Adding an Interoperable Server Interface to a Spatial Database: Implementation Experiences with OpenMap™*

Charles B. Cranston[1], Frantisek Brabec[1], Gísli R. Hjaltason[1], Douglas Nebert[2], and Hanan Samet[1]

[1] University of Maryland, College Park, MD 20742, USA,
{zben,brabec,grh,hjs}@cs.umd.edu
[2] U.S. Geological Survey, Reston, VA 22092, ddnebert@usgs.gov

Abstract. Many organizations require geographic data originating from diverse sources in their day-to-day operations. It is often impractical to maintain on-site a complete database, due to issues of ownership, the sheer size of the data, or its dynamic nature. OpenMap™ is a distributed mapping system that allows displaying together geographic data acquired from disparate data sources. In this paper, we report our experiences with building a "specialist" for OpenMap, allowing the OpenMap map browser access to data stored in SAND, a prototype spatial database system. DLG data from the U.S. Geological Survey was used to demonstrate the combined system. Key features of the OpenMap and SAND systems are described, as well as how they deal with the DLG data.

1 Introduction

Geographic data is being digitized at an ever increasing rate. In many cases this is driven by the day-to-day needs of the collecting entities: the public utilities whose paper maps are becoming frayed and unusable, various ecological and scientific entities who use the data in furtherance of their primary missions, or even mundane land-ownership tracking by local governments. In other cases new technologies, such as space shuttle imaging radar, are making digitized spatial information available almost faster than it can be recorded.

However, the Earth is very complex, far too complex to digitize in its entirety. Of necessity, it is *abstractions* of reality that are captured. In this process some details are inevitably lost. Moreover, each digitizing interest community has its own idea of which information it is important to retain. For example, given a river, the ecology community may be interested in the number of small frogs per kilometer of bank, a transportation agency may be more interested in the positions of current and future bridge crossing sites, while the Defense department may be more interested in knowing at what points the river can be forded by an M1 Abrams tank.

Because of these differing views, data digitized by one organization may not be easily shared with other dissimilar organizations, unless some method for interoperation can be devised. Yet such sharing is not only desirable but mandatory, as decision makers simply will not pay for essentially duplicative digitization efforts. Occasions arise when data simply *must* be shared. For example:

> An emergency response team requires the synthesis of a map that includes geological, soil properties, road network, water lines, demographic information, and public service facility locations such as hospitals and schools to be plotted for an urban area just impacted by an earthquake. No single agency is responsible for this variety of geospatial data yet "best-available" information must be assembled and printed for use by field personnel in paper and electronic form within 6 hours [9].

The OpenGIS (Open Geographical Information Systems) Consortium [4] is an open, industry-wide consortium of GIS vendors and users who are attempting to facilitate interoperability by proposing standards for GIS knowledge interchange. The goal of the consortium is to enable transparent interworking within any one community of interest, and to provide a framework for explicit conversion procedures when data is to be shared between differing interest communities. The consortium was founded relatively recently, so the standardization effort is still in its early stages. Nevertheless, a specification of an object model for GIS data has been approved by its members. The specification, called OpenGIS Simple Features, is in three variants,

* This work was supported in part by the U.S. Geological Survey under contract 1434HQ97SA00919, by the National Science Foundation under grant IRI-9712715, and the Department of Energy under Contract DEFG0295ER25237.

each tailored to a specific transport mechanism: ODBC/SQL92 (Open Database Connect/Structure Query Language) [12], Microsoft OLE/COM (Object Linking and Embedding/Component Object Model) [14], and CORBA (Common Object Request Broker Architecture) [13].

OpenMap™ is a product suite developed by BBN Technologies (now a division of GTE Internetworking) in a DARPA-sponsored project to demonstrate CORBA-based mapping. Whereas OpenGIS's Simple Features specification addresses the interface between a GIS database and a GIS application, OpenMap™ specifies an interface between a GIS application and its user interface (UI). OpenMap includes a user interface client, a client/server interface (implemented through CORBA), and a suite of *specialists* that implement the server side of the interface, making a particular kind of data source accessible to the user interface client. Thus, it provides a way to integrate geographical data from diverse data sources in a single map display. OpenMap has been used in technology demonstrations within the OpenGIS community, and its creators have initiated a dialogue concerning the need for an open application/UI interface.

