ABSTRACT

Title of dissertation: OPTIMIZING CLIENT-SERVER COMMUNICATION FOR REMOTE SPATIAL DATABASE ACCESS

František Brabec, Doctor of Philosophy, 2005

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

Technological advances in recent years have opened ways for easier creation of spatial data. Every day, vast amounts of data are collected by both governmental institutions (e.g., USGS, NASA) and commercial entities (e.g., IKONOS). This process is driven by increased popularity and affordability across the whole spectrum of collection methods, ranging from personal GPS units to satellite systems. Many collection methods such as satellite systems produce data in raster format. Often, such raster data is analyzed by the researchers directly, while at other times such data is used to produce the final dataset in vector format. With the rapidly increasing supply of data, more applications for this data are being developed that are of interest to a wider consumer base. The increasing popularity of spatial data viewers and query tools with end users introduces a requirement for methods to allow these basic users to access this data for viewing and querying instantly and without much effort. In

our work, we focus on providing remote access to vector-based spatial data, rather than raster data.

We explore new ways of allowing visualization of both spatial and non-spatial data stored in a central server database on a simple client connected to this server by possibly a slow and unreliable connection. We considered usage scenarios where transferring the whole database for processing on the client was not feasible. This is due to the large volume of data stored on the server as well as a lack of computing power on the client and a slow link between the two. We focus on finding an optimal way of distributing work between the server, clients, and possibly other entities introduced into the model for query evaluation and data management. We address issues of scalability for clients that have only limited access to system resources (e.g., a Java applet). Methods to allow these clients to provide an interactive user interface, even for databases of arbitrary size, are also examined.

# OPTIMIZING CLIENT-SERVER COMMUNICATION FOR REMOTE SPATIAL DATABASE ACCESS

by

František Brabec

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Page left intentionally blank

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Technological advances in recent years have opened ways for easier creation of spatial data. Every day, vast amounts of data are collected by both governmental institutions (e.g., USGS, NASA) and commercial entities (e.g., IKONOS). This process is driven by increased popularity and affordability across the whole spectrum of collection methods, ranging from personal GPS units to satellite systems. Many collection methods such as satellite systems produce data in raster format. Often, such raster data is analyzed by the researchers directly, while at other times such data is used to produce the final dataset in vector format. With the rapidly increasing supply of data, more applications for this data are being developed that are of interest to a wider consumer base. The increasing popularity of spatial data viewers and query tools with end users introduces a requirement for methods to allow these basic users to access this data for viewing and querying instantly and without much effort. In our work, we focus on providing remote access to vector-based spatial data, rather than raster data.

Traditionally, common spatial databases and Geographic Information Systems (GIS) such as ESRI's ArcInfo are designed to be stand-alone products. The spatial database is kept on the same computer or local area network from which it is visualized and queried. This hardware setup allows for instantaneous transfer of large amounts of data between the spatial database and the visualization module so that it is perfectly feasible to use large-bandwidth protocols for communication between these two. There are, however, many applications where a more distributed approach is desirable. In these cases, the database is maintained in one location, while users need to work with it from possibly distant places over the network (e.g., the public internet). These connections can be far slower and less reliable than local area networks, and thus it is desirable to limit the data flow between the database (server) and the visualization unit (client) in order to get timely response from the system.

Another approach has been adopted by numerous web-based mapping service vendors (MapQuest [23], MapsOnUs [24], etc.) that face the same challenge. Their goal is to enable remote users, typically only equipped with standard web browsers, to access the vendor's spatial database server and retrieve information (in the form of maps) from them. The solution presented by most of these vendors is based on performing all the calculations on the server side, and then transferring only bitmaps that represent results of user queries and commands. Although the advantage of this solution is one of requiring minimal hardware and software resources on the client site, the resulting product has severe limitations in terms of available functionality and response time (each user action results in a new bitmap being transferred to the client).

In our research, we explore new ways of allowing visualization of both spatial and nonspatial data stored in a central server database on a simple client connected to this server by possibly a slow and unreliable connection. We develop a new client-server approach as an answer to some of these drawbacks of traditional solutions. Our system aims to partitions the workload between the client and the server in such a manner that the user's experience with the system is interactive, with minimal delay between the user action and appropriate response. The design works around potential bottlenecks for the information transfer such as the limited network bandwidth or resources available on the client computer. To support multiple concurrent clients, limited resources on the server must also be considered.

Our solution is especially appropriate for usage scenarios where transferring the whole database for processing on the client is not feasible. This is typically due to the large volume of data involved, the lack of computing power on the client, and a slow link between the two. We focus on finding an optimal way of distributing work between the server, clients, and possibly other entities introduced into the model for query evaluation and data management. We address issues of scalability for clients that have only limited access to system resources (e.g., a Java applet). Methods to allow these clients to provide an interactive user interface, even for databases of arbitrary size, are also examined.

The rest of the thesis is organized as follows. In the remaining part of this chapter, we introduce existing spatially-enabled DBMSs and discuss their advantages and drawbacks. In Chapter 2, we provide a review of existing methods for remotely accessing spatial databases. In Chapter 3, we introduce the basic concepts of the client-server paradigm, and discuss its application to remote spatial operations. In Chapter 4, we describe our initial research efforts in this area and outline lessons learned for our further research. Chapter 5 introduces our client, the SAND Internet Browser. It describes its functionality and the user interface layout, as well as providing several examples of how to evaluate queries using it. In Chapter 6, we discuss our architecture based on pure client-server approach. Given a client that communicates directly to a server, we examine different deployment options and describe several methods that improve the performance that can be achieved in this environment. Chapter 7 extends the basic client-server approach by adding auxiliary servers. Such servers can be used as temporary data storage between the client and the server. We present typical deployment scenarios when this would be beneficial, as well as present methods for using this arrangement to further speed up its performance. In Chapter 8, we combine all the different design options and speed-up methods together and discuss how to choose the optimal deployment method for given specific usage scenarios. In Chapter 9, we discuss several related issues that we needed to address in order to be able to build a system useful in practice. Chapter 10 discusses our experience with spatial data and algorithm visualization and its application in geographic information system applications. Finally, in Chapter 11 we draw some conclusions and propose topics for further research.

## 1.1 Spatial DB Servers

### 1.1.1 MySQL

MySQL [10] is a fast, multi-threaded, multi-user SQL database server. MySQL implements spatial extensions following the specification of the Open GIS Consortium (OGC), in particular the OpenGIS Simple Features Specifications For SQL [22]. MySQL implements a subset of the SQL with Geometry Types environment proposed by OGC. The support for spatial attributes in MySQL is recent and so far only basic. MySQL is available under either GPL or commercial licenses.

### 1.1.2 PostgreSQL/PostGIS

PostgreSQL [26] is an open source object-relational database management system based on code originally developed at the University of California at Berkeley. It supports SQL92 [28] and SQL99 [29] and offers many modern features such as foreign keys, triggers, views and transactional integrity. PostgreSQL can also be extended by the user in many ways, for example, by adding new data types, functions, operators, etc. PostgreSQL can be used, modified, and distributed by everyone free of charge for any purpose.

PostgreSQL includes support for the two-dimensional spatial data types point, line, line segment, box, open and closed path, polygon and circle. It also implements various operations such as overlap and distance. However, these are not available on all combinations of data types. The spatial support in PostgreSQL is more mature and its features are implemented more thoroughly and completely than in case of MySQL.

PostGIS [16] is a GIS built on top of PostgreSQL, it adds support for geographic objects and effect spatially enables the PostgreSQL server. PostGIS is released under the GNU General Public License.

### 1.1.3 Oracle

Oracle facilitates support for spatial data through its Oracle Spatial [70] module. Oracle Spatial provides a way to store and retrieve multi-dimensional data in Oracle. It is primarily used for GIS to implement geo-reference and solve spatial queries.

Oracle Spatial stores individual features (point, line or polygon) in a single field within a table. A single Helical Hyperspatial code (HHCode[1]) [56] is used to store the Euclidean spatial dimensions and additional data dimensional include depth, elevation, or time. The types of multidimensional data are restricted only in that they

[1]HHCodes are in effect Peano-Hilbert [41] codes.

must be a numeric data type and have a bounded range. The HHCode is generated through the recursive decomposition of space of known dimensionality. Attribute data for specific multidimensional data is stored within columns of a table in the database. Access to the data for processing and manipulation is accomplished through extensions to Oracle PL/SQL. Oracle Spatial is a commercial product.

### 1.1.4 Informix

IBM Informix Spatial DataBlade [5] module extends the IBM Informix Dynamic Server (IDS) to support location-based data. It provides access to the SQL-based spatial data types and functions both through standard SQL queries or with client-side GIS software. Spatial data is indexed using a built-in R-tree multidimensional index to maximize spatial query performance. Informix is a commercial product.

### 1.1.5 ArcGIS

ESRI's ArcGIS [25] is an integrated family of products that allow deployment of GIS system on a range of platforms — desktops, servers, embedded and mobile devices. ArcGIS products allow integration with existing databases and display their records on maps for presentation and analysis purposes. ArcGIS supports implementation of the geospatial data model through a collection of files in a file system (for smaller projects) or through accessing an RDBMS (such as DB2, Oracle or Informix). While ArcGIS is not a core spatial DBMS, it can be used to aggregate data from other sources and to serve as a data source itself.

### 1.1.6 MapInfo SpatialWare

MapInfo SpatialWare [7] data management software enhances existing 3rd party RDBMSs (including DB2, Informix and SQL Server) to manipulate spatial data. SpatialWare also utilizes an R-Tree to index the spatial data. Implemented on top of one of the supported standard RDBMs, SpatialWare can also serve as a spatial data server.

### 1.1.7 SAND — Spatial And Non-Spatial Database

Spatial And Non-spatial Database (SAND) [44] was originally developed by Claudio Esperanca at the University of Maryland but many other people contributed to its extensions and improvements since then. While we use this particular database engine as our central spatial server in our specific implementation for providing the database management and query evaluation, the client-server approach for distributed spatial databases that we have developed does not depend on any specific software used and can be adapted to others, commercial or non-commercial alike.

The SAND kernel is based on extended relational data model. The data in SAND is organized into a collection of tables, each table consists of a set of tuples (records), where each tuple is a set of simple data types (attributes). There are three table types in SAND: relation, linear index and spatial index. Each of these tables supports operations that are appropriate to its function in the database environment. Most of the operators are used to alter the order in which tuples are retrieved.

Relations only support random access to individual tuples. In case of linear indexes, the 'find' operator allows performing searches for a tuple that is closest to a tuple value given in the argument. Range queries can be performed by using the find operator for finding a starting point and a stop condition for a linear traversal. Linear indexes are implemented as B-trees [42]. Spatial indices support several spatial search operators such as 'overlap' (for searching tuples that intersect a given feature) and 'within' (for searching tuples within certain distance from a given feature). Tuples can also be ranked, in which case they are retrieved in the order of distance from a given feature. Numerous spatial index types were implemented in SAND for the purpose of experimental evaluation and comparison. The default spatial index type used is the PMR quadtree [65, 66].

Most common non-spatial types are supported by SAND (integers, floating-point numbers, fixed and variable-length strings). Various two and three-dimensional geometric types (points, line segments, axes-aligned rectangles, polygons and regions) with both Euclidean and spherical distance metrics are also available[2]. The usual operators are supported for the non-spatial attributes, which includes the 'compare' operator that establishes total ordering among attribute values of the same type. For

[2]The Euclidean distance metric is used for small area maps. The spherical metric is utilized for maps that cover whole or major part of the world and when Euclidean calculations could be too distorted.

the spatial types, operators such as 'distance' (calculates distance between two spatial attributes), 'intersect' (determines whether there is any intersection between two spatial attributes) and 'bbox' (calculates the smallest enclosing axis-aligned rectangle for a given feature) are available.

The application programming interface (API) for the SAND kernel is derived from and embedded in the Tcl [19] language. This connects the high-level compiled code of the kernel with a front end based on an interpreted programming language. Most aspects of SAND can be controlled and programmed through this SAND interpreter.

# Chapter 2

# Spatial Server Access

## 2.1 Integrated Solutions

Some spatial database solutions, especially those intended for browsing and querying relatively small data sets, are created as standalone products with the database engine and the visualization unit bundled together. This allows a user and his or her computer to be completely independent in running the application. The obvious drawbacks include the limitations regarding the data volume that can be presented this way as well as complications resulting from keeping the locally stored data set up-to-date.

### 2.1.1 ArcGIS Desktop

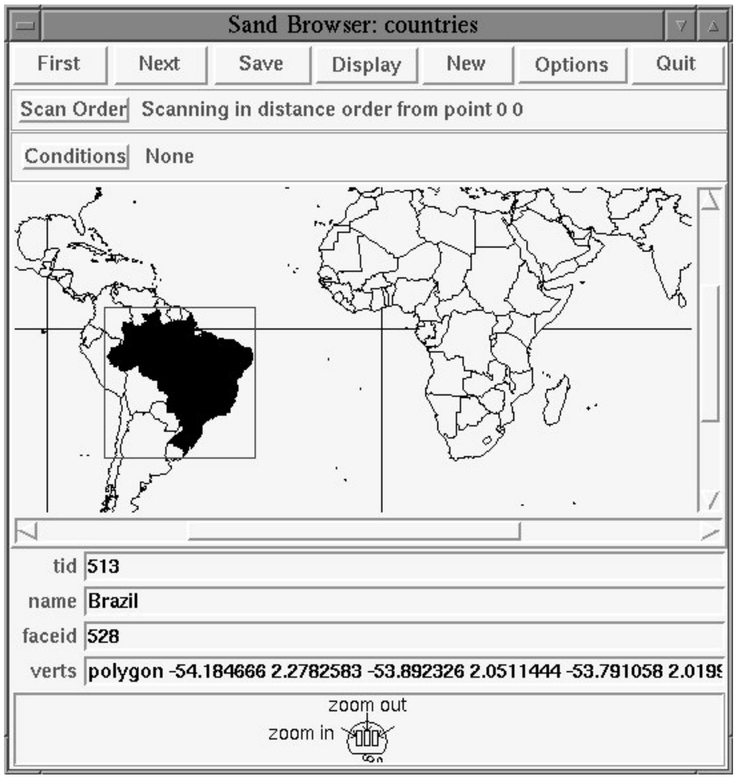
ArcView [1] — now part of the ArcGIS Desktop suite — was one of the first and remains the most popular representatives of this category. It provides extensive mapping, data use, and analysis along with simple editing and geoprocessing capabilities. Its design allows for usage of extensions, implemented by both ESRI and third party developers.

### 2.1.2 MapInfo Professional

MapInfo Professional [7] is a business mapping solution allowing users to perform data analysis and visualization. Similar to ArcView, it supports extensive data import and export and allows users to develop their own custom extensions.

### 2.1.3 Sand Browser — The Original SAND Visualization Tool

The SAND Browser is a tool designed to provide a graphical user interface to the facilities of SAND developed at the University of Maryland, originally by Claudio Esperanca [44] with further enhancements added by Gisli Hjaltason [53, 54]. It facilitates visualization of the SAND data by letting the user to specify several search criteria: the scan order in which tuples are to be incrementally retrieved, a spatial selection (overlap and within constraints) and an arbitrary selection predicate. The appearance of SAND Browser is illustrated in Figure 2.1.



Scan Order Scanning in distance order from point 0 0

Conditions None

tid 513

name Brazil

faceid 528

verts polygon -54.184666 2.2782583 -53.892326 2.0511444 -53.791058 2.019

Figure 2.1: Snapshot of a SAND Browser window. The relation being browsed contains data describing world political divisions. The current tuple corresponds to the main land mass of Brazil.

The SAND Browser user interface is divided into vertically stacked panels. These are (from top to bottom):

- The *command area*, containing buttons for several actions. The key functionality is provided through the **First** and **Next** buttons that retrieve the first and next tuple satisfying the currently set scan order and predicate respectively; **Display** facilitates operations on the graphical display, such as clearing it or selecting the color for drawing spatial features or displaying all the query matches in one step.

- The *scan order* button triggers several "pop-up" dialog boxes which allow the user to specify parameters to be used when the relation is being scanned with the help of one of the indices defined for it.

- The *conditions* field is used to enter a selection predicate. The expression must follow the syntactic rules of the Tcl language.

- The *graphical display* panel is the drawing area where spatial features are input and output. At any given moment of the interaction, the current value of the relation's spatial attribute is displayed in this area and highlighted by a blinking rectangle. When defining a query, spatial features are input by drawing on the graphical display. The display can also be panned and zoomed in and out.

- The *tuple display* panel contains a series of labeled entry boxes, one for each attribute in the schema of the relation. These are updated to reflect the value of the current tuple.

The browser supports selection and spatial join and semijoin queries, and their results are processed (and returned) in an incremental manner. This means that the user gets some visual feedback quickly. Additionally, receiving the results in some specific order is often a part of what the user is interested in. Returning objects in the order of distance from another object or a set of objects is an example of a fairly typical query.

## 2.2 Remote Access Tools

Accessing spatial databases is not the only area where the user interface may need to be hosted on a system different from the one that executes the operations resulting from the user's input. In some scenarios, the solution needs to be very general to support many different remote applications. In other scenarios, the remote access method needs to support a very specialized remote applications and the access-enabling technology can be developed and optimized for this specific usage.

### 2.2.1 Generic Remote Access Tools

Several software vendors have come to the market with products that let users utilize a remote machine in a such a fashion that their experience is similar to using the computer locally (Figure 2.2). While some operating systems come with this kind of feature built in (e.g., terminals in Unix), others (such as Microsoft Windows) were originally lacking this capability, and only third party applications or recent add-ons assured that these platforms would provide this functionality as well. Although these remote access tools were designed for general purpose remote access, they must solve problems similar to those present in the remote access of a spatial database. Notice that the task is similar in both cases. First, the system needs to report to the user the visual representation of the computer's activities in the form of a bitmap on the computer screen. Second, the user's input such as keystrokes, mouse movements and clicks need to be sent back to the remote computer. Since the networking connection could be slow and/or with significant latency, it is desirable to use a protocol that would minimize network utilization in order to maximize system performance and usability.

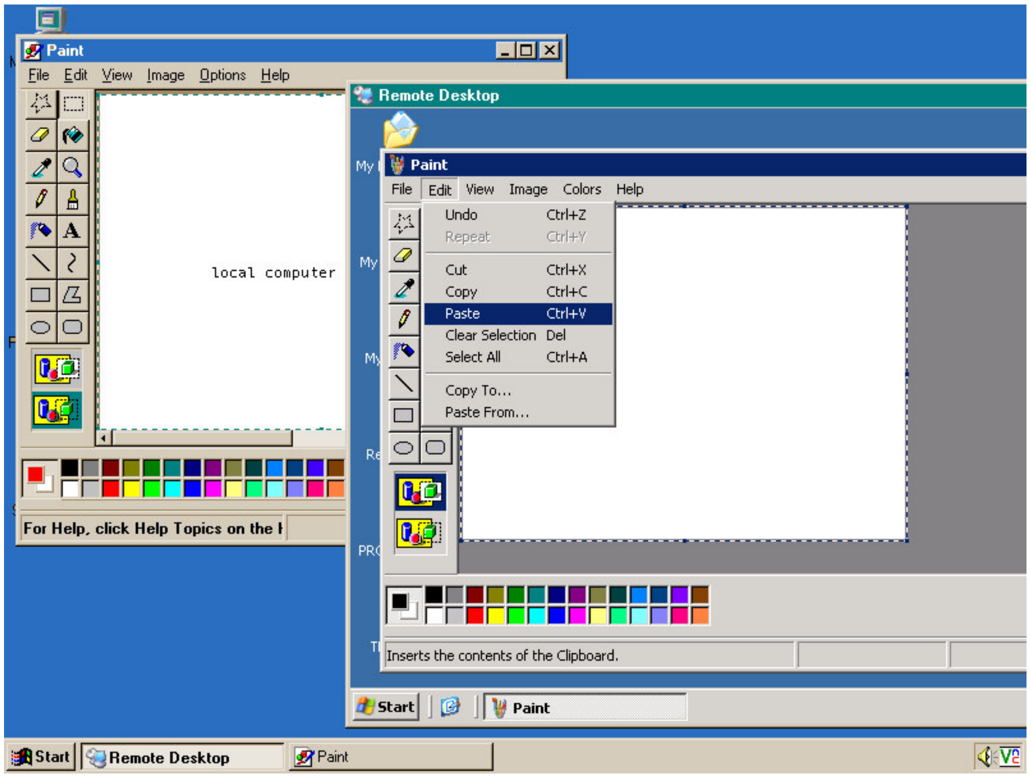


Figure 2.2: Remote Access — The user runs a sample application called "Paint" locally (left window) as well as on a remote computer (right inner window labeled "Paint" in its title bar). The remote computer's desktop is showing in the local window (right outer window labeled "Remote Desktop" in its title bar). The application's appearance is the same in both cases.

While the general principle of all the above mentioned systems is the same — to transfer screen bitmaps to the user and keystrokes and mouse events to the computer — the overall performance, and, ultimately the usability, of these products depends on how efficiently each system utilizes its available network resources. If no attempt is made to optimize the data transfer, a new screen is transferred to the client after every operation. A screen bitmap can occupy anywhere between tens of kilobytes to megabytes. Even if usual static bitmap compression methods (e.g., LZW [85] used in the popular GIF format) are employed, the data throughput cannot be decreased so that the screen updates are instantaneous and almost invisible to the user. White this may be acceptable for certain types of computer usage, generally faster updates are required for optimal user experience. For instance, smooth animation typically needs to refresh the screen in the order of tens of times per second. Obviously, if downloading a new bitmap takes a few seconds, the desired behavior is very far from achievable in reality.

There is no need to transfer the whole bitmap after every change though. Notice that in many cases, the new bitmap may remain identical in some areas. For instance, after a panning operation in a window, most of the old bitmap can be reused and only a small stripe at one of the edges of the window has to be newly rendered. The predominant similarities between the old and new bitmap may be different from application to application. While panning and zooming may be common operations in browsing a geographic database, these operations may not be used as often by a normal computer user. Therefore, even though some of the remote access packages already have powerful mechanisms that decrease the network utilization significantly, focusing specifically on support for spatial database browsing can allow for even better performance.

### 2.2.2 Web-Based Mapping Services

The above mentioned approach where mere bitmaps are sent to the client for viewing was employed by many vendors that provide access to maps over the web. The typical example of providers of these services are vendors such as MapQuest [23] and Switchboard/MapsOnUs [24] for street maps based on addresses; or TopoZone [20] for topographical maps. Their approach is simple, the server receives a location description (e.g., a street address, name of a place, etc), it queries its spatial database, retrieves a map, converts it into a bitmap image and sends it back to the user (their browser). The map retrieved from the spatial database may be in vector (MapQuest, MapsOnUs) or raster (TopoZone) format. In either case, it gets rasterized or subsampled respectively before sending the data over the network to user's browser (Figure 2.3).

This approach requires very little support from the client site, typically just a web-browser equipped computer or network appliance. The drawback of this solution is that it quickly reaches its usability limitations when more serious work is attempted. Such poorly supported operations include even basic zooming in or out or panning not to mention running queries. In particular, actions such as zooming or panning are very cumbersome with performance bordering unacceptable for many users as the response time is determined by the amount of data that needs to be transferred every time a new view is requested. Other operations such as querying the database beyond displaying all objects within a certain rectangle are not supported at all.

This behavior is a direct result of a trade-off faced by designers of these systems.

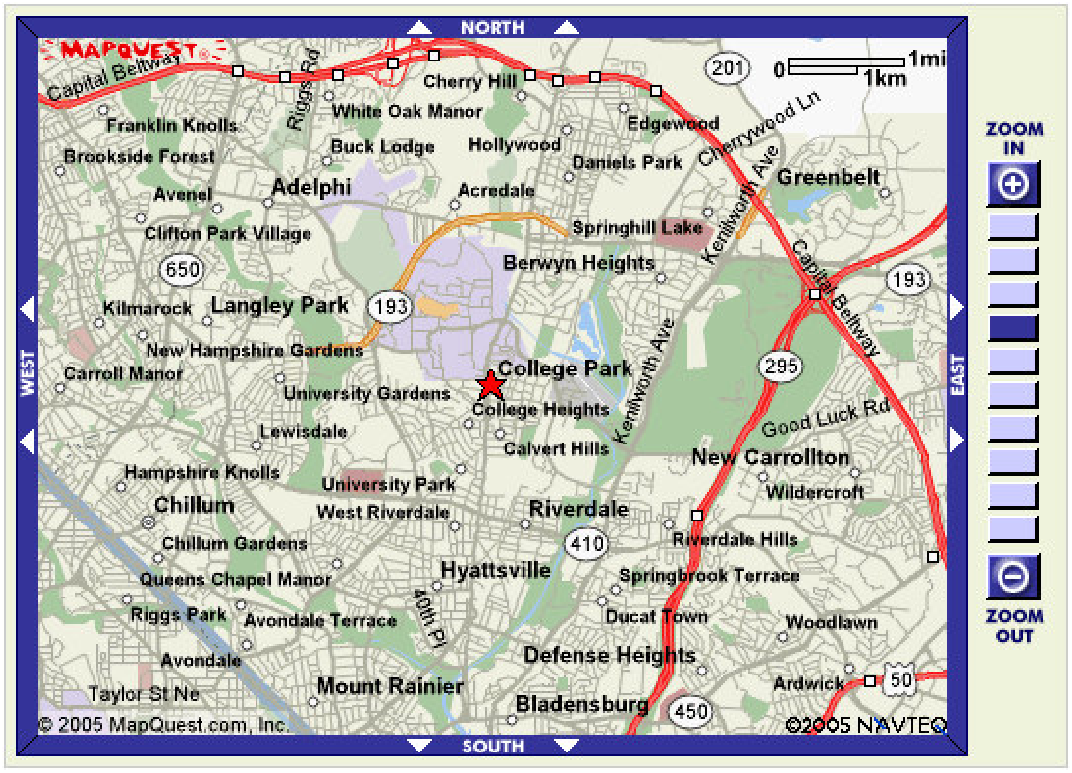


Figure 2.3: Map Quest — a sample map generated on the server and downloaded as a bitmap to the web browser (client)

On one hand, they could assume just the bare basic features on the client side and do all the work on the server side. Another approach would be to transfer a custom application to the client machine to provide for a more optimized operations and thus to improve the whole user experience. It is obvious that the designers chose to go the former way, probably to be able to serve as many users as possible without any need for initial setup or configuration. It is also obvious that having the advantage of a smart client could improve the overall user experience and performance of the system. In this research we have investigated various methods of utilizing such a smart client and found approaches that perform well for different scenarios.

An interesting enhanced raster-based design was recently presented by Google [4] and later also by Microsoft [9]. Similar to MapQuest, Google's map service is raster-based. However, this service does not send a single image covering the whole viewable area every time there is need for an update. Instead, the viewable map is divided into a grid of $128 \times 128$ small image cells. When a panning operation is executed, there is no need to download a new image that represents the whole viewable area. Only cells covering the area that just became visible need to be downloaded, others are reused by simply moving them on the screen.

Apparently, Google does not generate these image cells dynamically, instead the whole world has been pre-processed and a raster representation in the form of a collection of small image cells for each supported zoom factor has been created. Clearly, this pre-processing operation is expensive to run and produces a vast amount of data that is difficult to manage for most hosts. Additionally, this approach is not ideal for datasets that get updated often as the whole set would have to be pre-processed again. It is also not appropriate when users need to be able to select various combinations of layers to be displayed. Given the number of layers $n$, to pre-process the data into bitmaps for each combination of layers would require having $2^n$ different versions of the coverage. This is because each layer can be either displayed or hidden and they are all controlled independently, hence $2^n$ combinations. This would generate a vast amounts of data for even a few layers. Thus, while this approach is perfectly feasible for a organization with enough resources in terms of storage space and processing power that serves one-size-fits-all static data (which is the case for Google Maps or MSN Virtual Earth), it is impractical for most companies and institutions that are not capable of dedicating the resources that this approach requires.

It would also be possible to generate these tiles dynamically. However, this would increase server load as the servers would have to perform this rasterization operation in addition to serving the tiles to clients. Comparing to services such as MapQuest, the areas to be rasterized would be the same. However, the Google approach would drive larger overhead as one rasterization request would have to be processed for each tile. On the other hand, due to client caching, the Google servers would typically have to rasterize smaller total area as there would be no need to re-rasterize areas covered by previously processed tiles.

### 2.2.3 Existing Spatial Data Clients

In many respects, working with a remote spatial database is no different than doing many other tasks on remote computers. The common goal is to utilize remote resources and to subsequently retrieve and deliver the results of this utilization to the local computer. A very common example of this situation is accessing a mail server. In this scenario, a remote server collects all recipients' incoming email and holds it for them. All they have to do is to connect to the server, retrieve the email onto their local computer and process it locally. This approach is quite feasible as sending email is not as intensively interactive task as operating a program remotely using a keyboard and/or a mouse. Many times, when a remote access to a computer program is needed, some sort of proprietary protocol is developed that suits the particular application well. We can easily argue that this is the case with common protocols such as FTP (remotely operate a file server), HTTP (remotely operate a web server), SMTP (remotely operate a mail server) as well as numerous others less known and more proprietary solutions.

It may be useful to look for an alternative to developing a special communication protocol for each application that needs to communicate with the user over a network. One such an alternative is a solution that instead of transferring data for further processing to the client (e.g., email to be read, HTML to be rendered on the client browser, etc), has the server perform all the processing, thereby producing a human-understandable form (e.g., a screen shot), and only transfering this screen image — a bitmap — to the client. In this solution, the client can be very simple, barely more than a monitor, video memory, a network card and some input devices to allow users to communicate back (e.g., mouse, keyboard, touch screen, etc). While this solution is rather universal, it suffers from performance drawbacks. These drawbacks result from the fact that every user's activity triggers transfer of data between the client and the server. If the latency and the bandwidth of the network connection is less than perfect, this results in the user not being able to do truly interactive work as he or she has to wait for the results every time some operation is initiated. While these drawbacks can be worth the ease of use for a casual user, this approach is not necessarily desirable for serious work.

Other systems rely on a custom client module that communicates with the spatial server via a proprietary protocol. There are not many such systems, most of those that are available are in the stage of a research project rather than a proven commercial product.

Yap, Been, and Du [87] proposed a responsive visualization system that links a central TIGER database stored in PostgreSQL database with a Java-based client (over a JDBC interface). Their research focuses on fast viewing of static data (i.e., window queries) rather than a more powerful tool supporting a larger class of spatial queries and operations.

A new ESRI product called ArcGIS Server [25] provides an infrastructure for building centrally managed enterprise GIS applications. It can also be used out-of-the-box in conjunction with the standard ArcGIS Desktop for basic management and mapping over a network connection.

Oracle provides a rendering service called MapViewer [13] for their Oracle Spatial product. This is a Java component that accepts spatial queries and generates resulting bitmaps viewable by the client platform. Thus, from the perspective of a user, this is similar approach to that used by MapQuest and similar web-based services.

### 2.2.4 MapServer — The Map Visualization Tool

MapServer [8], a system originally developed at the University of Minnesota, is an environment for constructing spatially enabled Internet-web applications. MapServer is not a full-featured GIS system; instead, it provides a visualization layer for third party spatial data sources such as ESRI's Shapefiles or spatially-enabled DBMSs (e.g. Oracle, Sybase, MySQL). MapServer provides functionality to support those types of web applications that simply need to enable users to browse spatial data.

MapServer reads data from supported data sources and given its current configuration generates an image, presumably to be sent to the user's browser over the internet. The current configuration may control such aspects of the visualization as the area being viewed, data layers, projection, color scheme, level of detail, labels, etc.

Even though MapServer in itself does not provide GIS functionality, the method of facilitating interactive environment for users connected over the internet is a valid one. While serving images representing the current map view is similar to the method used by MapQuest and other mainstream commercial mapping services, the open source nature of the MapServer project, its wide acceptance as a map visualization tool, and its ability to serve as a single box solution make MapServer a good standard for comparing performance and user experience of different systems.

# Chapter 3

# Client-Server Paradigm

## 3.1 Introduction

A client-server computer network application is one where the client typically facilitates the user interface and connects to an application or database server to submit requests and receive responses whose nature is determined by the type of the application (Figure 3.1). Servers are powerful computers or processes dedicated to managing resources or performing tasks. Clients are usually lower end computers, workstations or hand-held devices.

The amount of business logic handled on the client side can differ from design to design. In one approach, the client can communicate directly with the server application (e.g., a database), and then process and visualize results on its own. A more common method introduces a middleman application on the server side that

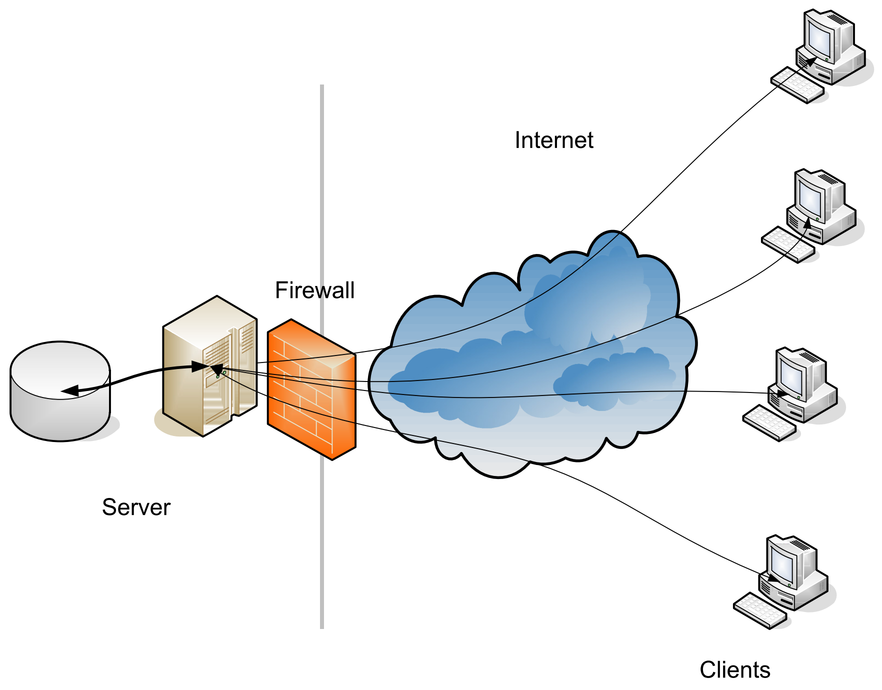


Figure 3.1: An example of a computer system deployment using the client-server architecture.

connects to the database and implements any necessary business logic. The client then simply visualizes already prepared text, images or other output.

This approach can be generalized into the application server connecting an arbitrary number of databases and other applications, and then aggregating the results and preparing them for the client to visualize. This type of system was developed under the OpenMap$^{\text{TM}}$ [12] project with our participation as described in [43].

## 3.2 Applications

The client-server paradigm is not new in the area of computer science. For many years, the only computers available used to be large machines spanning vast rooms.

Operators would then access these machines from consoles, some of them colocated, while others were in other rooms. This arrangement can be seen as an example of a client-server configuration where the main computer acts as a server and the console (usually a unit consisting of no more than a screen and keyboard that can process a basic protocol for controlling the cursor on the screen) as a client. Only with the advent of PCs in the early 1980s did the concept of an independent stand-alone computer that is widely available become common and gradually took over most IT deployments. While some specialized applications are still used in a client-server fashion, the most common applications in use today are run on the same hardware that the end user utilizes for controlling the application. This is appropriate when the application is not too complex to overwhelm the hardware available to individual end users and when the amount of data that the application needs to work with is not large or when this data is also available locally. It is interesting to note that some companies are revisiting the client-server approach even for basic computing such as utilizing office applications and e-mail [18]. This effort is however largely driven by the effort to simplify administration of a larger deployment and to allow users to hop between workstations easily rather than to solve performance issues.

Working with large spatial databases is not feasible within a stand-alone desktop machine, and thus utilizing some type of client-server approach is necessary. This is because unlike editing a letter or a spreadsheet, it is typically not feasible for all the spatial data to be copied over to the machine on which the user is going to work.

This can be due to the sheer volume or the combination of the volume and the need for frequent updates. Thus, it is necessary to load some data on demand, as the user works with the application. The exact nature of the data transfer and the design of the client-server infrastructure is the subject of our work.

# Chapter 4

# Initial Approach

In the previous chapter we discussed situations where a single spatial database server provides certain services to one or more remote clients. As we have seen, these clients do not have to be identical. By utilizing new technologies, various platforms can be supported and participate in the spatial database distributed environment. In this model, the spatial server is fixed and various clients can utilize its services.

However, we can ask whether we can allow certain versatility on the server side as well. Imagine that there are several different entities providing certain spatial data for a given location. It would be helpful if these servers could be accessed in a uniform fashion. It would be even more helpful if a client existed that would aggregate services provided by these individual servers. In this way, the user could work with several servers from one interface, possibly without even realizing the whole result is created from data residing on separate servers.

The need for merging spatial data from several servers into a single client comes from many real applications. Each community that manages spatial data may have 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 interoperability can be devised. Yet, such sharing is not only desirable but may be necessary, as funding may not be available 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 often both paper and electronic form within a few hours [63].

## 4.1 OpenGIS and SandBrowser

Initial stages of this research involved working on the OpenMap$^{TM}$ [12] project with the OpenGIS community and a report on our results was published in [43]. The OpenGIS (Open Geographical Information Systems) Consortium [40] 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.

OpenMap$^{TM}$ is a product suite developed by BBN Technologies (now a division of GTE Internetworking) in a DARPA-sponsored project to demonstrate CORBA [86]-based mapping. Whereas OpenGIS's Simple Features specification addresses the interface between a GIS database and a GIS application, OpenMap$^{TM}$ 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$^{TM}$ 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.

In this work, we have investigated methods for adding the SAND database into the OpenMap$^{TM}$ architecture. We built a SAND specialist that made data stored in SAND accessible to the OpenMap$^{TM}$ user interface client.

Issues similar to what we were working on during our involvement in the OpenMap project are discussed in [84]. These authors also study a situation when several servers running some sort of a geographic information system manage different data and the user needs to access more than one of them to get all the information he or she needs. Without any collaboration among these servers, the user would have to visit each of them one by one, execute the necessary queries on them individually and at the end aggregate the information obtained into a single result somehow. The authors present a system, where the servers would be aware of each other and would provide the user's client software with information about other potential sources of data. In this way, the user only needs to go to this single server site, and the client will learn about other servers that it would also automatically query and eventually present the user with a answer that was found by piecing several query results together. Notice that this solution is really more of a peer-to-peer architecture on the server level, where all servers are equal in terms of acting as an access point to the system for the client browser. The OpenMap approach on the other hand is client-server both on the end user and on the server level. This is because of the presence of the central server that acts as the sole contact point for the user's client browser. This central server then works as an interface between the user and the individual GIS servers.

