OSHADHI-3
A Biodiversity Information System

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
System Science and Automation

by

MANJUSHA N. GEDAM

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

Jan 1999

“Dedicated to my loving Brother”

Akhilesh

Contents

Acknowledgements

I would like to thank my project guide, *Dr. Jayant Haritsa*, for his excellent guidance throughout the course of this project. I am grateful to him for his constant support and encouragement, without which this project would not have materialized. I take this opportunity to express my deepest gratitude to him for the moral support that he gave me in my days of agony and trauma. I would like to thank Prof. Dr. U.R. Prasad, Dr. Preeti Shankar, Dr. Y. Narhari who gave me the encouragement and moral strength to carry on my project.

I would like to thank my advisor Dr. Vijay Chandru for his guidance during my M.E course.

I would also like to thank Mr. Janardhan Pillai, Mr T.V Ramachandran for providing me some tips on the data I required for the project.

I am very grateful to Vikram who me helped throughout the project and gave me valuable suggestions from time to time. I thank all my labmates who made *DSL* a pleasant place to work.

I express my special thanks to all my friends Santanu, Rupesh, Sonali, Pankaj and others who helped me throughout my stay in IISc and made my stay here a pleasant and memorable one. I would always cherish the good and sad times that I spent with them. Ajay has been encouraging, guiding and supporting me throughout my stay in IISc.

My *Aai* and *Baba* have been the greatest source of inspiration to me. They provide me with all the necessary strength for all my work. All the credit of my success goes to them.

Abstract

The biodiversity conservation of the large number of plant species of India has become very essential with the rapid growth in the number of plant species which strengthen the genetic base of India due to their rich economic importance. Several organizations undertaking the measures to conserve the biodiversity such as identification of species and monitoring the climatic conditions, have realized the need for efficient management of the large amount of biodiversity information which is currently available in various forms. Traditional database systems (built on relational, hierarchical and network models) which are widely used for commercial applications such as banking, fail to meet the modeling and processing requirements of the bio-diversity data. To address this lacuna, the OSHADHI project was started at *Database Systems Lab* (DSL), *Indian Institute of Science*. The goal of this project was to design and develop an object oriented bio-diversity information system called OSHADHI. There were two stages of this project OSHADHI-1 and OSHADHI-2. OSHADHI-1 addressed the problem of biodiversity data management and incorporates aspects of bio-diversity information and information about various geographical regions, climatic conditions, etc. It also supports an interface for asking some simple queries to the database through a graphic user interface. In OSHADHI-2 a query language (*Vaidya*) was designed and implemented over the OSHADHI object model to enable the user to ask complicated queries. Second, some newer indexing methods were added.

In this project, we have concentrated on the **spatial** part of the biodiversity data. The earlier two stages of OSHADHI biodiversity system mostly concentrated on the Taxonomy part of the biodiversity data. This project aims at finding the answers to all possible queries arising for geographic database. We use the **vectored** form of spatial data which make the implementation simple and flexible. We have added a Graphical User Interface through which the user interacts with the system and obtains the replies of his requests either as display on the Canvas or on Pop-up dialog boxes. We have built this system on a new Platform called **SHORE** unlike the previous stages of OSHADHI which were built on EXODUS. In addition to this, we have provided an indexing technique for Spatial data called the region quad tree index. This index is useful for efficient searching of maps over a range.

Chapter 1

Introduction

1.1 Motivation

The genetic base of India is represented by a large number of plant species which include the "local" species (species that grow naturally in the country) and the "foreign" species (species that have been brought in from various parts of the world). Over the past few years, several new local species have been identified and there has also been a rapid growth in the number of foreign species. The rich economic potential and uses of these plant species make it essential for their *biodiversity* to be conserved. Several measures for biodiversity conservation have been suggested by researchers and scientists in the area of environmental sciences. These measures include identification of species, monitoring the status of endangered species and monitoring the climatic conditions of various geographic locations [Decc95].

Several organizations all over the country are working towards the implementation of these measures in order to conserve the biodiversity of the species. However, one of the major hurdles faced by the scientists and researchers working in this area is the management of the large amount of biodiversity data. The biodiversity data includes a variety of information such as information about the species and their characteristics (such as flower, fruit, stem, habitat, etc.), information about endangered species and species of medicinal value, climatic conditions such as temperature, soils and land use, distribution of various species in different geographic locations, the details about botanical gardens and herbariums [1], references (bibliography) for various species and geographic regions, details about the experts in the related fields, etc. Biodiversity data may therefore be classified into four inter-related groups, namely, the *taxonomy* data (which includes details of all the species, their classifications and characteristics, etc.), the *geographic* data (which includes the details of the geographic locations, their environmental conditions, etc.), the *bibliographic* data (which includes the details of the journals, articles and textbooks in the field of biodiversity) and the *experts* data. The organization of the biodiversity data is shown in Figure 1.1.

