advantage of a triangulation is that it is a surface approximation that preserves continuity. There are many triangulation methods in use. Note that once the surface data is triangulated we need to find an efficient representation for it.

Representing the data as a collection of individual triangles is somewhat inefficient as each triangle vertex is stored as many times as the number of triangles in which it is a component. This storage problem is overcome by storing the triangles using the ''triangular strip format,'' which decomposes the data into linear sequences of adjacent triangles (i.e., they share a common edge). These strips can then be stored as a sequence of vertices. The first three vertices form the base triangle and then each additional triangle in the strip is represented by just one more vertex (which is combined with the last two vertices of the previous triangle to form a new triangle). The strips do not have any logical structure associated with them. We use algorithms in the Visualization Toolkit (Schroeder, Martin, and Lorensen 1996) to convert the original three-dimensional data to triangle strips. It is important to note that the SAND database views a triangular strip as an atomic object, which cannot be decomposed further (although we can perform operations like distance and intersection on it).

Using triangular strips is an efficiency issue. For example, the data we are using started as a three-dimensional grid of points. We used Volume Visualization Toolkit (Schroeder, Martin, and Lorensen 1996) to analyze these points and find a surface that represented a particular isosurface. This toolkit then presented that surface to us as a collection of triangles. The method of triangular strips served as a compact way for the program to store the triangles (and is a commonly used representation in the computer graphics community making it efficient to display).

In the three-dimensional extension of SAND, each triangle strip is stored in a tuple. Thus, the decomposition of a surface into triangle strips is analogous to the decomposition of a polygon into sequences of adjoining triangles. We can also make an analogy to a decomposition of a polyline into convex linear segments as is often done for roads. While there might be a natural way of connecting and ordering the convex linear segments that form a polyline, there is no such natural ordering on adjoining triangle sequences that form a polygon or more generally in three-dimensional space that form an isosurface.

Generalizing all this work for *k*-dimensional objects would enable SAND to have the capability to work with isosurfaces across time rather than having to deal with individual time slices as is now the case. Potentially, this could also allow some flexibility between when deciding whether an attribute is considered as a ''spatial attribute'' and when it is considered as a numeric ''nonspatial attribute.''

To fully support the three-dimensional surface data we augmented SAND to include three-dimensional geometric data types of the form of points, line

segments, axis-aligned rectangles, triangles, and triangle strips. The addition of these new data types caused us to have to support many more comparison routines such as between points and triangles, lines and triangles, etc.

The routines that make up the database structure of SAND are written in C/C++. Users interact with them through the use of an interface written in iTcl (an object-oriented version of Tcl [Ousterhout 1994]). This interface permits the visualization of data through two types of controls: the scan order in which tuples are to be retrieved and an arbitrary selection predicate that also enables the use of two relations via a spatial join. The queries are specified with the aid of menus and dialog boxes, while spatial values can be specified graphically by drawing them. Query results can be displayed interactively or by saving them in relations for use in subsequent SAND queries in a manner similar to the use of a spreadsheet.

Ranking

One of the key features of the SAND system is the support of the *ranking* operation. It enables us to find objects in the order of their proximity to other objects. Ranking can be viewed as a spatial analog of sorting. For example, it is not possible to order a collection of points in the same sense that a collection of numbers can be sorted in ascending or descending order. However, it is possible to *rank* points in ascending or descending order of distance from a given point or spatial object.

In SAND, ranking is performed incrementally (Hjaltason and Samet 1999). This means that once the parameters for a ranking operation are given, the first (and closest) tuple is returned as soon as possible. Thus the search 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 property of our algorithm is that having determined the first k elements, we can obtain the $(k+1)$st element without having to start the process from scratch and obtain the $k+1$ elements in order. In other words, we resume the search for the $(k+1)$st element from the point at which we obtained the kth 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.