While the work presented by Wang and Jusoh in [84] is relevant to our research, its authors address different issues that arise from implementing this client-server architecture on a GIS than what we have focused on. In particular, their interest is in the distributed aspect of the solution and they deal with issues related to being able to harness information stored on multiple servers and to combine them into a single result. These include problems such as designing a proper common interface for different modules involved in the product, converting the data into common formats so that they could be merged together, etc. They are less interested in making the system truly interactive. To achieve this also requires focusing on methods that would minimize the amount of data needed to be presented as query results, thereby minimizing the amount of data necessary to be transferred over the network connection to the client. Methods such as client data caching would help the system to provide some response to the user almost immediately. Our experience in the OpenMap project was that an overly general design with frequent transformations between proprietary and common formats in conjunction with utilization of Java [6] and CORBA technologies results in poor performance which can turn a potentially useful tool into one that is too sluggish, too cumbersome to use, and, ultimately, of little practical value.

## 4.2 OpenMap$^{TM}$ and OpenGIS

A GIS system can be viewed as being divided into three tiers: 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. 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, 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 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, 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 mouse gestures[1], which allow the specialist to respond to mouse actions on the displayed graphics.

Specialists can be implemented either in C++ or Java. Among the components of OpenMap are classes for each language (called CorbaSpecialist and Specialist, respectively) that encapsulate the common aspects of all specialists. 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.

[1]A mouse gesture is a way of combining mouse movements and clicks that the software recognizes as specific commands. A typical example is the click-and-drag operation to move objects on the screen.

## 4.3 Specialist

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 it is based on the Specialist class that was provided by BBN. Figure 4.1 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.

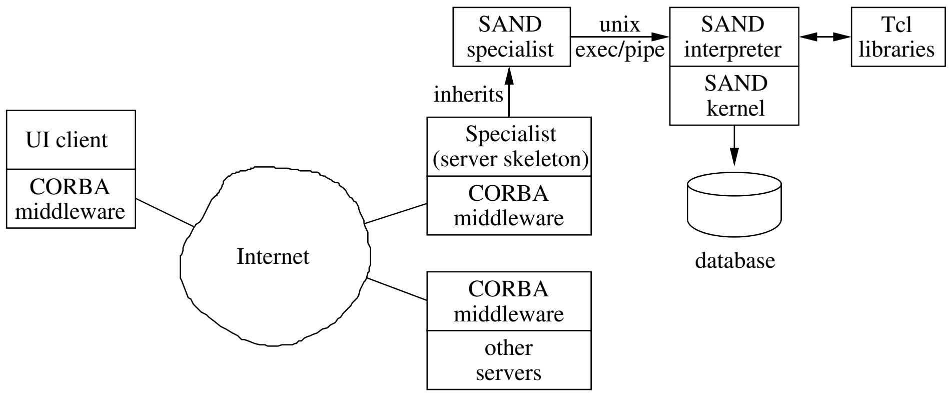


Figure 4.1: Structure of the SAND specialist for OpenMap

The data made available by the SAND specialist in our demonstration was obtained from USGS and was in the form of points, polylines and polygonal areas. 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.

## 4.4 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, 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 it 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, and 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.

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, 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 the Specialist's Point object creator takes binary floating arguments. Presumably, the

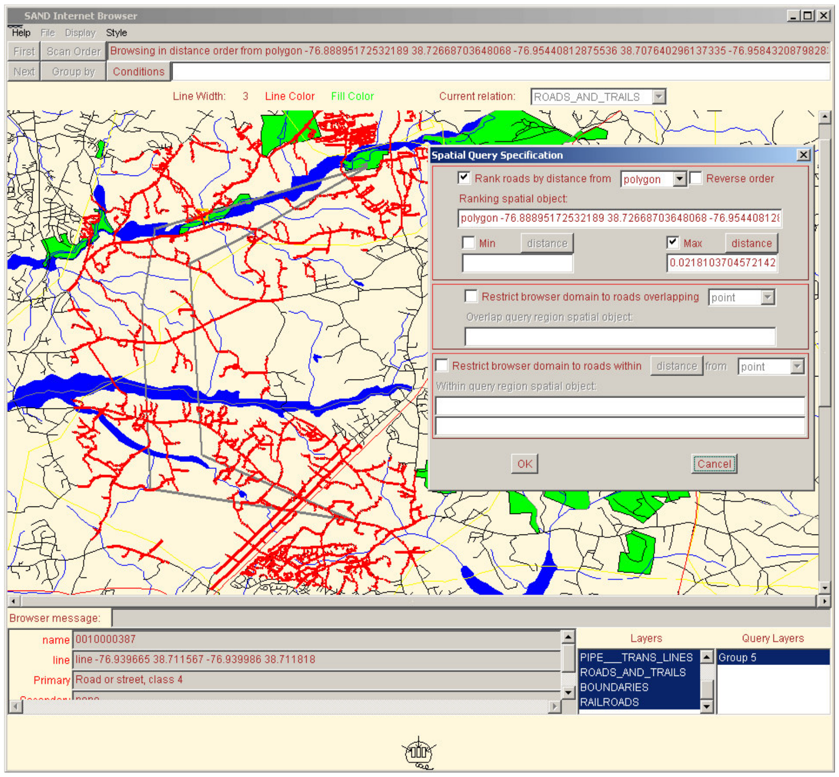
Specialist must convert this machine-specific binary floating value into some machine independent wire format (conversion 3). Finally, the display client must convert from the wire format to some display device-specific format (conversion 4) for eventual display.

This project presents one of the initial stages of our research efforts in the remote spatial data access area. It was mostly focused on interoperability and data synthesis, and less on optimizing performance and responsiveness.

# Chapter 5

# SAND Internet Browser — User Experience

The SAND Internet Browser is a Java applet or application that represents the client piece of our client-server solution for facilitation of remote access to spatial databases. The design of the SAND Internet Browser user interface (Figure 5.1) is rooted in the experiences gathered from using the original Tcl/Tk-based SAND Browser (Section 2.1.3) as well as the OpenMap project outlined in Chapter 4. While many concepts originally proposed in the traditional SAND were validated by years of usage, some drawbacks emerged as well. These weaknesses fall into two categories. One category contains items related to the user interface and its limits for the types of operations the user can perform. The other category includes problems hidden from the user involving various design-related limitations. Such problems are possibly more significant because they cause the original SAND Browser not to be able to work with certain types of datasets. Such limiting factors include the overall size of the dataset, the spatial and non-spatial types used, etc.



Browsing in distance order from polygon -76.88895172532189 38.72668703648068 -76.9544081287553 6 38.707640296137335 -76.9584320879828

Line Width: 3 Line Color Fill Color

Current relation: ROADS_AND_TRAILS

[✓] Rank roads by distance from polygon [ ] Reverse order

Ranking spatial object: polygon -76.88895172532189 38.72668703648068 -76.954408128

[ ] Min distance

[✓] Max distance 0.0218103704572142

[ ] Restrict browser domain to roads overlapping point

Overlap query region spatial object:

[ ] Restrict browser domain to roads within distance from point

Within query region spatial object:

Browser message:

| | |
|---|---|
| name | 0010000387 |
| line | line -76.939665 38.711567 -76.939986 38.711818 |
| Primary | Road or street, class 4 |
| Secondary | none |

Layers: PIPE___TRANS_LINES, ROADS_AND_TRAILS, BOUNDARIES, RAILROADS

Query Layers: Group 5

Figure 5.1: SAND Internet Browser — user interface

One major weakness related to the user interface involves the SAND Browser's design which limits it to working with only one layer of data per its instance. Typically, a single data set that represents a certain geographical area consists of several layers of data. For instance, in case of a general purpose map, these layers may include political boundaries, streams and rivers, lakes and ponds, roads and trails, contours, pipelines and power lines, etc. While the original SAND Browser was able to display all the layers as a background, any queries could only be done with respect to a single layer selected at the start-up time of the application. Switching between individual layers for the purposes of querying (e.g., finding all matching roads vs.all matching rivers) was impossible. Also, the display properties of the background layers was determined in advance and was fixed during the program execution. This caused problems when some of the background layers with extent overlapped, for instance a park layer (where parts were filled in green) and a water surface layer (filled in blue). Furthermore, not all queries require displaying of all the layers available in the given dataset. Thus, it became obvious that the user needs to be able to show and hide individual layers easily and that the user needs to be able to dynamically choose the primary layer, i.e. the layer with respect to which the queries are made. Numerous smaller problems were identified as well. For instance, forcing users to type a selection predicate in free text formatted to follow the syntax of the Tcl language was also identified by many users as suboptimal approach.

The other category of weaknesses includes items hidden from the user but obvious to the administrator of the system. One such problem results from the tight coupling between the data engine (SAND) and the visualization tool (SAND Browser). In order to enable a user to work with this system, the whole spatial server and the dataset had to be installed on the same computer that the user was working on. This may be acceptable for demo purposes or to a user who uses a limited number of datasets heavily. However, it is not realistic for centrally managed large datasets that need to be made available to multiple users. The SAND Browser made work with larger database impossible further by attempting to copy all spatial data into the main memory of the visualization module. While this is a reasonable approach for using the SAND Browser as a demo tool for the SAND DBMS, it made it impractical for any serious work.

We have investigated the properties of SAND Browser and using the experience we designed a new user interface for SAND Internet Browser. This new interface looks familiar to SAND Browser users which makes it easy for them to switch. It, however, also provides access to the improved SAND Internet Browser functionality. While the user interface improvements are naturally helpful to users, our primary focus was on creating the new SAND Internet Browser as a tool that provides visualization and query access to an arbitrarily large data set in a typical deployment scenarios.

Like the original SAND Browser, the SAND Internet Browser 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.

A SAND Internet Browser window is divided into several panels stacked vertically. These are (from top to bottom):

- The *command area*, containing menus and buttons for several actions: *First* and *Next* retrieve the first/next tuple satisfying the currently set scan order and predicate; *File* provides various basic operations such as *Open* or *Close* [the current relation] or *Quit* [the browser]; *Display* presents a menu of several activities connected to the graphical display, such as clearing it or drawing spatial features; *Style* allows changes of drawing attributes such as line or fill color.

- The *Scan order* panel contains a button that triggers the exhibition of several "pop-up" dialog boxes, and a message area that displays the currently selected scan order parameters. Each dialog box corresponds to parameters to be used when the relation is being scanned with the help of one of the indices defined for it.

- The *Conditions* panel allows specification of a predicate to be used to evaluate the query. Unlike in case of the SAND Browser, this client uses an intuitive interface for building arbitrary conditions based on fields of the current relation. In this way, the user need not to be aware of any specific syntax that must be followed.

- The *info panel* indicates the current line and fill colors (the colors that will be used next time the user chooses any displaying operation from the *Display* menu). It also contains a choice which allows the user to change the active data layer (the relation from the data set that all the queries are executed with respect to).

- The *graphical display* panel is the drawing area where spatial features are input and output. At any given moment of the interaction, the current value of the relation's spatial attribute is displayed in this area and highlighted by an orange rectangle. Most of the other user interface components which hold a spatial feature value support input by drawing on the graphical display. The display can also be panned and zoomed in and out. The zoom in and zoom out operations are available through left and middle mouse click respectively, while the right click finds the nearest object to the location of the click from the current relation and makes it the current tuple (i.e., a “pick” operation in computer graphics parlance).

- The *info line* will show various messages during the browser execution.

- The *tuple display* panel contains a series of labeled entry boxes, one for each attribute in the schema of the relation. These are updated to reflect the value of the current tuple.

- The *layer display* lists all the layers available for this relation. Clicking on the individual entries in this list toggles displaying of the respective layers in the graphical display area on and off.

- The *query history* lists all queries and their results (they can be assigned names) the user has performed so far. In this way, the user can easily visually compare results of multiple queries by flipping through them, the user can also return to an older query, use it to pre-populate it the query dialog box, then initiate a new query. Thus, it is easy to form new queries that share some parameters with previous ones.

Below, we give a few examples of how a user may utilize the user interface to obtain results to specific queries.

- Given a road map of Silver Spring, MD (single layer data set), find all the roads within distance $d$ from polygon $p$ (hand drawn to represent an area of interest, e.g., a flooded area and its immediate proximity) and return them in the order of distance from line $l$ (hand drawn to represent, for instance, an emergency vehicle route). The result will indicate the streets affected by the flood that are closest to where the emergency vehicles can get.

    1. Go to **File** menu and **Open Relation** Silver Spring.
    2. Press **Scan order**, choose **line** to open a dialog box
    3. Enable **Ranking by distance from**, choose **line** and select the line $l$ by drawing on the canvas.
    4. Enable **Restrict search to lines within**, choose **distance** and specify a line by drawing on the screen. The length of the line will define the distance used in this condition. Choose **polygon** from the object list next to it and define polygon $p$ by drawing it on the canvas.
    5. Click **OK** to submit the query.
    6. Click **First** and **Next** to retrieve the first and all the other lines within distance $d$ from $p$ ordered by distance from $l$.

7. The entire group could also be displayed at once by selecting **Display group** (uses the current line color) or **Display blended group** (changes the color for each returned line to indicate the order in which the lines were retrieved).

- Given a multi-layered map of Washington, D.C., find all the roads that pass through a park

  1. Use **Open Relation** to open the appropriate data set.
  2. Make the **Road** layer the current relation.
  3. Click on **Scan order**, choose **line**, and a dialog box will appear.
  4. Enable **Ranking by distance from**, choose **relation**, choose **Parks**. Enable **Max distance** and enter **0** (zero) to the text box below. This will look for all the lines and a maximum distance of zero from any park. In other words, for all lines that intersect some park.
  5. Click **OK** to submit the query.
  6. Set the line width and color to the desired values and use **Display group** to show the results.

- Find all roads that are within distance *d* from Georgia Avenue in the Silver Spring database

  1. **Open Relation** Silver Spring.

2. Click **Scan order**, choose **name type**, and both the min and max value of **Name**(**Type**) to **Georgia**(**Ave**).
3. From **File** menu, select **Save group** under a name of your choice.
4. Click **Scan order**, and choose **line** to open a dialog box.
5. Enable **Ranking by distance from**, choose **relation**, and choose the relation that you just saved.
6. Enable **Max distance** and click on the button to specify a line by drawing on the canvas. The length of the line will define the distance *d*.
7. Click **OK** to submit the query.
8. Use **Draw group** or **Draw blended group** to display the result of the query.

# Chapter 6

# Direct Server Access

Traditionally, a client-server computing paradigm only involves two computers — the client and the server (obviously not counting computers and devices in between the two that simply route or shape the traffic between them, such as routers, firewalls, etc). In some types of deployment, the server functionality may be actually performed by two (or more) machines to add redundancy (and thus reliability) or for load-balancing purposes. This however is transparent to the client and the cluster of servers acts as a single entity with a single interface.

In recent years, an alternative to a client-server approach emerged in the form of a peer-to-peer networking (e.g., [27]). The main idea is that instead of relying on a single machine[1] to support all the clients (or users), these clients all contribute some

[1]Or a single virtual machine — that is, a cluster of machines that act as a single entity, usually to improve performance or reliability

of their resources to a common resource pool which they all can utilize to obtain the services that they need (Figure 6.1).

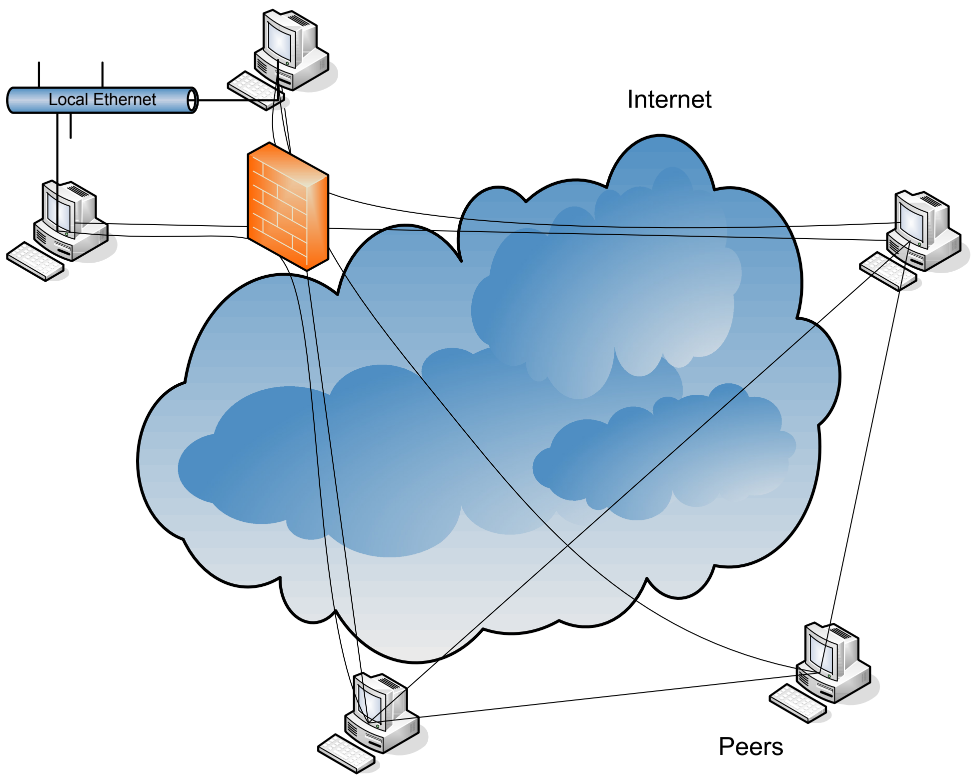


Figure 6.1: Example of a Peer-to-Peer network environment. Individual computers can directly communicate and exchange information with any other computers participating in the network.

The application of the peer-to-peer paradigm on remote spatial database visualization and querying has been investigated in [82]. Due to the large amounts of data usually involved in operations on spatial databases, inventive methods are needed to distribute data among individual peers in an efficient way [81].

## 6.1 Pure Client-Server Design

The simplest and most common design for the client-server architecture makes individual tasks such as data management, image rendering or query evaluation the responsibility of either the client or the server. When the spatial database application is implemented in this manner, the server handles all the data management and query evaluation. The client only facilitates data visualization while maintaining connectivity to the server. In this scenario, the client simply translates user input into queries and sends them to the server. It can also receive data sent by the server and visualize them. There is no data storage or processing on the client beyond these basic functions. Figure 6.2 indicates the data flow and processing. Note that this design corresponds to way in which many popular web-based mapping services such as MapQuest or Switchboard's Maps On Us operate.

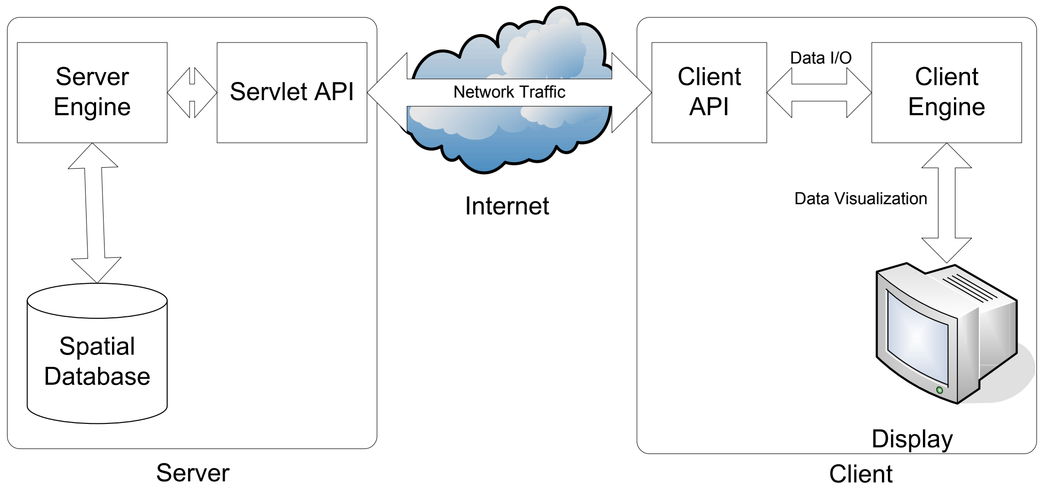


Figure 6.2: Diagram of basic Client-Server architecture

The advantage of this approach is that most users can utilize this service with the resources that they already have — a networked machine with a web browser. The users do not need to install or set up any additional hardware or software. However, the main disadvantage of this approach is that the client needs to communicate with the server every time the user requests even the most simple operation. This can slow down the user experience significantly if the network throughput and latency are a limiting factor or if the server is heavily loaded.

## 6.2 Memory Based Caching in Client

The first method that improves upon the basic design is one where the client utilizes some of its own main memory to store (cache) some of the data in the central database (Figure 6.3). This allows the client in some cases to rely on its own data repository to handle some of the user's requests thus cutting back on the network utilization and improving the system's responsiveness. Naturally, the spatial data stored on the client must be spatially indexed for fast approach. Note that in this approach it is no longer possible to use the standard web browser as a mere image viewer. A custom code needs to be loaded onto the client to facilitate the operations to be performed there. The Java environment has emerged in the past years as a platform of choice for most types of lightweight cross-platform applications.

The maximum amount of data to be stored on the client in order to optimize the overall performance will depend on the extent of client's available resources. The

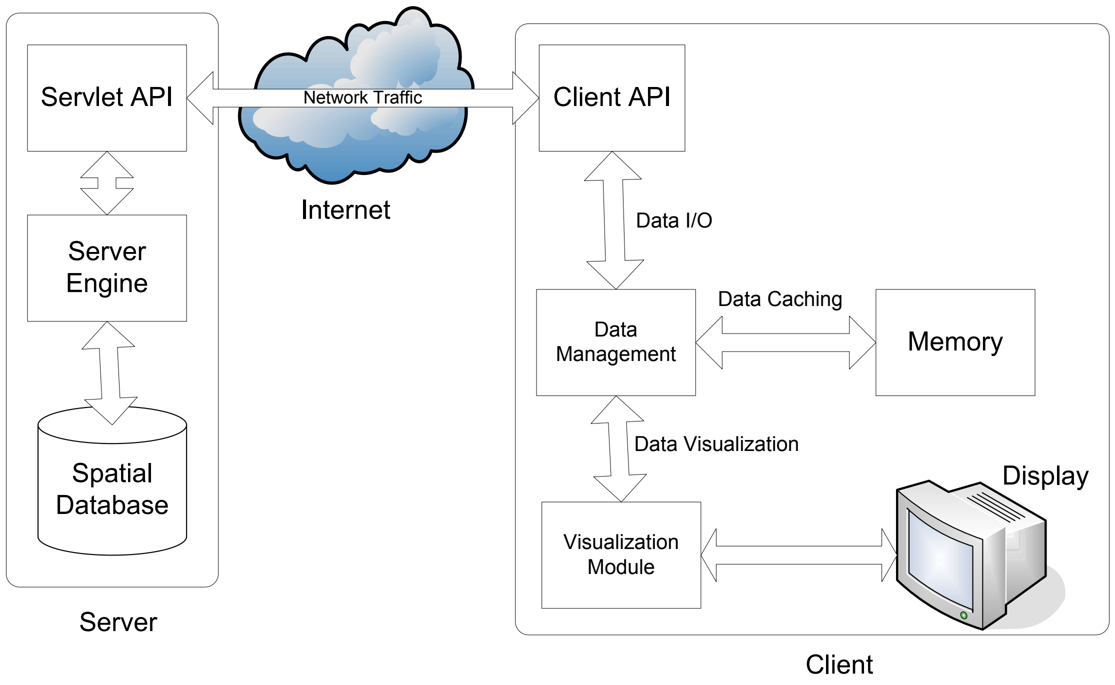


Figure 6.3: The thin client communicates directly with the main spatial server and utilizes its (limited) memory capacity to cache some spatial data locally. This saves many data transfers as the client does not need to ask the server for data after each screen update.

basic concept of this design is for the client to be fetching the requested data via fast memory-only operations whenever possible. This would be more efficient than retrieving the same data over the network from the central server. While retrieving data from the memory is usually more efficient than downloading the data over the network, a special consideration has to be made in case of environments that are unusual in some way. For instance, small devices with low processing power may still be providing a more efficient service by acting as terminals and without attempting to perform any data management operations themselves.

Given these facts, it's clear that there are no single settings that would assure the best performance for all types of devices. However, optimal values for some of the parameters can be estimated by the system via a self-test. For instance, it is important that the amount of memory allocated for the spatial data storage is lower than the total available main memory and that even after that enough memory is left for other parts of the application. It is necessary to use only as much memory as the Java Virtual Machine can provide without swapping on the disk. This would slow down the process significantly and cancel out benefits of internal caching over the network-only approach. Unfortunately, in Java it is difficult to find out the amount of memory available without swapping as the system seamlessly pools together the main memory capacity and the available disk space[2] to provide the information.

Similarly, the processing power of any given hardware platform can be assessed upon start up of the client application. This information indicates how much spatial data the platform can handle and sort through effortlessly.

Operations performed by the SAND system are primarily client-driven, i.e., any operation performed on either the client or the server is in response to some user-generated input. To minimize the amount of data that needs to be transferred from server to the client in response to each event on the client side, various techniques were developed and implemented in SAND.

To keep the amount of traffic between the client and the server low, we cache some data on the client in case the user requests another operation on data in the

[2]Up to the total memory allowed for the process to use by a runtime parameter.

same area. We store the data in their original vector format rather than the resulting bitmaps so that the client is able to generate new views and process some types of queries locally without having to request additional data from the server.

Only using the main memory results in lower requirements for the client in terms of access privileges to the client platform resources. This allows us, for instance, to run the client as a basic Java applet without requiring the user to perform any special configuration or installation. This is especially beneficial in cases where ad-hoc usages of the system are needed, when users are invited to work with the given SAND instance on a short notice (e.g., in case of emergency personnel accessing a spatial database specially set up to support handling of a specific crisis).

### 6.2.1 Internal Spatial Data Structures

The spatial data is stored on the client using a PMR quadtree [71] spatial data structure. This structure divides the plane into quadrants such that if an object is inserted into a certain quadrant, then if the quadrant already contains more than a predefined threshold of other objects, then the quadrant is split into its four children once and only once and the objects are reinserted into the children. Thus, the objects are always stored in the leaf nodes of this quadtree.

We establish and maintain the maximum amount of data that can be cached on the client in order not to overwhelm or crash the client platform. Since a vertex is an atomic building block of all the objects representable within the application, the allowable amount of data is measured in terms of the number of vertices occupied by all the objects stored within the tree. An alternative would be relying on the runtime environment's reports regarding the amount of occupied and free memory. Unfortunately, the amount of free memory reported by Java VM is not necessarily up-to-date (i.e., it does not account for available memory not yet claimed by the garbage manager) or authoritative (e.g., it may include memory that would only become available through disk swapping). Thus, counting the total number of vertices stored provides better control over the system performance. The goal is to keep enough memory free within the Java VM for all the temporary data allocation to be performed within the main memory. If there is not enough available main memory, the machine may start running the garbage collector more frequently, thereby slowing down our main process. This is discussed in more detail in Section 6.2.3.

Each leaf node of the PMR quadtree also contains a time stamp indicating when it was accessed (displayed) last. Together with the PMR quadtree containing the spatial data, we also maintain pointers to all the PMR quadtree leaf nodes in a variant of a balanced binary search tree[3]. The key for this tree is the time stamp stored in the PMR quadtree leaf nodes. This structure facilitates quick insertions, deletions and location of pointer representing the PMR node with the oldest time stamp. This arrangement facilitates our caching mechanism (Figure 6.4). When we

[3]The Java library $TreeMap$ class is used to build and maintain this structure. The Java implementation is based on red-black trees [47]

need to make more memory available for additional data, we use the least-recently-used (LRU) caching mechanism to delete as many PMR leaf nodes linked from the top of the balanced binary search tree as necessary. If all four children of some internal PMR quadtree node are removed, the tree automatically collapses and the internal node becomes an empty leaf node. A flag in each node indicates whether the node represents an area that is actually empty (valid node) or whether the node is empty because its elements are not available in the memory (e.g., page fault, invalid node).

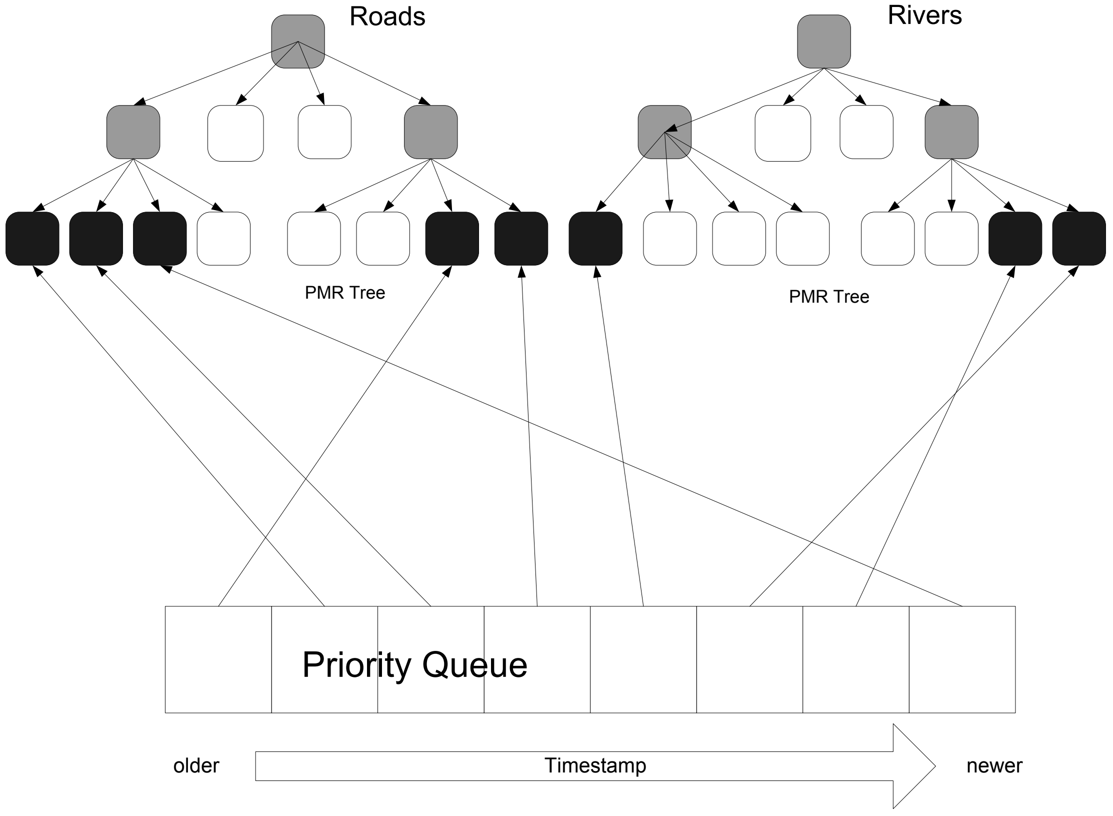


Figure 6.4: Individual spatial data layers are stored in separate PMR quadtrees. A priority queue shared by all of them maintains ordering of all the PMR leaves for all the PMR quadtrees based on the time of their last viewing.

Note that using this mechanism, the entire quadrant has to be contained in the memory for its node to be valid. This may be too inefficient as if we continuously work with only part of the quadrant and have no need to load the rest of the quadrant, then the node would never be marked as valid and the data from the part in which we are interested would be reloaded over and over. To prevent this, we add another field in each node indicating which part of it is actually valid. Thus, if we loaded data for only part of the quadrant, we mark the quadrant as valid but indicate which part of it is actually valid (i.e., the intersection of the quadrant and the query window). The next time we need to access data from this quadrant, if the area we need falls completely within the valid area of the node, we do not need to load any additional data. If the area we need is not fully enclosed by the valid area, then we load the missing part and increase the valid area of the node accordingly. This is discussed in more detail in Section 7.4.

The type definition (in Java syntax) of the node in our PMR quadtree data structure is as follows:

```
class RNode implements NodeAble {
        private RNode parent;
        private RNode[] son;
        private DrawableVector lo;
        private DRectangle validSubarea;
        private long timestamp;
        private boolean pagefault;
}
```

As discussed above, a typical dataset would contain several tables representing different layers of the map. While each layer is stored in a separate PMR quadtree, there is only a single balanced binary search tree for all the layers combined. This way, when a user stops working with one of the layers, its data will be automatically and gradually removed from the cache and will be replaced with the data needed currently.

To understand the memory management used to store data in our PMR quadtree, it is useful to compare it with the standard memory management. There, individual pages of main memory may be swapped out onto the disk if the processes running on the system require more memory than what is available. Analogously, if the client asks the PMR tree to store more data than its capacity (see discussion on page 54), the individual PMR nodes can be "swapped out". In the context of the PMR quadtree, this means that this node is freed from memory and pointers to this node are reset to indicate that the node is no longer available. Now, when the data for the same quadrant is needed again in the future, the client will request its download from the server. So in its simplest form, the PMR node is valid if and only if all the data in the database that overlap its quadrant are loaded into the client's memory.

Unfortunately, this approach alone does not produce good results because the user-driven accesses into the database (i.e., the window queries resulting from the user panning and zooming) do not necessarily align with the PMR quadtree boundaries. This means that some PMR nodes would not get loaded fully and thus would not become valid per the above definition (see Figure 6.5b). Therefore, not all the results of the window query would get cached and some pieces of the window would have to be reloaded from the server after each screen refresh even if the user's viewable area remains the same.

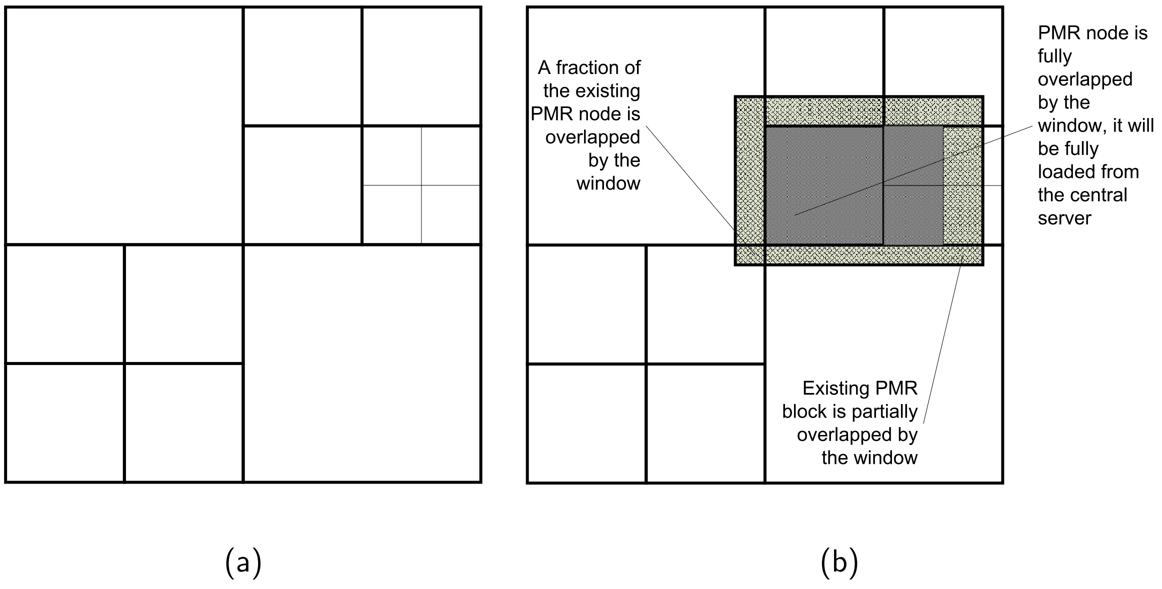

A fraction of the existing PMR node is overlapped by the window
PMR node is fully overlapped by the window, it will be fully loaded from the central server
Existing PMR block is partially overlapped by the window
(a)
(b)

Figure 6.5: Figure 6.5a shows spatial decomposition of a PMR quadtree. Figure 6.5b shows the same PMR quadtree after a window operation has been performed. Note that the window overlaps some nodes of the PMR quadtree fully, some partially and some not at all. All data will be loaded for the fully overlapped nodes and they will become regular leaves of the PMR quadtree. The non-overlapped nodes will not be affected. The partially overlapped nodes will have some data loaded typically causing some of them to split. Leaf nodes that were not loaded fully will have the covered area covered indicated in their $validSubarea$ field.

One solution would seem to be to load extra data from outside the query window in order to fill the enclosing PMR quadrant in its entirety (and allow it to become "valid" and thus cached) as indicated on Figure 6.6. Unfortunately, the data distribution in the set may be substantially non-uniform and without examining the data first, one cannot decide what enclosing PMR quadrant or a collection of quadrants to load. Picking a quadrant or a collection of quadrants without knowing how much data in the database overlaps these quadrants could result in loading much more data than what the user requested. In this case, this caching method would actually be slowing the user down rather than providing improvement.

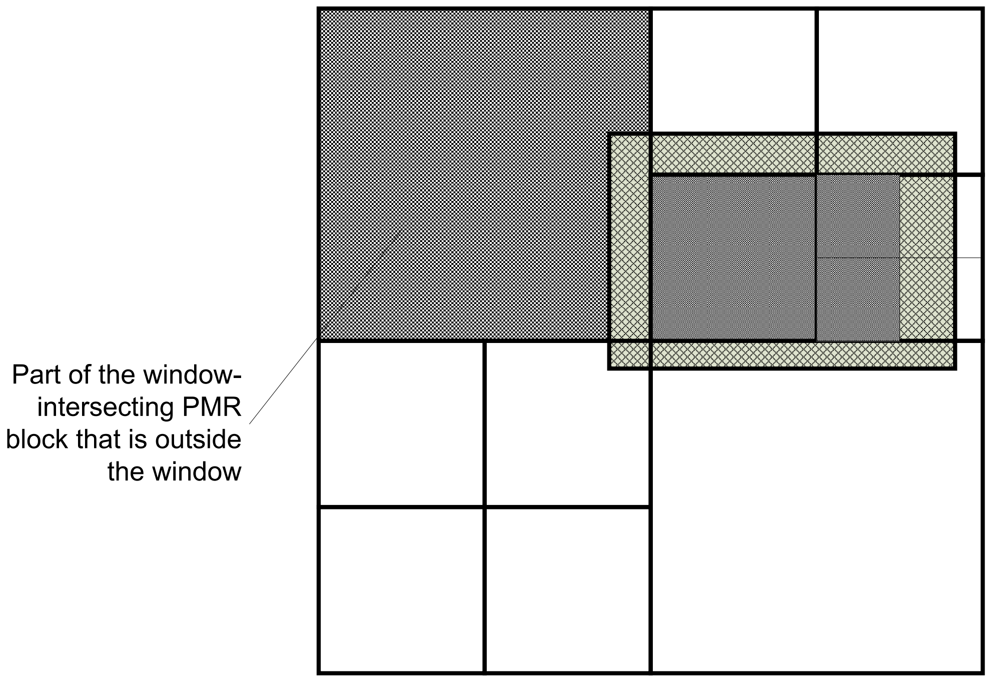

Part of the window-intersecting PMR block that is outside the window

Figure 6.6: In some cases, loading the partially overlapping quadrant in full would result in loading excessive amounts of data.