SAND (Spatial And Nonspatial Data) [1] is a prototype spatial database system developed by our group. Its purpose is to be a research vehicle for work in spatial indexing, spatial algorithms, interactive spatial query interfaces, etc. The basic notion of SAND is to extend the traditional relational database paradigm by allowing row attributes to be *spatial* objects (e.g., line segments or polygons), and by allowing spatial indexes (like quad trees) to be built on such attributes, just as traditional indexes (like B-trees) are built on nonspatial attributes.

As part of a demonstration project involving OpenMap™ we built a SAND specialist that makes data stored in SAND accessible to the OpenMap user interface client. In the demonstration, we used a SAND database populated with DLG (Digital Line Graph) data from USGS (U.S. Geological Survey). This paper discusses issues that arose in this work.

The rest of the paper is organized as follows. Section 2 discusses OpenMap™ in more detail. Section 3 describes the SAND database system. Section 4 presents the SAND specialist for OpenMap, while conclusions are drawn in Sect. 5. For the interested reader, we have included in an Appendix a discussion of the issues arising in converting DLG data to SAND's native format.

2 OpenMap™ and OpenGIS

A GIS system can be viewed as being divided into three tiers (see Fig. 1), the UI (User Interface), Application, and Database tiers. In the Database tier we have databases storing actual GIS data, in the Application tier we have applications that query the databases and process the result in some manner, and in the UI tier we have the graphical user interface where the query result is displayed to the user. The OpenGIS Simple Features specification addresses the interface between the Application and Database tiers (as well as between applications). OpenMap, on the other hand, specifies an interface between the UI and Application tiers. This interface is based on CORBA (Common Object Request Broker Architecture) [11], an industry standard middleware layer based on the remote-object-invocation paradigm. By *middleware* we mean shared software layers that support communication between applications, thereby hopefully achieving platform independence. Such a "layering" organizational paradigm has been extremely successful in networked computer communications (for example, FTP over TCP over IP over Ethernet). The recent adoption of the IIOP (Internet Inter-ORB Protocol) standardizes CORBA interoperation down to the TCP/IP protocol layer. Thus *any* two CORBA applications should be able to interwork.

The central component of OpenMap is the OpenMap Browser, its user interface client. It includes a map viewing area, navigation controls, and a layers palette, in addition to menus and a tool bar. A simplified version of the OpenMap Browser was implemented in Java (see Fig. 2), and a variant of it can be deployed on any Java enabled Web browser. The layer palette lists map layers available to the client. A map layer is a collection of related geographic objects, i.e., road network, railroad tracks or country boundaries. The layers come from data servers, termed specialists, that communicate with the OpenMap Browser using CORBA. The interface specification between specialists and the OpenMap Browser allows the Browser to request data objects intersecting a query rectangle, where the data objects are graphical objects of various types, including line segments, circles, rectangles, polylines/polygons, raster images, and text. These can be specified either in lat/long coordinates or in screen coordinates. In addition, the interface provides support for custom palettes that allow the user to configure the specialist, and support for *gestures*, which allow the specialist to respond to mouse actions on the displayed graphics.