Ranking in SAND is implemented using a priority queue, which contains both objects and blocks. The algorithm works for any hierarchical spatial data structure (including BSP trees [Fuchs, Kedem, and Naylor 1980], R-trees [Guttman 1984], k-d trees [Bentley 1975], and all variants of quadtrees [Samet 1990, 2005]). What makes the algorithm efficient is that instead of examining individual objects for proximity, we examine their containing blocks (which may contain more objects). This is done by a process analogous to a tree search, which processes the blocks in the order of their distance from the ranking object. For more details about the algorithm, see Hjaltason and Samet (1995, 1999, 2003).

Spatial join

A key feature of SAND is the focus on the spatial join as a major mode of interacting with the data. A join (e.g., Elmasri and Navathe 1989) of two relations is a special case of the Cartesian product of the two relations where values of designated attributes must satisfy a given condition called the *join condition*. A spatial join (e.g., Brinkhoff, Kriegel, and Seeger 1993; Brinkhoff and Kriegel 1994) is a special case of the join operation where the join condition involves a spatial attribute. For example, consider a rivers relation (where the spatial attribute is the set of line segments comprising the rivers) and a roads relation (where the spatial attribute is the set of line segments comprising the roads). In this case, a spatial join of rivers and roads where the join condition is that the spatial objects associated with the spatial attributes have at least one point in common. In other words, their spatial intersection is nonempty. In particular, the result of the join is the set of tuples (i.e., their Cartesian product) where each tuple in the result is formed by a pair of tuples (one from roads and one from rivers) whose spatial attribute values have at least one point in common. Thus, in SAND, whenever we want to ask a question about the relationship between the elements of two different data sets, the question is posed as a join (or a distance semijoin as discussed in the next section).

The three-dimensional version of SAND was developed to explore a set of isosurfaces extracted from meteorological data, specifically temperature measurements at various altitudes and locations at various times. One data set type of interest to us consisted of the isosurfaces from a particular time that corresponded to a particular temperature range. Note that it is not necessary that all the temperature readings with a particular value or range form a single connected region. Thus a particular temperature range region may be the union of many surfaces. Another data set type of interest is one where all the isosurfaces from a particular time period are in the same data set and each is labeled by its corresponding temperature value.

An example of a spatial join operation involving our three-dimensional data takes the spatial join of the set of locations of the cities in the two-dimensional plane of the surface of the Earth with the set formed by the collection of triangle strips corresponding to the isosurface for a particular temperature. The set of resulting tuples would be ordered (i.e., ranked) by their separation or distance between them using the algorithm described above in ''Ranking.'' This would allow us to find which cities are closest to a particular temperature isosurface.

It is interesting to note that neither the isosurface nor the connected components of the isosurface form natural entities in the isosurface data set. Instead, the isosurface data set is the union of a collection of triangle strips. Thus, in the above join, each city would appear in as many tuples of the resulting set as there are triangle strips in the data set. However, SAND has a means of dealing with this issue by ensuring that each isosurface would in fact be reported just once for each city (although a city would still appear in as many tuples as there are different isosurfaces) by making use of its *unit attribute* mechanism. This mechanism enables the

user to change the meaning of the entities that are ranked. In particular, instead of the default option of using individual tuples as the distinct entities, we now can specify that the distinct entities have a common attribute value for a given attribute name or set of attribute names.

Spatial semijoin

An important variation on the spatial join is the spatial semijoin. Briefly, the spatial semijoin is motivated by the regular semijoin operation in relational databases (e.g., Elmasri and Navathe 1989). The semijoin of relation A with respect to relation B with join condition C is formed as follows. We first compute the spatial join of A and B with join condition C. Recall from the previous section that this operation is computed taking the Cartesian product of sets A and B and then just retaining the tuple pairs where condition C is satisfied. For example, the result of this operation could be the set of tuples: (a_1, b_2) (a_1, b_3) (a_2, b_4) (a_2, b_5) (a_3, b_6), where a_i corresponds to the attributes from relation A and b_i corresponds to the attributes from relation B.