The solution that we chose involves loading data only for the part of the PMR block that overlaps the query window. As the data is being loaded, the node will be typically split as necessary. For each leaf node that was not loaded in full we mark its PMR block as valid (meaning the node is in the memory and contains data), however we indicate in a field within the node that not the whole area of the block is represented. Specifically, we use the field *validSubarea* (see page 60) to indicate the rectangular subarea of the PMR block for which the data is fully loaded in the memory. Obviously, when the node is fully loaded, then this field matches the block area itself.

The question now is how to update the *validSubarea* field after subsequent window queries that overlap different areas of the block but which still do not result in the block being fully loaded (see Figure 6.7). To analyze this situation, it is helpful to review the dynamics of building a PMR quadtree based on such window queries. When an empty PMR block overlaps a query window, the system loads data from within the area of the overlap. The amount of data loaded is typically much larger than the PMR splitting threshold.

So the originally empty PMR node is typically split over and over following the rules for building a PMR quadtree until all the objects are inserted. This means that the rectangle that indicated the valid area in the original leaf node is no longer assigned to the PMR block that was empty originally as it now turned into an internal PMR node. Instead, some of the smaller children of this block that intersect the boundary of the query window are not loaded in full and are assigned a valid area indicator. During a subsequent window query operation, if the query window

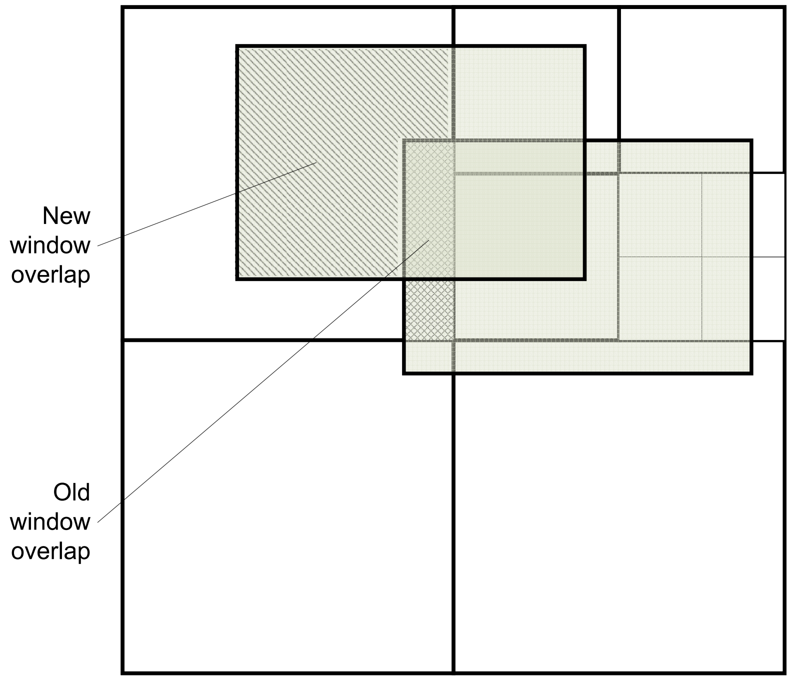


Figure 6.7: Two consecutive window operations result in partial loading of an existing PMR block with data. Since the union of the two loaded subareas is not necessarily a rectangle, we cannot represent the loaded area by $validSubarea$ directly. We handle this by calculating the bounding rectangle for these loaded areas, load the remaining overlapping data into the PMR quadtree as well, and use this bounding rectangle as the new $validSubarea$ for this PMR node.

intersects but does not overlap some of the PMR blocks, the system loads additional data that lies inside the intersection, thereby likely triggering further decomposition of the PMR node. Therefore, we see that updating a valid but incomplete PMR node via another window query that results in the same node still being incomplete happens only rarely, and the total number of original and newly added objects would still have to be below the PMR threshold. We see that this only happens when the amount of data within the area covered by this PMR block is low. Thus, when an incomplete node is filled through another window query and this operation does not result in splitting of the node, it is safe to calculate the union of the two rectangles (the original *validSubarea* and the intersection of the PMR block and the new window), load data that lies within the resulting rectangle, and use this rectangle as the new *validSubarea*. Of course, if this additional load finally results in splitting the PMR node, the *validSubarea* for the node is no longer relevant as the node becomes an internal node.

### 6.2.2 Data Traversal

As the user explores the content of the database using a graphical viewer, he or she is basically retrieving all the objects stored in the database that overlap the current viewing windows. Some of the content may be already available within the client while other parts of the database have either not been retrieved yet or were retrieved earlier but dropped again since then. It is difficult and computationally expensive to maintain the definition of geometry that identifies which parts of the database are currently in the cache and which are not. Such geometry would be represented by a collection of orthogonal polygons and every time a new requests is issued, an intersection between this object and the viewing area would have to be computed.

Instead of maintaining the definition of the available area, we store the information about availability within individual nodes of our spatial data structure. Each node can be either 'white' (no data available for the area represented by this node, and such a node is a leaf), 'gray' (some data is available for area represented by this node and further recursion is needed to get the exact answer) or black (the whole area under this node is available in the local cache). If a gray node is a leaf, it means that there is some data stored but not enough to cause another split. In such case, the node has a rectangle associated with it that specifies which part of the node is valid.

When the content of the spatial structure overlapping certain query object needs to be drawn, a tree traversal is performed to find all the objects in the tree that overlap the query object. Sometimes, the internal nodes may be “invalid”, they either were not loaded yet or were previously removed by the memory management process when they were not used for some time. In such a case, the data needs to be reloaded. Remember from the above discussion that is it not always necessary to reload the content of the entire invalid block. Only the intersection of the query object and the invalid block is considered and even in that case, reload is only performed if the valid area represented by the *validSubarea* field doesn’t contain this intersection.

As requests are received from the visualization module for more data, there are two ways to handle them:

1. The data can be requested from the server each time a missing block is found. The advantage of this approach is that the data can be drawn on the screen as the tree is being traversed. This has a benefit from the user’s perspective as it enables him or her to see some approximate results while waiting for the whole screen to update.

2. All the missing blocks can be gathered first and stored in a set. Only once the whole tree has been traversed, is a single query is created that references all the stored blocks at once. The advantage of this approach is the lower number of server requests which means less time spent on overhead associated with creating the queries and accepting the results.

Our research indicated that the overall behavior of the system is more acceptable when implemented in the latter fashion. The number of queries resulting from each screen update is cut from perhaps tens to only one, and the lower overhead cost significantly speeds up the whole redrawing process. So, while the user may need to wait a little bit longer to see the first response to his or her request, the process also finishes faster. Therefore, overall, the system react to any user input in a more instantaneous fashion.

The final algorithm contains two steps. In the first step, the system finds out what areas need to be loaded from the server and builds a collection of rectangles that represent this area (the algorithm is expressed in Java notation). The function $fetch$ is called with parameters $q$ representing the root of the PMR quadtree, *searchBlock* is the current viewable area, *block* is the rectangle represented by the node $q$ and *boxCollection* is used to store the set of rectangles whose overlapping data need to be loaded from the server. Once the function call returns, *boxCollection* will contain the set of rectangles whose content needs to be loaded from the server.

```
double xf[] = {0, 0.5, 0, 0.5};
double yf[] = {0.5, 0.5, 0, 0};

void fetch(RNode q, DRectangle searchBlock,
           DRectangle block, Vector boxCollection) {

  DRectangle inters = searchBlock.intersection(block);

  if (q.isLeaf()) {
    if (q.isPageFault() && !q.isStored(inters)) {
      if (r.getValidArea() != null) {
        boxCollection.addElement(inters.union(r.getValidArea()));
      } else {
        boxCollection.addElement(inters);
      }
    }
    return;
  }

  for (int i = 0; i < 4; i++) {
    DRectangle subblock = new DRectangle(block.x + block.width * xf[i],
    block.y + block.height * yf[i],
    block.width / 2, block.height / 2);
    if (subblock.intersects(searchBlock)) {
      fetch(q.son[i], searchBlock, subblock, boxCollection);
    }
  }
}
```

In the second step, the algorithm takes the list of rectangles *boxCollection* returned by the first step and loads all the data from the server that lie within the area defined by this collection of rectangles. Then for each rectangle loaded, it adjusts the corresponding PMR node status:

```
for (int i = 0; i < boxCollection.size(); i++) {
  DRectangle block = (DRectangle)boxCollection.elementAt(i);
```

```
    resetPages(root, bbox, block);
  }

  private void resetPages(RNode r,
                          DRectangle block, DRectangle search) {
    if (r.isLeaf()) {
      if (r.isPageFault()) {
        if (search.contains(block)) {
          r.resetPageFault();
        } else {
          // store in the node the intersection of block
          // and search that was loaded with data
          r.setValidArea(block.intersection(search));
        }
      }
      return;
    }
    for (int i = 0; i < 4; i++) {
      DRectangle subblock = new DRectangle(
                               block.x + block.width * xf[i],
                               block.y + block.height * yf[i],
                               block.width / 2, block.height / 2);
      if (subblock.intersects(search)) {
        resetPages(r.son[i], subblock, search);
      }
    }
  }
```

Given a PMR block $n$, the function `n.isPageFault` checks whether all the data that lie within the $n$ are loaded in the client, i.e., it returns true if the node $n$ is completely invalid or if only part of the node's area is loaded. It returns false if the node's data are fully loaded in the memory.

The function `n.setValidArea(a)` adjusts a partially valid PMR node $n$ to indicate that data overlapping $a$ (a part of the block termed *search* in the function parameters that is overlapped by $n$) is loaded in the memory.

The function `n.resetPageFault` marks node $n$ as fully valid.

Now, when we need to display all data that overlaps a given window $w$, we can look at not just the valid/invalid identifier of each PMR node that overlaps $w$, we can also check the *validSubarea* field of the invalid nodes. If the intersection of the window $w$ with the PMR block is fully contained in the node's *validSubarea*, we know that all the necessary data for this window query is already in the database, even if the PMR node is not loaded fully. This test is performed by the *fetch* function above, when deciding what data needs to be fetched from the server. When the drawing function is called, it already knows that all the data is already loaded in the memory and it simply steps through the overlapping PMR nodes and displays their contents, as shown in the algorithm below:

```
void draw(RNode q, DRectangle searchBlock,
          DRectangle block, CoordSystem cs) {
  if (q.isLeaf()) {
    q.draw(cs);
    return;
  }

  for (int i = 0; i < 4; i++) {
    DRectangle subblock = new DRectangle(block.x + block.width * xf[i],
    block.y + block.height * yf[i],
    block.width / 2, block.height / 2);
    if (subblock.intersects(searchBlock)) {
      draw(q.son[i], searchBlock, subblock, cs);
    }
  }
}
```

The function `n.draw(cs)` simply draws all data stored in PMR node $n$ using the coordinate system $cs$.

We have mentioned the utilization of a balanced binary search tree to facilitate caching. The balanced binary search tree data structure supports quick (`log n`) insertions, deletions and lookup operations. In the system, a single balanced binary search tree is shared by all data layers of the current data set. Each PMR leaf node in each layer contains a time stamp[4] field (see page 60 for the data structure definition). References to individual PMR nodes are then stored in the balanced binary search tree and the time stamp associated with each node serves as the key. When the content of a PMR node is displayed, the time stamp of that node is updated. This requires reordering of the balanced binary search tree, which, in practice, is achieved by simply reinserting the node into the tree using the updated time stamp. This mechanism provides us with the list of all the PMR leaf nodes currently in memory sorted in the order of their last usage. This allows us to implement the least-recently used caching mechanism on the PMR nodes. When memory needs to be freed, i.e., the total number of vertices stored in the memory exceeds the threshold (see Section 6.2.1), PMR nodes are removed from the memory in the order of their last usage until the number of vertices decreases below the threshold. The removal of PMR nodes from the PMR quadtree follows the standard PMR-deletion algorithm. It means that if

[4]Since operations happen quicker than the granularity of the system clock, the time stamp is really a large counter that increases every time a new element is stored into the tree. This guarantees uniqueness of the keys.

the combined number of elements stored in all four children of a PMR node is below the splitting threshold, the tree collapses and the parent node now becomes a leaf itself. This can repeat recursively.

Since the single tree handles all data layers (i.e., all PMR quadtrees, one for each layer), it means that the system automatically purges data that belong to layers that the user turned off. Since the layer was turned off, its PMR nodes no longer get accessed (i.e., they are no longer used for visualization). Therefore, the PMR leaves of hidden layers do not have their time stamps updated and thus they gradually become the oldest in the queue (in terms of the time stamp). This means that they will be freed first when more memory is needed.

### 6.2.3 Memory Management Considerations

Unlike some development platforms such as C/C++, the Java environment does not provide the code to access memory management functions on the OS level. Specifically, in Java, the virtual machine handles memory management. It monitors the references to all allocated objects and when an object is no longer referenced, it can free its memory and make it available for new objects in a process called "garbage collection". The application layer code has only limited control over this process.

Thus, when elements are removed from the tree, they cannot be explicitly and immediately released from memory. Instead, the client makes sure that there is no link to the removed object and as the independent Java garbage collector (GC) thread eventually locates this data, it will recognize it as no longer in use and will make its memory available for other objects. Thus, the memory will not become available immediately but only after it is found to be free by the GC process.

Another problem is caused by the fact that the remaining available memory reported by Java can only be considered to be a rough estimate. This is because the system does not know what the amount of the free memory is at all times and thus, it has to run the garbage collector first before it can find out how much memory is really occupied. This leaves two run-time options, either build the system on numbers that may be vastly inaccurate, or, run the garbage collector explicitly every time the memory status is needed. The former method is not suitable in our case as we need more precise numbers than what this method offers. The latter method is not acceptable due to performance reasons.

So we see that any fine tuning of the memory management based on the amount of remaining available memory is very difficult. Many times, the garbage collector, which runs continuously in the background, would increase the intensity with which it looks for available memory when it detects that more memory is needed. This happens before the application layer can detect this state so it is possible that the main thread responsible for user interaction suddenly slows down significantly without any apparent reason. As this is very undesirable from the user's point of view, it is important that we take precautions to avoid this. Our approach is based on ensuring that the total amount of memory that the program has allocated at any given time does not exceed the amount of memory easily available within the virtual machine. Therefore, we need to ensure that the garbage collector will not have to struggle to find a few more blocks of memory when most of the total memory is already allocated. This amount of memory that is 'safe' for the application to utilize without affecting the performance throughout the execution is determined when the application starts. In this way, the garbage collector works with the same intensity all the time and does not cause any application performance fluctuations.

To compute the 'safe' amount, our system utilizes the number of vertices of all the objects in the tree as the identifier of memory occupancy. A safe threshold is determined from the total available memory at the start. During the execution of the application the memory manager starts dropping nodes from the memory once the number of vertices exceeds this predefined threshold. To establish the ideal threshold turns out to be as difficult as it is important. Setting the threshold low and storing less data on the client would result in more data flow between the client and the server due to more frequent "page faults". Storing more data on the client would mean that the amount of memory being utilized would come closer to the maximum amount available. This means that the Java garbage collection thread could become much more aggressive trying to locate and reclaim unused memory. As discussed above, this could result in noticeable slowdowns in the main user interface thread, or even bring the execution of these main threads to a standstill.

The internal caching method relies solely on the resources available on each client's platform. This method allocates a portion of client's available memory space for storing spatial data that has been received from the spatial server. The efficiency of this method depends on the amount of memory available and the processing speed of the platform. Additionally, as with any other caching method, the usage pattern also affects the benefits of this method. If the spatial operations are localized, i.e. users tend to access the same part of the global map for a while before moving to another part, it can also improve the efficiency dramatically as the data turnaround within the cache is smaller. If the data needed can be found in this local repository instead of having to fetch it from some external source, it completely avoids delays caused by the data transfer over the network layer. Unfortunately, in typical scenarios the amount of data that can be stored internally compared to the amount of data available on the spatial server is very small.

Our system relies on the client to keep track of what data it has available. The client is responsible for requesting only the data that it is missing to draw the user-requested area. In this way, the server can be designed as stateless. It does not need to keep track of what data was sent to which client in order to only send data that the client has not received yet. The client can do its own memory management internally, drop data it no longer needs, and download targeted data from the server.

In summary, we have shown how the client treats leaf blocks of its PMR quadtree as "pages" of memory for the purposes of caching. Each block is marked as either valid or not valid in order to indicate whether the data needs to be loaded from the server or whether it is available locally. In addition, however, some invalid blocks may contain an internal indicator (a rectangle not necessarily aligned with the node's own block) showing which part of it is actually available locally after all and thus does not have to be loaded.

## 6.3 Disk-Based Caching

An obvious limitation of the memory-only approach is the maximum amount of space that can be utilized for local data storage. This approach is the only one available when the client runs on a platform that does not allow access to its secondary memory (e.g., disks). In a Java-based platform, this is usually the case when the client runs as an applet, a lightweight variant of a downloadable executable code with minimal requirements in terms of management, security privileges, etc. It is certainly easier for a user to run the code in such environment, however he or she gives up access to resources that, if available, could significantly increase the performance of the system. This environment may also be the only one available on smaller devices that do not have any secondary memory such as disks and only offer a limited amount of battery-powered RAM. Examples of such devices include various PDAs and similar handheld platforms.

One way access privileges to additional system resources can be utilized to improve the performance is to store some data on the local disk. While the disk operations are naturally slower than operations in the main memory, they are faster than fetching data over the network. Thus, it is beneficial to store all data that is downloaded from the server into the disk-based cache. Then, when the data has to be released from the main memory but later needs to be reloaded, the system can fetch the data from the disk cache rather than re-request the data from the data server. The amount of time that data should be cached on the disk naturally depends on the specific application. For instance, if the data in the database is constantly changing, a fresh reload may be needed more often. In such a case, it does not actually matter whether the data is locally stored in the main memory or on the disk, a forced reload may be needed every predetermined period of time. However, if the data in the database is relatively stable, then reloads do not need to be performed too often and in some cases perhaps never. This would allow the local client to eventually develop a cache as large as the source database (or as large as feasible given the disk space and performance of the hardware, we assume that the central database can manage large amounts of data more efficiently than the lightweight client).

Thus, on platforms where this arrangement is feasible, we can deploy two layers of cache. One layer is based in main memory whose capacity would be limited to a safe amount we know the platform can manage without excessive memory management and garbage collection overhead. This safe capacity can be associated with specific platforms in advance, or a safe capacity can be estimated upon launch of the client during its initiation. The second caching layer would have theoretically unlimited capacity but in practice we would still want to impose some restrictions. The exact amount of disk space to which the application would be restricted depends on many factors including remaining free disk space, speed of the disk access vs. speed of the network access, frequency of data updates, etc. Generally however, we can utilize two or three platform profiles that would define this limit in a way suitable for all devices that fall into its respective category. The very basic categories would probably include one entry for small (wireless) devices and another for PC-like platforms with faster network connection.

## 6.4 Disk-swapping in Local Caching

One way to utilize the disk space to gain access to more local storage is simply to rely on the operating system's or JVM's capability to provide more main memory by swapping memory pages onto the disk. However, this is a general purpose mechanism for allowing processes to access more memory than what is physically available. This approach is not optimized to store structured database data and spatial data in particular. We have not pursued this option as it is obvious that the next method is more efficient.

## 6.5 DBMS-driven Local Caching

Instead of letting the main program store more data in the memory using standard memory allocation functions and letting the operating system to swap on disk as necessary, we investigated utilization of tools specifically designed to store data by accessing disk when necessary.

Various DBMSs have been developed for many platforms and small-footprint products are available even for handheld devices. However, only few of these products have support for efficient handling of spatial data. We have identified two such products that are easily available, popular and free — MySQL and PostgreSQL. Using a DBMS-driven local caching is a special case of methods discussed in more detail in Chapter 7.

## 6.6 Locally Cloned Servers

The above methods discuss storing parts of the data on the local machine, usually small fraction of the whole dataset representing the parts that were used recently. The central data server still serves as a master data source and clients store data only temporarily, for their own use and the amount of data stored is typically much smaller compared to the data available on the server.

A different approach has been investigated in [82]. There, the authors look into automatically creating a local copy of the server content. This eventually turns the local machine into a stand-alone platform where queries and visualization can be performed without further network operations. Such a server clone can then be used as a server for other clients and data modifications performed on this server can be propagated back into the original source or parent of this clone. As the data on a single server propagates to other locations by creating clones that can be used interchangeably, a peer-to-peer network of spatial servers is in effect built. Clients to such a system can then connect to any of the participating peers. Further optimization then allows clients to select a more efficient server to connect to. One time clients can even become servers in this infrastructure themselves. Obviously, the purpose of this solution is primarily dissemination of the complete database data across multiple network nodes for redundancy and load balancing purposes. Our work focuses on allowing thin clients to communicate with data servers, whether they are isolated or participate in such a peer-to-peer system.

# Chapter 7

# Utilizing Auxiliary Servers

Development of internet technologies has introduced various methods for utilization of additional servers to improve performance for end users who connect to external servers using common internet protocols (such as HTTP). One of the first and most popular methods is caching. Caching can be implemented directly within the end user's browser, or it can also be implemented within the user's or ISP's[1] network, on the gateway (proxy) between the network and the outside internet. In the latter case, the same cache can be shared among several users. Additionally, a cache can attempt to predict what files the user may want to download in the future and it can pre-fetch the data accordingly. For instance, in the case of the HTTP protocol, the caching server can automatically follow all hyperlinks listed on the web site that the user requested. This obviously speeds up serving up of this data significantly if the

[1]ISP = Internet Service Provider, e.g. a phone or a cable company

user actually requests the pre-fetched data later. The term 'caching proxy' has been established for servers that provide such functionality.

Obviously, the reason for introducing these proxy servers between the client and the internet is because the responsiveness of the proxy server with respect to the end user's browser is much higher than if the data was requested directly from the original host. This is due to both higher network speed between the client and the proxy server compared to the network speed between the client and the original host. Another factor can possibly be lower load and higher responsiveness of the proxy server since it only handles traffic for a few users and therefore can process requests more efficiently.

In the context of spatial databases, some services will naturally be provided by large corporations who made proper investments into their hardware equipment and can handle direct connections of individual users in large numbers. However, for smaller outfits, hosting a spatial database is much more complex task than hosting a simple web server as many do today. This is because the amount of traffic generated by browsing a spatial database can be so much larger.

In case of these smaller outfits, we expect that more often than not the spatial data provider will have a working relationship with the individual users. For instance, a spatial database may be set up by a rescue agency and the services provided by the server will be utilized by individual rescuers carrying laptops and other mobile devices. Or, a real estate maps will be provided by a real estate agency and utilized wirelessly by agents in the field.

In such scenarios, whether because the dynamic situation does not allow the time to set up a robust hardware system or because the spatial server is run by a smaller entity, we cannot assume that the central spatial server is going to be a well established high power machine hosted in a professional environment. It is possible that an ad-hoc set up may be needed on a short notice. In this case, offloading some of the work away from the central server is an important prerequisite for an efficiently working system. A real-life scenario when a spatial database technology may be used in a rescue operation is shown in Figure 7.1.

The solution is to bring a small server or a server-like device closer to the end clients and use it for certain types of proxy services. The rest of this section discusses how various methods introducing additional hardware could improve the experience of end users by speeding up the system. When a small spatial proxy is added in the above sample scenario, the resulting hierarchy looks as shown in Figure 7.2.

As indicated above, these smaller servers supporting the system may need to be affordable, easy to transport, cheap to run. For some deployments, it could be beneficial to create single purpose “black boxes” that run an operating system and application underneath exposing only a basic user interface. This is similar to routers, printers, firewalls and other network devices.

While there are several spatial DBMSs on the market, we have identified two that we believe would be uniquely advantaged for this role — MySQL and PostgreSQL.

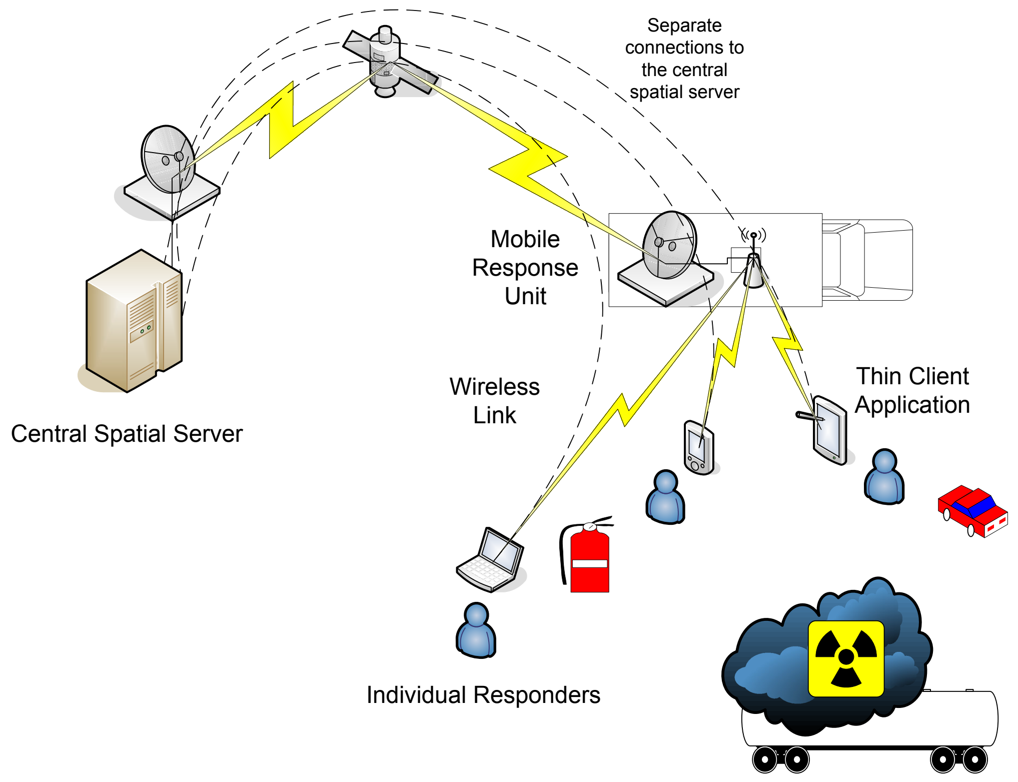


Figure 7.1: In an emergency situation, individual responders in the field communicate directly with the central server. Such communication can be facilitated by relatively slow wireless carrier data networks or it can be provided through a wireless LAN linked with the central server via a satellite or other limited-bandwidth link.

Both are very popular, well understood with a small footprint, operate on many hardware platforms, and can be used at no charge and contain some level of support for spatial operations[2].

[2]MySQL supports spatial operations in versions 4.0 and newer.

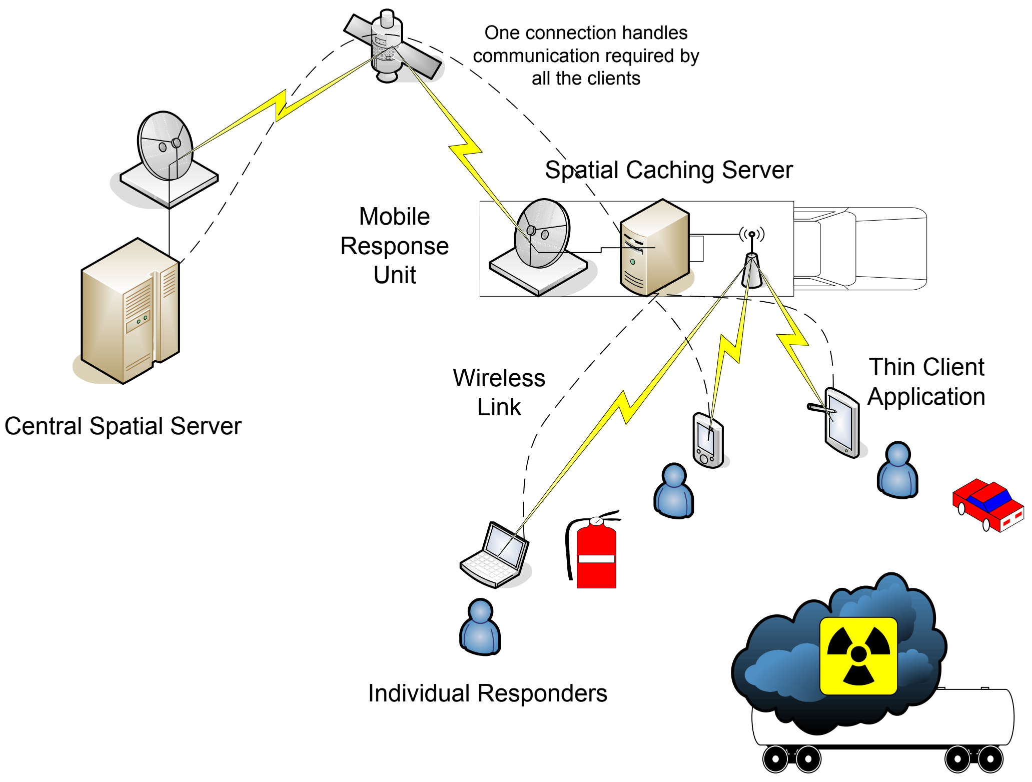


Figure 7.2: Emergency response service deployed a mobile unit in support of the operations. This unit can cover the area with a fast wireless network access and provide a proxy service for spatial operations. Individual responders can utilize the applications on their mobile devices more efficiently.

## 7.1 Using External Data Storage

As discussed, sometimes the clients can be hosted on very small devices, small in terms of size as well as performance. The internal caching method can produce only limited improvements. However, it still may not be necessary to go back to the off-site main spatial server for each screen update. If the client is run from within a network under the same jurisdiction (e.g., not a mobile device over a ISP connection but a small/mobile device hosted within the customer's own network), it is possible to offload the functionality normally provided by the internal caching method to a third server. Such a server would be hosted within the same network as the client devices and thus would be easily accessible to the clients. On the other hand, such a server would be powerful enough to store significant amounts of spatial data so that it could process most queries issued by the client for the purposes of visual map browsing.

As discussed above, the types of queries that such server needs to be able to process are rather simple so even databases with basic spatial support can be considered. In our research we have investigated two commonly available database engines, both SQL-based, that have some sort of spatial support — MySQL and PostgreSQL. These engines' support for spatial operations is rather limited, especially in comparison with SAND and other more mature systems. However, the level of spatial data support that they provide is sufficient. An added benefit is that these systems are immensely popular and their installation and management would be easy for many customers' IT staff.

## 7.2 Static Proxy

In some cases, the main spatial server provider and the individual users of this database are from within the same organization or these organizations collaborate closely. If this is the case and the spatial data is rather static (i.e., updates in the database are not done frequently), it may be feasible to perform a one-time step of copying all the spatial data stored in the main spatial database onto the auxiliary database running on the proxy server. In such scenario, the auxiliary database needs to be preloaded with the spatial data from the central SAND server when the system is being installed as well as possibly periodically after that. The frequency would depend on how often the data on the central server changes. This approach is especially effective if updates on the central spatial server are performed in regular intervals rather than dynamically. For instance, a new data set may be released once a month or once a year instead of applying partial updates continuously.

Since the complete valid map resides on the proxy server, there is no need for the client to ever connect to the central spatial server for window queries. There is also no need for the proxy server to talk to the spatial server, to receive updates or for any other reason. Therefore, the only traffic generated by this scheme involves the SAND Internet Browser clients communicating with both the central spatial server (e.g., SAND server) and the SQL proxy server. This situation is outlined in Figure 7.3.

## 7.3 Dynamic Proxy

In some cases, the amount of data stored on the central server would overwhelm even a normal server-level machine. Or, the data on the central server gets updated continuously and any information stored on the server may potentially only be valid for a short period of time. In such scenarios, preloading the proxy server with all

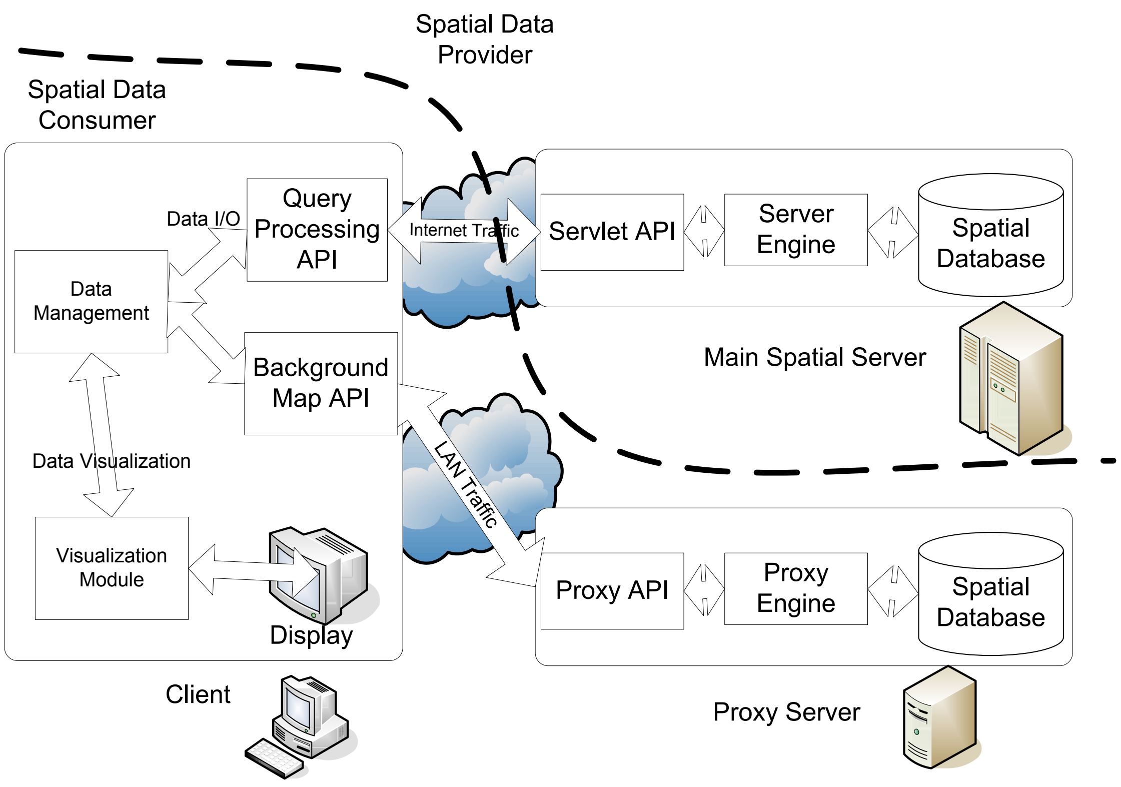


Figure 7.3: Static Proxy — The proxy server is pre-loaded with all the spatial data from the central spatial server during its setup. The client submits complex spatial queries to the central server but retrieves all the spatial data required for visualization from the proxy.

the spatial data from the main spatial server is not possible and/or useful. For this situation, we have developed a design that involves deploying the proxy server with no data preloaded on it. As the individual clients start working with the data, they still go directly to the central spatial server to get results for custom queries and to the proxy server to get results of window queries. This time however, the necessary data may or may not be available on the proxy server. If the data is available, it is sent back to the client immediately. If the data is not available, the proxy connects to the central spatial server, retrieves the necessary data and stores it in its database.

Once this is finished, it evaluates the window query locally. Since the data was just loaded, the server retrieves all the data successfully and sends it back to the client. The layout of this scenario is illustrated in Figure 7.4. Since the dynamic proxy loads the data from the central spatial server on as-needed basis, it is not a problem if some data is not available locally. The proxy can utilize this to drop data when necessary, e.g., to keep the amount of data stored locally under the a prescribed limit or to assure that the data served is not older than a certain predetermined age.

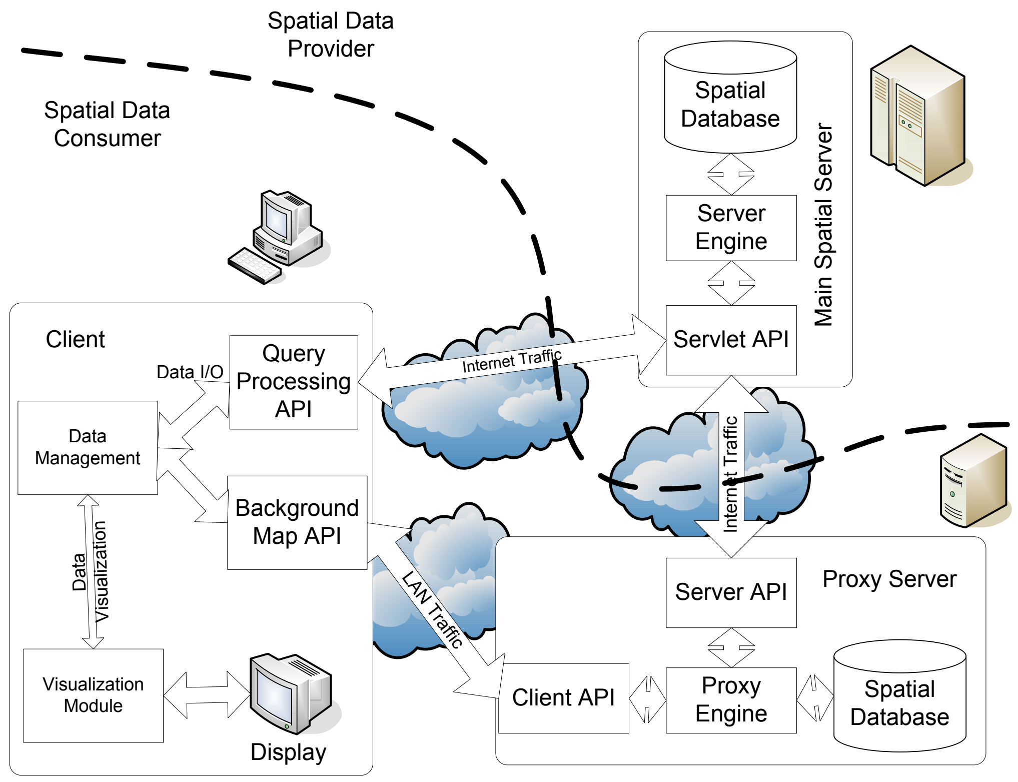


Figure 7.4: Dynamic Proxy — The proxy server is installed with no data on it initially. It connects to the central spatial server and if a request comes from a client for data not available locally, the proxy retrieves the data from the central server, caches it locally, and sends it back to the client.

When data needs to be dropped would depend on the particular scenario. Furthermore, the data release may be triggered by the proxy itself, or, it can be triggered by the central spatial server. For instance, the proxy may decide to drop the least recently used data to maintain a predetermined maximum total amount of data stored. Or it can drop data older than a predetermined threshold based on the properties of the data on the central database. For instance, in case of weather information, the data may be considered expired after 30 minutes. So the proxy could periodically drop all data older than 30 minutes and thus force a fresh download the next time a client needs the same segment of the database.