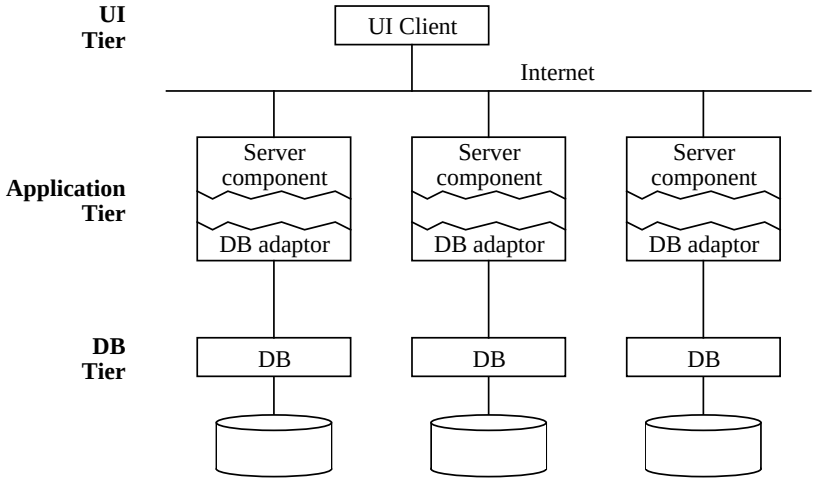

Fig. 1. A three tier view of a distributed GIS system

Specialists can be implemented either in C++ or Java. Among the components of OpenMap are classes for each language (called CorbaSpecialist and Specialist, respective) that encapsulate the common aspects of all specialists [2]. A custom data server can be created by extending these classes, adding only the specialized routines required to access a particular target database. Details of CORBA and session initialization, transfer of query rectangles from client to server, and transfer of GIS feature information from server to client are handled transparently.

3 SAND

SAND [6, 7], the spatial database system developed by our research group, is divided into two main layers, the SAND kernel, and the SAND interpreter. The SAND kernel was built in an object oriented fashion (using C++) and comprises a collection of classes (i.e., object types) and class hierarchies that encapsulate the various components. Since SAND adopts a data model inspired by the relational model, its core functionality is defined by the different types of tables and attributes it supports. Thus, the table and attribute class hierarchies are among the most important. The SAND interpreter provides a low-level procedural query interface to the functionality defined by the SAND kernel. Using the query interface provided by the SAND interpreter, we have built a number of useful tools. For example, the SAND Browser is an interactive spatial query browser, that allows the user to pose queries through graphical input. Also, we have built a prototype for a high-level declarative query interface to SAND, modeled on SQL.

3.1 Table Types

The table abstraction in SAND encapsulates what in conventional databases are known as relations and indexes. 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. All open tables in SAND respond to a common set of operations, such as **first**, **next**, **insert**, and **delete**.

SAND currently defines three table types: relations, linear indexes and spatial indexes. Each table type supports an additional set of operations, specific to its functionality. The function of most of these operations is to alter the order in which tuples are retrieved, i.e., the behavior of **first** and **next**.

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

Linear indexes for non-spatial attributes are implemented using B-trees [5]. Tuples in a linear index are always scanned in an order determined by a total ordering function. Linear indexes support the **find**