The difference between the semijoin and the join operation is that in the semijoin we also specify a surviving relation. The term *surviving* means that we apply a projection on the attributes of the surviving relation after computing the spatial join. This means that after computing the spatial join, we only retain the attributes of the Cartesian product subject to the join condition that came from relation A (this is the projection). This means that in our example, the set of tuples (a_1, b_2) (a_1, b_3) (a_2, b_4) (a_2, b_5) (a_3, b_6) would be replaced by (a_1) (a_1) (a_2) (a_2), (a_3). Now, notice that there are two instances of each of the attribute values a_1 and a_2. At this point, we must decide which one of the two instances of a_1 and a_2 to retain. The conventional definition of the semijoin picks one of them at random and the same is done for the spatial semijoin with the difference that the join condition had to involve the spatial attribute.

At this point, let us define variants of the spatial join and spatial semijoin that we call the *distance join* and *distance semijoin*, respectively (Hjaltason and Samet 1998). The *distance join* is a spatial join where the join condition involves the distance (i.e., separation using a suitably defined distance metric) between the objects associated with the spatial attributes of the two joined relations. The *distance semijoin* (also referred to as the *ranking spatial semijoin*) differs from the conventional semijoin and spatial semijoin in the following two ways:

1. If there are several duplicate instances of the tuples resulting after the projection, then the distance semijoin is formed by retaining the tuple instance with the smallest distance between the objects associated with the spatial attributes of the two joined relations.
2. We also retain the attribute values associated with the tuple of the nonsurvivor relation that participated in the join. This is different from the conventional definition of the semijoin.

As an example, of the utility of the ranking distance semijoin, suppose that we have one relation called stores and another called warehouses. In this case if we compute the distance semijoin of the stores and warehouses relations with stores being designated as the surviving relation, then we have the result that of the (store, warehouse) tuple pairs, we retain the (store, warehouse) pair with the smallest distance between the two spatial objects associated with the corresponding tuples that form the pair. In essence, what we have done is use the distance value between the two objects associated with the spatial attribute values of the relation to determine which of the many tuples with the same store to retain (i.e., just one of them). We could have decided to retain several of them in which case we have a variant of a *k*-nearest-neighbor operation (e.g., Xia et al. 2004). In this case, the ranking comes into play in the sense of which of the objects of the surviving relation is reported first (actually its corresponding tuples). The result is analogous to a discrete space Voronoi diagram (e.g., Preparata and Shamos 1985) where the sites are the locations of the warehouses and the discrete space is the locations of the stores. This is also known as a *clustering join* (Wang and Shasha 1990). The advantage of using the distance semijoin over the more conventional method of constructing an entire Voronoi diagram is when we only want a few results as now we do not have to compute the entire Voronoi diagram.

As an example of the utility of the distance semijoin in a visualization context, suppose that we compute a distance semijoin between a data set of cities and a data set of all isosurfaces from a particular time where the join condition is the distance of their separation. This would yield pairs of cities and isosurfaces where each city is associated with its nearest isosurface (temperature reading). The distance semijoin is computed with the aid of the incremental nearest-neighbor algorithm described above in ''Ranking.'' The efficient computation of the distance semijoin is an area of active research (e.g., Shin, Moon, and Lee 2000).

Example usage of SAND for scientific visualization

We have used the three-dimensional extension of SAND to visualize meteorological data in conjunction with our work on the OPTIMASS (Optimizing Mass Storage Organization and Access for Multi-Dimensional Scientific Data) project. The OPTIMASS project was, in part, a collaborative research effort between researchers at the Lawrence Berkeley Laboratory, the Lawrence Livermore Laboratory, and the University of Maryland. Our data set represented the surface of the world as a 320×160 grid above which 19 ranges of height values (also termed *height levels*) existed. Measurements, such as air temperature, were stored for many time intervals. This data was stored in DRS (data retrieval and storage system) format provided by the Program for Climate Model Diagnosis and Intercomparison at Lawrence Livermore National Laboratory. Instead of working directly with this four-dimensional array of measurements, we converted it to isosurfaces. Each iso-

surface corresponded to a boundary in the original data at a particular time step and with regards to a particular measurement type (such as air temperature).