Data invalidation driven by the spatial server would be triggered by changes on the main spatial server. When a part of the central database is updated, the server would instruct its proxy servers to drop the corresponding part of the database from their local repository. Alternatively, it could instruct the proxy servers to refresh the data immediately instead of waiting for the first time that a client requests data from the dropped area. The communication between the individual elements of the solution is outlined in Figure 7.5.

## 7.4 Implementation Details

The SAND Internet Browser running on clients is implemented in Java and its connection with the external servers is facilitated via Java Database Connectors (JDBC) modules provided by the respective database vendors (the specifics related to indi-

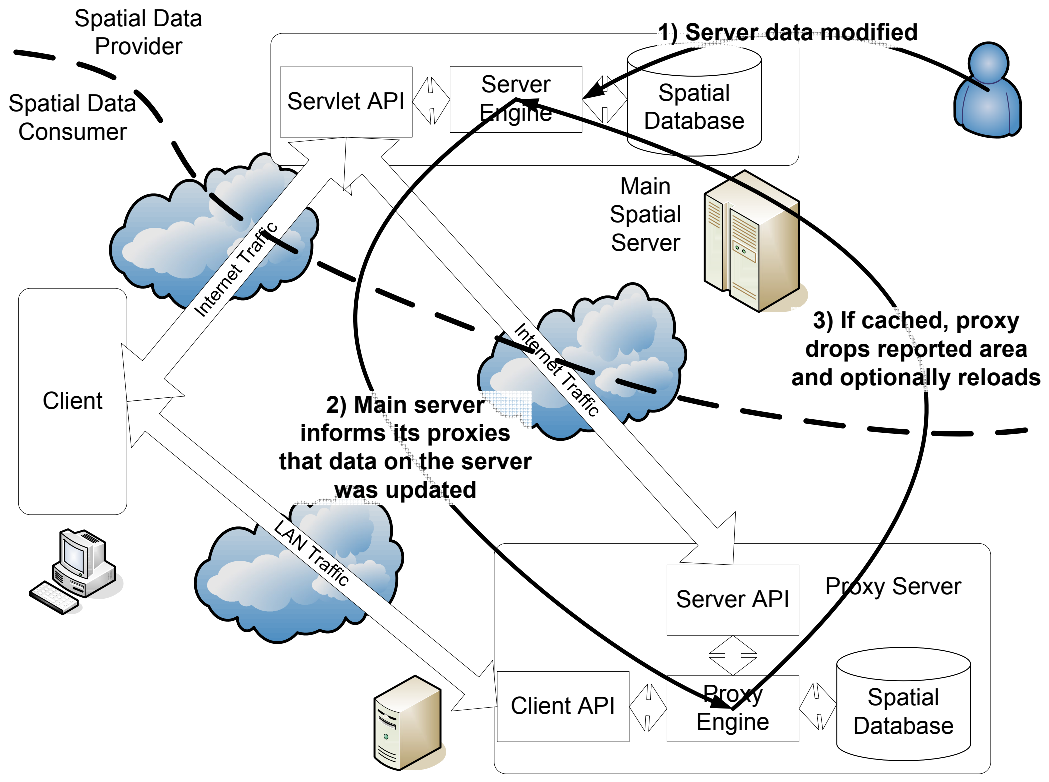


Figure 7.5: Diagram showing how database updates performed by the spatial data provider get propagated to individual proxy servers

vidual DBMSs are discussed in Section 8.5).

The external servers to be used are specified on startup of the client using a uniform resource locator (URL) that defines the database type (vendor), username and passwords required to authenticate access to the database as well as the server name and the database within the DBMS on that server to use.

For PostgreSQL, the external data source would be identified by a URL such as:

```
postgresql://username:password@servername/database
```

For MySQL, the URL would be similar:

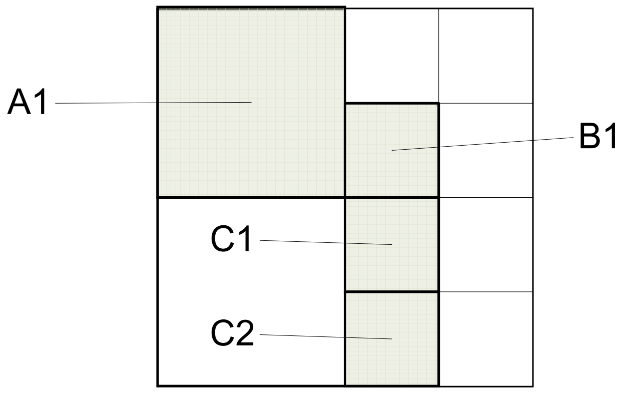
```
mysql://username:password@servername/database
```

SAND Proxy, our implementation of the proxy server outlined in general above, is a combination of two modules. The first is an off-the-shelf SQL database responsible for spatial data storage and window query handling. The second is a Java module responsible for communication with the clients and, in case of the dynamic proxy, with the SAND server that performs the role of the central spatial database. In addition, this module also maintains information about which parts of the SQL database are currently valid (i.e., which parts fully mirror the content of the central SAND database).

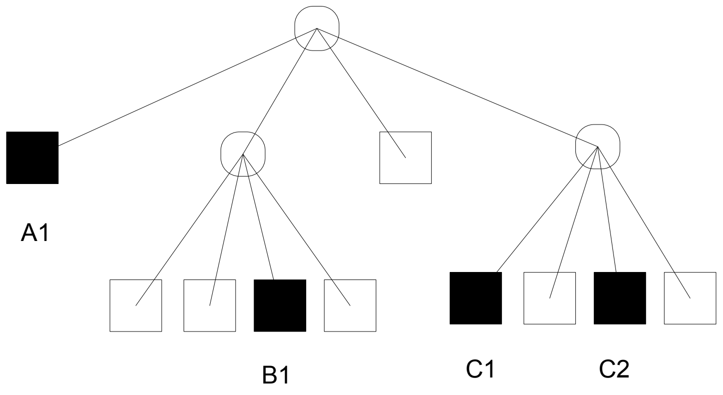
The first module is implemented through one of the supported SQL databases described in Section 1.1. This part is responsible for storing the bulk spatial data and for efficiently responding to window queries. It is important to note that this SQL database does not have any information regarding what data it contains compared to the content of the main spatial server. Therefore, the information about this relationship needs to be managed elsewhere.

We have created a second database implemented within the Java module of the SAND proxy to maintain the information about which area of the "world" that is stored in the central database is covered in the local SQL database and which is not. The SAND Proxy utilizes the Region Quadtree [72] data structure to manage this information. The problem of determining which areas of the world are represented in the SQL database translates into evaluating window queries on this data structure.

The Region Quadtree allows the SAND Proxy to identify quickly and efficiently which part of the main database is available through the local SQL database. Figures 7.6a and 7.6b indicate how the region quadtree complements the SQL database to manage the cached data.



(a)



(b)

Figure 7.6: A region quadtree indicates which areas of the world are stored within the SQL database. The areas present in the local SQL are indicated by darker blocks in Figure 7.6a. This information is represented in the region quadtree, the covered area shown in Figure 7.6a corresponds to the region quadtree shown in Figure 7.6b.

When the proxy server is first started, the auxiliary SQL database is empty and the region quadtree is correspondingly all 'white'. As the clients start connecting and requesting spatial data, the proxy server initially forwards these requests to the central spatial data server as it does not store the required information locally yet. Once the data arrives over the network back to the proxy server, the Java code in the application layer fetches the data from the communication layer and inserts it into the database through its JDBC connection. Once the data is stored in the database, it means that the gaps in the coverage are filled. Then, the local database can be queried directly and the result is then returned back the respective SAND Internet Browser clients.

For any query window $R$, some data overlapping the window may already be available locally and some may not be. We see that calculating the exact difference of $R$ minus the area for which the data is available locally would produce a sequence of orthogonal polygons. However, such a sequence is expensive to calculate on the proxy side and it would also be expensive to find out the overlapping data on the spatial server side (expensive especially compared to a simple window query). Therefore, for every window query $R$, we first test whether the data overlapping $R$ is available locally in full by recursively traversing the region quadtree. If all the data for $R$ is fully available, no download from the central server is needed. The local database can be used to fetch all the overlapping objects and the resulting data stream can be sent back to the client. If the region quadtree reports that some data overlapping $R$ is missing in the local database, then a download of all the data overlapping $R$

in its entirety is requested from the server. While it will re-load some data that are already present locally, the benefit is that the overhead is much smaller, as only a single window query is submitted to the central server. Any would-be duplicates are ignored by the SQL databases as the data table structure is set up to enforce uniqueness of individual objects stored.

After the data overlapping $R$ is loaded from the server, the region quadtree is updated to mark $R$ as fully loaded. This is done through top-to-bottom insertion into the region quadtree — by recursing into all overlapping nodes, marking them black if they are fully overlapped. Or, in case of a partial overlap and unless the maximum depth was reached, the node gets divided into four children and the same operation is performed recursively. If there is still just partial overlap of $R$ and leaf node $N$ when the algorithm reaches the maximum allowed decomposition level, we mark the node as black. This assures that any subsequent window query that is simply a result of a lateral movement (i.e., a scroll operation) along the same axis as the window edge that intersects $N$ won't report missing data due to the same $N$ and cause another download request to the central server. Of course, the drawback is that the region tree reports $N$ as available in the SQL database while part of data overlapping $N$ is in fact missing. In reality, this area is very small (a fraction of the node on the region quadtree maximum depth level) and will be loaded before the data is needed — once the window $R$ moves such that it overlaps $N$ in full. This is because $N$'s white neighbors will trigger download of data overlapping $R$ thus filling the gap in $N$'s coverage as well.

Given the root node of the region quadtree $Q$ and the window query provided by the client $R$, the algorithm is then as follows:

1. traverse $Q$ to find out whether data overlapping $R$ is available locally in full
2. if some areas of $R$ are not stored locally
   (a) request the download of data overlapping $R$ from the central spatial server
   (b) insert the downloaded data to the local SQL database
3. retrieve all elements overlapping window $R$ from the local spatial database (note that at this point it is known that the data is actually in the database)
4. return all elements back to the client

This approach guarantees that the proxy is always able to provide the data requested by the client, while efficiently caching the data for future use.

## 7.5 Automatic Multi-Level Displaying

In Chapter 5 we discussed the concept of layers within a spatial data visualization tool. The most intuitive application for the data layers is assigning a single data layer to each type of objects represented by the source data set. For instance, if the source data set contains information on roads, parks, cities, rivers and water surfaces, the SAND system can be set up with five layers, each corresponding to one of the types of data stored in the database. The user then can choose to display any or all of the data sets in the same time. The display order of the individual layers is such that layers containing object with spatial extent (lakes, parks) are drawn first, one dimensional objects (roads, rivers) are displayed next. That way we minimize the number of situations when one object would substantially or completely hide another. In case of polygon overlap (e.g., a lake is a part of a park) or overlapping lines (e.g., a metro line runs under a road), the user can choose to show or hide individual layers to highlight the data of interest.

Besides this obvious application, the same layer system can be used to handle situations when the original data set contains more elements that would be feasible to display on the map in the same time. By not feasible we mean that the it is possible to choose such a viewable area or window that drawing all elements that fall within this area would either take too long or displaying of all such elements would virtually completely cover the whole window. This would make such a view pointless.

The solution SAND provides for this type of scenario is to use a layer system to store several levels of details for each data type where this excessive data density could occur. So, for the above mentioned example, we may choose to establish three levels of detail for roads, cities and rivers and only two levels for parks,and water surfaces. Thus, there would be the total of 13 layers in such a SAND deployment.

As SAND displays visible objects, it steps through the layers of each object type starting with the layer with the least amount of detail. It displays all the objects in the current layer that fall within the viewable area, and then moves to the next layer. If drawing all the elements from that level still results in acceptable data density, the elements get drawn and the algorithm repeats this step by going to the next layer. Once the loop reaches the first layer that contains too many objects to be displayed, the execution exits the loop.

This algorithm is captured in the following Java code. `sl` is an array of layers, each storing different level of detail for a given object type. The first array element (index zero) represents the layer with full details, the last layer is the most general one. `wholeView` represents the current viewable area.

```
for (int i = sl.length - 1; i >= 0; i--) {
   if (sl[i].isFeasibleToDraw(wholeView))
      sl[i].draw(wholeView);
   else
      break;
}
```

It is important to note the `isFeasibleToDraw` function. Given the current viewable area and the contents of this layer, it determines whether we want to display all the data or whether it would result in too many objects being drawn. Seemingly, this function needs to examine the data within the viewable area to decide whether it is feasible to draw all of it. However, this function is actually used in the opposite way. It needs to provide a result without necessarily having access to the data. Only depending on the result of this function, the data will get loaded from the server and displayed on the screen.

We are primarily concerned about responsiveness of the system. Therefore, this function bases its decision primarily on the amount of data that would be drawn if the current layer were to serve as a data source for drawing on this viewable area. For the purposes of this calculation, the amount of data is represented by the number of vertices used to represent the stored objects. A preset threshold on the number of vertices is used to decide whether it is feasible to draw this layer using this viewable window or not.

Obviously, if this layer stores points (e.g., cities) or line segments (e.g., roads, rivers), then there are always one or two vertices per single object respectively. When polygons are stored, the number of vertices per object varies. In this case the average number of vertices per object is calculated for the whole dataset and this ratio is used to approximate the number of vertices for any given viewable area given the number of polygons that overlap it. The results of this approach can be skewed if there were large discrepancies between the number of vertices for represented polygons. However, practical experience with actual real datasets such as lakes, states, countries or parks showed that this method works adequately.

Similarly, if the data that falls within the viewable area is not available on the client, it is unknown how many objects would be drawn on the map. Initially, no data is available on the client but after a while some data is already cached and some may still be missing or might have been dropped in the past. So the content of the viewable area may be partially known. By using the spatial representation of the data, we find all the quadtree nodes that overlap the viewable area. Black nodes (those that have data available locally) give us directly the number of objects stored in them. For white areas we approximate the number of vertices expected to be drawn in them by multiplying their area by the global vertex per area rate. After we add up all the vertex counts (actual and approximate) for all the blocks that overlap the viewable area, we can compare with the preset threshold and the result is the response from the `isFeasibleToDraw` function.

Finally, it should be noted that when the source data is being pre-processed and distributed into the individual detail layers, in order to maximize efficiency, we need to make sure that the more detailed layer does not contain data already represented in the less detailed layer. This is because as we saw in the algorithm, the more general object will have already been drawn in the previous iterations of the loop.

# Chapter 8

# Building Combined Solutions

This section describes how the individual building blocks presented previously can be combined together to build a complete spatial database visualization solution. Results of experiments are given that provide guidelines for selection of appropriate designs given specific deployment scenarios.

## 8.1 Modular Design and Chaining

While different host types may be used to cache spatial data, their functionality is similar. Their goal is to store the data that have passed through up to their efficient capacity. In our design we do not attempt any collaborated pre-fetching where several different proxies would fetch pieces of data from the central database. We are not looking at keeping a global plan as to what piece of information is stored on which participating node. This has been looked at in [83]. In our case, the clients or proxies are stacked on top of each other, where the node closest to the actual displaying client has the smallest capacity and usually stores a subset of data of its successor in the chain. The further in the chain that we go from the client, the more data and processing power the machine within the chain has.

This is because when a client requires a certain data range and cannot find this information locally, it sends the request to the next cache/proxy node. If the data is available there, it is served. If it's not available there, the cache/proxy requests the same data further up the chain. This process repeats until the data is reached, in the worst case in the main spatial data server. Once the data is reached, it is sent back the same way the requests came, i.e. all caches/proxies on the way between the client and the successful data repository will get the chance to store the data as well. Since the layers closer to the client have typically smaller capacity, they would usually have to drop some of the data first and thus end up storing subsets of data available on the proxy. This proxy hierarchy is outlined in Figure 8.1. Of course, what data can be expected to be stored on the proxy becomes less clear once the proxy serves multiple clients. In such a case, the proxy may get overwhelmed by requests from another client in such a way that it is forced to drop all data loaded for our client. In this case, our client may still hold some data while the proxy no longer does.

Regardless of the type of platform managing the data, the data is always stored in a spatial data structure (e.g., a quadtree). The main data server runs a full-blown

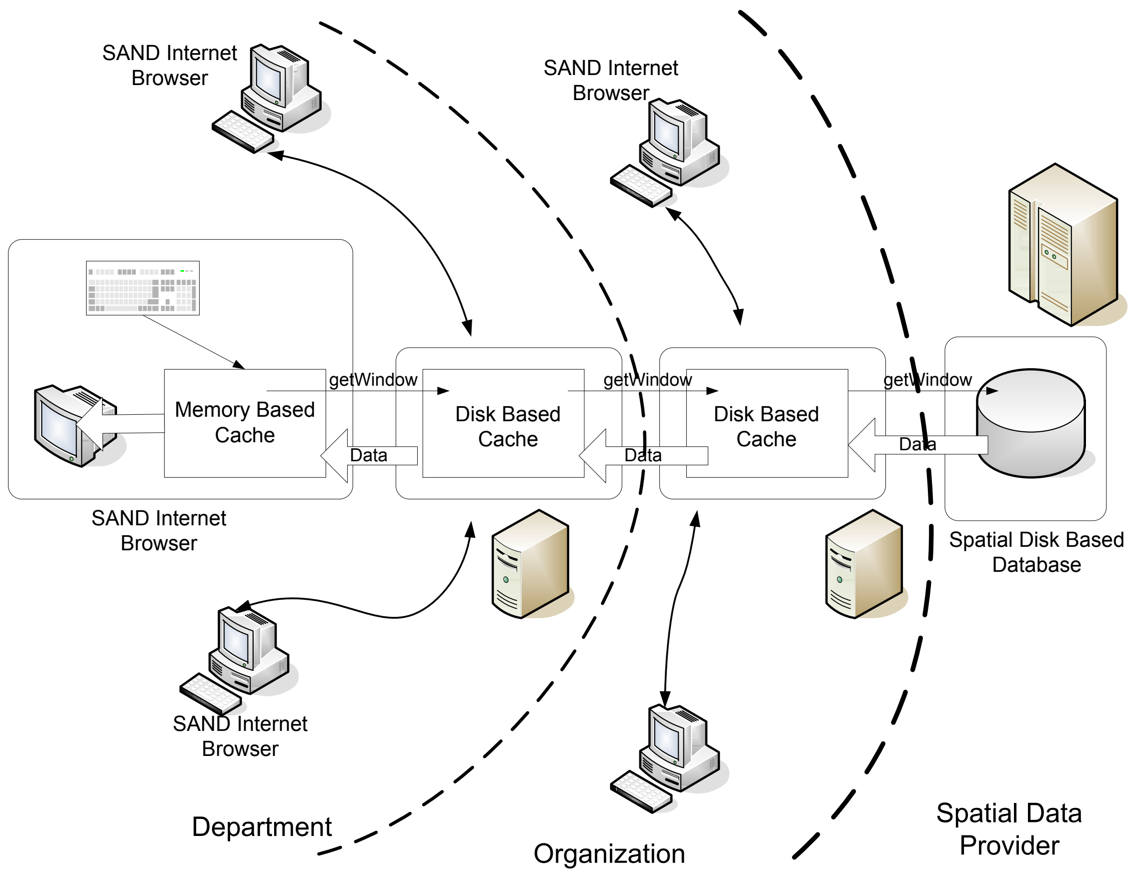


Figure 8.1: SAND Internet Browser and proxies chained together

spatially-enabled DBMS. The proxies and clients however only perform a subset of operations of a normal DBMS in order to support the limited functionality required by this layered system of caches/proxies.

This layered system is only used for base map visualization, it is not used to evaluate queries. Arbitrary query evaluation would basically involve installing the whole DBMS again closer to the client. Additionally, while the base map is being used repetitively and possibly by many users, queries tend to be unique and non-repeating. Thus, there is a much smaller chance that this caching stack could improve the query evaluation performance.

The common interface for nodes participating in the stacked caching system turns out to be very simple:

*implements:* `getArea(Rectangle area)`

*requires:* `getArea(Rectangle area)`

This means that each participant in the infrastructure needs to be able to perform a remote procedure call (RPC) representing a window query on its parent within the hierarchy (where the parent means the node closer to the main server). It also needs to provide a window query interface, i.e. allow nodes closer to the client to submit window queries (RPCs) to it.

Above we have shown that individual computing platforms can be linked together to create a chain of caching proxies that link the client's visualization module with the central spatial database. Not all computers within this chain need to employ the same caching method. They only need to implement the above interface. The actual implementation can vary based on the hardware parameters of that platform as well as other factors. However, even within each computer, the individual caching methods do not need to be used in an isolated fashion. The caching concept can be generalized to involve an arbitrary number of caching layers that can be stacked on top of each other in the order of speed in which they are able to serve the content. Many times, the speed of delivery is inversely proportional to the volume of data any given layer can store efficiently or at all.

Note that accessing the central data server can be considered within the framework of such a layer as well, and it would be the last and slowest layer that however always succeeds (never generates a page fault). So we see that it does not matter whether the data served by any given layer is stored locally or in a remote location. Thus, this concept allows us to generalize the caching into multi-server setups, or even a peer-to-peer environment. All the client needs to know is in which order it should turn to individual data providing layers. Note that the border between data cache and data server is fuzzy as individual clients can share caches on servers closer to them than the original server, in which case such caches would actually serve as sort of proxies in such environment.

For instance, imagine an environment where multiple clients contain their own internal memory cache as the first data provider layer, in which case the next layer is a disk cache stored locally on the client's platform. In other words, the next layer is the "proxy" server hosted within the same location (in terms of local area network). This proxy server can be shared with other clients so even if this client never accessed certain data from the server, another client might have. If so, it facilitates faster loading all other clients that utilize the same proxy server. Finally, as the last resort, the last data provider layer is the main spatial data server.

## 8.2 Chaining in SAND Environment

Above we outlined the general principle of the proxy chain and showed that each node only needs to implement the common interface while the actual implementation is irrelevant to the other nodes. Examining this arrangement in more detail, we see that two nodes in the chain are in a special position. The ends of this chain correspond to the visualization module of the clients and the central spatial database. They do not need to implement half of the common interface. The visualization module does not serve as a data source for any other node in the chain and so it does not need to implement the window query service. Looking at the other end of the chain, the central database that serves as the ultimate data source does not need to query anywhere else for additional information and so it does not need to implement the *getArea* (see page 106) callout function.

From the global perspective, on the beginning of the chain is the data consumer, the visualization module of SAND Internet Browser. It is set up to talk to a single data producer, which can be specified at run time. During compilation, nothing needs to be known besides the fact that the producer is an instance of the remote interface, i.e. that it implements the *getWindow* function as a remote procedure call. An instance of the remote interface can be either the final data producer, the central SAND database (at the end of the chain) or a proxy/caching module (a chain link in the middle). The proxy/caching module responds to *getWindow* calls from modules in front of in in the chain as well as calls the *getWindow* procedures on another instance of this interface placed behind it in the chain. Figure 8.1 illustrates a basic situation where the SAND Internet Browser and SAND server are linked through several proxy modules.

The most basic scenario is for a chain containing only two links, the SAND Internet Browser and the SAND server. This special case is shown in Figure 8.2. In this case, the SAND Internet Browser requests the spatial data directly from the SAND server every time a screen update is needed. This method may be appropriate for standalone deployments, for instance in scenarios where a mobile computer needs to be preloaded with data and taken into field where no connectivity is available. In this setup, the SAND system is no different than traditional self-contained GIS products such as ArcView [1].

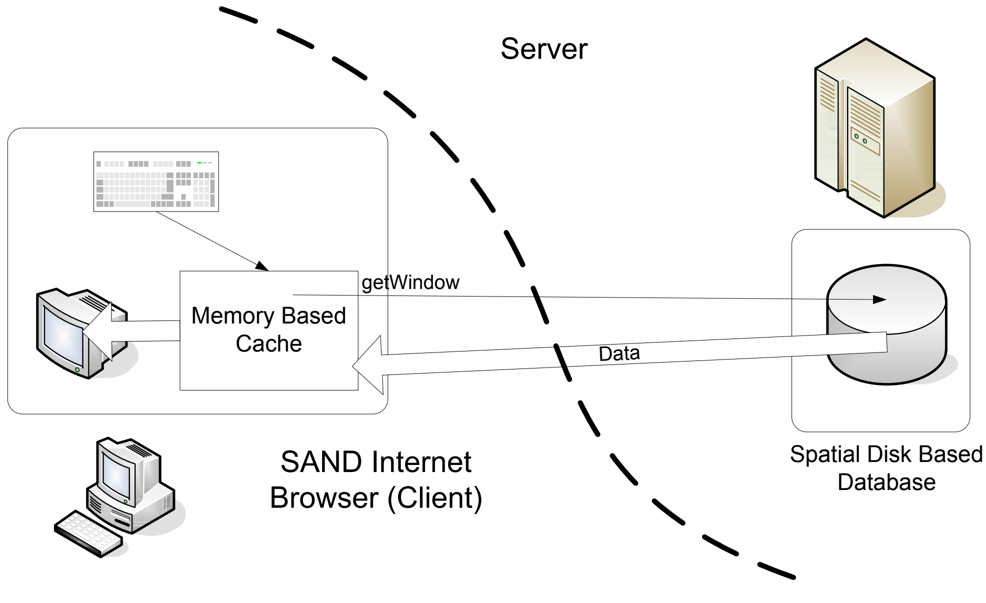


Figure 8.2: SAND Internet Browser and SAND Server chained together in a basic setup

A slightly different approach is achieved in a case where a simple memory-only proxy is implemented to run in the same space as the SAND Internet Browser itself. In this scenario, the SAND Internet Browser client stores some data locally within its own internal data structures. How much data is stored depends on the amount of resources available on the given platform. Only if the data needed is not stored in the local structures, the central SAND database is used. This approach is actually the traditional method of how the data flow was designed in the original SAND Browser [44]. This method saves some calls to the database by allowing the client to evaluate window queries internally. While the SAND database is better suited to handle arbitrary amounts of data, somewhat larger overhead of the database calls and high enough speed of the internal method (for limited amounts of data) makes the usage of the internal cache helpful. Additionally, the internal cache can store the data in ready-to-use objects, there is no need for any data conversion and memory management resulting from temporary data creation and removal.

The last method for chaining modules together with using at most two hardware platforms (one for client and one for server) involves adding a file-based data repository for the client. This module is similar to the one outlined in the previous paragraph. However, instead of storing data in the main memory, it stores data on the disk. Thus, more data can be stored locally as the main memory capacity is no longer a constraint. On the other hand, the data access is more costly. This approach is similar to utilizing a SQL database on the client as it would also store its data on the disk. The advantage is that the actual DBMS does not have to be installed which makes this method appropriate in scenarios where the client platform has a disk access but does not have DBMS available or installation of one is not feasible. Example of this scenario would be various portable computers that are either running somewhat unusual operating systems (e.g., Psion) or that are only used sporadically or on a short notice where installation of the DBMS is not possible or feasible for logistical reasons. This method can still use the memory-based caching as well or it can use the disk caching exclusively. The two scenarios are outlined in Figure 8.3.

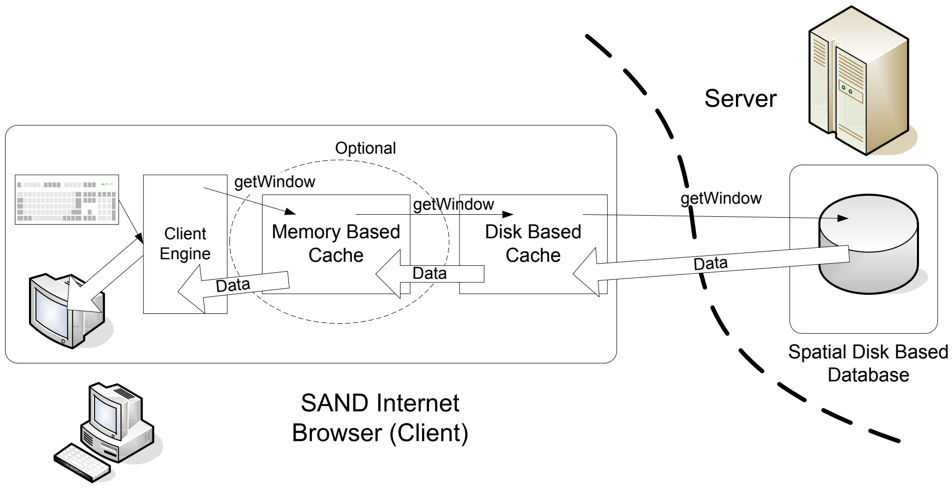


Figure 8.3: SAND Internet Browser uses disk-based caching possibly in conjunction with memory-based caching

Chaining together modules hosted on three hardware platforms allows us to benefit from instant access to data stored in client's local memory as well as from somewhat slower access to much larger amount of data stored on a dedicated server within the same LAN/WAN. In this scenario, we link together the SAND Internet Browser viewer, the memory-based caching method (both on the user's platform), the SQL-based proxy server (on a dedicated server-level platform typically within the same

LAN/WAN as the end user), and finally the SAND central server. This setup in shown in Figure 8.4.

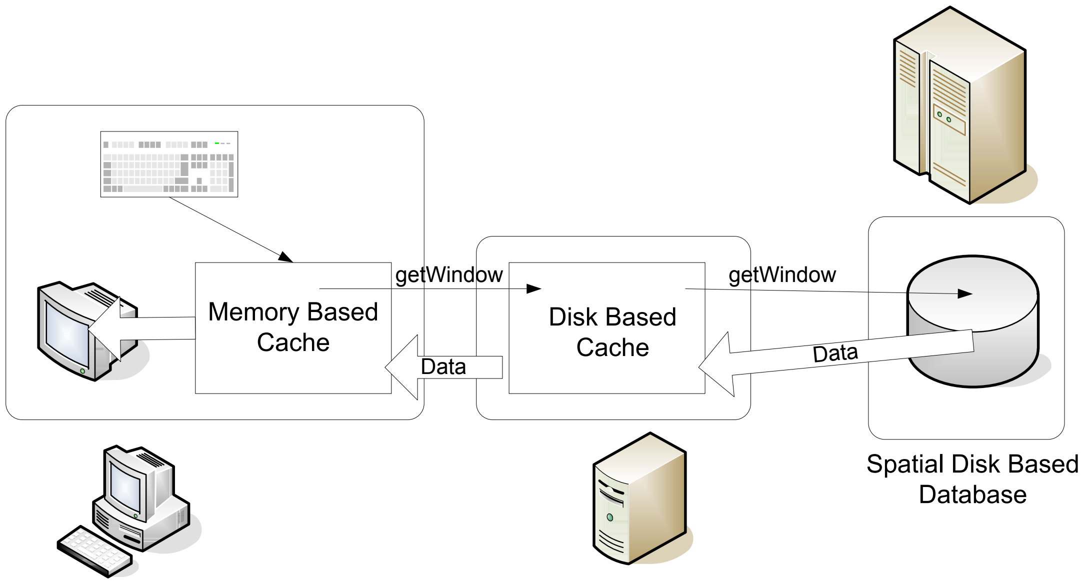


Figure 8.4: SAND Internet Browser, a proxy and the central server in a three-platform setup

This design allows for unlimited chaining, i.e., an arbitrary number of nodes can be linked together to create a data channel between the end client and the master server. However, there are only a few distinct types of caching proxies, most typical include:

1. The fast but resource-wise limited client's internal memory.

2. The central database that contains all the data but that is slow in providing it.

3. An in-the-middle link that is faster than the central database but does not necessarily contain all the data. Although it contains more data than the internal memory, it is slower.

While in some specific scenarios it would be beneficial to utilize several proxies of the same type within each channel, typically such additional modules would introduce higher overhead without bringing much speed-up that could offset it. So we expect that a typical setup of this system would either involve just two physical nodes (the client and the server) or it will chain together three nodes — the client, the central server and a single caching proxy server between them.

The proxy server platform can be chosen and various parameters adjusted to bring its overall behavior and performance either closer to the user client or to the central database as desired. For instance, by providing the proxy server with substantial processing power, main memory and disk space we can make it work more like the central SAND database. Bringing it closer to the user's client within the network topology (or even as close as installing it on the same hardware altogether) will make this proxy module more responsive (due to less networking overhead) but may decrease the overall performance due to having access to less resources and thus caching less data.

## 8.3 Peer-to-peer Options

If the system is to be utilized by a single user at a time, there is no need to link several same-type proxy servers together to improve responsiveness. This assertion assumes the data hand-off speed and networking delays are the bottleneck rather than the disk and processor of the server being fully utilized. However, in situations where multiple users are trying to access data through a given proxy server, it may be beneficial to link such servers in parallel fashion to split the user accesses among several servers and thus improve performance in such multi-user environment. These proxy servers can either act completely independently or they can share the actual database and only offer their processing power to run the application code that accesses the database.

An alternative set up that does not involve dedicated proxy servers would be a peer-to-peer environment. If there are multiple clients running nearby, they can register each other as potential data sources for individual data layers. Then it would be more efficient to attempt to fetch the data from the peer rather than going back to the main server. The exact collaboration strategies naturally depend on specifics of the platforms involved. For instance, a proxy server would actually load and cache any data that was requested from it but not found. In a peer-to-peer environment, the peer client could be queried for data but probably should not be forced to load the data if it is not available. This is because the client has limited resources and should primarily contain only data that it needs itself, not data that other client needs at this time. Peer-to-peer spatial data sharing concept has been explored in [80], however in that work the author focused primarily on cloning the 'heavy' primary data servers. Our focus is more on cache/proxy sharing and more lightweight collaboration.

Finally, the multiple server approach can be also used to relieve the load off of the central server in scenarios where high server load would become the bottleneck.

In this case, multiple data servers could be installed and provided to clients, and the clients would then utilize one of the servers spreading the load across all of them. The strategies for choosing the server can range from round robin or random to more sophisticated ones that measure current load and connect to server that is least busy. Note that this switching method does not need to be implemented on the application level, as appropriate hardware devices exist that handle this service automatically and seamlessly in such a way that they basically build one virtual superserver. In other words, the clients are not even aware that several different data servers are ready to handle the incoming requests.

The individual multi-server setups are illustrated in Figures 8.5–8.8.

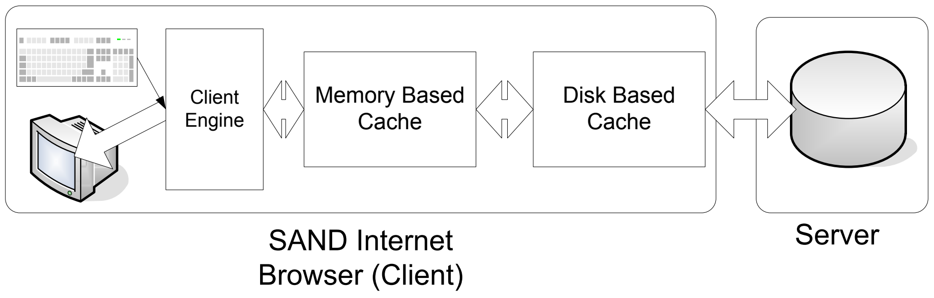


Figure 8.5: A client running the SAND Internet Browser utilizes local memory as well as the local disk for internal caching and connects directly to the central spatial server.

## 8.4 Caching vs. Proxying

In web terminology, terms such as caching and proxying have their long accepted meanings. Browser caching refers to an operation where the client, for some period of

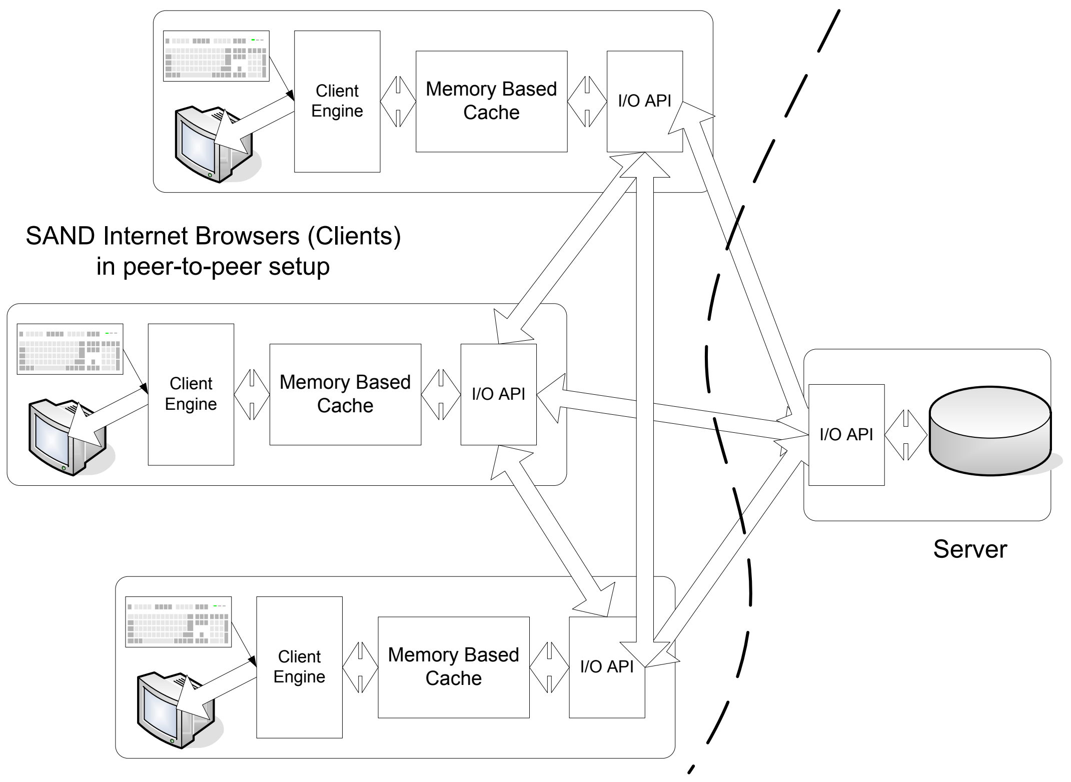


Figure 8.6: Individual SAND clients utilize each other in peer-to-peer fashion for localized caching. They connect directly to the central server if necessary.

time, locally stores data that it has previously received and displayed to the user. In this way, the data can be quickly retrieved if a request for the same data comes again in the near future. The type of data that is handled in this way includes HTML code describing the content of visited web pages as well as images, Java applets, and other objects to which the visited pages may refer.