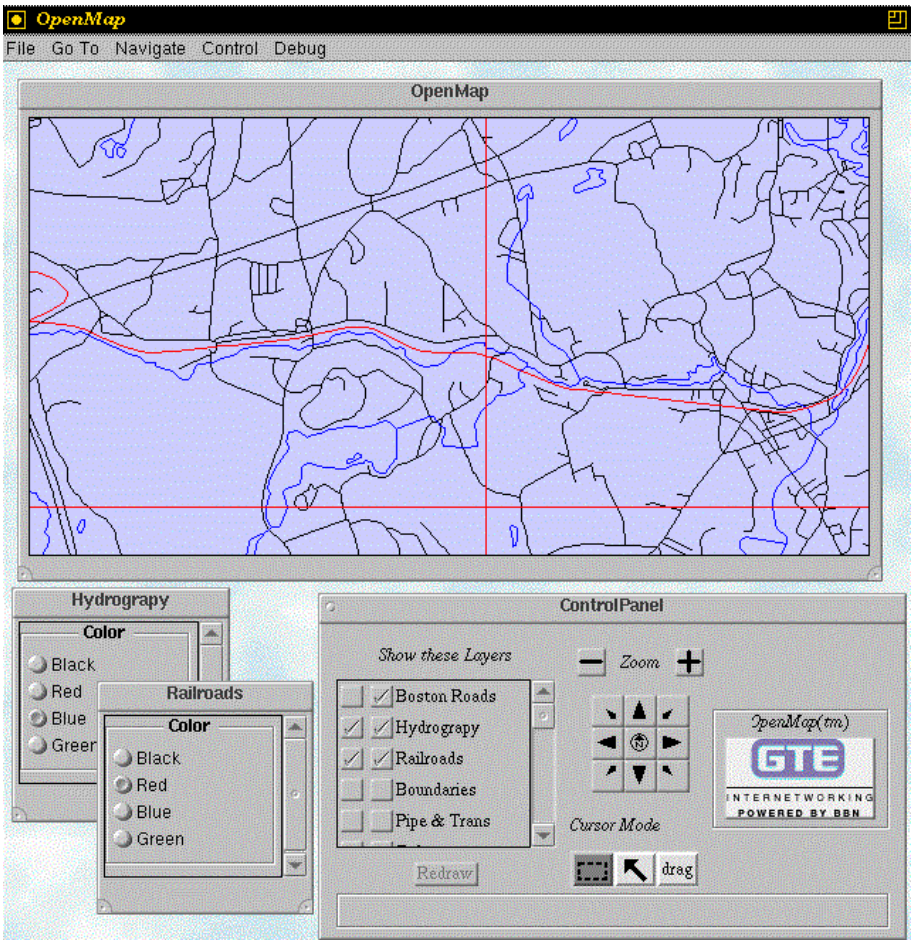

Fig. 2. Client display

operator, which retrieves from the disk storage the tuple that most closely matches a tuple value given as an argument. The **find** operator can also be used to perform range searches.

Spatial indexes are implemented using PMR-quadtrees [10, 16, 17]. They support a variety of spatial search operators, such as **intersect** for searching tuples that intersect a given feature, or **within** for retrieving tuples in the proximity of a given feature. Spatial indexes also support **ranking** [8], a special kind of search operator whereby tuples are retrieved in order of distance from a given feature.

3.2 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 and three-dimensional geometric types (points, line segments, axes-aligned rectangles, polygons and regions). All attribute types support a common set of operations to convert their values to and from text, to copy values between attributes of compatible types, as well as to compare values for equality. Non-spatial attribute types also support the **compare** operator, which is used to establish a total ordering between values of the same type. This is required so that non-spatial attributes can be used as keys in linear indexes. Spatial attribute types support a variety of geometric operations, including **intersect** which tests whether two features intersect, **distance** which returns the Euclidean distance between two features (used for the **ranking** operator), and **bbox** which returns the smallest axis-aligned rectangle that contains a given feature (i.e., its minimum bounding rectangle). Some spatial types support additional operations. For instance, the *region* type supports operations like **expand**, which can be

used to perform morphological operations such as contraction and expansion, and **transform**, which can be used in the computation of set-theoretic operations.

3.3 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 scripting language, *Tcl* [15]. Tcl offers the benefits of an interpreted language but still allows code written in a 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* [15], a toolkit for developing graphical user interfaces.

The SAND interpreter provides commands that mirror all kernel operations mentioned in the previous section. In some cases, a single command may cause more than one kernel operation to be performed. In addition, the interpreter implements data definition facilities. The processing of spatial queries is supported by interpreter commands associated with spatial attributes, spatial indexes and bounding structures.

4 SAND Specialist for OpenMap™

In this section we describe a specialist for OpenMap that provides access to geographic data stored in SAND relations. We implemented this specialist in Java, and thus it is based on the Specialist class provided by BBN. Figure 3 shows the software components of an OpenMap session, where the structure of the SAND specialist is detailed. The user interface client uses CORBA middleware to communicate with various specialists, each of which provides access to a specific type of data source. The SAND specialist code communicates with the UI client with methods inherited from the Specialist class, and in turn invokes the SAND interpreter to perform the actual data access. The SAND specialist responds to requests for objects in a particular map layer intersecting a query rectangle. (In this case, each map layer corresponds to a SAND relation.) In addition, the SAND specialist directs the UI client to display a custom palette for each layer, where the color of the data objects in the layer can be set.