Our example application makes use of three data collections. The cities collection consists of five-tuples where the attributes are: a geometric location, a Boolean flag as to whether or not it is a capital, a population count, the name of the city, and a unique tuple identifier (tid) that tags it in the system. The isosurfaces collection consists of five-tuples where the attributes are: a triangular strip representing a portion of the isosurface, a temperature value for the isosurface, the time at which the temperature was measured, the name of the isosurface that the triangular strip belongs to, and a unique tuple identifier. Although the countries collection is not used in the queries, it does allow us to display the geographic boundaries of different countries to help orient us when viewing the other data.

The utility of the three-dimensional extension of SAND is that it provides a capability of formulating a wide range of queries. We executed a number of queries on this dataset to illustrate the use of our system. A general query with this data is

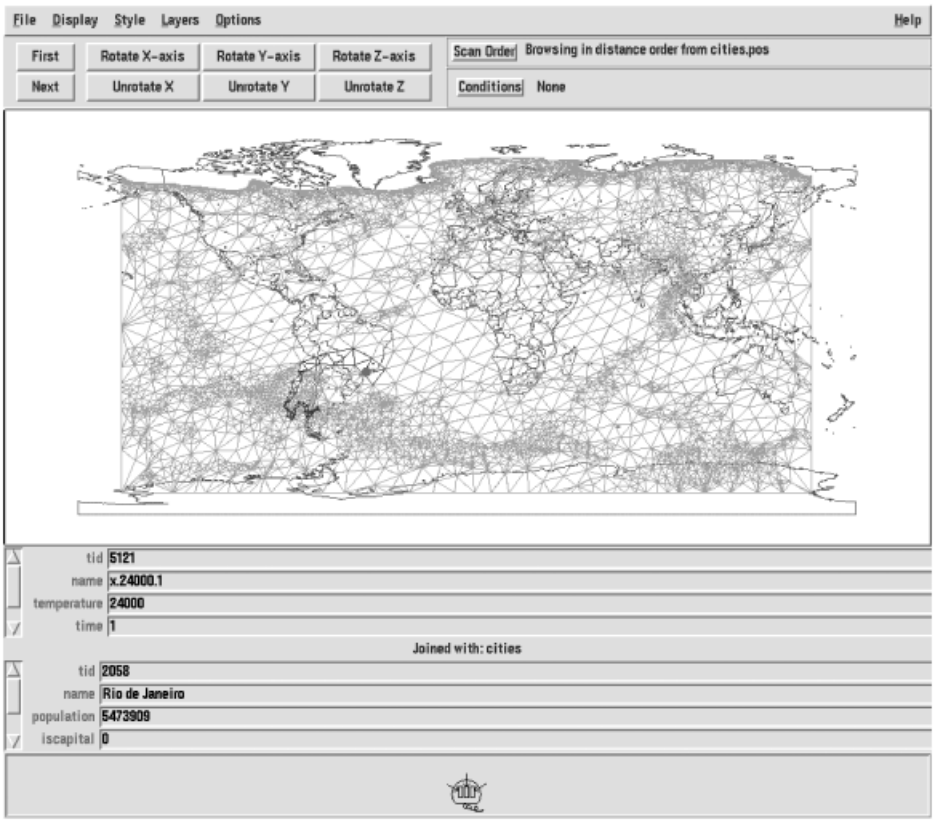

Figure 1. Query to find the closest cities to isosurfaces of different temperature values and time periods.

one that seeks to find the locations of the closest cities to isosurfaces of different temperature values and different time periods in increasing order of distance (i.e., a distance semijoin). In this case, once the closest city has been found for a particular isosurface *i* (for a specified temperature value or range and time period), that isosurface will not be reported again. An example of the response to such a query (just the closest isosurface–city pair) is given in Fig. 1. There we see the closest city (i.e., Rio de Janeiro) and isosurface fragment pair listed in the text portion at the bottom of the display. The isosurface fragment (triangular strip) is colored blue rather than green to indicate its position. The city, Rio de Janeiro, is the large red dot near the wide end of the isosurface fragment.