A (caching) web proxy is a server application that is placed between the user's browser and the internet. This application receives all the requests from browsers on the local network. If a request comes through for data that the proxy does not store locally, then this request is forwarded by the proxy through the internet to the original

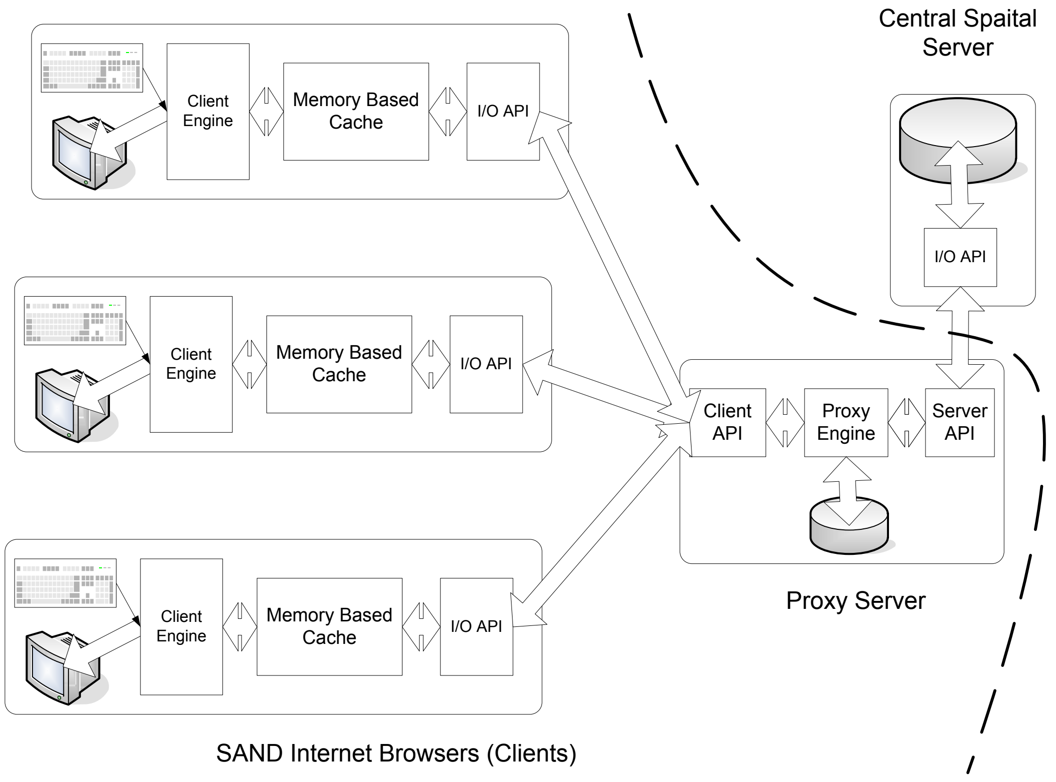


Figure 8.7: Clients utilize their own memory for internal caching and a proxy server for local caching. They access the central spatial server through the proxy as necessary.

provider of the requested content. As the data is received, the proxy again forwards it to the user's browser but it also stores the data locally. In this way, the next time one of the users utilizing this proxy server requests the same data, the proxy can respond immediately, thereby limiting any traffic resulting from the request to just the local network. This obviously makes the request resolution much faster. In essence, a caching web proxy provides the same functionality as the browser's cache but it is shared by multiple users and typically hosted from a more powerful and/or dedicated machine. Neither the browser-based cache not caching web servers typically pre-fetch any data, i.e. they do not download data from the internet which is not specifically

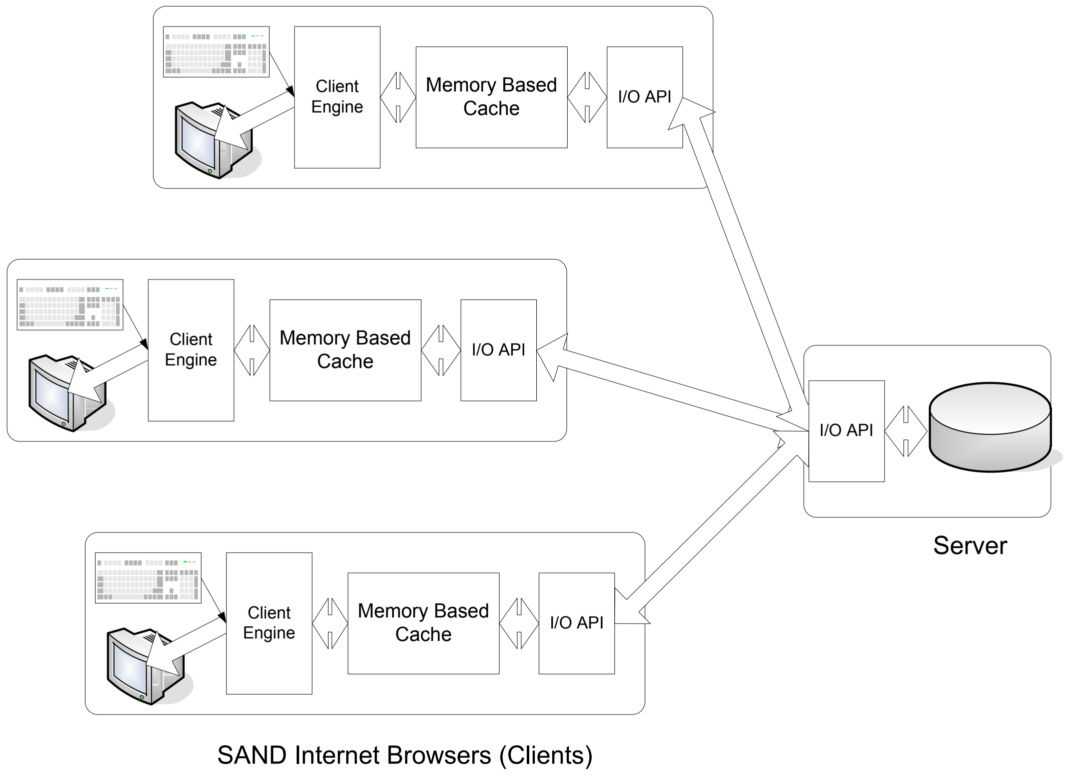


Figure 8.8: Each client uses only own memory for local caching and otherwise connects directly to the central spatial server.

asked for by users, perhaps by attempting to predict future users' requests. This is usually because many pages that users tend to visit often update their contents frequently (e.g., news sites). Obviously, in such a scenario, utilizing the proxy server is only of limited benefit.

In our spatial database scenario, we can expect that users will work with only a small number of spatial databases at a time, and that the content of these databases, or at least their base data set (e.g., road map or water surfaces layer), will be mostly static. In such a situation, it makes sense to explore the possibility to pre-load all the relevant data onto a proxy server. In this way, when the user submits a complex database query or accesses more dynamic layers, the request will be handled by the primary spatial server. When the user only needs to access the background layers (usually to display the map's background), it is feasible and more efficient to retrieve the necessary data from the locally available proxy server instead.

This local proxy server can either replace the internal client's caching altogether or it can be used as a second level cache and thus participate in chaining potential data sources together, chaining between the client on one end and the primary spatial server on the other. The proxy can also be either preloaded with data (Figure 7.3) or it can be initially empty and fill itself with data as users start to utilize the system (Figure 7.4). While the pros and cons were generally discussed above, we have also compared the performance of these approaches in our research and the results are presented below.

## 8.5 SAND Proxy Implementation

In Section 7.4 we have introduced embedding of external DBMSs into the SAND infrastructure. In this section, we outline the implementation specifics related to the individual DBMSs that we have worked with. We have employed two off-the-shelf databases that contain some level of support for spatial indexing. Specifically, we looked at popular open-source databases MySQL (version 4.1+) and PostgreSQL.

Compared to database engines that were designed primarily as a spatial database (e.g., SAND), the spatial data support in these products is only rudimentary. However, for our purpose of using them as auxiliary data repositories, the level of support is sufficient. From the two DBMSs that we work with, MySQL is the one with more limited capabilities (the spatial data extension was also added more recently). For instance, MySQL JDBC does not even support MySQL's own spatial types, so wasteful conversions from and to plain text representation are necessary. This means that all spatial objects returned by the database need to be expressed as strings, passed through JDBC into the SAND Internet Browser, where the strings needs to be parsed before the objects that they represent can be displayed. This continuous encoding/decoding taxes the performance and decreases the benefit of secondary caching based on MySQL.

PostgreSQL JDBC supports spatial types directly. This means that in response to a spatial query, the caller receives back a sequence of Java objects that implement and represent the spatial types returned. Such spatial objects include points, lines and polygons. These objects can be directly used by the drawing module of the SAND Internet Browser after the coordinates are converted from the world coordinates to their corresponding screen coordinates. There is no need for any string parsing as was the case for MySQL. The result of this benefit is much faster query processing and more significant improvement in caching using PostgreSQL.

In the assumed scenario, the main spatial database (in our case represented by SAND) contains all the available data, both spatial and non-spatial and its sophisticated database engine enables evaluation of complex queries. Installing and maintaining such a massive database is appropriate for a dedicated spatial data managing organization that has the proper hardware and human resources. However, to extend this amount of effort is generally not feasible for a consumer — a customer trying to benefit from such a data collection. This is the primary reason for investigating the client-server architecture for spatial data access in the first place. Thus, when looking for a secondary cache to be installed closer to the users' clients (possibly on customer's internal network), utilization of SAND or a similar sophisticated database would negate many benefits of deploying the system through the client-server architecture.

For the purposes of secondary level caching, no sophisticated queries are needed, only support for window queries is required. A window query refers to finding all objects in the database that intersect a given rectangle. Thus, we can successfully deploy simpler database systems such as MySQL or PostgreSQL that may have only rudimentary spatial data support but even such limited support is sufficient for such purpose. Additionally, management of such a system is much easier for the customer, especially since such secondary cache can run autonomously (as we will show below).

Objects represented in the SAND database include points, lines,[1] and polygons.

[1]The "line" data type in SAND actually represents what is elsewhere referred to as line segments. In the literature, the term "line" is often used to denote an infinite line in a geometric sense. At other times, the term "line" denotes a collection of adjacent line segments, i.e. a multi-line or a polyline.

Thus, the secondary database system needs to be able to query the data to find points included in a rectangle, lines overlapping (crossing) a rectangle, or polygons overlapping a rectangle. It turns out the support for such operations is not always available and many times the query needs to be simplified into looking for objects whose bounding box intersects a given window. This is obviously a much easier and faster query but may result in false positives. Specifically, at this time, MySQL syntax supports queries using arbitrary spatial objects (e.g., users can enter a query with respect to a polygon and a window) but such queries when executed only use the object's bounding box to calculate the results. This may surprise some users who expect to get the exact results, not results with respect to the bounding boxes. Presumably, this will be improved upon in future versions.

PostgreSQL performs calculations exactly as stated in the SQL but it does not support many of the geometric operations for many spatial type pairs. For instance, the "overlap" operation is only supported for boxes and lines, not for polygons, i.e., determining whether a line or a rectangle overlap a given window is possible, while determining whether a polygon overlaps the window is not.

Fortunately, for the purposes of visualizing objects visible within a given view, utilization of the limited overlap operation (i.e., that only works with respect to objects' bounding boxes) is sufficient. In this case, the possible extra objects will be clipped by the drawing algorithm (as will any hidden parts of the objects that do overlap the view).

The SAND Proxy module implements the design outlined in Section 7. Its purpose is to link one or more SAND Internet Browsers (clients) with the main SAND server, while optimizing the performance by decreasing the network traffic and absorbing some load off of the clients as well as the SAND server. The proxy is designed to be easy to install and to manage a lightweight component that would typically be installed within the network of entities that frequently utilize services provided by spatial servers. This proxy is meant to handle the bulk of “get window” queries that result from browsing the data, thereby allowing the main server to focus primarily on evaluating unique queries that individual users form themselves using their clients.

Since the SAND Proxy software is hosted from a server-class machine, the module has access to all resources of the computer including the disk. This differentiates it from the client software and allows it to process large amounts of data more efficiently.

We have investigated several different methods of how this module works, each method’s properties make it more suitable for a different specific usage scenario.

## 8.6 Implementation Details

The SAND Internet Browser client takes the following options from the command-line:

1. `server` — This parameter provides the client with the host name or the IP address of the central SAND server which hosts the data the client will be

working with. This option is required regardless of the proxy situation since as outlined above the proxy is not meant to completely replace the central server and some communication with the central server is always needed.

2. `proxy` — This parameter is optional and instructs the client to use the proxy specified by this parameter to speed up the client's operations. The format of this parameter follows the standard URL[2] format, i.e.:

   ```
   <protocol:>//[user@password]<host>[:port]/<database>
   ```

   The individual parts of the URI are defined as follows:

   - `protocol` — One of the supported proxy protocols — the options are postgresql, mysql, sandproxy and hybridproxy. These are discussed in greater detail below.
   - `user` — The username needed to obtain access to the proxy.
   - `password` — The password needed to obtain access to the proxy.
   - `host` — The hostname or the IP address of the proxy server.
   - `port` — The port number where the client should be connecting to on the proxy server.
   - `database` — The name of the database used to store the spatial data on the proxy server.

[2] URI — Uniform Resource Identifier [17]

The individual supported protocols provide the following functionality:

- `postgresql` — The client connects to a proxy that runs the PostgreSQL database that was preloaded with spatial data from the corresponding SAND server. As discussed above, this is an appropriate scenario for the situation that the data on the central server is relatively static. The communication between the SAND Internet Browser and the SAND Proxy is done through PostgreSQL's JDBC protocol.
- `mysql` — This case is equivalent to the one above, except that this time the database running on the proxy server is MySQL. The communication is done through MySQL's JDBC protocol.
- `sandproxy` — This protocol assumes that the proxy server is running a standard SAND spatial database and that the communication between the SAND Internet Browser and the proxy is done through the SAND protocol. Note that while the standard SAND DBMS is running on the proxy, it is only used to store spatial attributes of the central SAND server. In other words, the proxy is still a auxiliary database that works in conjunction with the central SAND server. It does not act as a complete copy of the central SAND database.
- `hybridproxy` — This is the most complex environment used in scenarios where the proxy needs to be loading data from the central database on demand. This could be the case when the data on the central server

changes frequently or when it is impractical to pre-load all the spatial data stored on the central server into the proxy (due to the volume constraints, time constraints, etc.). In this scenario, the proxy acts as a pass-through module that captures requests from the server. If it has the requested data immediately available, then it sends the data back to the client. Otherwise it requests the data from the central server, loads the data it obtains into its local database (PostgreSQL), and then it executes the client's query locally (now the data is definitely available). The proxy runs a DBMS (e.g., PostgreSQL) as well as a Java servlet that provides communication to the central SAND server (over the SAND protocol), the SAND Internet Browser client (over the SAND protocol) and to the internal PostgreSQL database (over PostgreSQL's JDBC).

## 8.7 Communication Protocol

The communication between the SAND Internet Browser client and the SAND server and between the SAND Proxy server and the central SAND server run over a TCP[3] connection. The Java servlet on the server side acts as both a simple standard web server and as a specialized SAND protocol servlet. The reason for adding the web server capability is to allow running the SAND Internet Browser client as a Java applet within standard web browsers. The security limitations of these browsers

[3]Transmission Control Protocol

result in only allowing applets to establish network connections back to the server from which they were downloaded. Thus the applet has to be downloaded from the same computer that the SAND Internet Browser server part runs on. While our servlet could be simply installed on the computer that is already running an existing web server of the organization deploying the SAND Internet Browser, attaching web server functionality to the SAND Internet Browser server part makes the deployment easier. There is no need to interact with the IT personnel and the system can be installed on any networked computer, instead of just the web server, which is valuable especially in the testing and trial phase of the deployment.

The Java servlet listens on two TCP ports and waits for a client to initiate a connection. If the client is run as an applet, the first connection would come to the port that handles the HTTP (web) traffic. The only operation supported by this simple web server is downloading of a specified file through a *GET* command [69]. These files can either be the *index.html* file that serves as a wrapper page that the applet is embedded in or either *sandjava.jar* or numerous *class* files (depending on whether *JAR* files are supported by the user's web browser). Thus ordinarily, first the HTML page would be served which would trigger another request for either the *JAR* file or several *class* files. After all this data is received by user's web browser, it is able to start the Java applet. If the client is run as an application, no download of the code is needed since the application was installed separately beforehand. Users simply start the client piece by double clicking the appropriate icon on their desktop or launch the application in any other way specific to their platform.

Once the client piece is launched (either by going to certain web site which results in downloading and running the applet or by launching the application manually), it connects back to the SAND servlet, but this time on the TCP port dedicated to SAND Internet Browser traffic. The communication is driven by the client piece, the server only responds to client's queries. The client initiates the transaction by sending a query. The query is sent in text format. It consists of the query keyword (see Table 8.1 for list of query keywords) followed by matching arguments, the number and type of arguments vary from query type to query type (and thus keyword to keyword).

Upon receiving the query, the Java servlet uses it to create a SAND-Tcl expression or script in SAND kernel native format and sends it to the kernel for evaluation or execution. The SAND kernel responds accordingly, and the response is given in text format. Naturally, the response depends on the query and can be a boolean value, a number or a string representing a value (e.g., a default color), or the whole tuple (e.g., in response to the nearest tuple query). If a script was sent to the kernel (e.g., requesting all the tuples matching some criteria), then an arbitrary number of lines can be returned by the SAND server. Passing this data over the network directly to the client in plain-text would naturally be very inefficient. Thus in case a query is submitted that returns an arbitrary number of tuples, the data stream is compressed using the standard LZW [85] algorithm at the servlet and decompressed at the client before the result is parsed. Results of queries that are known to return only a single tuple are sent uncompressed.

It is obvious that this protocol is not as optimized for carrying the SAND Internet Browser specific traffic as it could be. On the other hand, it is very flexible and easy to use which is especially valuable during the SAND Internet Browser development phase. Moreover, we do not expect more efficient encoding of the data within the protocol to improve the overall performance of the system much.

It is also obvious that if another spatial database is used instead of SAND kernel, only a simple modifications to the servlet would need to be made in order for the SAND Internet Browser to function properly. The queries sent by the client would need to be recoded into another query language, one that is native to this different spatial database. The format of the protocol used for communication between the servlet and the client will be unaffected.

The following table explains the protocol used for client-server communication. The client always initiates the communication with the server. It is done by sending a request keyword followed by a variable number of arguments to the sender. In response, the server executes the operation encoded by the request, and, if applicable, it sends the response back to the client.

| Protocol format | |
|---|---|
| Query Keyword | Explanation |
| cce | choose catalog entry — open a specified mapset |
| cdd | change working directory of SAND kernel |
| cls | close table |

| | |
|---|---|
| crr | create random relation — used to store temporary results |
| drp | remove relation (typically the temporary relation) |
| ext | exit |
| fbq | calculate results of query, insert resulting tuples into a given group |
| flt | retrieve tuples via filter |
| fst | find the first tuple from the given table |
| gad | get auto display – retrieve the information about the default coloring scheme for this table |
| gat | request list of attributes for the given group |
| gav | get attribute value — get the attribute value for a given tuple and its attribute |
| gcl | list all relations available on the server |
| gfc | get fill color — get the default fill color for a given relation |
| ggb | get group by — get all tuples with the same unit attribute as the given tuple that also match the query |
| gib | get indexed by — find all indices of a given table |
| gid | requests raster data associated with a given tuple |
| glc | get the default line color for a given group |
| gle | get the number of tuples of the given group |
| glw | get line width — get the default line width for a given table |
| gon | get the object name (singular) for the given table (e.g., road, river, etc) |
| grl | get list of relations of the given mapset |
| gsi | get the spatial index of the given group |
| gsn | get name of objects (plural) for the given table (e.g., roads, rivers, etc) |
| gst | get the spatial type of the given table (e.g., line, rectangle, etc) |
| gun | request the default group-by attribute of a given relation |
| gwr | get dimensions of the world |
| nea | get the nearest tuple to a given location |
| nxt | get the next tuple within the table |
| otb | open table |
| scg | save current group (on the server) |
| ssc | sand keep-alive message (generates periodical traffic to keep the connection alive) |
| sts | get statistics about the current session |

Table 8.1: SAND Protocol keywords

## 8.8 Evaluation

Our research explores the impact of various types of techniques of chaining different caching layers together on the performance of the solution. We investigate different scenarios and suggest ideal combinations of caching based on the types of devices used, usage model (e.g., number of users looking at the same data), network speed, and other factors.

Specifically, we have designed and implemented the following caching methods and investigated properties of SAND systems created by chaining them in various combinations:

- Client
    1. direct access — the client communicates directly with the main spatial server without any local caching
    2. local caching — the client caches data in its memory
- Proxy
    1. pre-loaded data — local SQL database is pre-loaded with all spatial data from the server

2. dynamically-loaded data — local SQL database is loaded dynamically based on the requests coming from the clients

It is important to realize that our contribution involves the design of the computational infrastructure put in place to improve the overall user experience by speeding up the system's response in comparison with other traditional approaches in given scenarios. The behavior of the whole system depends on a number of factors, many outside our reach (e.g., the network latency, number of concurrent users, or even the exact implementation of the garbage-collection algorithm in the underlying operating system or virtual machine, etc.). This also makes rigorous comparison with other existing systems that aim to serve the same goal (e.g., MapQuest) difficult as we are not able to run performance tests of both systems in identical environments. MapQuest is deployed in a professional hosting environment on powerful hardware while it is being accessed by tens, hundreds or possibly thousands of people at any given time. Our system can be tested in a lab environment where the hardware and network have different parameters. Furthermore, the target audience for an established service such as MapQuest is different than the one intended for the SAND service. For MapQuest (and similar services), the system only aims to display a map for the area that surrounds a given location. The SAND system is designed to allow the user to perform more complex spatial queries.

Therefore, the nature of the SAND system and a MapQuest-type system makes their comparison difficult. This is because they may behave slightly differently each time the experiment is run. Thus, other similar experiments or real deployment experience may produce results that would differ from ours. Nevertheless, as we will see, where we claim that one of the methods is better than the other, the performance difference is substantial enough so that the same conclusions can be drawn even when allowing for rather generous deviations.

Of course, we have tried to minimize the impact of external factors. This was done by utilizing the same hardware and software platforms for both systems, the same networking environment as well as identical data sets, queries or sequences of queries. In addition, the parameters of the server platform, the networking environment, and the type of datasets and queries that were run on them were selected to be typical for the types of deployments that we suggest would benefit from this system. The selection of test scenarios is discussed further below.

Finally, the goal of the evaluation is not to determine that one approach is better in any scenario. Instead, we aim to identify what approach is the best one for different types of deployments and provide the system administrator and user with guidelines for selecting a solution best suitable for their specific needs. Besides comparing vector-based SAND against a bitmap solution, we also deployed SAND in several different ways utilizing its modularity as described in Section 8.1.

## 8.9 Comparison with MapServer-Based Visualization

For our performance evaluation, we used TIGER datasets from the U.S. Census, specifically the street maps for states in the Mid-Atlantic region. This includes all the roads and streets in Virginia, Maryland, District of Columbia, New Jersey, and Pennsylvania. There are over 7,500,000 entries in this combined dataset. Each entry corresponds to a single line segment, each actual street may be represented by one or more line segments in the map. The total size of the data stored in the format distributed by U.S. Census is over 700MB.

The relevant datasets can be obtained from the U.S. Census Bureau in many ways including downloading through the agency's HTTP/FTP server[4]. The U.S. Census Bureau publishes the specifications of the TIGER format in [21]. ESRI converted the TIGER data into their proprietary Shapefile format [2] and offers the converted data for download on their web site[5].

U.S. Census Bureau TIGER data contains numerous layers of spatial and non-spatial data. For the purposes of this test, we only worked with the road data layer represented within the TIGER set. In ESRI's filename scheme, the names of the data files start with `tgr` followed by the five-digit county FIPS[6] code followed by the layer

[4]`ftp://www2.census.gov/geo/tiger`

[5]`http://arcdata.esri.com/data/tiger2000/tiger_download.cfm`

[6]Federal Information Processing Standards

suffix. "Line Features — roads" are stored in files with suffix `lkA`. This layer alone provides enough data to test and compare performance of different spatial mapping systems. These road segments are also moreless evenly distributed across the covered area.

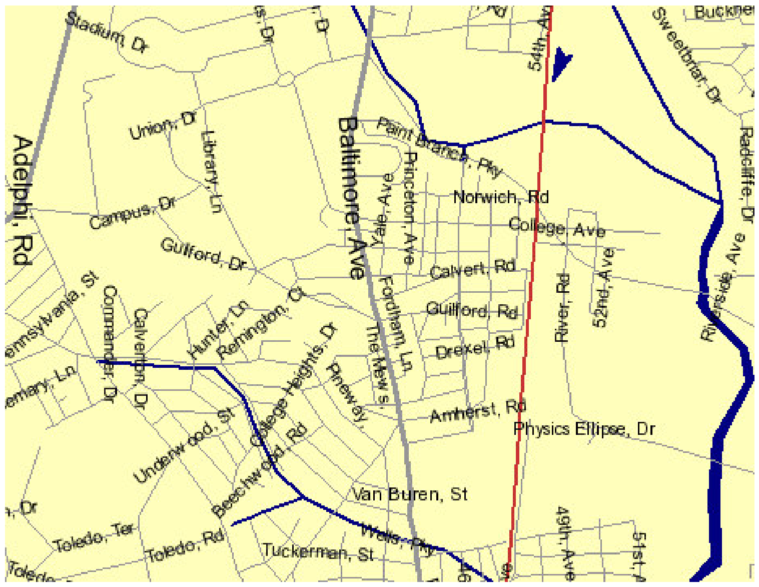
In an actual deployment, it would probably be helpful to include other data layers available in TIGER to create a background map for users' specific data sets. Such layers could include railroads, other transportation-related objects (e.g., airports), streams and rivers, parks, water surfaces (e.g., lakes, oceans), boundaries, power lines, etc. Additionally, we may choose to create several separate layers out of some of the data types available in TIGER. For instance, while all the roads in TIGER are stored together within the same layer, it may not be desirable to display all the roads all the time. This situation arises as a result of using a lower zoom factor which means that a larger area of the covered world is visible, and thus only major roads need to be displayed. Only when the user zooms in to examine a smaller area in more detail do we need to show all the roads represented in the data set. Thus, the information stored in the TIGER road data layer can be used to create several road layers, each representing roads with a different level of detail. Perhaps, on one end of the spectrum the layer would only list major highways while the other end would show even the smallest neighborhood streets.

With different levels of detail available, the mapping application can decide which layer to fetch the data from based on the current zoom level, the desired maximum number of elements to be displayed in each view, etc. The exact setup can be configured based on the requirements of each specific deployment. This adaptive system is available in SAND and is discussed in more detail in section 7.5.

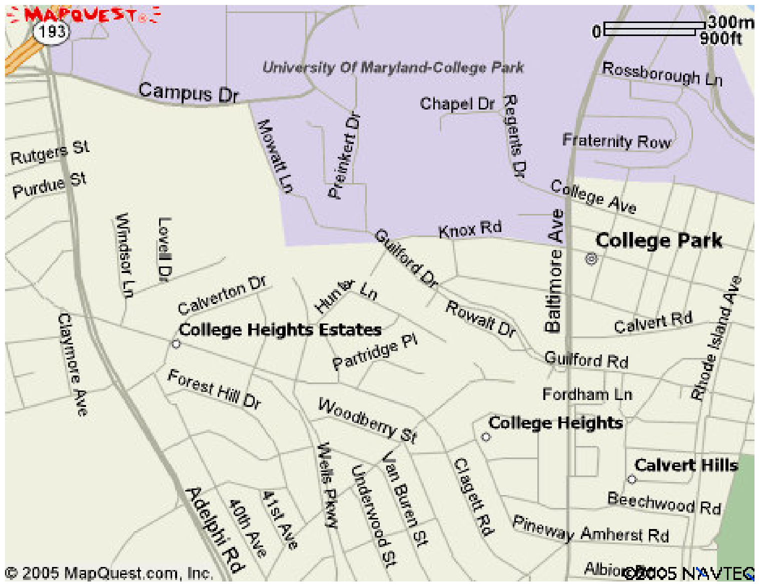
When selecting data elements (such as individual roads) for inclusion into the individual layers, we need to know the "significance" of each element. The more significant elements would be visible in the more general views while less significant elements would only be visible in more detailed levels. The meaning of "significance" depends on the type of data as well as the particular deployment scenario. For instance, for the above mentioned road layer, interstates and other highways would be typically considered the more important roads while smalled local streets would be less important. Similarly, on the park layer, national parks may considered more important while state and regional parks would be less important. For other types of data, the importance may be reflected by the spatial extent (e.g., size of a lake), shape (e.g., an object represented by a straight line may be more important than one represented by a jagged line) or total length. As we see, the importance can be deduced from the non-spatial data that accompanies the layer elements (e.g., the type of the part, the class of the road) or from the spatial properties of each element itself as well as elements around it. Selecting elements according to their non-spatial attributes is simple as a search for all elements whose non-spatial attributes match a given criteria (e.g., cities with more than $x$ residents) will produce the desired results. However, the situation gets significantly more complex when selecting objects by their spatial properties (either because non-spatial attributes are not available or because they do not reflect the desired search criteria closely enough). This topic is discussed in more detail in section 9.2.

A sample map created from U.S. Census TIGER data is shown in Figure 8.9a. Note that we have split the TIGER road data set into two map layers, one that contains major roads and one for minor roads. We have used the non-spatial attribute 'CFCC' associated with each road which contains information about the class of each road stored in the dataset. Both layers are then drawn using different graphical parameters to differentiate major and minor roads when the level of detail is large enough for both layers to be displayed. For comparison, a MapQuest map capturing the same area is shown in Figure 8.9b. We see that both data sources present feasible option for online mapping in terms of coverage, level of detail and visual presentation so results obtained using the TIGER data should be applicable to maps created from other data sources such as MapQuest as well.

Our performance testing aims to compare different types of SAND's vector based approach to remote mapping with the bitmap based approach employed by such popular systems such as MapQuest or Switchboard's MapsOnUs. In order to run both systems in the same environment, we chose MapServer (see Section 2.2.4) to represent the bitmap approach. This allows us to deploy both systems on the same hardware, using the same operating system and within the same networking environment. This helps us minimize performance differences caused by factors that are not directly



(a)



(b)

Figure 8.9: Comparison of a sample map generated from TIGER data and a MapQuest map of approximately the same area and zoom factor. Figure 8.9a shows the TIGER map, Figure 8.9b is the corresponding MapQuest map

related to design of spatial data management.

The MapServer system can operate with spatial data represented in ESRI's Shapefile format. Once the U.S. Census TIGER data is converted to ESRI's Shapefile format, the MapServer system can utilize the data set directly. We acquired our data from ESRI who, as mentioned above, makes it freely available for download.

As mentioned above, various types of SAND deployment may use auxiliary SQL servers that support spatial data. In particular, we implement support for MySQL and PostgreSQL. SAND, MySQL or PostgreSQL do not work directly with TIGER data or Shapefiles, so we need to convert the Shapefile-formatted data into the native formats of these database applications. We used publicly available libraries [3] to develop tools that perform this conversion. Thanks to this conversion process both MapServer and SAND (in standalone configuration or supported by MySQL or PostgreSQL) can operate on data represented in their own native format. Since all these native sets originate from the same U.S. Census data set, both systems work with identical content which makes their side-by-side comparison valid.

Our work focuses on optimizing methods that facilitate remote access to spatial data. Therefore, in this evaluation we investigated the performance of the module that is responsible for presenting the user with the data retrieved from the spatial data repository. While the retrieval of visible data from the spatial repository (the window query) is naturally a part of the visualization process, for these services we rely on technology that was not part of our research. So in our evaluation we need to make sure that any possible difference in the performance of the DMBSs does not affect our overall results.

### 8.9.1 Evaluating Performance of Underlying DBMSs

As we indicated above, our approach to remote visualization is not necessarily dependent on the SAND server. Any DBMS with spatial capabilities could perform the function of the central spatial server or of any of the auxiliary servers. Our implementation indeed allows the system administrator to use SAND, MySQL or PostgreSQL interchangeably on the auxiliary servers. Therefore, when evaluating our remote visualization approach against others, we do not need to choose any specific DBMS. This means that in our MapServer (representing a bitmap approach) vs. SAND (representing our vector data approach) comparisons, we need to investigate the relative performance of all the DBMSs involved to confirm that their individual properties are not the decisive factor in the overall results. While these DBMSs are needed for the whole remote visualization solution to work, we only want to compare methods that are employed after the matching objects have been identified by the DBMS.

To assess the performance of the spatial engines separately, we ran tests on the same data and queries that we later use for assessing the performance of the whole system, but without transferring the matching objects anywhere. As we will discussed in further detail below, the operations used include zoom in, zoom out, slow scroll (the new view overlaps 90% of the area of the previous view) and fast scroll (the new view overlaps the previous view by 50%, i.e., half the window). The data density (i.e., the number of objects visible within the view) for all these operations is kept within ranges to be expected during normal usage. Specifically, the zoom factor was selected such that the number of objects shown in any view is around 10,000-20,000 objects. We observed that with a larger number of objects, the content of the window gets too busy and individual features start to blend together, thereby filling continuous areas of the display with the feature color and making individual features indistinguishable. Of course, for the zoom operations the number of visible objects can be much lower as the user zooms farther inwards.

We have used MySQL as the DBMS running on the central spatial server for our evaluation. Other DBMSs could have been used as easily though, we chose MySQL because it is a popular and affordable product available for many platforms and thus it is easier for readers to relate to experiments that utilize this product. Our goal is to evaluate the performance of the overall design and to do so we need to find out if using different DBMSs for the bitmap and vector approaches skews the results. Thus, in order to isolate the performance of the underlying spatial engine from the overhead cost related to further data formatting, and linearization for output or visualization, we use the same set of operations on both MapServer and on MySQL.

The sequence of operations that we use to emulate typical usage scenarios and to measure the performance of various types of deployment are given in Section 8.9.2. Here we use the same sequence of views to measure the performance of the DBMSs alone. This enables us to see whether there are significant differences between the query evaluation time in MapServer and MySQL.

For MySQL, we measure its core performance by executing the following SQL command[7] that calculates the number of elements within a given window:

```
select count(*) from roads where
                    within(location, rectangle(minx,miny, maxx, maxy))
```

As the test iterates over the predefined set of rectangles, the values *minx*, *miny*, *maxx*, *maxy* are replaced by the actual values of each test rectangle.

To access MapServer functionality, we used the PHP-based [14] scripting tool MapScript [15]. We have developed a script implementing the same functionality that was used for MySQL — that is, iterate over all test rectangles and query the number of elements (using the `queryByRect` operation) that lie within each test window.

Both methods and their native data sets utilize spatial indices built on the data. In the case of MapServer, the Shapefiles themselves are distributed together with spatial indices. In the case of MySQL, the spatial index was built after completion of the conversion and import process.

Table 8.2 shows the performance of MapServer vs. MySQL. The DBMS were

[7]Note that the actual SQL command is somewhat more complex due to syntax restrictions and requirements of MySQL. In particular, the command is: `select count(*) from roads where within(location, GeomFromText("Polygon((minx miny, minx maxy, maxx maxy, maxx miny, minx miny))"))`

evaluating 20 window queries for the scroll operations and five window queries for the zoom operations.

| **Spatial Engine Performance** | | |
|---|---|---|
| Operation | Time (in sec) | |
| | MySQL | MapServer |
| Fast Scroll | 4.3 | 4.6 |
| Fine Scroll | 5.5 | 6.8 |
| Zoom In | 1.7 | 2.0 |
| Zoom Out | 1.5 | 2.0 |

Table 8.2: MySQL and MapServer spatial engine performance comparison

As we see from the results, the performance of both systems is comparable. Once we obtain the total cost of retrieving, transporting and visualizing the results for the same sequence of queries, we will be able to judge how large a role the query execution performed by the DBMS plays in the overall cost of the remote visualization operation and whether the differences in performance between the underlying DBMSs are significant.

Data retrieval from the database is an integral part of this process, it is always the first step in the visualization of the area requested by the client. After the data has been retrieved from the database, MapServer always creates an image file and pushes it to the client. In case of SAND, retrieved data will be passed to the client in vector format, possibly merged with other data already available there and finally rasterized for viewing directly on the client side.

Note that to obtain the cost of the visualization alone (i.e., not counting the cost of running the DBMS query), we cannot simply subtract the time it takes the DBMS to fetch all matching elements within each window in the query window sequence from the total running time. This is because the number and size of the query windows requested by the visualization module from the database is also determined by the design of the module. Due to deployments of caching and other methods within SAND, spatial database supporting the SAND-based system would typically process queries encompassing smaller areas and thus return fewer elements. This is because often the query objects are only fractions of the current total viewable area and are often only used to fill gaps in cached coverage.

In the case of the MapServer system, the visualization process involves the following steps:

1. On the server side
    - Given the current viewable area (calculated from instructions sent by the client), perform the 'window query' on the spatial database.
    - Retrieve all elements within the viewable window and render them into a raster image.
2. Between the server and the client
    - Transport the resulting raster image to the user's web browser.
3. On the client side

- Display the raster image within the web browser

In the case of SAND with local caching, the visualization process is as follows:

1. On the client side

   - Given the current viewable area (calculated from the user's input such as clicking the navigation bars), using the locally managed spatial structure calculate which areas of the viewable areas are available locally and which are missing. The missing areas are expressed in terms of a sequence of quadtree blocks. Send the request for objects overlapping this sequence to the server.

2. Between the client and the server

   - Transport the sequence of window queries to the server

3. On the server side

   - Perform 'window queries' on the spatial database using the sequence of rectangles received from the client.
   - Combine all the resulting objects into a single data package to be transferred to the client.

4. Between the server and the client

   - Transport all the matching objects to the client.

5. On the client

    - Load the received objects missing from the local cache into the local cache (possibly dropping some older cached objects)
    - Render a bitmap using the vector data stored in the local cache (we know all the necessary data is available locally after the gaps have been filled by the download from the server).
    - Display the bitmap on client's screen.

### 8.9.2 Performance Comparisons for Typical Usage Scenarios — MapServer vs. SAND Internet Browser

We have identified several typical operations that a user of a mapping or GIS system would perform frequently while navigating around the map. These operations include:

- Zoom in — this operation allows the user to view an area of interest in more detail. This operation is usually available through clicking at a map near the area of interest while the application is in the 'Zoom in' mode. The application then centers the next view around the selected point and makes the viewable area smaller, possibly while showing more details.
- Fast Scroll — this operation allows the user to move the viewable area left and right or up and down by large increments. For dynamically visualizing

applications (such as SAND) this is typically done by clicking and dragging the scroll bar on the side of the viewable window. As the user scrolls vertically or horizontally, the application attempts to refresh the view dynamically. For applications that respond to operations one click at a time (such as MapServer), this is typically implemented by adding four discreet buttons next to the image, each representing a direction the map would move if the button is clicked (i.e., left, right up and down). In this scenario, the map often moves by one half of the window size, e.g., when the 'move down' button is clicked, the elements near the bottom edge of the map will now be showing near the center and there is 50% overlap between the old and new views.

- Fine Scroll — this operation allows the user to move the viewable area left and right or up and down by small increments, perhaps only by a fraction of the window width or height. In dynamically updating systems (such as SAND), this is typically implemented by responding to clicks on arrows located at the end of scroll bars. In discreet systems (e.g., MapsOnUs) this is often implemented by adding another set of buttons similar to the ones used for fast scrolling. These buttons, however, trigger a much smaller move in the given direction and the old and new viewable areas overlap significantly.