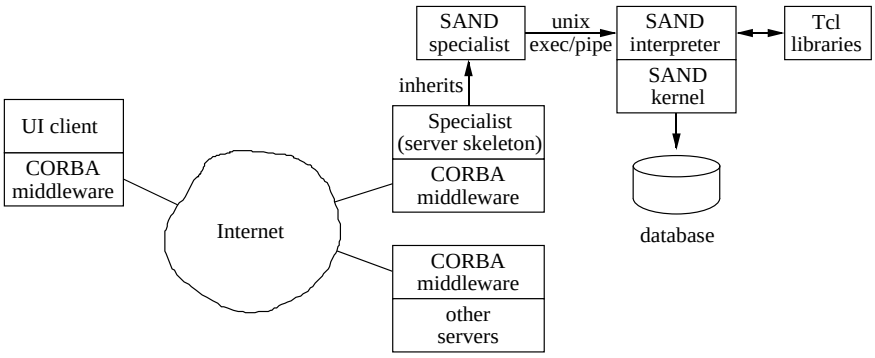

Fig. 3. Structure of the SAND specialist for OpenMap™

The data made available by the SAND specialist in our demonstration was obtained from USGS and is in the form of points, polylines and polygonal areas (see Sect. 5 for details of how this data was imported into SAND). Since the goal of the demonstration was to show how multiple maps from diverse sources could be overlayed on top of each other, it was undesirable to display filled polygons as they obliterate any map features in lower layers. Thus, we chose to focus on polylines and polygon boundaries, stored as line segments in the SAND relations representing each map layer. (An alternative approach would have been to represent polygonal areas with the SAND polygon attribute type, and convert from polygons to polylines or line segments in the server at run time.) A spatial index was built on the line segment attribute in the map layer relations in order to allow efficient spatial lookup.

4.1 Implementation of the SAND Specialist

The SAND specialist communicates with the SAND interpreter by passing it Tcl scripts that implement specific database queries using the low-level SAND query interface, and receiving back textual output through an I/O pipe. GIS features satisfying the query are translated into OpenMap Specialist objects, which are then passed on to inherited Specialist methods for transport to the client display. The Tcl script that the SAND specialist passes to the SAND interpreter selects a database, opens a spatial index, and then executes a query passing a rectangle as an argument (the only query currently specified in the OpenMap specialist interface is such a rectangle intersection query):

```
sand cd <directory>;
set index [sand open <indexname>];
$index first -intersect \
       {rectangle <left> <bottom> <width> <height>};

while {[$index status]} {
  puts [$index get];
  $index next;
}
$index close
```

The `<directory>` argument to the `sand cd` command specifies the file system directory that holds the database. The `sand open` command returns a handle to an open table (see Sect. 3.1), in this case an index on a spatial attribute. The handle, which corresponds to an underlying C++ spatial index object, can subsequently be used to perform actions on the index. The script initiates a spatial window query by invoking the table command `first` with a query rectangle, which loads the first tuple satisfying the query (if any exists), i.e., a tuple corresponding to a line segment that is intersected by the given rectangle. The table command `get` returns the contents of the current index tuple, in this case storing a line segment. The table command `next` loads the next tuple satisfying the query, or sets the table status to false if none exists. In fact, not only does the SAND kernel return the tuples satisfying the query one-by-one (through the SAND interpreter), but actually executes the query incrementally rather than batch style. The `while` loop outputs the line segment (which is received by the SAND specialist), and fetches another one until all line segments intersecting the query rectangle have been exhausted. At that point, the index file is closed.

This query plan, which makes use of a spatial index, is more efficient than the straightforward one of initiating a sequential scan of a relation, then testing each line segment for intersection with the query rectangle and only outputting those segments that actually intersect it. However, as it turned out in our experiment, the time saved was actually drowned out by communication costs. (We will discuss this at greater length in Sect. 5.)