We can vary this query by stipulating that we want the closest cities to a particular temperature value or range regardless of the time periods. In this case, we are only looking at isosurfaces for one temperature value or range. Fig. 2 is an example of the result of the execution of such a query for temperature 260.00. In this case, we used the *condition* menu of SAND to set the condition on the

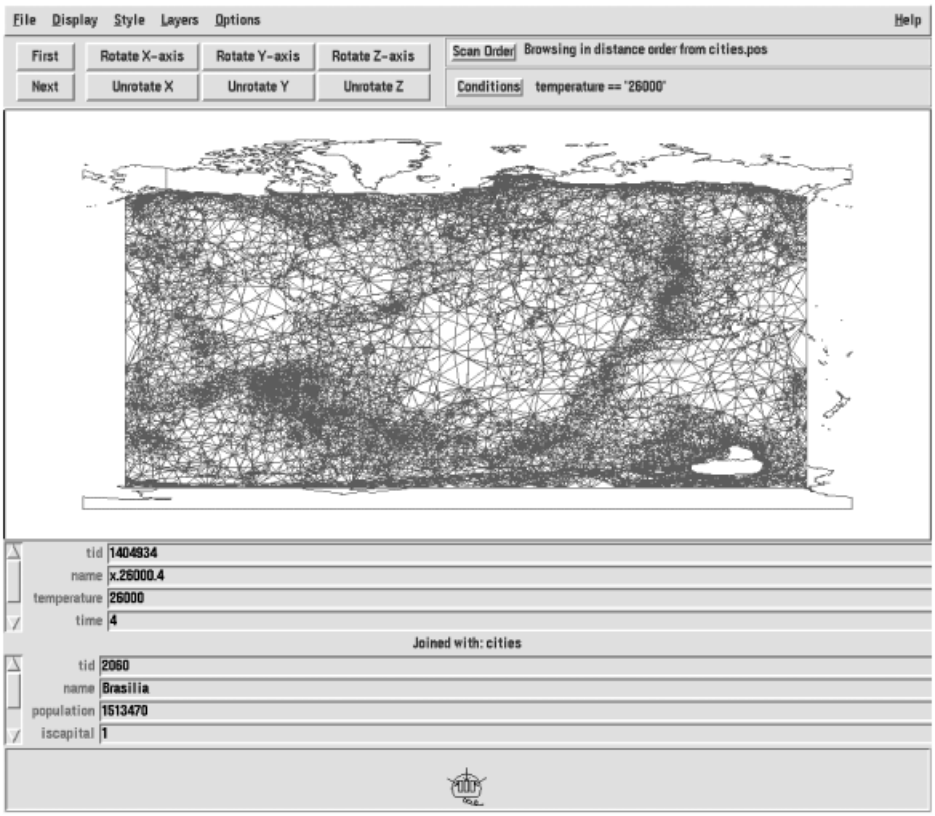

Figure 2. Query to find the closest cities to isosurfaces for the particular temperature value 260.00.