- Zoom out — this operation allows the user to see a larger area of the map within the viewable window. Since more objects fall within the viewable area, it may be desirable to select and show only a subset of the elements (e.g., the more important ones) that are viewable at the more detailed zoom level. This operation is typically available by clicking at a map near the area where the zoomed out view should be centered while the application is in 'Zoom out' mode. The application then makes the viewable area larger while possibly selecting only some of the objects for display.

We expect (and confirm our expectations by running experiments) that the cost of each visualization operation (zoom, pan) for the MapServer approach will be approximately constant given a constant data density (i.e., the number of objects to be visualized for per the fixed view area size) and viewable area size. If the number of elements within the viewable area remains the same, then the cost of the spatial query and the subsequent rendering cost remains the same as well. Similarly, the cost of converting the resulting bitmap into a compressed raster format commonly used in web applications (e.g., GIF or PNG) remains the same. Finally, the size of the resulting GIF or PNG image remains the same and so does the cost of transporting it over the network to the user's web browser. Thus we see that when the number of objects visible as a result of a visualization operation remains the same, the cost of updating remains constant as well. Given a server platform, the MapServer system responsiveness will depend on the network speed and latency.

The situation for the SAND Internet Browser is different. There, the system takes a more complex approach when processing visualization requests and the response time will depend on the nature of the request as well as on the history of similar requests preceding this one.

As mentioned above, we have selected several typical operations that users of a mapping system or GIS would perform most often while navigating around the map. These operations include zooming in and out and panning/scrolling.

First, we compare MapServer with the standard SAND setup that only involves the central data server and the SAND Internet Browser client. No auxiliary servers are involved at this time. The spatial server uses MySQL to process the window queries.

As discussed, the bitmap-based approach (MapServer) always loads the full bitmap in response to a single click. Once the request to the server is sent, the user cannot operate the client until this operation is over. If the user tries to initiate further operations while another one is still in progress, all such attempts are either ignored or they are queued up and executed one by one. This means that the user may inadvertently trigger a given operation several times (perhaps as a result of not seeing any response from the system he or she was not sure if the operation request was registered by the application). This could result in the user waiting seconds or even minutes for the queued up operations to clear. SAND uses a different approach. While an operation is being executed, the client does not ignore any subsequent operation requests nor does it queue them. Instead, it only keeps track of the last operation requested. So if the user tries to scroll several times in a row, once the client becomes available, it only executes a single scroll operation that takes the user to the latest requested position. Clearly this is a more desirable method. However, in order to evaluate the visualization system without influence of this additional optimization, when measuring the scrolling performance in SAND we always wait for the last scroll to finish (i.e., update the screen) before triggering the next one.

In order to prevent the user's response time[8] from affecting the performance timing, we have created a special SAND Internet Browser module that emulates a sequence of user's operations while always waiting for the operation to finish before launching the next one. Similarly, we have developed a user emulator for the MapServer system which requests a new operation as soon as it receives results from the previous one.

For the SAND Internet Browser, we measure the execution time in two scenarios:

- The data to be visualized as a result of the user's operation is already cached on the system.

- The data to be visualized as a result of the user's operation is not yet cached on the system and has to be loaded dynamically from the server.

For MapServer, the bitmap is always downloaded from the server for each new operation.

[8]The amount of time it takes the user between noticing the last operation finished and when he or she requests the next operation.

In our sequence of operations, each view shows about 25,000 line segments. Each single fine scroll update “recycles” 9/10th of the screen (using a quick bitmap copy) and updates 1/10th of the screen from the cache or by download from the server, (i.e., about 2,500 line segments). The fast scroll operation reuses one half of the screen and rasterizes vector data loaded either from the local cache or downloaded from the server for the other half of the screen. For the zoom in operation, the data for the next operation is always cached in the system (since the next view shows the subset of data shown in the previous view). For the zoom out operation, the data for the next operation may either already be in the local cache (e.g., if the zoom out operation was preceded by a zoom in operation) or it may need to be loaded from the server (e.g., if the user panned into the current view on the same zoom level and now he or she needs to zoom out to view previously unseen data).

In order to measure the performance across the various deployment scenarios (here represented by different properties of the network connection), we emulate networking environments that correspond to several typical methods of achieving connectivity on mobile devices as well as fixed workstations. The methods that we investigated are given in Table 8.3 along with the properties of such connection methods.

To emulate different networking properties in our test environment, we have utilized NIST Net [11], a general-purpose tool for emulating performance characteristics in IP networks. The tool was designed to allow controlled, reproducible experiments with applications that are sensitive to network performance. NIST Net can emulate

| **Connectivity methods** | | |
|---|---|---|
| Connection Type | Bandwidth (bytes/sec) | Typical delay (sec) |
| Dial-up modem 56kbps | 7,000 | 0.300 |
| Cable/xDSL | 182,000 | 0.200 |
| Satellite | 62,500 | 1.000 |
| LAN | 1,250,000 | 0.002 |

Table 8.3: Properties of various network connection types

end-to-end performance characteristics imposed by various wide area network situations (e.g., congestion loss) or by various underlying subnetwork technologies (e.g., asymmetric bandwidth situations of xDSL and cable modems).

NIST Net is implemented as a kernel module extension to the Linux operating system. The tool allows a Linux server to emulate numerous complex performance scenarios including tunable packet delay distributions, congestion and background loss, bandwidth limitation, and packet reordering or duplication.

We have configured NIST Net using networking parameters typical for individual connectivity methods (Table 8.3) to measure the performance of the SAND system in different deployment scenarios. Since NIST Net only affects incoming traffic (i.e., the traffic that is arriving into the machine running the NIST Net emulation), we had to make sure to run the module on the machines that are receiving data from upstream. This would include the machine running the client and also the machine running the auxiliary proxy in case of the dynamically loaded version of the setup.

As discussed earlier in Section 7.4, the actual DBMS running on the server can be one of many supported systems. The communication protocol between the client and the server-based DBMS used to submit queries and retrieve results depends on the specific DBMS used. As such, some protocols may be more efficient than others in encoding the results in the least amount of space possible. For some DBMSs this optimization may not even be a concern as they expect other layers of the network connection to handle this.

When the native SAND server is used as the central DBMS, the objects returned as a query response are represented in their textual form but the stream of text is compressed using the LZW algorithm before the data is pushed over the network (the SAND protocol is discussed in more detail in Section 8.7). On the other hand, MySQL and PostgreSQL do not explicitly compress the data sent between the client and the server. In order to remove this potential inefficiency and to level the playing field with MapServer (that always implicitly compresses the datastream by by representing the bitmaps in formats such as GIF or PNG that are natively compressed), we establish a compressed IP tunnel[9] between the client and the server. In this way, any communication between the client and the server is compressed even when MySQL or PostgreSQL is used. Note that in the case of a fast network connection, using compression can actually slow the system down as the time spent by the system to compress and decompress the data may exceed the time saved from the faster data transfer.

[9]This was achieved by employing the Secure Shell (ssh) utility and its port forwarding and compression functionality.

For our testing we have used a Dell PowerEdge 300 server running RedHat Enterprise Edition 3.0 AS operating system. The server is powered by a 800MHz Pentium III processor with 192MB of RAM. The client ran on a Compaq Armada E500 laptop with 700MHz Pentium III and 192MB of RAM with the same operating system.

For the pure client-server environment (i.e., no auxiliary servers), the performance was tested for the following three basic client-server architecture states. First, the *cached* SAND Internet Browser state refers to a scenario where the SAND Internet Browser provides local caching and the data to be displayed as a response to the sequence of scroll operations is already available in the client's memory. This type of memory-caching client is discussed in Section 6.2. Second, the *direct* SAND Internet Browser state refers to a scenario where the client does not cache data locally and downloads all the data from its server. This represents the pure client-server setup where the client communicates directly with the central server (Section 6.1) but note that essentially the same environment is created in a setup that involves a static (preloaded) proxy (Section 7.2) or a dynamic proxy which already contains the required data (Section 7.3). It is also similar to a scenario where the client's internal caching is implemented but the data for the current viewable area is not available on the client (Section 6.2). In this scenario, in addition to being displayed, the data would also be stored locally in the PMR quadtree as it is downloaded from the server so that the performance of such a setup in this particular scenario may be slightly lower. Finally, the *dynamic* SAND Internet Browser state refers to a scenario where the client provides local caching but the necessary data is not available in the local cache yet.

Table 8.4 shows the results of a performance comparison of MapServer with the SAND Internet Browser for scrolling. During a sequence of fine scroll operations, the previous window overlaps the next window 90% of the window area. This means that the SAND Internet Browser can use a fast bitmap copy operation to transfer the part that can be reused to an other location of the screen and it needs to rasterize only 10% of the window using vector data stored either locally or downloaded from the server.

| **Fine scrolling/Local Panning** | | | | |
|---|---|---|---|---|
| Connection type | MapServer | SAND Internet Browser | | |
| | | Cached | Dynamic | Direct |
| Dial-up modem | 179 | 6.6 | 80 | 124 |
| Cable/DSL Line | 52 | 6 | 38 | 20 |
| Satellite | 181 | 5 | 85 | 81 |
| LAN | 18 | 5 | 33 | 10 |

Table 8.4: Performance comparison of the SAND Internet Browser and MapServer for the fine scroll operation. The table indicates the time in seconds it took to perform 20 subsequent fine scroll operations.

Table 8.5 shows the results of a performance comparison of MapServer with the SAND Internet Browser for zooming in. The starting viewable window showed 25,000 line segments and each zoom in operation doubled the map scale, i.e. both the $x$ and $y$ coordinate ranges were halved. Thus, the area before the zoom in operation is four times as large as the area displayed after the zoom in operation. We measured the time it took to execute five subsequent zoom in operations, the last view was showing only dozens of line segments.

Note that the viewable area resulting from the zoom in operation is always a subset of the viewable area that existed prior to the zoom in operation. Thus, for the caching SAND Internet Browser, the data to be displayed after any zoom in operation will always be available in the cache.

| **Zoom in** | | | | |
|---|---|---|---|---|
| Connection type | MapServer | SAND Internet Browser | | |
| | | Cached | Dynamic | Direct |
| Dial-up modem | 44 | 0.5 | N/A | 10 |
| Cable/DSL Line | 12 | 0.8 | N/A | 3 |
| Satellite | 44 | 0.5 | N/A | 10 |
| LAN | 5 | 0.8 | N/A | 1 |

Table 8.5: Performance comparison of the SAND Internet Browser and MapServer for the zoom-in operation. The table indicates the time in seconds it took to perform five subsequent zoom-in operations. The results for the dynamic SAND Internet Browser method are not applicable (N/A) since the data will always be cached from the previous operation.

Table 8.6 shows the results of a performance comparison of MapServer and the SAND Internet Browser for the zoom out operation. This test is essentially a reverse of the zoom in operation with a single important distinction in the caching SAND Internet Browser. While each zoom in operation can expect to have all the necessary data cached from the previous step, in the zoom out operation this is not necessarily the case. Consider a scenario when the user moves around in a zoomed-in (e.g., street) level and then tries to zoom-out (e.g., to city level). As the viewable area grows, not all the data objects that overlap this area are necessarily cached.

For the zoom out operation, the starting viewable window showed a large detail containing only a few dozens of line segments. Each zoom out operation expands both $x$ and $y$ coordinate ranges twice. Thus, the area before the zoom out operation is four times smaller than the area showing after the zoom out operation. We measured the time it took to execute five subsequent zoom out operations, the last view was showing about 25,000 line segments. We considered both scenarios outlined above for the SAND Internet Browser. One scenario captures the situation where the data to be shown after the zoom-out operation is already in the cache (i.e., the zoom-out operation was preceded by a zoom in operation without any panning operations in between). The other scenario explores a situation when the data to be shown after the zoom-out operation is not in the cache and has to be fetched from the spatial server.

| **Zoom out** | | | | |
|---|---|---|---|---|
| Connection type | MapServer | SAND Internet Browser | | |
| | | Cached | Dynamic | Direct |
| Dial-up modem | 45 | 1.8 | 48 | 26 |
| Cable/DSL Line | 12 | 1.6 | 22 | 5 |
| Satellite | 45 | 3.2 | 36 | 17 |
| LAN | 5 | 2.3 | 20 | 2 |

Table 8.6: Performance comparison of the SAND Internet Browser and MapServer for the zoom out operation. The table indicates the time in seconds it took to perform five subsequent zoom out operations.

Table 8.7 shows the results of a performance comparison between MapServer and the SAND Internet Browser for global panning. Unlike in the Local Panning/Fine Scrolling scenario evaluated above, in the global panning operation, a large portion of the post-panning viewable area does not overlap the pre-panning viewable area. This means that the SAND Internet Browser must load a large portion of the new viewable area from the locally cached data or from the central spatial server. Given this, we again measure the performance of the SAND Internet Browser for two distinct scenarios:

- The data to be visualized as a result of the user's operation is already cached on the system.
- The data to be visualized as a result of the user's operation is not yet cached on the system and has to be loaded dynamically from the server.

MapServer, as always, generates a new bitmap on the server and pushes it onto the client. Each view was showing about 25,000 line segments during this panning operation. As we can see, each of the tests performed above repeats the same operation under the same conditions. This provides us with a comparison of each possible operation under given conditions (in terms of network parameters) separately. While in a real life deployment the network parameters will likely remain fixed during each session, the sequence of operations will probably be a combination of the available operations. In other words, the user will probably not use solely the fine scroll or the zoom operations, instead they would typically do some scrolling, then zoom in, scroll some more, zoom out, etc. The typical sequence structure and the duration of such a session would depend on the nature of the scenario. Reviewing a larger area for certain properties may involve much scrolling and a minimum of zooming. Investigation of multiple separate locations may involve more zooming in and out with a minimum amount of panning.

| **Fast Scrolling/Global Panning** | | | | |
|---|---|---|---|---|
| Connection type | MapServer | SAND Internet Browser | | |
| | | Cached | Dynamic | Direct |
| Dial-up modem | 161 | 3.9 | 109 | 108 |
| Cable/DSL Line | 44 | 3.9 | 54 | 19 |
| Satellite | 165 | 3.9 | 104 | 80 |
| LAN | 14 | 3.8 | 48 | 9 |

Table 8.7: Performance comparison of the SAND Internet Browser and MapServer performance comparison for the fast scroll (global panning) operation (in sec)

Furthermore, the user will rarely work under conditions when the spatial data is either fully cached all the time or not cached at all in any step. Depending on the exact usage patterns, the user can expect to benefit from the caching for some portion of his or her operations. The success rate of the caching mechanism will depend on numerous factors. The first is the time at which the operation is executed. The cache will be empty right after the start-up of the client application. So the user can expect to be fetching data from the server for most such operations initially. Thus, the initial performance of the caching SAND application will appear close to what we have shown above under the non-cached data columns (i.e., direct or dynamic). Once the cache is filled with data, the success rate will depend on the extent to which the user's spatial operations are localized. If the user visualizes information mostly within the same limited area, then most of the operations will utilize the cached data.

In such a scenario, the performance will be close to what we have shown above in the cached data column. Most of the time the sequence of operations generated by the user will trigger a mixture of cached and non-cached data retrievals. Therefore, we can consider our cached and non-cached results to be the extreme cases of what a user may expect and the typical experience will lie somewhere inbetween.

The data in tables 8.4–8.7 is displayed graphically in Figures 8.10–8.13. From these figures we see that in most deployment scenarios, network environments, and usage patterns the user can expect to have substantially better experience using the SAND system than when using a pure bitmap system. In addition, we see that the performance differences of the underlying DBMSs (as shown in Table 8.2) are negligible in comparison to the cost of running the whole remote visualization operation. Thus we can safely conclude that the differences that we observed are really due to the remote visualization design rather than due to the use of different DBMSs for the examples that we used.

### 8.9.3 Performance Comparisons for Deployments Utilizing Auxiliary Servers

In the previous section we compared the SAND-based system that involved a caching and non-caching client and a central spatial server with a bitmap based system represented by the MapServer application. We have seen that this type of SAND deployment in which the client communicates directly with the central server is a valid

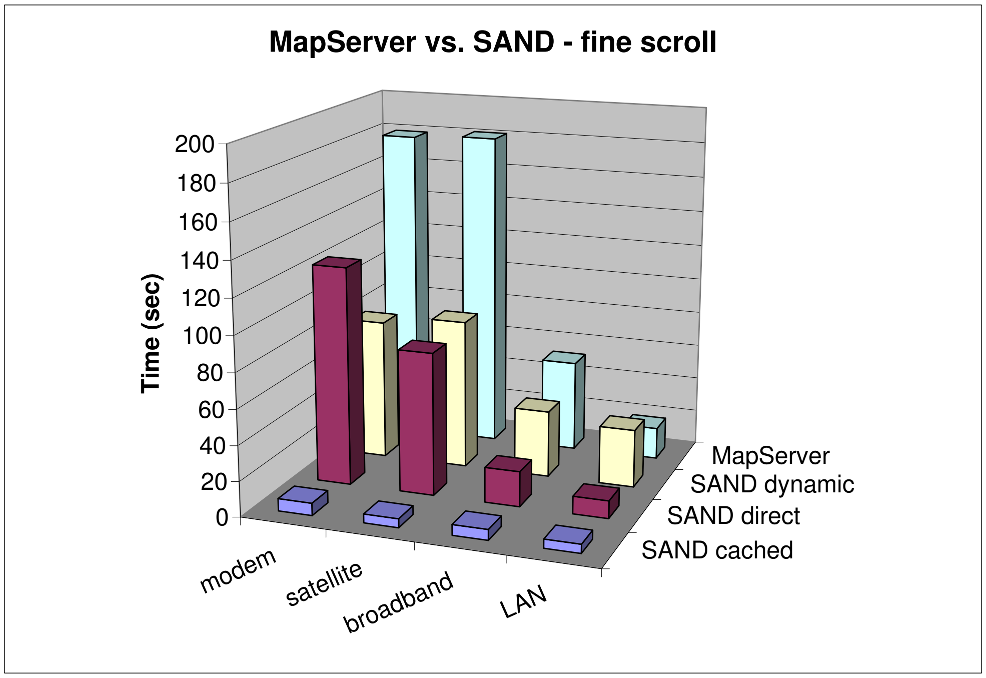


Figure 8.10: Comparison of a bitmap (MapServer) approach with the vector-based SAND approach for remote spatial data visualization. The figure shows results for 20 subsequent fine scroll operations as tabulated in Table 8.4.

approach when the prerequisites for its efficient usage are satisfied, namely the capability of the client to cache non-trivial amounts of data or when the network speed is such that data can be downloaded from the server over and over quickly. With the growing popularity of small footprint wireless-capable handheld devices (e.g., smart/cell phones, PDAs and other similar devices), we also need to look at how the SAND-based architecture handles scenarios where the client cannot store larger amounts of data locally. Note that the bitmap approach is still valid as the client does not store any data locally and thus this method is still applicable even on these mobile devices (see Figure 7.1).

An example of the deployment of the handheld wireless technology in conjunction

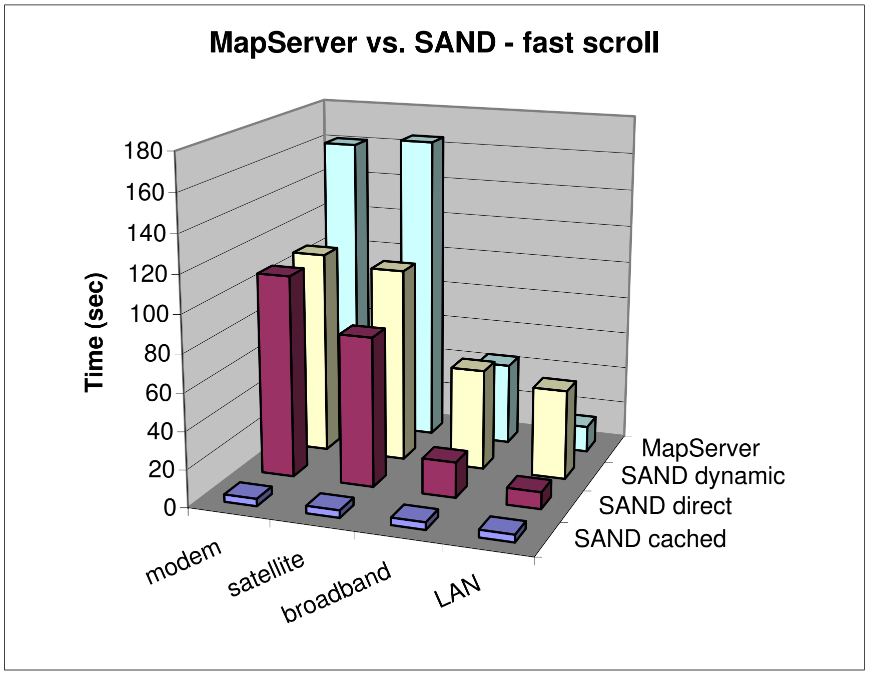


Figure 8.11: Comparison of a bitmap (MapServer) approach with the vector-based SAND approach for remote spatial data visualization. The figure shows the results for 20 subsequent fast scroll operations as tabulated in Table 8.7.

with the introduction of a third auxiliary server is presented in Figure 7.2. This scenario can be modeled using the SAND system by deploying the non-caching SAND client on the handheld device and by installing an auxiliary SAND proxy server within a fast network connection. In this way, the client is requesting all the data it needs to visualize from the proxy server (possibly repeatedly) while the proxy server either has all the data already available or it needs to fetch the data from further upstream (i.e., from the central data server).

We evaluate the performance of different approaches below. We compare the bitmap MapServer approach with a SAND system utilizing an auxiliary proxy server. This proxy server can either be preloaded with the base spatial data or it loads the

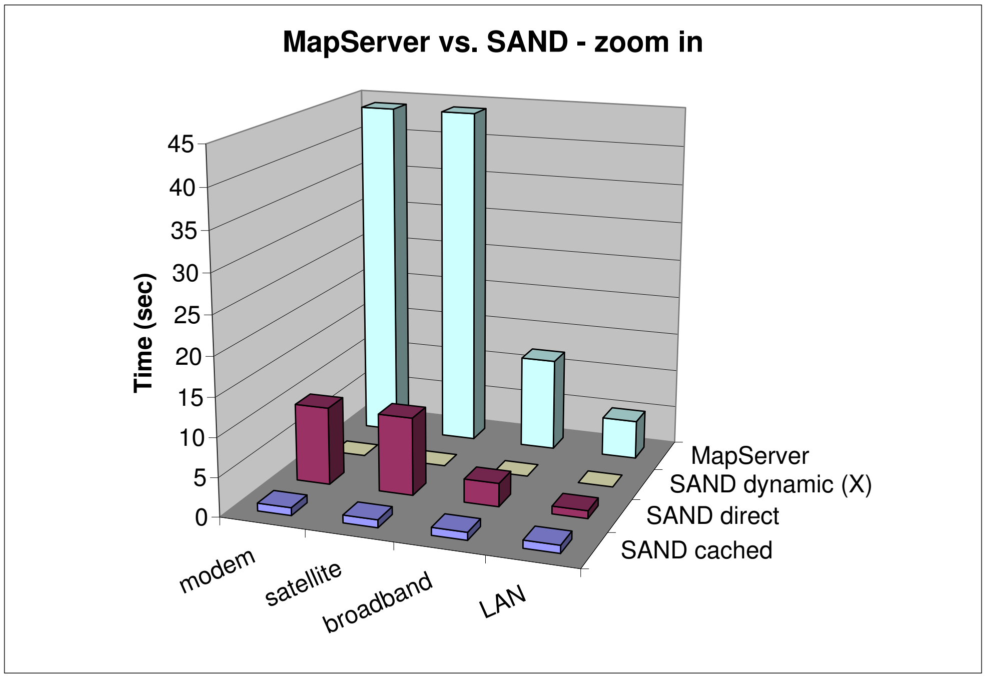


Figure 8.12: Comparison of a bitmap (MapServer) approach with the vector-based SAND approach for remote spatial data visualization. The figure shows the results for five subsequent zoom in operations as tabulated in Table 8.5.

data from the central spatial server as the requests for the data come in.

As before, we examine at two different usage scenarios. The first one assumes that the user just started the application so that no cached data is available yet. Since the proxy server can provide its services to multiple users, we also assume that this user is the first one to request this particular data. The second scenario assumes that the same data was already accessed before (by this or another user) and thus it is already available on the proxy server. The proxy server in our environment is implemented as a Java servlet that manages connections between the clients and the central server, and utilizes MySQL DBMS over a JDBC connector for data storage (Figures 7.3 and 7.4). In these scenarios, we assume a spatial database system needs

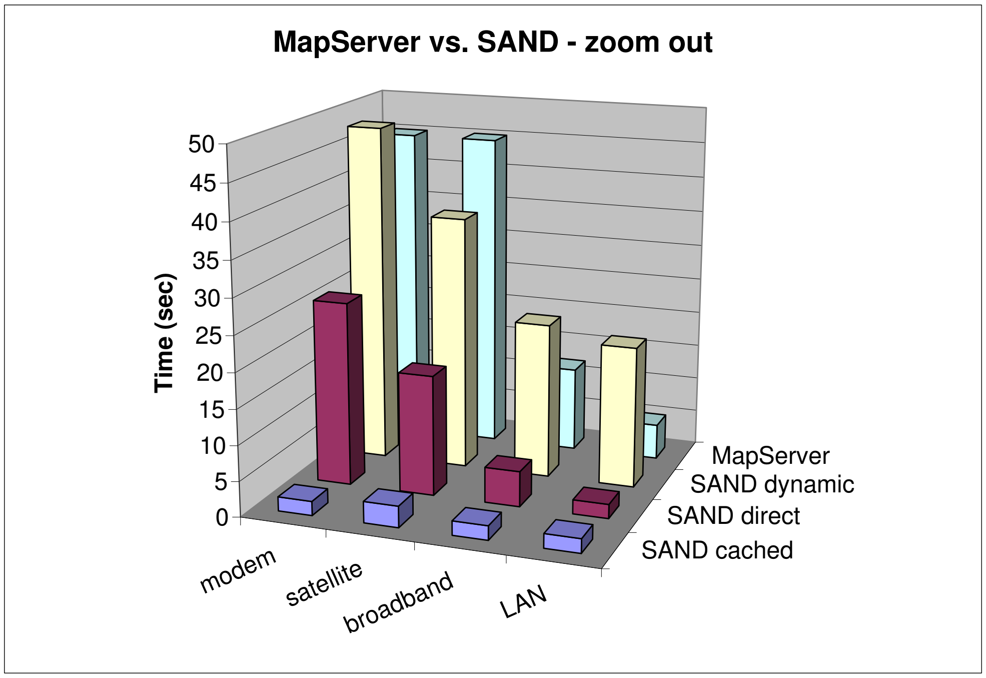


Figure 8.13: Comparison of a bitmap (MapServer) approach with the vector-based SAND approach for remote spatial data visualization. Figure shows the results for five subsequent zoom out operations as tabulated in Table 8.6.

to be implemented quickly with minimum time and effort spent on preprocessing and setting up. Examples of this type of deployment are discussed in Section 8.9.3 and illustrated in Figures 7.1 and 7.2.

In this environment, the communication link consists of two parts. The first leg connects the handheld clients with a local connectivity provider. In the emergency scenario used to illustrate this case, a mobile communication van or similar vehicle equipped with a wireless router as well as with satellite or similar type of link to the central computing facilities could serve as such a provider. Thus the data received by the handheld devices may need to travel across two network segments, one between the device and the mobile van, while the other must travel between the van and the central facilities. We presume that in emergency scenarios such as these, the connectivity between the handheld devices and the van is faster than the connectivity between the van and the central facility.

We perform our experiments with different ad-hoc style usage scenarios on the same set of typical operations used in the experiments described in Section 8.9.2, i.e., fast scroll, fine scroll, zoom in and zoom out. Based on the assumptions for such an emergency response deployment, we assume that the mobile teams will be able to connect to the central facilities over a satellite link. Locally, the connection between the individual response team members will be wireless (e.g., WLAN 802.11b/g). This emulation is again facilitated by the NIST Net product. Table 8.8 contains the exact network layer parameters that we assumed.

| **Connectivity methods** | | |
|---|---|---|
| Connection Type | Bandwidth (KB/sec) | Typical delay (ms) |
| Satellite | 62 | 1,000 |
| Wireless LAN (802.11b) | 1,300 | 25 |

Table 8.8: Properties of network connection types typically available in deployments involving an auxiliary server

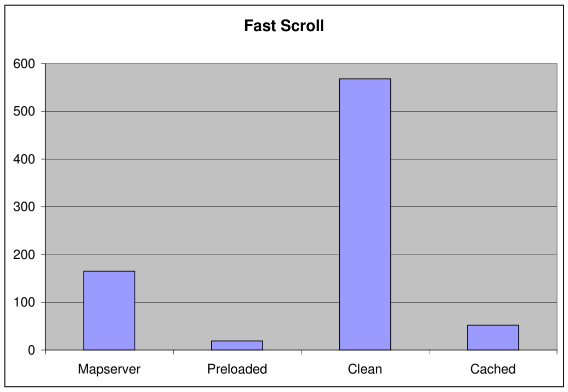
Since we assume that the need for quick response time won't allow establishment of any sophisticated data center, any hardware deployed on the site will have to be portable and thus possibly not particularly powerful. It is reasonable to expect that emergency response teams would use laptop-type equipment for any server tasks. In our experiments, we have used a Pentium III 700MHz/192MB RAM platform running RedHat Enterprise Linux to represent such an underpowered mobile platform.

Table 8.9 shows the results for different ad-hoc style usage scenarios. Figures 8.14a, 8.14b, 8.14c and 8.14d indicate the performance graphically. As we see, MapServer performs better on a freshly installed system where no data has been pushed through the infrastructure yet. In all scenarios, MapServer produces the results much faster than the SAND caching-type deployment where the cache is still empty. This is because of the additional overhead of copying the necessary data from the central data server to the auxiliary server, a step that MapServer completely bypasses. Once the cache is loaded with data, we see that it performs at least as well as, and most of the time significantly better than, MapServer. If the auxiliary server is preloaded with the data, then the difference is even more pronounced.

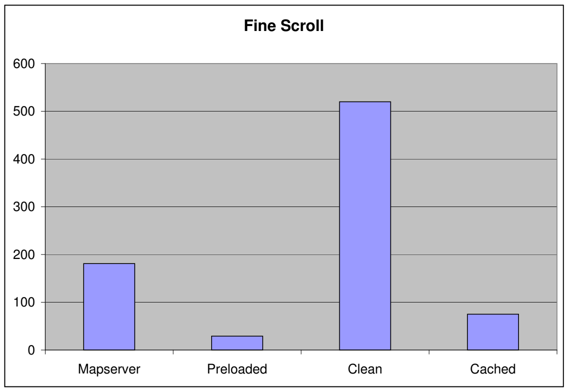
| **Auxiliary Server-based Deployment Performance Comparison** | | | | |
|---|---|---|---|---|
| Operation | MapServer | Preloaded Proxy | Dynamic Proxy (clean) | Dynamic Proxy (cached) |
| Fast Scroll | 165 | 19 | 568 | 52 |
| Fine Scroll | 181 | 29 | 520 | 75 |
| Zoom In | 44 | 2 | N/A | 12 |
| Zoom Out | 45 | 6 | 58 | 48 |

Table 8.9: Performance comparison of various operations for MapServer and the SAND Internet Browser using the auxiliary server deployment method. The scroll operation values represent the time (in seconds) it took the system to process 20 subsequent scroll operations. The values associated with the zoom operations indicate the number of seconds it took the system to process five consecutive zoom operations. The client to auxiliary server link is of a wireless LAN type. The link between the auxiliary server and the central spatial server is a satellite connection. The result for the zoom in operation in the dynamic SAND Internet Browser method is not applicable (N/A) since the data will always be cached from the previous operation.

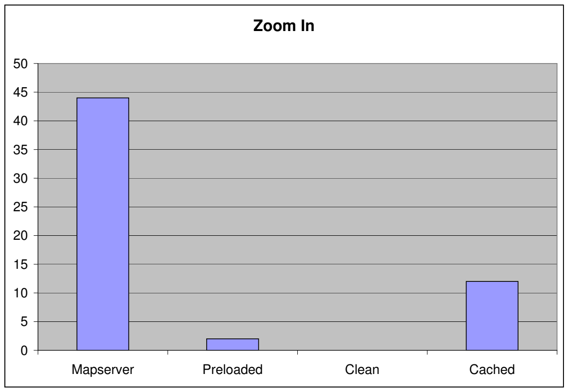
While intuitively we would expect the performance of the preloaded auxiliary



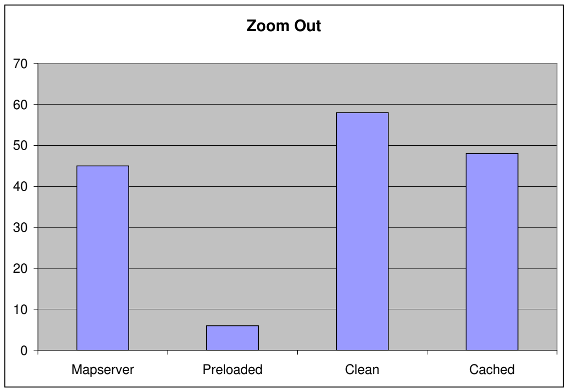
(a)



(b)



(c)



(d)

Figure 8.14: Performance comparison of various operations for MapServer and SAND Internet Browser using the auxiliary server deployment method. Note that non-cached (clean) scenario for the zoom-in operation is not applicable.

server and the cached auxiliary server to be virtually the same, the results indicate that this is not the case. The reason is the additional overhead that the caching server needs to manage the data. In particular, whereas in case of the preloaded server, the client always goes directly to the data stored on the auxiliary server using its DMBS's native protocol, in the case of the caching server, the client communicates with the

Java middleware that first needs to determine what data is available and then serves as the client's data provider. While some extra overhead is unavoidable and that the caching method can always be expected to be slower than the direct access to preloaded database, we believe that with some additional optimization of the Java middleware, the difference could be decreased.

### 8.9.4 Latest Development in Remote Mapping

In early 2005, a new variant of the bitmap approach has emerged. Google and Microsoft have introduced their own mapping systems (Google Maps and Virtual Earth respectively) aimed to compete with established providers such as MapQuest and Switchboard/MapsOnUs. Their developers realized the deficiencies of the classic bitmap approach, especially the slow response time due to the need to load the full bitmap after each operation. The principles of the Google Maps application are outlined in Section 2.2.2, while the Virtual Earth application is implemented in a very similar way.

The viewable area is divided into many tiles and as the user pans, previously downloaded tiles are are cached and reused where possible and only newly visited areas need to have their tiles downloaded. The client logic is implemented in Javascript which makes this application directly usable in most browsers without any additional software download and installation.

Because both Google and Microsoft serve one-size-fits-all maps built from data that is not being updated, they were able to generate the tiles from the source datasets in advance. After this pre-processing, no further spatial operations are needed and their infrastructure simply serves existing tiles to individual clients.

While this need for pre-processing disqualifies serving of real-time or frequently updated data, in principle it should be feasible to generate such tiles from vector data on demand. This would be similar to SAND except that in SAND the client is responsible for rasterizing missing pieces of the viewable area from the vector data while here this operation would still be done on the server. The Google/Microsoft approach however would not scale as well as SAND since all the work would have to be done on the server. While we naturally cannot run formal experiments comparing Google Maps or Virtual Earth with SAND, we can estimate what the performance of these technologies would be within the same environment MapServer and SAND is deployed. Assuming that the cost of locating the right tiles on the server is negligible, the decisive factor for the cost is the amount of data transmitted from the server to the client. Since the tile approach allows for tile reuse, only the newly visible areas will trigger further download. For our fast scroll operation, we are reusing 50% of the visible area and so Google Maps/Virtual Earth would be twice as fast as MapServer (Table 8.11). For our fine scroll operation, 90% of the area is reused so the tile applications would be about ten times faster than MapServer (Table 8.10). The zoom in and out operations (if offering the view for the first time) would take as long as MapServer because the specific bitmaps are not yet available on the client so they have to be loaded from the server in full.

| **Fine scrolling/Local Panning** | | | |
|---|---|---|---|
| Connection type | Tile Method (estimated) | SAND IB dynamic | SAND IB direct |
| Dial-up modem | 18 | 80 | 124 |
| Cable/DSL Line | 5 | 38 | 20 |
| Satellite | 18 | 85 | 81 |
| LAN | 2 | 33 | 10 |

Table 8.10: Performance comparison of the SAND Internet Browser and the estimated tile method for the fine scroll operation. The table indicates the time in seconds it took to perform 20 subsequent fine scroll operations.

| **Fast Scrolling/Global Panning** | | | |
|---|---|---|---|
| Connection type | Tile Method (estimated) | SAND IB dynamic | SAND IB direct |
| Dial-up modem | 80 | 109 | 108 |
| Cable/DSL Line | 22 | 54 | 19 |
| Satellite | 82 | 104 | 80 |
| LAN | 7 | 48 | 9 |

Table 8.11: Performance comparison of the SAND Internet Browser and the estimated tile method for the fast scroll (global panning) operation (in sec)

So we see that for fine scrolling, the tile method would be faster than SAND when the data is not already cached (the performance would be similar for cached areas). For fast scrolling, the performance would be comparable. For zoom in and out operations, the tile method would provide no benefit over MapServer-type approach as the available bitmaps cannot be reused. So in many scenarios, especially where the client is not expected to perform more operations than viewing the map, the tile-based approach is a valid alternative to SAND. The drawback of the tile method is that all the work is concentrated on the server so as the number of clients connecting to a server rises, the performance decreases more rapidly than in case of SAND where the client is responsible for more work. Additionally, the tile method does not allow for development of more sophisticated clients that would perform more operations locally. While in the case of SAND, the client stores the vector data and can therefore perform many operations (such as window or nearest neighbor operations), in the case of the tile method, the client only has access to the bitmap tiles that do not provide data for such localized calculations. So we see that even though SAND may be slower in some scenarios, it is a better platform for developing smarter, more independent client applications.

Most recently, both Google and Microsoft linked their symbolic maps with satellite photography so a symbolic map can be viewed as overlaid on top of a satellite image. The challenge there is to align the symbolic data (e.g., road segments) with their corresponding representation on the satellite map. As the symbolic map and the satellite images usually come from different sources, relying simply on matching the coordinates of the symbolic objects with the satellite images does not necessarily work well enough. This is because of small discrepancies due to measurement errors, rounding errors during numerical calculations, and so on that cause corresponding objects not to be aligned precisely. While the resulting differences may seem to be relatively small, even a shift of a few feet can cause the symbol of a road intersection to be displayed on the sidewalk or beyond which is undesirable from a user's perspective.

# Chapter 9

# Practical Considerations

The results of our work involve not just a theoretical design or a proof-of-concept implementation. We went further and developed a software platform that could provide core functionality for a production spatial data visualization system. In this section we discuss several items that are not directly related to the design of client-server communication but that we had to investigate and resolve in order to create an overall usable system.