The line segments as returned over the I/O pipe are of the form:

```
{line <x1> <y1> <x2> <y2>}
```

The SAND specialist parses this format, creates two OpenMap Specialist points, creates a Specialist line between the two points (using either black or a color determined by a control subpanel), and adds the line to the display list for return to the client.

Each coordinate value undergoes four data conversions. The data in the index file is in binary floating format. The SAND interpreter converts this to an ASCII string representation (conversion 1) to return it over the I/O pipe to the SAND specialist. The SAND specialist reads the ASCII string representation from the pipe and converts it back to binary floating (conversion 2). It must do this because Specialist's Point object creator takes binary floating arguments. Presumably Specialist must convert this machine-specific binary floating value into some machine independent wire format (conversion 3). Finally, the display client must convert from the wire format to some display-device specific format (conversion 4) for eventual display.

The Java source file for SAND specialist contains 275 lines of code. Of this, roughly 200 implement the basic server and another 50 implement the palette for control the color of the line segments in a SAND specialist map layer. The rest are comments and housekeeping "import" statements.

4.2 Sample OpenMap™ Session with the SAND Specialist

Figure 2 shows the OpenMap UI client displaying three map layers obtained from the SAND specialist, Hydrography (i.e., rivers, lakes, etc.), Boston Roads, and Railroads. The window on the lower right is a control panel, where the user can select the layers to display as well as pan and zoom on the map display. The layer list on the left side of the control panel has two check boxes for each available layer. The right-hand one selects the layer for display, whereas the left-hand one causes the creation of a custom palette specific to the layer. The Hydrography and Railroads layers each have a palette that allows setting the color of their line segments (the two windows on the lower left). The color for the Hydrography layer is set to blue, and the color for the Railroads layer is set to red. The color of the Roads layer is set to the default color, black, since it does not have a palette associated with it. When the user clicks on the "Redraw" button, a query is sent to each selected server for any geographic objects visible in the display area.

5 Concluding Remarks

Our research agenda includes developing efficient ways to access spatial data using advanced indexing structures and thus one of our goals for the demonstration software was to showcase the speed advantage these make possible compared to the simple sequential scan paradigm commonly used in industry. This desire caused us to prematurely optimize our software, which became a problem during development. For example, we spent too much time worrying about cross-language linkage delay, the delay caused by using an I/O pipe to communicate with the SAND interpreter (as opposed to assembling all the code into one binary), the delay caused by our need to translate each data point between binary floating point and string representation four times, etc.

When the demonstration was finally assembled, we found that the speed advantage gained from our indexed access was completely swamped by the delays in other software components and by the communication cost. In addition, there is an inherent conflict between the incremental nature of our software and the batch nature of other software components in the demo. SAND was designed to be incremental; data are typically released from the query as soon as they become available. On the other hand, the Specialist class (which implements the CORBA communications) seemed to gather the complete data set before transmitting any data to the client, and in fact seemed to do significant processing before doing so. As an indication of this difference, for a typical query result size of a few thousand line segments, SAND would start returning data within five seconds and complete the query within 30 seconds. However, the Specialist class might then spend over two minutes marshalling the results before the client could show the user any feedback to the user's query.

The design decisions we faced in implementing the SAND Specialist were largely those common to most software development projects. We had to decide on an implementation language (Java or C++; we chose Java), and how our code would couple to existing systems (i.e., loosely or tightly coupled; we chose a loosely coupled approach). We needed to develop a data philosophy that would support the desired results, and to make that philosophy work with the data we could easily obtain. In retrospect, many of the problems we faced were due to premature optimization and a failure to test error paths early enough (i.e., errors occuring in various parts of the system weren't apparent, the only visible result being that no data was displayed).