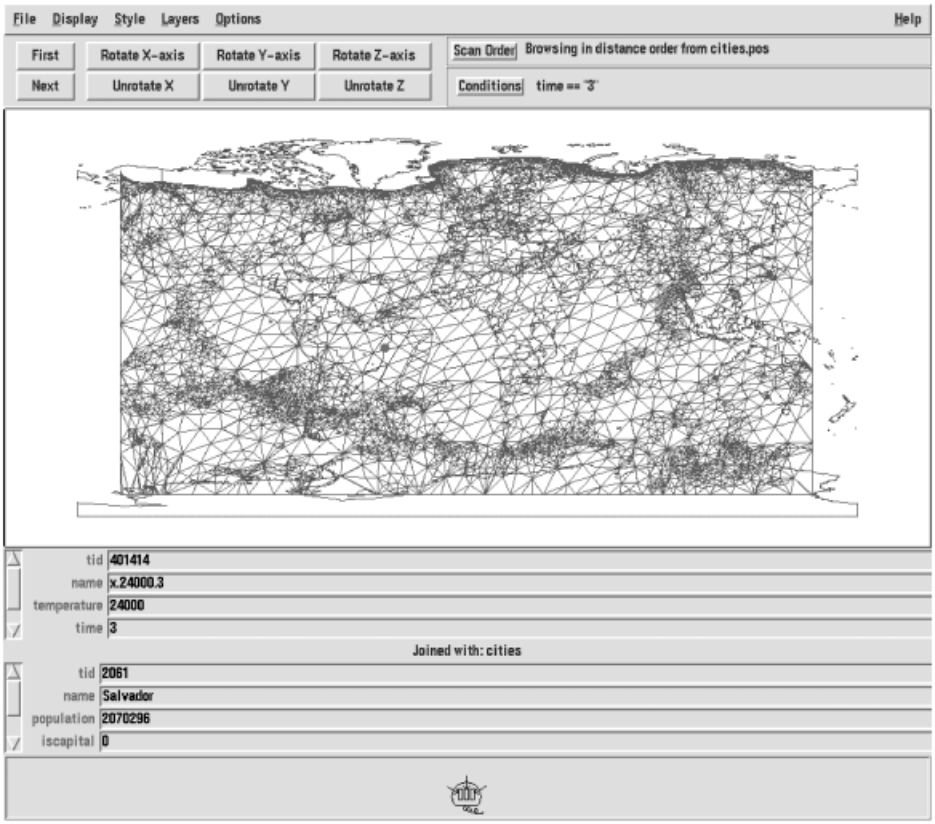

Figure 3. Query to find the closest cities to isosurfaces of a particular time period 3.

temperature value. Again we see the isosurface fragment in blue and the city, this time Brasilia, is the large red dot that nearly covers the isosurface fragment (this particular fragment is rather small).

The query can also be varied further by stipulating that we are only interested in the closest cities to any temperature value but for a particular time period or range. Fig. 3 is an example of the result of the execution of such a query for time period 3. In this case, again, we used the *condition* menu of SAND to set the condition on the time period. The relevant isosurface fragment is highlighted in blue and the city, this time Salvador, is the large red dot along the coast of Brazil.

Notice that in all three examples we assumed the use of the *unit attribute* feature of SAND to examine all triangle strip tuples associated with a particular time period and temperature value or range isosurface. It should be clear that we could have also executed the reverse form of the query in the sense that we would take the distance semijoin of the cities with the isosurfaces. In this case, for each city, we would obtain the closest isosurface, and thus be able to determine the time period and temperature value at which it occurred. Fig. 4 is an example of the result of the