## 9.1 Multi-Threaded Design

Normally, a single computer program executes its code sequentially. This means that the program handles individual tasks one at a time. While a single such task is being executed, any other tasks have to wait for this one to finish. In the context of a spatial data visualization tool, such individual tasks could involve checking user's input, rasterizing the map, communicating with the server, etc. Clearly, in this scenario we cannot afford for the program to stop everything else while it is downloading data from the server or rasterizing the vectors using the objects from the PMR quadtree. Such a program would appear non-responsive to the user and would provide poor user experience.

Thus, we have employed Java's support for multi-threaded design which allows us to perform some of these tasks in parallel within the single program. For instance, one of the threads would manage screen redraws, i.e., it would be always ready to refresh a part of the application window that has been obscured by another window on the desktop. By running in a separate thread, the code is always ready to do so by copying the last known bitmap to the viewable are, regardless of what the rest of the application is doing at that time.

Another thread is responsible for preparing a raster image based on user's input. This process involves fetching all objects that overlap the current viewable window (e.g., by traversing the internal PMR quadtree, by downloading data from the server or the proxy, etc.) and rasterizing it using the proper zoom factor, colors, etc. Once the thread finishes, it makes the resulting bitmap available as the last known valid bitmap. If the user requests a different view while this thread processes the previous user's request, the calculations get interrupted, the current plan abandoned and the thread starts working on rasterization of a new view according to the user's latest input. The user's input is continuously monitored by yet another thread.

So, as we see, the SAND Internet Browser can receive user's input any time and if a new command arrives while the last one is still being processed, the work on the previous command stops before it is finished and a new round of calculations starts. This is different from most bitmap-based systems where the user has to wait for the last command to be processed before execution of any new command can start.

In this way, the user interface is always responsive. In particular, the user doesn't have to wait for his/her mouse click to be accepted or for a part of the user interface to be refreshed until drawing of the current scene is completed. Moreover, the system responds to user's map navigation quickly by abandoning a previous map rendering and redirecting the processing power to render the most recently requested view.

## 9.2 Maintaining Maximum Map Density

Since our client-server approach supports any size of the data set, it is obvious that it may not always be feasible or desirable to display every single element of the data set that lies in the current view window. If the window is large enough, such an operation would call for displaying of too many elements which would both take too much time to render and cause the drawing canvas to be too busy. Therefore, it is desirable to have a way to select and display only a subset of the elements to keep the density of the information within the view reasonable.

In some cases, the solution may be to examine values for some nonspatial attribute or attributes to decide which elements should be still displayed and which ones should be omitted for a given view. An example of such an attribute could be the class of the road in case of a road map. All roads would be displayed only in a detailed view that zooms in on a small area of the map. On the other hand, for a more global view, only interstate highways would be shown. This concept is familiar to many users of various popular web-based mapping services such as MapQuest. Assuming that an appropriate set of nonspatial attributes is available for the working data set, this method produces very natural results.

However, an appropriate set of nonspatial attributes may not always be available, or the nonspatial data may not be reliable or it may not provide enough information to separate data into as many levels of details as necessary. For instance, in the case of a road map, each line segment may have an attribute associated with it that identifies the class of the road and this attribute can have one of three possible values, e.g., interstate, main road, or local road. If we are working with larger datasets, it's possible that we need to be able to display the map with six different levels of detail. In this case, relying on the road class attribute only won't provide the necessary depth of line segment classification.

Therefore, in some scenarios it is necessary to rely only on the spatial properties of data or some combination of spatial and nonspatial properties to create a classification of spatial objects for the purpose of creating the required number of detail levels.

When choosing the more important elements to be representatives of all the elements in less detailed map, the topology of the map may provide valuable information. For instance, if individual line segments are grouped together to represent complete streets and roads within the spatial data set, then we can draw conclusions about the importance of individual roads from their total length. It is likely that longer roads are more important than shorter ones. However, while this grouping may be part of one data format, it may not be available in others. Thus, in order to make an algorithm work on any data set regardless of what topology information the particular data format contains, we need to accept the input as an arbitrary list of line segments without relying on information about their mutual relationship.

Given such a list of individual line segments and the desired selectivity factor (for instance in terms of percentage of the original data set size), the goal is to select a subset of the original data set of the specified size so that it contains the most representative roads, rivers, or other spatial features within the map. For instance, we may need to find the most representative 10% of the roads from the original set.

In our algorithm, the first step is to group line segments so that each group represents a continuous road, river or other line feature. Notice that these groups may not necessarily match streets or roads grouped by their name or number as sometimes a street may continue in the same fashion under a different name or on the contrary, a single street may suddenly turn and start heading in a completely different direction than its earlier part.

The next step can then be to compare spatial features of these individual polylines found in the previous step and rank them based on their perceived importance for the simplified map. Notice that it is possible to look at the polylines separately, one by one, or alternatively the ranking of these polylines could depend on their own features as well as on features of their neighborhood. For instance, if a similar polyline was found close-by, it may not be as important to display this one as well. Thus, the current polyline may be ranked lower than some other one whose own properties may make it look less important than that of the polyline we are currently looking at.

Thus, some of the attributes of each such polyline that will contribute to the total ranking may include:

1. Total length of the polyline — typically, the longer is the road, the more important it is. However, this cannot be the only criterion as we cannot guarantee that these polylines match real roads exactly.

2. Total number of segments in the polyline — more segments in the polyline may indicate a more winding road which may make it less likely to be one of the major connectors. Thus given two polylines of similar length, the one with less segments will rank higher than the one with more segments.

3. Total number of other polylines nearby — a major road may not have as many intersections as a more local road of the same length. This criterion will require looking at the buffer of certain size around this polyline and count the number

of other polylines within that buffer.

4. Region coverage — some regions contain more major roads than others. Thus, it is important to choose such polylines for the simplified map so that the whole area is still represented. To achieve this, the whole area of the map will have to be subdivided into multiple regions and each region should be represented by such a number of polylines that is proportional to the total number of polylines in the region. The total number of regions that need to be created will depend on how evenly are the polylines represented throughout the map. The more irregular distribution, the more regions need to be considered.

5. "Windingess" — calculate the total deviation from a smooth curve. If the polyline is winding, then this is an indication that it represents a smaller local road or river.

The above outlined problem is a part of a more general discipline called map generalization [61]. The problem of extracting major map features such as primary roads into another map so that this new map still resembles the old more detailed one has been addressed by numerous researchers. Since the quality of results is determined subjectively, it is difficult to compare similarly performing methods in a definitive way. As our research has not been focused on finding the best generalization method, we do not offer more detailed comparison with existing work or a formal evaluation. We developed a generalization method because we needed to be able to generate sparser datasets to be used within SAND's layer system and it is outlined in this section for the sake of completeness. Nevertheless, our method has been shown to perform well. In particular, an informal sampling of subjective opinions of users indicated that the selection of elements from the detailed map preserves the overall visual impression of the represented area. Moreover, our algorithm is simple, scales well, and is based purely on SQL and spatial database queries.

The generalization process is typically executed in form of pre-processing such as upon loading the dataset onto the server. On-line generalization is not feasible in general. The server can only process a given amount of data within the time frame required for a user-friendly system responsiveness. However, since the dataset can be arbitrarily large, there is no guarantee of any upper limit on the amount of data that would have to be generalized given the user's current view. Therefore, it is necessary to create several layers of generalized data that would reside on the server along with the original data set. The server will send data from the layer corresponding with the level of detail requested by the client.

# Chapter 10

# Visualization of Spatial Data

## 10.1 Introduction

In the previous chapters we focused on optimization of the transport of the spatial data from the data repository and the user's client (and its visualization module specifically). However, providing the visualization module with the data is not the last operation that the GIS system and its clients need to take care of. Before the user can benefit from the GIS system, the spatial data needs to be converted from the incoming stream (regardless whether the data source is the internal cache or a remote spatial server) into an image presented to the user, i.e., visualized.

The visualization process is simpler when the data is coming in from known sources and in a unified format. This is usually the case when the client-server application is produced in a way such that both the server and the client are developed together as a part of a single project. This allows the system architects to design them so that they are optimized to work well in tandem. This is the case of SAND, where both the SAND Internet Browser client and the Java servlet that provides interface to the SAND server are application-level programs that were designed as part of the same effort. As such, communication between these two counterparts as well as the client and server modules themselves can be fully optimized. In similar scenarios, the hardware used in the system can also be custom-designed to correspond with the needs of the system, both on the server and the client ends. This kind of approach is typically applicable for scenarios where both client and server are maintained by the same entity or by separate but closely collaborating entities.

However, there are numerous examples of applications where the server side of the solution is facilitated by an entity independent of the client users. This makes the visualization more difficult because for a variety of reasons data coming from different sources may not work well together. The objects in the data may not be properly aligned, there may be unexpected overlaps, they may have incompatible drawing attributes, etc. This can be the case when the server provides services to a large number of clients from different vendors. Our earlier experience with such an environment are described in Chapter 4. Similar problems may arise where data provided by other entities need to be reformatted and imported for use in the system. We have explored such integrations in [75] in the context of providing general public access to government-maintained spatial data. Additional aspects of this problem were investigated in [77]. Our research in the area of thin-client access to government-provided data is presented in [73] and [74].

One of the most popular platform choices for implementing applications that run over the internet is Sun Java environment. Nowadays, most commonly used computer systems have the Java Virtual Machine implemented and usually available for free. Thus, any Java-based software solution is immediately deployable over most platforms that users wish to use on the client side of the solution. While running a Java application is a relatively simple task, for certain audiences an even easier solution on the client side may be needed when no explicit installation of any additional module would be necessary. This can be provided by making the Java client in the form of a applet which is automatically downloaded and executed from within a standard web browser. Unfortunately, due to various security reasons, Java applets do not have access to as many system resources as do Java applications. While for some scenarios this doesn't pose any problems, others may need to avoid these restrictions by requiring the software to be run as an application after all.

Below, we discuss issues related to using Java and Java applets especially for displaying spatial data.

## 10.2 Visualization of Spatial Data Structures

The representation of spatial data is an important issue in spatial databases [48, 78]. Generally, there are two types of spatial data: locational data and object data. Locational data consists of points while object data consists of spatial objects that have extent (i.e., they occupy space) such as lines, rectangles, regions, surfaces, volumes, etc. The representation of spatial data involves the selection of an appropriate decomposition of the underlying space that contains the spatial objects as well as a spatial index to facilitate finding the objects. Spatial indexes usually provide a quick way to access the objects given a specific location or set of locations. Spatial database systems make use of a number of different space decompositions and access structures. For example, Oracle [50, 68] makes use of a quadtree-block decomposition (e.g., [55, 58, 71, 72]) and a B-tree access structure [42]. The result is known as *SDO* denoting *Spatial Data Option.* IBM Informix [79] makes use of data blades that combine a hierarchy of minimum bounding boxes (often known as an R-tree [49]) with a B-tree access structure. ESRI makes use of a two-level grid (as well as a three-level grid) which is a variant of a grid file [67] that uses a regular decomposition partition. The result is known as *SDE* denoting *Spatial Data Engine.*

Although we have mentioned a number of general representations above, actual implementations make use of variants of these representations. The relationship between these variants is not immediately clear, although a well-trained computer scientist can consult the original references or books (e.g., [71, 72]). Often, this form of consultation requires an effort that is beyond what users and database designers are willing to expend. In addition, it may not provide the intuition that is desired. What is really wanted is a hands-on experience with the ability to vary the data assuming a rather limited volume of data so as to make the task manageable.

In previous work [36, 37] extended in [39] we developed a set of spatial index JAVA (e.g., [32]) applets that enable users on the worldwide web to experiment with a number of the most popular spatial representations and spatial data types, and, most importantly, enable them to see in an animated manner how a number of basic spatial database search operations are executed for them. The decompositions can be disjoint or non-disjoint. Due to the limitations of our displays, our examples are restricted to a two-dimensional domain where the objects can be points, lines, and rectangles. The decompositions can be regular (i.e., based on a recursive halving or quartering of the underlying space into blocks of equal size), or non-regular (in which case the blocks can have arbitrary size). The decomposition process can partition all of the axes at once, or one axis at a time (e.g., a k-d tree [33]). The non-disjoint decompositions are based on a hierarchy of aggregations of minimum bounding boxes for the spatial objects (e.g., an R-tree). In this case, we show the effects of choosing between a large number of competing methods of aggregating the bounding boxes.

In addition to seeing how the underlying space is decomposed through the ability to insert data in an incremental manner, we also enable users of the applets to delete data so that effect of any sequence of arbitrary insertion and deletion operations can be understood. Most importantly, users of the applets can see how the space decompositions support the most common database search operations. In particular, we show in an animated manner how spatial selection [30] is executed for an arbitrary rectangular search region (also known as a window operation or a spatial range query), and how nearest neighbors are found. Both the spatial selection and the nearest neighbor operations are executed in an incremental manner which means that the data objects that satisfy the query are returned and displayed one-by-one. The nearest neighbor algorithm is particularly noteworthy as when the algorithm is run to completion, it provides a full ranking of the data objects in terms of their distance from the query object [51, 52]. The animations are performed in a consistent manner for the different spatial decompositions and data types through the use of a consistent user interface and colors.

### 10.2.1 Data Structure Population and Modification

In order to be able to visualize the decomposition induced by the data structures as well as the associated algorithms, we need a way to populate and modify them which means that we must make it easy to insert and delete spatial objects. As soon as an object is inserted using the graphical user interface, the decomposition of the underlying space into blocks is updated and displayed, and likewise when an object is deleted.

Points are inserted by simply depressing the mouse inside the drawing canvas (i.e., the display area). The point is inserted at the position of the mouse when it is depressed. Points are deleted by using the same process except that the user must be in deletion mode which is done by selecting the “delete” option. Since it is difficult to position the mouse precisely over the point to be deleted, we simply delete the closest point (using the Euclidean distance metric) to the position of the mouse.

Lines are inserted by depressing the mouse at a particular point, which serves as a starting vertex, and then dragging the mouse (while depressed) to the point which is to serve as the terminating vertex, at which time the mouse is released. This technique is fine for creating new lines. However, at times, we also need to connect existing lines by making use of existing vertices thereby permitting the creation of polygonal maps such as a triangulation. This is achieved by making use of the control key to augment the line insertion process. In particular, whenever the control key is depressed at the start (end) of the line creation process, the starting (terminating) vertex of the line is snapped to the nearest existing vertex. Thus two existing vertices are joined by a newly created line by keeping the control key depressed before selecting the starting vertex, during the dragging process, and after selecting the terminating vertex by releasing the mouse.

Lines are deleted in the same manner as points in that we position the mouse somewhere near the line we wish to delete while in deletion mode. Once again, the closest line (using the Euclidean distance metric) to the position of the mouse is deleted.

Rectangles are inserted in the same manner as line segments in the sense that the user specifies two diagonally opposite vertices. The difference is that the dragging process results in drawing a rectangle rather than a line. In addition, unlike the case of lines, there is no notion of snapping although this technique could be used in the future to permit the construction of collections of rectangles with shared sides. Rectangles, are deleted in a manner similar to that used for points and lines with the exception that we usually position the mouse somewhere within the rectangle that we wish to delete. If the mouse is in more than one rectangle (as may be the case for the rectangle representations that permit overlapping rectangles — e.g., the MX-CIF quadtree and the R-tree), then delete the containing rectangle $r$ whose boundary is the closest (note that $r$ is not necessarily the closest rectangle as the rectangle whose boundary is the closest need not be a containing rectangle). If the mouse is not within any rectangle, then delete the rectangle whose boundary is the closest.

Since one of the goals of the visualization is to be able to compare the different representations, the applets must keep track of the order in which the objects were inserted. This is necessary because for some of the representations the decomposition of the underlying space depends on the order in which the objects were inserted. For example, this is the case for the point quadtree, k-d tree, PMR quadtree, and R-tree.

### 10.2.2 Algorithm Visualization and Animation

We have tried to use the same color conventions for distinguishing between the data, query objects, and the intermediate results of the search operations. In order to understand the motivation for our choices, we now give a brief explanation of the algorithms that we use to implement the search operations. We first explain the incremental nearest neighbor algorithm which locates the objects in the database in the order of their distance from the query object $q$ [51]. Next, we explain the window query algorithm.

The incremental nearest neighbor algorithm makes use of a priority queue where the queue elements are the blocks of the underlying data structure as well as the objects themselves. The priority queue is ordered on the basis of the distance of its elements from the location of the query object, which is a point in our implementation. The algorithm works in a top-down manner in the sense that as elements are removed from the queue, they are checked if they correspond to blocks that are not at the lowest level of the hierarchy (i.e., nonleaf nodes). If this is the case, then their immediate descendants (i.e., the sons) are inserted in the queue ordered according to their distance from the query object. Otherwise, the objects that they contain are inserted into the queue ordered according to their distance from the query object. If the element $e$ that has been removed from the queue is a data object, then $e$ is reported as the next nearest neighbor of the query object.

In order to be able to visualize the behavior of the incremental nearest neighbor algorithm, at any instance of time (see [59, 62] for alternative nearest neighbor algorithm animations) we distinguish between the following entities by using different colors in a consistent way for all of the data structures and data types:

1. Blocks in the priority queue denoted by light blue.

2. Objects in the priority queue denoted by green.

3. Objects that have not yet been processed (i.e., entered explicitly into the queue or output into the ranking) denoted by red.

4. Objects that have been processed and hence have been ranked denoted by blue. The numeric position of the object in the ranking is displayed in blue next to the object.

5. Blocks that have been processed (although their objects may still be in the queue) denoted by gray.

6. The next item in the queue to be processed (could be a block or an object) denoted by yellow.

7. The query object denoted by orange.

It is important to note that we distinguish between objects that have yet to be processed and those that are explicitly in the queue. Objects that have not yet been processed are those that have not been enqueued explicitly although some of their containing blocks, say the set $C$, are in the queue, while the blocks that contain the elements of $C$ have already been dequeued. $C$ may contain more than one block for a particular data object when the representation is one that decomposes the blocks containing the objects into disjoint blocks at a particular level (e.g., the rect quadtree for rectangles and the PM quadtree family for lines). This distinction enables users to watch the progression of the algorithm.

By the same line of reasoning, an object can be in the queue several times. However, the incremental nearest neighbor algorithm ensures that the object will be reported just once in the ranking. The object will be inserted into the queue several times when the object is a member of several blocks whose contents are inserted into the queue. The insertion algorithm performs some checks to reduce the number of times an object $o$ is inserted into the queue; however, these checks are not guaranteed to prevent the object from ever being inserted into the queue more than once. For example, if the distance from $o$ to the query object $q$ is less than the distance from the containing block to $q$, then $o$ must have already been encountered and thus it is not even inserted into the queue. However, the algorithm does have the property that once an object is reported as the currently closest object, all remaining instances of the object are found immediately afterwards as they have the same distance and thus they are not reported; instead, they are removed from the priority queue [51].

Observe that if a block $b$ of size $2^m \times 2^m$ is in the queue, then all of the blocks of size $2^i \times 2^i$ $(i < m)$ contained in $b$ are also implicitly in the queue although we do not distinguish between them and $b$. This means that we really don't have an entity such as a block that is unprocessed in contrast to having objects that have not yet been processed.

An interesting question is why we differentiate between enqueued objects and enqueued blocks by using a different color for them (i.e., green and light blue, respectively). It would appear that we should use the same color for both of these entities as they can both be members of the queue. However, using such a convention would not enable us to properly deal with representations that store objects in both leaf and nonleaf nodes of the hierarchy.

For example, this is the case for the MX-CIF quadtree [57] which associates each rectangle with its minimum enclosing quadtree block. In particular, when executing the incremental nearest neighbor algorithm, after removing from the queue a block $b$ corresponding to a nonleaf node, we insert into the queue the four blocks corresponding to $b$'s children as well as the objects that are associated with $b$. Thus if we were to use the same color for the objects in the queue and the blocks in the queue, then when the queue contains a block $b$ as well as the objects in $b$, then we would not be able to see the objects even though they are present. Notice that this situation is different from that where the block is in the queue but the objects contained in it have not been entered into the queue (in which case the objects are denoted in red).

It is important to note that using the same color for the objects in the queue and the blocks in the queue also causes a problem for both the point quadtree and the k-d tree as for these representations the data corresponding to the point objects is associated with nonleaf nodes rather than just with leaf nodes as is the case for representations based on a regular decomposition. Thus when we enqueue the point, we also enqueue the blocks corresponding to its sons. Since the point serves as a corner of the son blocks in the case of the point quadtree and a side of the son blocks in the case of the k-d tree, if the point and the son blocks are in the queue at the same time, then we will not be able to see the point even though it is present. In other words, again, it should be clear that we need to use one color for enqueued objects and another color for enqueued blocks as both a block and the overlapping objects may be in the queue.

Observe that we distinguish between the elements in the priority queue, those that have been processed, and the one that is currently being processed (i.e., the next one to be processed). Initially, we only made a distinction between the elements in the priority queue and the ones that have already been processed. However, this made the visualization of the animation very difficult to follow as users could not easily detect what had changed from step to step. In particular, without making the explicit distinction between the elements that have been processed and the element to be processed (or currently being processed) we would have to do so via the process of a mental visual subtraction of 'before' and 'after' states of the animation process. Clearly, our approach is preferable.

As an example of the execution of the incremental nearest neighbor algorithm, consider Figure 10.1 which shows an intermediate stage of the algorithm for a PR quadtree immediately after retrieving the 11 nearest neighbors. Observe that the manner in which the algorithm proceeds leads to a constantly growing circle centered at the query point where the blocks inside the circle are gray corresponding to the blocks that have been processed while the area outside the circle is light blue corresponding to blocks in the queue which remain to be processed.

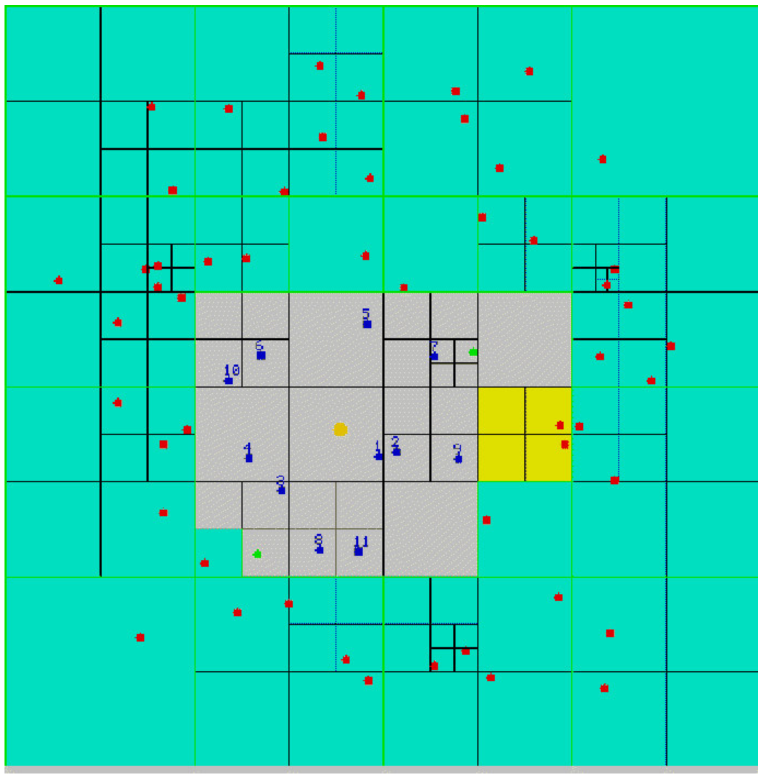


Figure 10.1: PR quadtree applet showing an intermediate stage in the incremental nearest neighbor algorithm immediately after finding the 11th nearest neighbor. The query object is orange. The 11 nearest neighbors are blue with their position in the ranking. All blocks in the priority queue are light blue while the points in the priority queue are green. The current item being processed, in this case a block, is yellow. All points not yet processed or not in the priority queue are red. All blocks that have been processed are gray.

The window query algorithm is a simple tree traversal that visits the blocks of the representation in a top-down manner checking at each stage if the block $b$ overlaps the query window. If there is no overlap, then exit. Otherwise, check if $b$ is not at the lowest level of the hierarchy (i.e., $b$ is a nonleaf node), in which case the algorithm is applied recursively to the immediate descendants of $b$. If $b$ is at the lowest level of the hierarchy, then check if the objects contained in $b$ are in the query window and report them as satisfying the query.

Once again, in order to be able to visualize the behavior of the window query algorithm, at any instance of time we distinguish between the following entities by using different colors in a consistent way for all of the data structures:

1. Blocks that remain to be processed (i.e., they partially overlap the query range) denoted by light blue.

2. Objects that have not yet been processed but whose smallest enclosing block has been found to be in the query range denoted by green.

3. Objects that have not yet been processed in the sense that their smallest containing block has not been tested with respect to being in the query range or has been found to be outside the query range denoted by red.

4. Objects that have been processed and that have been found to be in the query range denoted by blue.

5. Objects that have been explicitly processed and that have been found to be outside the query range denoted by violet.

6. Blocks that have been processed denoted by gray (although some of the objects that they contain remain to be processed).

7. Blocks that have not been processed as they are outside the query range denoted by white.

8. The next item to be processed (could be a block or an object) denoted by yellow.

9. The query range denoted by orange.

When the objects have extent (e.g., lines, rectangles, etc.), we need to be a bit more precise as to what is retrieved by the window query. The issue is whether the retrieved object $o$ must be contained in its entirety in the query window $w$, whether $o$ must enclose $w$, or whether $o$ need only have a nonempty intersection with $w$ (i.e., a partial overlap). For line segments, we have the following three options:

1. The entire line (i.e., both of its endpoints) are in the query window.

2. The entire line passes through the query window (i.e., both of its endpoints lie outside the window).

3. At least some part of the line crosses the boundary of the window.

The applets enable the user to specify which of these variants of the window query is to be used.

The concept of blocks to be processed as well as objects to be processed in the window query is similar to the blocks and objects that are stored in the priority queue for the incremental nearest neighbor query. In fact, we can make use of a queue to keep track of the blocks and objects that intersect the query range. However, in the window query algorithm we only place blocks and objects in the queue that are guaranteed to be processed in the future. These are the blocks and objects (by virtue of their smallest containing block) that intersect the query range. The difference from the priority queue of the incremental nearest neighbor algorithm is that the elements in the priority queue are ordered by their distance from the query object while there is no required order in the list of blocks or objects to be processed in the case of the window query algorithm. In other words, once a block is inserted in the list of blocks to be processed it can be processed at any time. On the other hand, in the interest of minimizing the size of the queue, we usually prefer to process the objects before the blocks as once the objects are processed, the queue is guaranteed to decrease in size while this is not the case after processing the blocks. Note also that if we order the list of blocks to be processed according to time of insertion, then we get some variant of a consistent top-down traversal of the blocks that intersect the query range (e.g., `NW`, `NE`, `SW`, `SE`).

The visualization of the window query algorithm differs from that of the incremental nearest neighbor algorithm in that we need to distinguish between objects that have been explicitly determined to be outside of the query range (denoted by violet) and those that have been implicitly determined to be outside of the query range by virtue of the fact that their containing blocks are outside of the query range (denoted by red). Of course, this distinction would also be needed in the visualization of the incremental nearest neighbor algorithm if we would set a bound, say $k$, on the number of nearest neighbors that we want to determine. In this case, when the algorithm runs to completion, all objects remaining in the queue (denoted by green) play a similar role to the objects that have been found explicitly to be outside the query range (and thus they will have to be converted to violet) in contrast to those in the blocks that have not even been inserted into the queue (denoted by red). Note that the blocks that remain in the priority queue have been explicitly tested in the sense that their distance from the query object was computed before they were inserted into the priority queue.

Observe that the window query could also be implemented in a bottom-up manner in which case we would be visiting the blocks in another order (e.g., by adjacency) as exemplified by algorithms that visit the blocks using neighbor finding (e.g., [31]). However, even if we change the algorithm, we can still use the same color conventions. The difference is that the blocks would be processed in a different order and the objects that overlap the query range would also be obtained in a different order.

Another important point to notice is that the algorithms that we have implemented must be visualized in the way we have described regardless of whether or not the underlying data structures are implemented as trees. This point is particularly relevant to the representations that are based on a disjoint decomposition of the underlying space (e.g., quadtrees). This is because these representations could also be implemented as collections of blocks corresponding to the leaf nodes, or even all of the nodes, and organized using some other data structure such as a B-tree. An example of such an implementation is the linear quadtree [46] where each block is represented by a pair of numbers corresponding to its depth and a location code denoting the result of interleaving the bits of the binary representations of a distinguished point in the block such as its lower-left-most corner. These pair of numbers are concatenated and stored in a B-tree. When such a representation is used, the algorithms are visualized in the same way. The difference is that when the nonleaf nodes are not explicitly stored in the B-tree, the algorithm must artificially create blocks corresponding to them as both the incremental nearest neighbor and window query algorithms simulate a tree traversal where the blocks corresponding to these nodes are indeed present.

Using the above coloring conventions, the animated visualization of the algorithms is quite straightforward and easy to follow. Initially, all objects in the database are displayed in red. The query regions for the window query and the query object for the nearest neighbor query are displayed in orange. As the algorithms for the various operations are executed, the data objects that have been found to satisfy the query are displayed in blue while the data objects that have been found explicitly (implicitly) not to satisfy the query are displayed in violet (red). The data objects that have been enqueued for a future decision are displayed in green.

As the search algorithms are executed, they are animated in the sense that the objects that satisfy the queries are returned one-by-one using a programmable time delay factor which is under the control of the user with the aid of a slider. The animation consists of showing the objects and blocks of the decomposition of the underlying space that are visited during the search in the order in which they are visited. Of course, since our algorithms are inherently top-down, they visit the elements of the hierarchy (which are blocks) in order of decreasing size whether or not the implementation of the data structure actually makes use of a tree. Thus the animation also displays the blocks at the intermediate levels regardless of whether or not they are physically present (i.e., this would be the case for a pointer-less quadtree representation such as a linear quadtree).

As blocks are visited during the search, they are displayed by filling the space spanned by them with yellow (actually this is done immediately prior to visiting them). Once the blocks have been processed, they are displayed by filling the space spanned by them with gray. Those blocks that have never been visited (e.g., as they are completely outside the query range) are displayed by filling the space spanned by them with white. The blocks that have been queued for processing (although not yet been processed) are displayed by filling the space spanned by them with light blue.

As mentioned above, since the search algorithms are often tree traversals (or at least visit the blocks in an order often related to their size), the animation has the effect of starting out with one yellow block that covers the entire underlying space and then decomposing it into smaller and smaller blocks all of which are light blue with the exception of one which is shown in yellow corresponding to the next block to be processed by the algorithm. Whenever a large block has been visited, we may need to generate blocks corresponding to its descendants at which time one of the descendants is visited while the remaining brothers at this level are enqueued for future processing. The blocks that are enqueued for future processing are the ones that are displayed by changing their fill color to light blue, while the blocks that have actually been visited have their fill color changed to gray. Blocks that will never be visited (e.g., they are outside the query range) have their fill color changed to white. As another example, consider Figure 10.2 which shows an intermediate stage of the window query algorithm for a bucket PR k-d tree with bucket capacity 3.

### 10.2.3 Extension to Other Representations

In order to test the generality of our ideas, we adapted our system and display conventions to deal with additional representations that are radically different. Most of the data structures that we discussed in Section 10.2 were dynamic in the sense that they did not have to be rebuilt when data was added and deleted. We now examine the two-dimensional range tree [34, 35] and the priority search tree [60]. They are designed primarily for range queries and thus we did not implement the incremental nearest neighbor query for them. These are static representations which have better worst-case execution times for range queries than quadtree variants or R-trees. As these are static representations, our applets do not show their construction in an in-

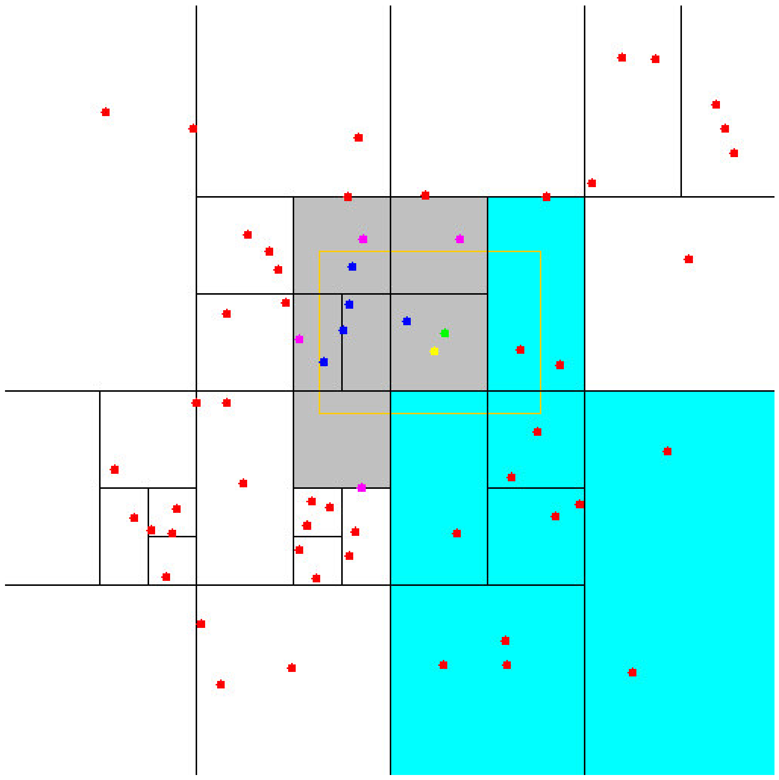


Figure 10.2: Bucket PR k-d tree (with bucket capacity 3) applet showing an intermediate stage in the window query. The query range is orange. The points that have been found so far to be in the query range are blue. All blocks in the queue are light blue while the points in the queue are green. The current item being processed, in this case a point, is yellow. All points that have been processed and found not to be in the query range are violet. All points not yet processed or not in the queue are red. All blocks that have been processed are gray. All blocks that have not been processed, as they are outside the query range, are white.

cremental manner. Instead, in the interest of speed, the points are added in sequence and the data structure is only built once the user has indicated that the input process is finished.

The two-dimensional range tree is designed to facilitate the execution of two-dimensional range queries. It is a tree of trees where the main tree $T$ is a one-dimensional range tree along the $x$ coordinate value. Each nonleaf node $c$ of $T$

contains a one-dimensional range tree along the $y$ coordinate values of all points in the subtree rooted at $c$. Thus we see that the $x$ coordinate value serves as the primary key, while the $y$ coordinate value serves as the secondary key. Given $n$ points, the structure requires $O(N \cdot \log_2 N)$ storage.

The two-dimensional range tree window query search algorithm first locates the nodes $a$ and $b$ in the main tree that contain the $x$ coordinate boundary values of the query range. It then performs a one-dimensional search on the one-dimensional range trees associated with the nonleaf nodes found in the paths from the nearest common ancestor of $a$ and $b$ to $a$ and $b$. This process takes $O(\log_2^2 N + F)$ time where $N$ is the number of points in the data set and $F$ is the number of points found.

This algorithm is visualized and animated in the same way as the quadtree and R-tree algorithms described in Section 10.2.2. The only modification we have made is to replace the block decomposition lines in the case of the quadtree variants and the object aggregate boundaries in the case of the R-tree by a set of vertical lines showing the partitions of the underlying space that are made by the one-dimensional range tree built on the basis of the $x$ coordinate values. As the partition lines go through the entire range of $y$ coordinate values, we use the visual cue of line thickness to differentiate between the levels of decomposition, with the thicker lines corresponding to the partitions closer to the root of the tree. Since there is a limit to the line thicknesses between which we can visually discriminate, we only show the partitions for the first four levels (labeled 0–3). The partition lines are also labeled with numbers to their left to show the depth of their corresponding nonleaf node. We do not show the partitions induced by the $y$ coordinate values as they differ at each nonleaf node and their multitude would only confuse the viewer. Keeping with our display color conventions, we show the partition lines in black.

Once the algorithm has found the boundary nodes $a$ and $b$ in the main tree corresponding to the $x$ coordinate values and their nearest common ancestor node $c$, it proceeds to search the one-dimensional range trees attached to the nonleaf nodes along the paths from $c$ to $a$ and $b$. During this process we use the color yellow to denote the space spanned by the current partition of the main range tree that is being searched, while the space spanned by the remaining partitions to be searched is shown in light blue. The light blue area corresponds to the blocks in the queue in our earlier algorithms. The space spanned by partitions that have already been searched is shown in gray while the space spanned by those that are outside of the query range is shown in white.

Once the search of a one-dimensional range tree $r$ has started, all points in the leaf nodes of $r$ are shown in green as they are analogous to those that are placed in the queue in the other representations that we implemented. At the same time, the color of the space spanned by the partition is changed from yellow to gray denoting that it has been processed. The search of $r$ first determines the nodes $d$ and $e$ in $r$ that contain the $y$ coordinate boundary values of the query range. All points corresponding to nodes before $d$ and after $e$ will not be examined and their color is changed from green to red as they were not examined prior to being implicitly placed in the queue.

The process of determining $d$ and $e$ may examine at most two points which are not in the range in which case their color will be changed from green to violet. All remaining points are processed one-by-one, at which time their color changes from green to yellow while the color of the previously processed point is changed from yellow to blue denoting that it has been found to be in the query range. As an example, consider Figure 10.3 which shows an intermediate stage of the window query algorithm for a two-dimensional range tree.

The priority search tree is similar to the two-dimensional range tree with the difference that it is just one tree and thus requires only $O(N)$ storage for $N$ points. It is designed to facilitate the execution of semi-infinite range queries which are range queries where the upper bound on the $y$ coordinate value is infinite. Conceptually, the priority search tree is a range tree for the $x$ coordinate values and a heap for the $y$ coordinate values. The structure is implemented as a binary tree where the leaf nodes contain the points and are linked together in ascending order of the $x$ coordinate value. Each nonleaf node $i$ contains a midrange $x$ coordinate value and a pointer to the node in the subtree rooted at $i$ with the largest $y$ coordinate value that has not already been associated with a node at a shallower level in the tree. The semi-infinite range query traverses the priority search tree in the same manner as the window query algorithm for the two-dimensional range tree and is visualized in a similar manner.