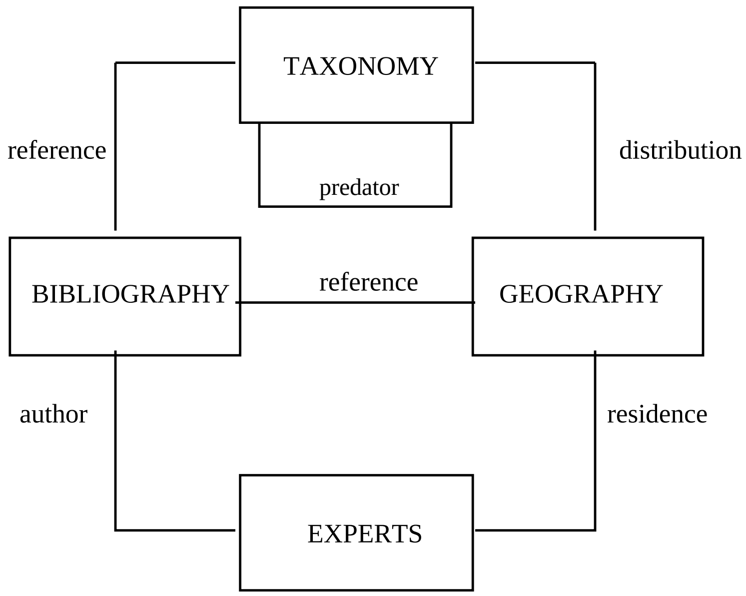

Figure 1.1: The biodiversity data organization

The biodiversity information mentioned above is currently available in various forms and there are several sources for this information. For example, a large amount of information is available in the form of textbooks and herbarium sheets [2], and in various places like colleges, universities and botanical gardens. However, maintaining the information in these forms makes its management extremely slow and inefficient. Hence, what

[1] Places where plant specimens are stored
[2] These are sheets that contain a plant specimen and its details.

is required is a system to manage the biodiversity data in an efficient way and to provide quick retrieval facilities. Apart from the need to aid the process of biodiversity conservation, maintenance of information such as whether a species is a local or foreign species is essential in the light of the *General Agreement on Trade and Tariffs (GATT)*. Also, it is essential to know in which region the species is found, its density of occurence, etc.

1.2 The OSHADHI system

To address the lacuna mentioned above the OSHADHI [3] biodiversity database system was started in *Database Systems Lab(DSL), Indian Institute of Science*. The project aims to design and implement an object model that best represents the biodiversity data, and a query engine to enable querying on the database. The initial prototype of OSHADHI was developed in an earlier work [Vid95], which we refer to as OSHADHI-1 in this thesis. The various entities of bio-diversity data were identified and modeled, and the OSHADHI architecture was designed. The data model was implemented and the system was tested on a toy database.

OSHADHI-2 [Mou97] was developed as an extension of OSHADHI-1. A query language called *Vaidya* was designed and an interpreter for that language was implemented. In addition, some indexing techniques that are specific to object oriented models were implemented to improve system performance.

However, in the above two works, support for geographic data was minimal. Therefore, we have concentrated mostly on modelling geographic data and related queries. Further, we have used the *Sequoia 2000 Storage Benchmark* [Seq93] as a guideline for representing geographic data and related queries. We also use a new backend *SHORE* [Sho97], which is a persistent object system, that represents a merger of object-oriented database (OODB) and file system technologies.

[3] plant kingdom in Sanskrit.

1.2.1 OSHADHI-1

Object oriented databases appeared to be most suited to handle the type of data encountered in bio-diversity information. Initially, the bio-diversity domain was studied and the objects of this domain were identified. Since the object-oriented approach helps to represent the real world objects naturally, it was used to model the bio-diversity. Subsequently, the architecture of the system was designed. The model was carefully analyzed and adapted to a database schema. The OSHADHI system uses **EXODUS** [Care86a] as the backend which is an *Extensible Object Oriented Database Management System* [DBE87]. The implementation included, simple query processing routines on the bio-diversity object model and a Graphical User Interface for querying the database.

In the next subsection, we describe the additional features incorporated in OSHADHI-2 as compared to OSHADHI-1.

1.2.2 OSHADHI-2

Some new features were added in OSHADHI-2 namely: A textual interface was provided. The query language that was designed and implemented is called **Vaidya**[4]. The performance of the OSHADHI system was improved by adding indexing methods such as hcC tree [SS94] and Nested Index [BK89]. A few geographical queries could be answered like whether a given point is inside a given country or not.

1.2.3 Limitations

While the above prototype implementation, along with the data model does simple query processing on the data model, it has some limitations.

- Most of the queries that could be answered were for the taxonomy part of the bio-diversity data. The above systems could not answer complex spatial queries. For example, a spatial query could have complex spatial join, overlapping regions to be found, etc.