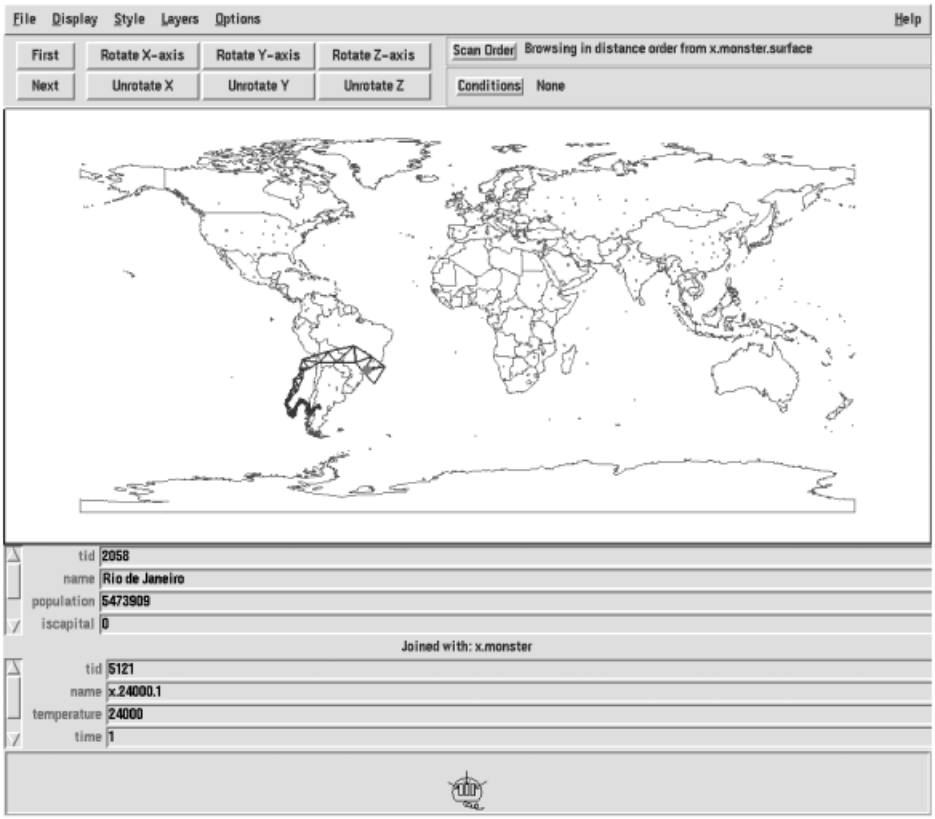

Figure 4. Query to find the closest isosurfaces to cities over all temperatures and time periods.

execution of such a query. Again the relevant isosurface fragment is highlighted in blue and the city, Rio de Janeiro, is the large lavender dot near the wide end of the fragment. In this particular query a different color had been chosen for the city database.

Concluding remarks

We have described the extension of a two-dimensional spatial database system to deal with data found in scientific visualization and described how some database operations could be used. The SAND system is not a commercial system. Instead, it is a prototype developed for research purposes that is constantly undergoing revision. We are working on a number of such extensions as well as on the development of efficient algorithms for computing distance semijoins. In addition, the real test of the viability of our approach is embedding it in a commercial database management system, which requires a considerable amount of additional issues to

be resolved such as the incorporation of spatial data into a query optimizer and other core database issues.

Acknowledgements

This work was supported in part by the Analysis and Visualization of Massive Data Sets projects administered by the University of California at Davis under Grant 98RA12807; by the National Science Foundation under Grants EIA-99-00268, IIS-00-86162, and EIA-00-91474; the Department of Energy under Contract DEFG0295ER25237; and Microsoft Research.

References

Aref, W. G., and H. Samet. (1991). ''Extending a DBMS with Spatial Operations.'' In *Advances in Spatial Databases—2nd Symposium, SSD '91*, Zurich, Switzerland, August, 299–318, edited by O. Günther and H. -J. Schek.

Bentley, J. L. (1975). ''Multidimensional Binary Search Trees Used for Associative Searching.'' *Communications of the ACM* 18(9), 509–17.

Brinkhoff, T., and H.-P. Kriegel. (1994). ''The Impact of Global Clustering on Spatial Database Systems.'' In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago, Chile, September, 168–79, edited by J. Bocca, M. Jarke, and C. Zaniolo.

Brinkhoff, T., H.-P. Kriegel, and B. Seeger. (1993). ''Efficient Processing of Spatial Joins Using R-Trees.'' In *Proceedings of the ACM SIGMOD Conference*, Washington, DC, May, 237–46.

Elmasri, R., and S. B. Navathe. (1989). *Fundamentals of Database Systems*. Redwood City, CA: Benjamin Cummings.