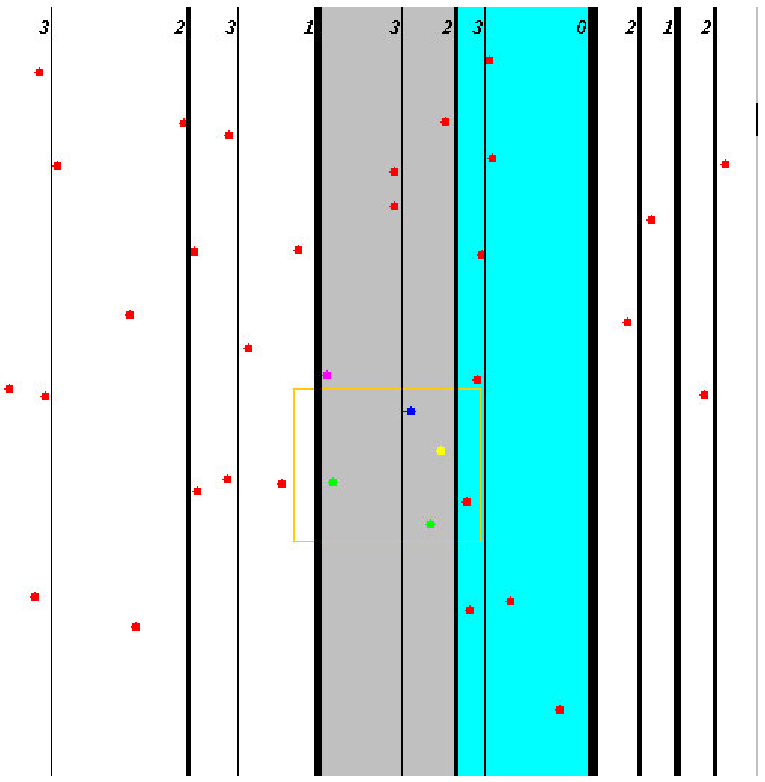


Figure 10.3: Two-dimensional range tree applet showing an intermediate stage in the window query. The black vertical lines indicate the partitions at the first four levels of the range tree for the $x$ coordinate value with the thicker lines corresponding to the shallower levels. The lines are labeled with the depth value of their corresponding partition. The query range is orange. The points that have been found so far to be in the query range are blue. All blocks in the queue are light blue while the points in the queue are green. The current item being processed, in this case a point, is yellow. All points that have been processed and found not to be in the query range are violet. All points not yet processed or not in the queue are red. All blocks that have been processed are gray. All blocks that have not been processed as they are outside the query range are white.

### 10.2.4 History List

It is desirable to be able to switch between different data structures in order to see how the same data set is represented by different spatial data structures. This is achieved by keeping a separate representation $R$ (termed a *history list*) of all objects of a given spatial data type. $R$ is ordered according to the time $t$ at which the object $o$ was created and thereby inserted into the data structure that was being displayed at $t$. In addition, when an object $o$ is deleted from the data structure that is being displayed, $o$ is inserted again into $R$ and marked as being deleted. This is important when the data structure depends on the order in which the objects were inserted (e.g., a PMR quadtree [64]) and the data structure is being rebuilt after a data structure switch that may have been accompanied by some insertion operations since the last time it was built. Thus we see that an object may appear more than once in $R$. Maintaining such an ordering ensures predictable and consistent results as we switch between representations.

The history list is also useful for dealing with illegal configurations of objects. In particular, some configurations of objects are illegal for one representation while legal for another. For example, intersecting lines (at points other than vertices) are allowed for the PMR quadtree while they are not allowed for the $PM_1$, $PM_2$, and $PM_3$ quadtrees [76]. Thus if we first build a PMR quadtree with intersecting line segments, followed by a switch to display the $PM_1$ quadtree for these line segments, then we do not allow any intersecting lines to be displayed. However, when we switch back from the $PM_1$ quadtree to the original PMR quadtree, then the intersecting line segments are displayed. The history list facilitates this action. Notice that similar problems arise when switching between representations that allow intersecting (i.e., overlapping) rectangle objects and those that do not. In the case of object types which can be described in terms of other simpler object types, the history list also contains a representation of the simpler object types using a particular data structure in order to facilitate operations that involve the simpler object type. For example, in the case of line segment objects, each line segment consists of two vertices which are point objects. The set of vertices is represented by a PR quadtree.

### 10.2.5 Semantics and Mechanics of the Move Operation

In the previous section, we discuss visualization of nearest-neighbor and window query operations. However, it is also important to look into operations that build the data structure to run the queries on. While insertion and deletion operations were briefly discussed in Section 10.2.1, in this section we will focus on the move operation. The move operation allows relocating an existing object within the data structure while keeping the properties specific to the structure. These can for instance be the order, in which the elements have to be inserted to recreate the current structure from scratch.

The first task in implementing the `move` operation is to decide which object to move. This is done with the aid of a pointing device (e.g., the mouse). In particular, once the `move` operation has been selected, the closest object to the position of the mouse is displayed (in orange) and the underlying data structures and the display canvas are continuously updated as the mouse moves. This is achieved by invoking the incremental nearest neighbor algorithm [51, 53] to find just one neighbor. This is the same algorithm used in the `nearest` operation to rank the objects with respect to a given query object.

In the case of objects with extent such as rectangles, the situation is more complex. Assume that rectangle object $o$ is the closest to the position $m$ of the mouse. The identity of the closest rectangle is determined in the same way as done for the `nearest` operation (see Section 10.2.2). However, since $o$ is not a point object, $o$ cannot be co-located with $m$. Thus we need to find some representative point for $o$ which is made to coincide with $m$. The easiest solution (and the one we adopted) is to represent $o$ by its centroid $c$. As the mouse is moved, the rectangle object is moved in such a way that its orientation is preserved (i.e., all horizontal sides remain horizontal and all vertical sides remain vertical).

At times, we may want to simply modify an individual rectangle by moving one of its constituent primitive elements (i.e., a vertex or an edge). In order to do this, we add the `move vertex` and `move edge` operations. The `move vertex` operation proceeds in the same manner as the `move` operation for a point object except that the point that is being moved is a vertex of a rectangle. The `move vertex` operation moves the closest vertex $v$ of the nearest rectangle object $o$ to position $m$ of the mouse. The identity of $o$ is determined by the `nearest` operation for rectangle objects (see Section 10.2.2). Once vertex $v$ of rectangle object $o$ has been determined, $v$ is moved so that $v$ is coincident with $m$ and all subsequent motions of the mouse result in a maintenance of the coincidence while preserving the orientation of $o$. The result is that $o$ is grown or shrunk in size through the motion of $v$ while the vertex diagonally opposite from $v$ is not moved.

The `move edge` operation proceeds in a similar, but more restricted, manner as the `move vertex` operation. The restriction is that the moved edge $e$ maintains its length and orientation throughout the motion, and all remaining edges are not moved although $e$'s two adjacent edges are allowed to grow longer and shrink. The `move edge` operation moves the closest edge $e$ of the nearest rectangle object $o$ to position $m$ of the mouse. The identity of $o$ is determined by the `nearest` operation for rectangle objects (see Section 10.2.2). Once edge $e$ of rectangle object $o$ has been determined, $e$ is moved in the direction of its normal so that the infinite extension of $e$ is coincident with $m$ and all subsequent motions of the mouse result in a maintenance of the coincidence while preserving the length of $e$ and the orientation of $o$. The result is that $o$ is grown or shrunk in size through the motion of $e$ while the edge of $o$ that is parallel to $e$ is not moved.

Moving line segment objects is more complex than moving rectangle objects although they are handled in a similar manner as point and rectangle objects. The key difference is that a collection of line segment objects can take on two forms. The first is as a collection of individual objects which is treated in a similar way to a collection of point and rectangle objects. The second is as a collection of connected objects formed by sharing their vertices. This makes the members of a collection of line segment objects inter-dependent whereas the members of a collection of point and rectangle objects are independent. The result is known as a polygonal map or subdivision.

As in the case of rectangle objects, we must make a distinction between moving a line segment object or moving its constituent elements of which, in this case, there is just one kind — that is, a vertex. The difference in the nature of the possible collections of objects between the `move` operation for line segment objects and the `move` operation for rectangle objects is that when we have a connected collection, we no longer can move the line segments and vertices as independent entities. In other words, the `move` operation must move all line segments incident at the vertices of the line segment being moved or at the vertex being moved.

The application of a `move` operation to a line segment object proceeds in a similar manner as the `move` operation for a rectangle object. Once again, we assume that line segment $o$ is the closest to the position of the mouse $m$. The identity of $o$ is determined by the `nearest` operation for line segment objects (see Section 10.2.2). In addition, we also determine the identity of its two vertices as well as the identities of all of the line segments incident at them. As in the case of moving entire rectangle objects (rather than their constituent parts — that is, vertices and edges), we use the centroid of the line segment as its representative point. As the mouse is moved, the line segment object (as well as all line segments that are incident at its vertices) is moved in such a way that its length, its orientation (i.e., slope), and the topology of the underlying data set are preserved. In addition, the resulting configuration of line segments must be legal for this data structure. For example, Figure 10.4b shows the result of moving the rightmost edge in the polygonal map in Figure 10.4a (i.e., in the `NW` quadrant) to the right (via a `move` operation). Notice that the motion of the vertices that comprise this edge result in considerable changes to the original polygonal map as well as to the resulting decomposition.

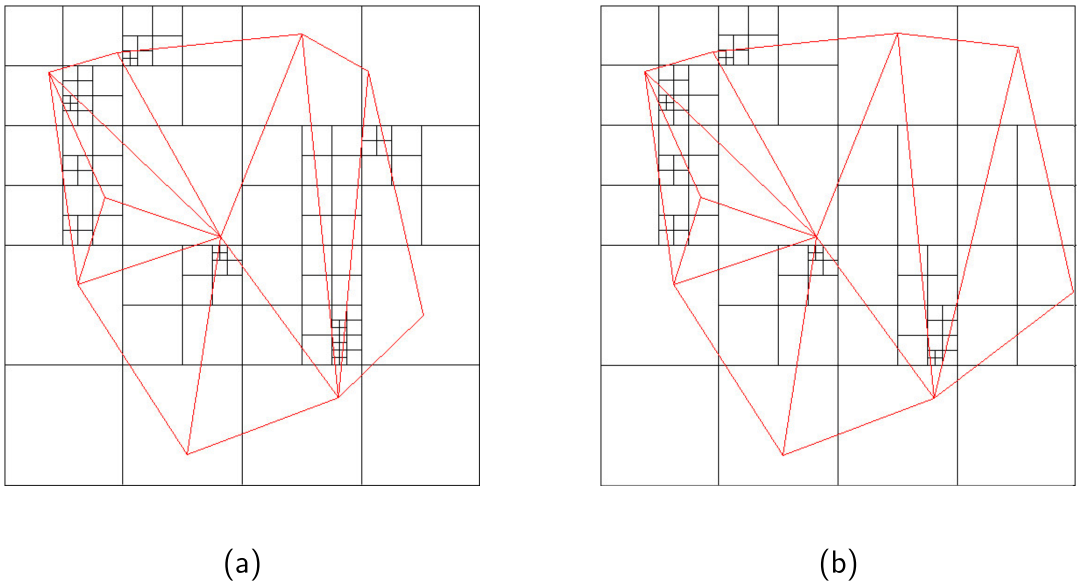


Figure 10.4: Figure 10.4b shows the result of moving the rightmost edge (via a 'move' operation) in the polygonal map in Figure 10.4a (i.e., in the NW quadrant) to the right.

Note that the fact that we may need to identify the vertices of the nearest line segment object differentiates this process from the `move` operation for rectangle objects. The principal difference is that since the primary entity is a line segment, the vertices are usually not stored explicitly in the data structure. However, in order to support operations such as deletion, nearest line segment object to a point, etc., recall from Section 10.2.4 that as part of the history list we maintain an auxiliary data structure in the form of a PR quadtree for the vertices of the line segment objects.

Having located the vertices $v_1$ and $v_2$ of the closest line segment $o$, we find all the line segments that are incident at $v_1$ and $v_2$ using the incremental nearest neighbor algorithm. In particular, for each of the vertices, the algorithm is invoked to find all line segments in the structure with distance 0 from them. Although this guarantees finding all line segments that share these vertices, unfortunately it also returns those line segments that pass through them without having them as an endpoint. A simple test weeds out these line segments from further consideration.

The `move vertex` operation proceeds in a similar manner as the `move` operation for point objects in that we find the closest vertex $v$ to the position $m$ of the mouse using the incremental nearest neighbor algorithm as was done for point objects. In this case, we are searching the auxiliary PR quadtree for the vertices. Notice that there must be at least one line segment connected to $v$ since otherwise $v$ would not be represented in the PR quadtree. In addition, we also determine the identity of all of the line segments incident at it using the incremental nearest neighbor algorithm. As the mouse is moved, $v$ (as well as all line segments that are incident at $v$) is moved in such a way that the topology of the underlying data set is preserved. In addition, the resulting configuration of line segments must be legal for this data structure. It is important to note that only vertex $v$ and the line segments incident at $v$ are actually moved. The remaining vertices of the line segments incident at $v$ are not moved. A useful byproduct of the `move vertex` operation is that it can be used to merge vertices so that the set of line segments incident at them are merged. This is difficult in the sense that it is hard to detect visually when two vertices are coincident.

A similar problem arises when creating a polygonal subdivision through the addition of a new line segment and it is desired for the new line segment to share a vertex with an existing line segment. This is achieved during line segment insertion by keeping the ‘Control’ key depressed. We use the same principle for merging vertices by stipulating that if the ‘Control’ key is depressed when the mouse is released during the execution of a `move vertex` operation on vertex $v_1$, then the system locates the nearest vertex $v_2$.

The line segments incident at $v_1$ and $v_2$ are joined at $v_2$ unless the resulting configuration leads to an illegal arrangement of line segments (e.g., intersecting lines in the case of a $\mathrm{PM}_1$ quadtree which, though, is legal for a PMR quadtree) in which case no merge takes place and the operation ceases with $v_1$ being located at its final legal position for the data structure currently being displayed. Observe that any existing edge between the merged vertices is removed as a result of the merge as we do not allow line segments of zero length.

We believe that our solution is preferable to the introduction of an additional operation named `merge vertex` which would merge the selected vertex with its nearest neighbor.

### 10.2.6 Effect of Move on the Underlying Data Structures and the Display Canvas

There are a number of ways of updating the display canvas as well as the history list to reflect the results of the `move` operation. In part, they depend on the granularity of the `move` operation. In particular, an important question is how often do we update the display canvas during the motion of the mouse and the object that is bound to it. As the applet senses the motion of the mouse, an event is generated indicating the new position and the applet updates the display canvas. During this update, no further sensing of motion is done and thus the update operation is not interrupted by subsequent mouse motions.

When the `move` is executed, only the currently displayed data structure is updated. No modifications are made to the other data structures as they do not physically exist at this time. However, we do need to modify the history list in some way to account for the motion. In this section, we discuss a number of possible methods of updating the history list, as well as explain the one that we have chosen.

The simplest and most naive way of updating the history list is to add a pair of entries corresponding to (`delete` $o$ at $p_1$, `insert` $o$ at $p_2$) each time the applet senses the motion of the mouse, when bound to object $o$, from $p_1$ to $p_2$ (i.e., as the mouse is dragged). This is very cumbersome and results in dramatic growth of the history list. An alternative is to replace the sequence of (`delete` $o$ at $p_1$, `insert` $o$ at $p_2$)

by one pair of (`delete` $o$ at $p_s$, `insert` $o$ at $p_e$) entries corresponding to the starting location $p_s$ and ending location $p_e$ of the dragging process. This method does not get rid of all inefficiencies as users could conceivably drag an object from $p_1$ to $p_2$ and then continue to drag the same object from $p_2$ to $p_3$, $\ldots$ $p_{i-1}$ to $p_i$. In this case, we can replace the sequence by the pair of (`delete` $o$ at $p_1$, `insert` $o$ at $p_i$) operations.

The above updating methods assume that the path traced by the dragging process has no effect on the resulting data structure. This is not true in general for all of the data structures. Point data structures such as the PR quadtree, which recursively quarter the underlying space into equal-sized parts until each block contains at most one point, are completely independent of the order in which the objects (i.e., points in this case) are inserted and thus they are also independent of the path followed by the dragging process. However, during this path, node merges and/or splits may occur. Data structures such as the priority search tree and the two-dimensional range tree are also independent of the path traced by the dragging process. However, they are static data structures which must be rebuilt in their entirety each time an object is deleted or inserted, and thus they must also be rebuilt after an object has been moved.

Data structures such as the point quadtree [45] (which is the two-dimensional analog of the binary search tree) and the k-d tree (which is like a point quadtree except that at each level of decomposition only one axis is split) whose shape is affected by the order in which the objects are inserted are also independent of the path followed by the dragging process as the same object is being dragged. Thus once the object (i.e., point) is initially removed from the data structure, its subsequent insertions result in its placement in leaf nodes and thus the subsequent deletions do not have an effect on the remaining data structure with the exception of the possible merging of four leaf nodes and one split of a leaf node. In other words, only the first deletion has any effect.

However, some data structures such as the PMR quadtree are affected by the path followed by the dragging process. In particular, in the PMR quadtree, a deletion causes merging to be applied repeatedly as long as the occupancy of the resulting node is less than or equal to the value of the splitting threshold, while insertion causes any node intersected by the inserted object to be split once and only once if the splitting threshold is exceeded. Thus the shape of the structure is very sensitive to the path followed by the dragging process.

Nevertheless, keeping track of the dragging process in the history list is not really practical for large data sets as the storage requirements will no longer depend on the volume of the data; but, instead, will depend on the granularity of the system's reaction to the motion of the objects. In fact, even keeping one entry in the history list for every motion of an object is impractical and thus we turn to an alternative method of keeping track of the effect of the motion.

This alternative is based on the observation that the most natural way to deal with the path followed by the motion of the objects in the history list is to ignore it. Instead, we find the object that is moved and modify its actual location in the history list while keeping intact its position in the relative ordering.

Thus the object that is moved is not deleted which represents quite a departure from our previous proposed approach for handling data structures whose shape depends on the order of insertion (e.g., the point quadtree and the k-d tree). Notice that in some situations, more than one object will be modified in the history list. For example, when moving line segment objects, we usually preserve the topology of the underlying set of line segments (unless merging vertices in which case we may have to remove a line segment from the history list which will be treated as a deletion). Thus we don't only move the line segment that is closest to the position of the mouse but also both vertices of the line segment and all other line segments incident at them. The same holds when moving the vertex of a line segment object in which case we also move all line segments incident at it.

The only remaining question is how to update the display canvas to reflect the changes in the underlying data structure. The simplest solution is to rebuild the entire data structure from the newly modified history list and redraw the display canvas. However, rebuilding the data structure and redrawing the display canvas each time a mouse motion is detected can be time-consuming and thus we would like to minimize such rebuilding and redrawing. The key is to differentiate between the data structures on the basis of the degree of their dependence on the order in which their constituent objects are inserted (and deleted). There are four classes of data structures with the following actions.

1. Dynamic data structures which are independent of the order of insertion such as the MX and PR quadtrees for point objects; $\mathrm{PM}_1$, $\mathrm{PM}_2$, and $\mathrm{PM}_3$ quadtrees for line segment objects; and rect and MX-CIF quadtrees for rectangle objects. As the objects appear in leaf nodes, all that is needed is the execution of a simple deletion and insertion operation, and a redrawing of the relevant portions of the display canvas. This may include some node merges and/or splits.

2. Dynamic data structures which are dependent on the order of insertion but independent of the path followed by the dragging process such as the point quadtree and the k-d tree. As motion is started, we only need to rebuild the structure from the position of the moved object $o$ in the history list. However, this would require us to have access to the partial representation of the data structure which may not be feasible. So, instead, we rebuild the data structure in its entirety from the history list.

3. Dynamic data structures which are dependent on the path followed by the dragging process such as the PMR quadtree for point and line segment objects. The data structure is rebuilt from the history list each time a mouse movement is detected.

4. Static data structures which are always independent of the order of insertion but must be rebuilt each time there is a change in their membership or in any

of the elements that they contain.

It should be clear the animation of the `move` operation in classes 1 and 2 is simple for most data sets including large ones. Our experience has also been that the animation for classes 3 and 4 has also been smooth.

An interesting byproduct of our solution is that it avoids some tricky problems that arise as a result of moving a sub-primitive of an object. For example, when we move a vertex $v$ of a line segment object, if we would have used the (`delete`, `insert`) method, then we would have had to reinsert all of the line segments that are incident at $v$. The order in which we reinsert them may affect the shape of the data structure when the data structure is order-dependent (e.g., the PMR quadtree). The simplest solution is to reinsert them in the order in which the nearest neighbor algorithm found them. However, this is somewhat arbitrary and is not repeatable. Fortunately, our method of simply updating the locational description in the history list of the moved object preserves the order of insertion and thus we are not bothered by this problem.

## 10.3 Spatial Data Visualization in GIS

Careful consideration of client-side operations, both the user interface design and implementation of operations executed on the client side aspects, as discussed above, is an important prerequisite for developing an efficient client-server system. In real-life deployments of client-server architecture for accessing remote spatial data servers, visualizing spatial structures and understanding of their inner working mechanisms, which was our focus in [36, 37, 38], is typically not of high importance. What matters more are the ways in which the actual objects stored in a spatial data structure are visualized. Here, difficult issues arise even when only a single dataset is involved. For instance, in the case of overlap between individual objects in the set, we need to decide on a way to display the data so that no information is lost by not showing an object or a part thereof that happens to be hidden behind another object. This becomes an even more burning issue when more datasets get involved in creating the final view on the database. This is the case in many typical applications of spatial data structures, such as a map representation.

Most maps are created by overlaying multiple layers of spatial data. For instance, a typical auto map can consist of road layer displayed as a network of black lines, a river layer displayed as network of blue lines, forested areas displayed as a layer of green polygons, a field layer displayed as a layer of yellow polygons etc. Based on what type of features the individual layers represent, certain rules regarding possible overlaps between individual layers can be established. For instance, it is obvious that a certain area cannot be included in the layer that represents forested areas and in the layer that represents lakes at the same time. To make things more complicated, this assumption can be made if the map captures an approximation of the reality at a single moment (as is usually true). Special provisions may need to be made in cases that the reality changes and this change needs to be somehow represented in the map. For instance, the water level in a lake may vary by the season and the map preparers could decide to represent this fact by representing both the lake and the surrounding soil in the map in their largest extent. Thus an overlap between the land and the lake is made even though normally this situation would not be expected.

While overlap of land and water may be unusual, overlaps of other types of layers are to be expected. This is especially true if different types of information are to be displayed in a single map. For instance, users may want to display both physical features (rivers, roads, forests, etc) as well as administrative subdivisions (counties, national parks, etc). While some of these can be represented by their contours, sometimes it is inevitable to display two or more overlapping layers that use solid objects (e.g., polygons) to represent their data. Obviously, in this case the top layer may be obstructing views into the other layers and this is an issue that needs to be addressed.

# Chapter 11

# Conclusion

The main focus of our work was to investigate and optimize remote access to spatial databases. We have reviewed existing solutions, most of them bitmap-based, and identified some of their advantages and drawbacks. We have proposed, designed, and implemented a new vector-based system that could be used in situations where the traditional approaches do not work too well. We have compared performance of a bitmap-based system with our vector-based SAND system. The results of our experiments allow us to suggest the best type of a remote spatial data visualization tool for a given the specific deployment scenario.

We have developed a modular design for infrastructure that facilitates remote spatial data access. We applied this design to implement several specific types of SAND system deployment. Which type of the deployment performs best depends on the specific environment in which the system is to be used. Generally, the system can either be deployed in such a fashion that the clients communicate directly with the central spatial server. Alternatively, in situations where the client runs on a thin platform or where the service is shared among several co-located clients, an auxiliary server could be used to improve the performance of the overall solution.

## 11.1 Direct Client-Server Communication

In the simpler setup, the clients communicate directly with the central server. The bitmap approach works better if only a few operations are needed per session. This is the case when the user only needs to see a basic map of a given area and does not plan on any further exploration or querying. If the user performs a sequence of operations (zooming, panning) in a given area, then the vector-based SAND approach performs better than the bitmap-based solution across all types of connections, all operations, and for all possible SAND deployments (i.e., the requested content is fully cached, not cached, or caching is not involved at all and the data is always fetched directly from the server). The more the user revisits previously seen areas, the greater are the benefits to be derived from SAND's caching.

In the case of the fast scroll operation where the new view overlaps 50% of the old view, improvements from using SAND range from a factor of 4 (using a fast LAN connection) to 40 (using a slow dial-up modem or a high-latency satellite link) over using the bitmap solution when the content is fully cached. If the content is not cached, then SAND performs between 1.5 and 2 times better. If caching is not available on the client and all data has to be downloaded from the server, then SAND performs 1.5 times better for slower network connections and up to 3 times worse for a fast LAN-type connection with both approaches being comparable for network speeds in between.

In the case of the fine scroll operation, where the new view overlaps 90% of the old view, improvement from using SAND ranges from a factor of 3.5 (using a fast LAN connection) to 35 (using a slow dial-up modem or a high-latency satellite link) over using the bitmap solution when the content is fully cached. If the content is not cached, then SAND performs between 1.5× and 2.5× better. If caching is not available on the client and all data has to be downloaded from the server, then SAND performs 1.5× worse than the bitmap solution for a LAN-type connection and between 1.5 and 2.5× faster for slower connections.

For zoom in operations, the improvement from using SAND ranges between 5 and 50 times faster than a bitmap system. For the zoom out operation, the improvement from using SAND is 2 to 20 times faster when client-side caching is available, and slightly better to 4 times worse for the scenario where caching is not available locally and all data has to be fetched from the server.

Therefore, we see that if users are only expected to use the system briefly, perhaps a quick address lookup followed by one or two zoom or scroll operations, the deployment of a bitmap system would be more efficient. However, when users are expected to use the system beyond just a few operations, it is better to deploy a vector-based system such as SAND. If possible, local caching should be used but better results are achieved even when data is always fetched directly from the server.

The tile-oriented bitmap method of accessing spatial data as introduced by Google Maps and Microsoft Virtual Earth would provide better results than the traditional bitmap method represented by MapQuest or MapsOnUs. While direct comparison in an identical environment is not possible, we can estimate what its performance would be in our test environment by extrapolating results from our MapServer experiments. If the data is already cached on the client, then the tile method and SAND would perform the same. If the data is not cached, then the tile method would perform about 4 times better on a fine scroll operation. Both methods would perform about the same for the fast scroll operation. The results of a comparison for the zoom in and zoom out operations will remain the same as they were for the regular bitmap approach. Therefore, while the tile method performs very well as a pure viewer, the SAND system offers a platform for better load sharing between client and the server while providing similar performance. This is due to having the vector data present on the client. In this way, the SAND Internet Browser can handle certain operations and queries locally, thereby speeding up the response time and relieving the server's load.

## 11.2 Auxiliary Server Deployment

In some cases, multiple co-located clients need to utilize a spatial server, while the connection speed between the location of the clients and the server is poor. In other cases, the clients do not support internal caching. For such scenarios, we proposed to introduce auxiliary proxy servers that speed up response time for the clients, while decreasing the load on the central spatial server. Such a proxy can be either pre-loaded with the spatial data stored on the central server or it can be installed without any data and allowed to download the data from the central server as necessary. Assuming a scenario where the local network connection between the clients and the proxy is fast (e.g., they are on the same LAN) and that the connection between the proxy and the central server is substantially slower (e.g., a satellite link), we saw that the scroll operations are about 6–8× faster for SAND. The zoom in operation can be 20× faster while zoom out is about 8× faster.

The extra overhead of managing the proxy makes the SAND solution slower initially when the proxy server does not contain any data. In this phase right after deployment, SAND can be 3× slower than the MapServer system. However, once the data becomes available on the proxy, SAND becomes about 3 times faster.

## 11.3 Future Research

There are many directions for future research in the area of spatial database remote access. One area of interest would involve investigation of methods for caching frequently used data in the form of bitmap tiles instead of vectors. While these tiles would only be usable in given views (in terms of zoom factor and layers displayed), they would also allow skipping of repeated rasterization steps. Such a system would in effect be a hybrid between SAND as it is today and the tile approach introduced by Google Maps.

# Bibliography

[1] ArcView — the geographic information system for everyone. `http://www.esri.com/software/arcgis/arcview`.

[2] ESRI Shapefile technical description. `http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf`.

[3] Geo::ShapeFile — Perl extension for handling ESRI GIS Shapefiles. `http://search.cpan.org/~jasonk/Geo-ShapeFile-2.51/`.

[4] Google Maps. `http://maps.google.com`.

[5] IBM Informix Spatial DataBlade. `http://www.ibm.com/software/data/informix/blades/spatial/`.

[6] Java — programming language. `http://java.sun.com`.

[7] MapInfo Professional. `http://www.cmcus.com/Products/Vendors/MapInfo/mapinfopro.asp`.

[8] MapServer — open source development environment for constructing spatially enabled internet-web applications. http://mapserver.gis.umn.edu.

[9] MSN Virtual Earth. http://virtualearth.msn.com.

[10] MySQL — the world's most popular open source database. http://www.mysql.com.

[11] NIST Net — National Institute of Standards and Technology network emulation package. http://snad.ncsl.nist.gov/itg/nistnet.

[12] OpenMap$^{TM}$ — open systems mapping technology. http://www.openmap.bbn.com.

[13] Oracle Application Server MapViewer. http://www.oracle.com/technology/products/mapviewer/index.html.

[14] PHP — PHP: Hypertext preprocessor. http://www.zend.org.

[15] PHP MapScript. http://www.maptools.org/php_mapscript.

[16] PostGIS — gis extension for postgresql object-relational database system. http://www.postgis.org/.

[17] RFC 2396 — uniform resource identifiers (URI): Generic syntax. http://www.faqs.org/rfcs/rfc2396.html.

[18] Sun Microsystems SunRay. http://www.sun.com/sunray/.

[19] Tool Command Language (Tcl). http://www.tcl.tk/software/tcltk/.

[20] TopoZone — the web's topographic map. http://www.topozone.com.

[21] U.S. Census Bureau — TIGER/Line. http://www.census.gov/geo/www/tiger.

[22] OpenGIS simple features specifications for SQL, 1999. http://portal.opengeospatial.org/files/?artifact_id=829.

[23] MapQuest: Consumer-focused interactive mapping site on the web. http://www.mapquest.com, 2002.

[24] MapsOnUs: Suite of online geographic services. http://www.mapsonus.com, 2002.

[25] ArcGIS server: ESRI's enterprise GIS application server, 2004. http://downloads.esri.com/support/whitepapers/other_/arcgis-server_90.p%df.

[26] PostgreSQL — the world's most advanced open source database, 2004. http://www.postgresql.org/about/.

[27] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126, ACM Press, Copper Mountain, CO, December 1995.

[28] ANSI. Database language SQL (X3.135-1992), 1992.

[29] ISO & ANSI. Database language SQL - part 2: Foundation, 1999.

[30] W. G. Aref and H. Samet. Optimization strategies for spatial query processing. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, G. M. Lohman, A. Sernadas, and R. Camps, eds., pages 81–90, Barcelona, Spain, September 1991.

[31] W. G. Aref and H. Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[32] K. Arnold and J. Gosling. *The JAVA*$^{\text{TM}}$ *Programming Language.* Addison-Wesley, Reading, MA, 1996.

[33] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[34] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, June 1979. (see 1980 article with the same name).

[35] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[36] F. Brabec and H. Samet. The VASCO R-tree JAVA$^{\text{TM}}$ applet. In *Visual Database Systems (VDB4)*, Y. Ioannidis and W. Klas, eds., Chapman and Hall, London,

United Kingdom, May 1998. Also *Proceedings of the IFIP TC2//WG2.6 Working Conference on Visual Database Systems 4 (VDB4)*, L'Aquila, Italy, May 1998.

[37] F. Brabec and H. Samet. Visualizing and animating R-trees and spatial operations in spatial databases on the worldwide web. In *Visual Database Systems (VDB4)*, Y. Ioannidis and W. Klas, eds., Chapman and Hall, London, United Kingdom, May 1998. Also *Proceedings of the IFIP TC2//WG2.6 Working Conference on Visual Database Systems 4 (VDB4)*, L'Aquila, Italy, May 1998.

[38] F. Brabec and H. Samet. Visualizing and animating search operations on quadtrees on the worldwide web. In *Proceedings of the 16th European Workshop on Computational Geometry*, K. Kedem and M. Katz, eds., pages 70–76, Eilat, Israel, March 2000.

[39] F. Brabec, H. Samet, and C. Yilmaz. VASCO: visualizing and animating spatial constructs and operations. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 374–375, San Diego, CA, June 2003.

[40] K. Buehler and L. McKee, eds. *The OpenGIS Guide — Introduction to Interoperable Geo-Processing*. OpenGIS Consortium, Wayland, MA, 1996. `http://www.opengis.org/guide`.

[41] A. R. Butz. Convergence with hilbert's space filling curve. *Journal of computer and system sciences*, 3(2):128–146, May 1969.

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

[43] C. B. Cranston, F. Brabec, G. R. Hjaltason, D. Nebert, and H. Samet. Adding an interoperable server interface to a spatial database: Implementation experiences with OpenMap$^{\text{TM}}$. In *Interoperating Geographic Information Systems — 2nd International Conference, INTEROP'99*, A. Včkovski, K. Brassel, and H.-J. Schek, eds., pages 115–128, Zurich, Switzerland, March 1999. Also Springer-Verlag Lecture Notes in Computer Science 1580.

[44] C. Esperança and H. Samet. Spatial database programming using SAND. In *Proceedings of the 7th International Symposium on Spatial Data Handling*, M. J. Kraak and M. Molenaar, eds., pages A29–A42, Delft, The Netherlands, August 1996. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information.

[45] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[46] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[47] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th IEEE Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, MI, October 1978.

[48] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):401–444, October 1994.

[49] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[50] J. R. Herring. Oracle7 spatial data option$^{\text{TM}}$: Advances in relational database technology for spatial data management. Technical report, Oracle Corporation, September 1996.

[51] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 83–95, Portland, ME, August 1995. Also Springer-Verlag Lecture Notes in Computer Science 951.

[52] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. Computer Science Department TR-3919, University of Maryland, College Park, MD, July 1998.

[53] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also University of Maryland Computer Science TR-3919.

[54] G. R. Hjaltason and H. Samet. Improved bulk-loading algorithms for quadtrees. In *Proceedings of the 7th ACM International Symposium on Advances in Ge-*

*ographic Information Systems*, Claudia Bauzer Medeiros, ed., pages 110–115, Kansas City, MO, November 1999.

[55] G. M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[56] K. V. R. Kanth, S. Ravada, J. Sharma, and J. Banerjee. Indexing medium-dimensionality data in racle. In *Proceedings of the ACM SIGMOD Conference*, Philadelphia, June 1999.

[57] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982.

[58] A. Klinger. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, ed., pages 303–337. Academic Press, New York, 1971.

[59] H. P. Lenhof and M. Smid. An animation of a fixed-radius all-nearest-neighbors algorithm. In *Proceedings of the 10th Annual Symposium on Computational Geometry*, page 387, Stony Brook, NY, June 1994.

[60] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[61] R. B. McMaster and K. S. Shea. *Generalization in Digital Cartography*. Association of American Geographers, Washington, DC, 1992.

[62] M. Murphy and S. S. Skiena. Ranger: A tool for nearest neighbor search in high dimensions. In *Proceedings of the 9th Annual Symphosium on Computational Geometry*, pages 403–404, San Diego, CA, May 1993.

[63] D. Nebert. WWW mapping in a distributed environment: Scenario of visualizing mixed remote data, 1997. (see `http://javamap.bbn.com/projects/matt/development/specialist.html`).

[64] 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.

[65] R. C. Nelson and H. Samet. A population analysis of quadtrees with variable node size. Computer Science TR-1740, University of Maryland, College Park, MD, December 1986.

[66] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, San Francisco, May 1987.

[67] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[68] Oracle Corporation. Advances in relational database technology for spatial data management. Oracle spatial data option technical white paper, Oracle Corporation, September 1996.

[69] D. Raggett. HTML 3.2 reference specification. `http://www.w3.org/TR/REC-html32`, 2001.

[70] S. Ravada and J. Sharma. Oracle8i spatial: Experiences with extensible databases. In *Advances in Spatial Databases — 6th International Symposium, SSD'99*, R. H. Güting, D. Papadias, and F. H. Lochovsky, eds., Hong Kong, July 1999. Also Springer-Verlag Lecture Notes in Computer Science 1651.

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

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

[73] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *Communications of the ACM*, 46(1):63–66, January 2003.

[74] H. Samet and F. Brabec. Remote thin-client access to spatial database systems. In *Proceedings of the 2nd National Conference on Digital Government Research*, pages 75–82, 409, Los Angeles, CA, May 2002.

[75] H. Samet, F. Brabec, and G. R. Hjaltason. Interfacing the SAND spatial browser with FedStats data. In *Proceedings of the dg.o 2001 Conference: Connecting Government and the People Electronically*, pages 41–47, Redondo Beach, CA, May 2001.

[76] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985. Also *Proceedings of Computer Vision and Pattern Recognition'83*, pages 127–132, Washington, DC, June 1983 and University of Maryland Computer Science TR-1372.

[77] J. Sankaranarayanan, E. Tanin, H. Samet, and F. Brabec. Accessing diverse geo-referenced data sources with the SAND spatial DBMS. In *Proceedings of the 3rd National Conference on Digital Government Research*, pages 331–334, 297, Boston, May 2003.

[78] M. Scholl and A. Voisard, eds. *Advances in Spatial Databases — 5th International Symposium, SSD'97*, Berlin, Germany, July 1997. Also Springer-Verlag Lecture Notes in Computer Science 1262.

[79] M. Stonebraker. Limitations of spatial simulators for relational DBMSs. Technical report, INFORMIX Software, Inc., 1997. `http://www.informix.com/informix/corpinfo/zines/whitpprs/wpsplsim.pdf`.

[80] E. Tanin, F. Brabec, and H. Samet. Remote access to large spatial databases. In *Proceedings of the 10th ACM International Symposium on Advances in Ge-*

*ographic Information Systems*, A. Voisard and S.-C. Chen, eds., pages 5–10, McLean, VA, November 2002.

[81] E. Tanin, A. Harwood, and H. Samet. Indexing distributed complex data for complex queries. In *Proceedings of the 4th National Conference on Digital Government Research*, pages 81–90, Seattle, WA, May 2004.

[82] E. Tanin and H. Samet. APPOINT: An Approach for Peer-to-Peer Offloading the INTernet. In *Proceedings of the 2nd National Conference on Digital Government Research*, pages 99–105, Los Angeles, CA, May 2002.

[83] E. Tanin and H. Samet. Improving access to large volumes of online data. In *Proceedings of the 3rd National Conference on Digital Government Research*, pages 99–104, Boston, May 2003.

[84] F. J. Wang and S. Jusoh. Integrating multiple web-based geographic information systems. *IEEE MultiMedia*, 6(1):49–61, January–March 1999.

[85] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[86] Z. Yang and K. Duddy. Corba: a platform for distributed object computing. *SIGOPS Oper. Syst. Rev.*, 30(2):4–31, 1996.

[87] C. Yap, K. Been, and Z. Du. Responsive thinwire visualization: Application to large geographic datasets. In *Proc. 14th Ann. Symp., Electronic Imaging 2002.* IS&T/SPIE, 2002. 19-25 Jan, 2002, San Jose, California.