The work described here can be extended in a number of ways. Some directions for future work include the following:

- Currently the SAND specialist invokes the SAND interpreter separately for each query, thereby incurring the overhead of opening and closing an I/O pipe. In order to allow the I/O pipe between the SAND specialist and the SAND interpreter to remain open between queries, we need to invent some explicit synchronization scheme.
- At this time, the non-spatial attributes of the SAND relations are not used by the SAND specialist. We plan to extend the SAND specialist to make use of these attributes by modifying the display of the spatial features. This could be as simple as feature coloring, or as complicated as automatic legend placement.
- The OpenMap specialist interface supports gesturing thereby allowing the specialist to act on user interface actions. A simple example might be to display (in textual form) the non-spatial attributes of the database tuple closest to a mouse click.

- Extensions to deal with areal (i.e., polygon) data such as choropleths.
- Multi-resolution display: a road drawn as a line segment on a low-resolution display might be better represented at higher resolutions as a long thin rectangle or polygon. An intelligent specialist could make this translation.
- The addition of other interoperable interfaces to SAND, e.g., the OpenGIS SimpleFeatures interface. We expect that most of the lessons learned in implementing the OpenMap™ server will be directly transferrable to work on implementing other interfaces. One such lesson is that one should take a careful look at network communication costs before expending too much effort in optimizing the local query execution of the database server.

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 — Second Symposium, SSD'91*, pages 299–318, Zurich, Switzerland, August 1991. (Also Springer-Verlag Lecture Notes in Computer Science 525).

[2] BBN Corporation. *Designing CORBA(Orbix/VisiBroker) Specialists for BBN's OpenMap*, 1997. Available as `http://javamap.bbn.com/projects/matt/development/specialist.html` on the web.

[3] K. J. Boyko, M. A. Domaratz, R. G. Fegeas, H. J. Rossmeissl, and E. L. Usery. An enhanced digital line graph design. U. S. Geological Survey Circular 1048, 1990. (Also see `http://edcwww.cr.usgs.gov/glis/hyper/guide/usgs_dlg`).

[4] K. Buehler and L McKee, editors. *The OpenGIS Guide — Introduction to Interoperable Geo-Processing*, Wayland, MA, 1996. OpenGIS Consortium. OGIS TC Document 96-001, available as `http://www.opengis.org/guide` on the web.

[5] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[6] C. Esperança and H. Samet. Spatial database programming using SAND. In M. J. Kraak and M. Molenaar, editors, *Proceedings of the Seventh International Symposium on Spatial Data Handling*, volume 2, pages A29–A42, Delft, The Netherlands, August 1996. International Geographical Union Comission on Geographic Information Systems, Association for Geographical Information.

[7] C. Esperança and H. Samet. An overview of the SAND spatial database system. *Communications of the ACM*, to appear.

[8] 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).

[9] Doug Nebert. WWW mapping in a distributed environment: Scenario of visualizing mixed remote data, 1997. Available as `http://www.fgdc.gov/publications/documents/clearinghouse/wwwmap_scenario.html` on the web.

[10] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. (Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986).

[11] Object Management Group. *CORBA 2.0/IIOP Specification*, 1997. OMG formal document 97-09-01, available as `http://www.omg.org/corba/c2indx.htm` on the web.

[12] Open GIS Consortium, Inc. *Open GIS Simple Features Specification for SQL Revision 1.0*, March 1998. Available as `http://www.opengis.org/public/sfr1/sfsql_rev_1_0.pdf` on the web.

[13] Open GIS Consortium, Inc. *OpenGIS Simple Features Specification for CORBA Revision 1.0*, March 1998. Available as `http://www.opengis.org/public/sfr1/sfcorba_rev_1_0.pdf` on the web.

[14] Open GIS Consortium, Inc. *OpenGIS Simple Features Specification for OLE/COM Revision 1.0*, March 1998. Available as `http://www.opengis.org/public/sfr1/sfcom_rev_1_0.pdf` on the web.

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

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

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

Appendix: Data Conversion – DLG to SAND

As the source for data in our demonstration we used data sets from USGS (U.S. Geological Survey), encoded in the DLG (Digital Line Graph) format [3]. In this section, we briefly describe the DLG format, and present issues that arose in the conversion of DLG data to a format readable by SAND.