Ertl, T., K. Joy, and A. Varshney, eds. (2001). *Proceedings IEEE Visualization '2001*, San Diego, CA, October.

Esperança, C., and H. Samet. (1996). ''Spatial Database Programming Using SAND.'' In *Proceedings of the 7th International Symposium on Spatial Data Handling, Volume 2*, Delft, The Netherlands, August, A29–A42, edited by M. J. Kraak and M. Molenaar. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information.

Esperança, C., and H. Samet. (2002). ''Experience with SAND/Tcl: A Scripting Tool for Spatial Databases.'' *Journal of Visual Languages and Computing* 13(2), 229–55.

Fuchs, H., Z. M. Kedem, and B. F. Naylor. (1980). ''On Visible Surface Generation by a Priori Tree Structures.'' *Computer Graphics* 14(3), 124–33. Also *Proceedings of the SIGGRAPH '80 Conference*, Seattle, WA, July.

Guttman, A. (1984). ''R-trees: A Dynamic Index Structure for Spatial Searching.'' In *Proceedings of the ACM SIGMOD Conference*, Boston, June, 47–57.

Hjaltason, G. R., and H. Samet. (1995). ''Ranking in Spatial Databases.'' In *Advances in Spatial Databases—4th International Symposium, SSD '95*, Portland, ME, August, 83–95, edited by M. J. Egenhofer and J. R. Herring.

Hjaltason, G. R., and H. Samet. (1998). ''Incremental Distance Join Algorithms for Spatial Databases.'' In *Proceedings of the ACM SIGMOD Conference*, Seattle, WA, June, 237–48, edited by L. Hass and A. Tiwary.

Hjaltason, G. R., and H. Samet. (1999). ''Distance Browsing in Spatial Databases.'' *ACM Transactions on Database Systems* 24(2), 265–318. Also *Computer Science TR-3919*, University of Maryland, College Park, MD.

Hjaltason, G. R., and H. Samet. (2003). ''Index-Driven Similarity Search in Metric Spaces.'' *ACM Transactions on Database Systems* 28(4), 517–80.

Moorehead, R., M. Gross, and K. I. Joy, eds. (2002). *Proceedings IEEE Visualization '2002*, Boston, October.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley.

Preparata, F. P., and M. I. Shamos. (1985). *Computational Geometry: An Introduction*. New York: Springer-Verlag.

Rigaux, P., M. Scholl, and A. Voisard. (2002). *Spatial Databases with Application to GIS*. San Francisco: Morgan-Kaufmann.

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

Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley.

Samet, H. (2005). *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA: Morgan-Kaufmann.

Samet, H., H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. (2003). ''Use of the SAND Spatial Browser for Digital Government Applications.'' *Communications of the ACM* 46(1), 63–66.

Schroeder, W., K. Martin, and B. Lorensen. (1996). *Visualization Toolkit*. Englewood Cliffs, NJ: Prentice-Hall (available at http://gd.tuwien.ac.at/visual/vtk/vtk-www/).

Shekhar, S., and S. Chawla. (2003). *Spatial Databases: A Tour*. Englewood Cliffs, NJ: Prentice-Hall.

Shin, H., B. Moon, and S. Lee. (2000). ''Adaptive Multi-Stage Distance Join Processing.'' In *Proceedings of the ACM SIGMOD Conference,* Dallas, *May 2000*, 343–54, edited by W. Chen, J. Naughton, and P. A. Bernstein.

Wang, T. L., and D. Shasha. (1990). ''Query Processing for Distance Metrics.'' In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, Brisbane, Australia, August, 602–13, edited by D. McLeod, R. Sacks-Davis, and H. -J. Schek.

Xia, C., J. Lu, B. C. Ooiand, and J. Hu. (2004). ''Gorder: An Efficient Method for KNN Join Processing.'' In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, Toronto, Canada, September, 756–67, edited by M. A. Nascimento, M. T. Özsu, D. Kossman, R. J. Miller, J. A. Blakely, and K. B. Schiefer.