Prior to describing the DLG format we point out that a DLG map of the same geographic area is divided into several layers. Each layer is represented in a separate DLG file:

- Hydrography: flowing water, standing water, and wetlands.
- Roads and trails.
- Railroads.
- Pipelines, transmission lines, and miscellaneous transportation.
- Hypsography: contours and supplementary spot elevations.
- Boundaries: state, county, city, and other national and state lands such as forests and parks.
- Public Land Survey System including town-ship, range, and section information.

Typically each of these layers is displayed in a different color in order for it to be easier to tell them apart. Since the DLG files are distributed in datasets covering a rather small region (or .5° square for 1:100,000-scale DLG maps), we merged several datasets together to represent a larger area.

The DLG format encodes information about geographic features, in the form of points, polylines and polygonal areas, together with associated non-spatial information. It has primarily been used to encode printed cartographic maps into digital form. As the name suggests, it is assumed that the map being represented forms a planar graph. Thus, DLG files are composed of node, line and area identifier elements. A single node is defined by its coordinates and may mark the start or the end of one or more lines or a point feature. Therefore nodes occur where lines intersect and at places on linear features where they are subdivided into separate line segments. A line (corresponding to an edge in the graph) is defined as sequences of line segments, with a node anchored at each end. Lines do not cross over themselves or any other lines in the map. An area is a contiguous region of a map bounded by lines. It is defined by a sequence of line references.

Along with their spatial information, nodes, lines and areas can carry feature codes[1]. These are numerical codes, used to describe the physical and cultural characteristics of the corresponding geographic features. For example, a feature code for an area might identify it to be a lake or swamp, and a feature code for a line might identify a road, railroad, stream, or shoreline. Features in DLG can have any number of feature codes.

In order to make the data stored in DLG files usable by our SAND system, we had to convert the DLG data to SAND's native format, which consists of a set of relations. Each relation in a SAND database contains an arbitrary, but fixed, number of attributes. Usually, only one of these attributes is of a spatial attribute type, but this is not a requirement. For our purposes we decided to focus on the line data provided by the DLG files. Nodes are mostly used to define line segments, and the spatial objects represented by a node type (e.g., wells, tunnel portals) were found to be of limited interest. The areas stored in DLG files are defined as a sequence of lines so each area was exported to the SAND database as such, but provisions were made to indicate that a set of lines originally defined a single area.

Although not currently used in the SAND specialist, we represented the feature codes of each feature in non-spatial attributes in the corresponding relation. We faced the dilemma that in DLG, a feature can have any number of feature codes, whereas the number of attributes in tuples of a given relation is fixed. To solve this, we divided the set of feature codes into three classes, primary, secondary, and the rest. Primary and secondary feature codes are meant to represent the most important characteristics of a feature. They are chosen in such a way as to make it very likely that there will be at most one primary and one secondary feature code for each feature. For example, if the feature is a road, the primary feature code would specify the type of the road (primary route, trail, footbridge, etc.), the secondary feature code would provide additional information (number of lanes, interstate route number, county route etc.) and all other feature codes would be stored together in a third attribute of the relation tuple storing the feature (in tunnel, on bridge, private etc.). Unfortunately, this strategy sometimes fails, i.e., a feature can have more than one primary or secondary feature code; for instance, a certain road segment could be part of several different interstate routes. In this case, only one of the feature codes is stored as the primary or secondary one.

Another issue that had to be resolved was the conversion of coordinates used to define the locations of spatial features. In our demonstration we used DLG files digitized from maps of scale 1:100,000; such DLG files define the location of objects with respect to the Universal Transverse Mercator (UTM) Projection, which is a map projection that preserves angular relationships and scale. However, OpenMap (and some

[1] The usual term for the codes is "attributes". However, since they are a very different concept from "attributes" as used in SAND (to mean fields in tables), we use the alternative term "feature codes" to avoid confusion.

of the spatial functions in SAND) assumes that spatial objects are specified using latitude and longitude coordinates. Thus we had to convert each DLG coordinate pair into latitude/longitude coordinates.