[4] ‘physician’ in Sanskrit

- Both OSHADHI-1 and OSHADHI-2 systems use **EXODUS** as backend. However, EXODUS suffers from several limitations which will be discussed in detail in this thesis.

- The quad tree indexing method that was used in OSHADHI-1 covers only point indexing. What is required for efficient recovery of maps i.e. spatial objects is what is called as the region quad trees, which was implemented in OSHADHI-3 and will be discussed.

- These systems do not provide any of the functionalities which a Geographic Information System must have, for example zooming for a map.

In OSHADHI-3 we have emphasised on all the above mentioned points. In the next subsection, we describe these features incorporated in OSHADHI-3.

1.2.4 OSHADHI-3

The features that are incorporated in OSHADHI-3 are as follows:

Spatial Queries

Complex spatial queries like joining a point with a polygon or a polygon with a polygon are handled which are discussed in [Seq93]. OSHADHI-3 answers most of the queries discussed in this benchmark.

Geographic Information System

OSHADHI-3 provides certain functionalities which a Geographic Information System must provide. These functions include zooming, etc. Also given a map, one can find its parent (meaning that if a country is displayed on the Canvas you can display the parent namely the Continent to which the country belongs) or given a map you can find its children (meaning given a country you can find all its states of the country and rivers that flow in the country).

Backend

OSHADHI-3 is built on **SHORE** *(Scalable Heterogeneous Object Repository)* [Sho97], unlike OSHADHI-1 and OSHADHI-2 which were built on EXODUS *Extensible Object Oriented Database Management System*, which is a persistent object system, as backend. Shore is a merger of object-oriented database and file system technologies. It overrides most of the drawbacks in EXODUS, which will be discussed in detail.

Indexing

The indexing technique that is followed in OSHADHI-3 is quad tree indexing. This technique was used in OSHADHI-1. However, as mentioned above it could handle only point queries meaning it could only efficiently search for a given point. In OSHADHI-3 we have implemented region quad trees which can efficiently search for a region in a given rectangle.

1.3 Related Projects

Several projects have been undertaken by various organizations with the aim of tackling some subsections of the biodiversity data management problem. For example, *INMED PLAN (Indian Medicinal Plants National Network of Distributed Database)* is a database built for the medicinal plants of India. The other related projects are the *Mangrove Ecosystems Information Service*, the *Indo-based PGR project*, the *ERIN (Environmental Resources Information Network)*, the *PLATO (PLAnt TOxins)-UK* project [Poun95] and the Australian *INTKEY* project.

Most of these projects have been built either using commercial database systems such as ORACLE [Orac88], or using other technologies like Hypermedia, etc. Some projects like the PLATO-UK and INTKEY use a database in the DELTA (DEscriptive Language for TAxonomy) format [Poun95].

Most of these systems deal with the Taxonomy part of the bio-diversity data. For the geographic part, dealing mostly with spatial queries, a project has been completed at the

University of Wisconsin, *Paradise* [Par97].

There are also some products for Geographic Information System, namely, **ArcView GIS** *The Geographic Information System for Everyone* [Arc97].

We will discuss the comparison of our system with the above mentioned product and project in Chapter 7.

1.4 Organization of the Report

The rest of the report is organized as follows: Chapter 2 gives overview of some of the aspects of bio-diversity data. Chapter 3 gives overview of what was previously done as part of OSHADHI-1 and OSHADHI-2 and gives the architecture of OSHADHI-3. The SHORE architecture and facilities are described in Chapter 4. Chapter 5 lists all the queries that can be handled by OSHADHI-3. Chapter 6 discusses some of the implementation details of the system. In Chapter 7, we compare the system with some standard GIS systems and geo-spatial databases. We highlight some special features of OSHADHI-3 and the possible extensions that can be made to OSHADHI-3 in Chapter 8 and finally conclude in Chapter 9 by presenting a summary of the features of OSHADHI Biodiversity Database System.

Chapter 2

The Bio-diversity Domain

This chapter gives a brief description of the various aspects of bio-diversity data. We will first describe the classification of the bio-diversity data. In the later sections we discuss about features of bio-diversity database and the kind of queries that need to be answered by a bio-diversity database system. We will also discuss the three stages of OSHADHI bio-diversity system.

Most of the material in this Chapter is taken from [Vid95] and from [Mou97].

2.1 Classification

The bio-diversity data is classified into four major types – the taxonomy data, the geographic data, the bibliography data and the experts data.

2.1.1 Taxonomy Data

Each unit of classification is called a *taxon*. The four major taxons in the classification of species are *Order*, *Family*, *Genera* and *Species*. An order contains several related families, which means that families that exhibit certain similar characteristics are grouped into an order. Likewise, a family is represented by a collection of genera and a genera is a collection of related species [Vasi74]. Each taxon has a biological name. The classification hierarchy of species and an example are shown in Figure 2.1.1.

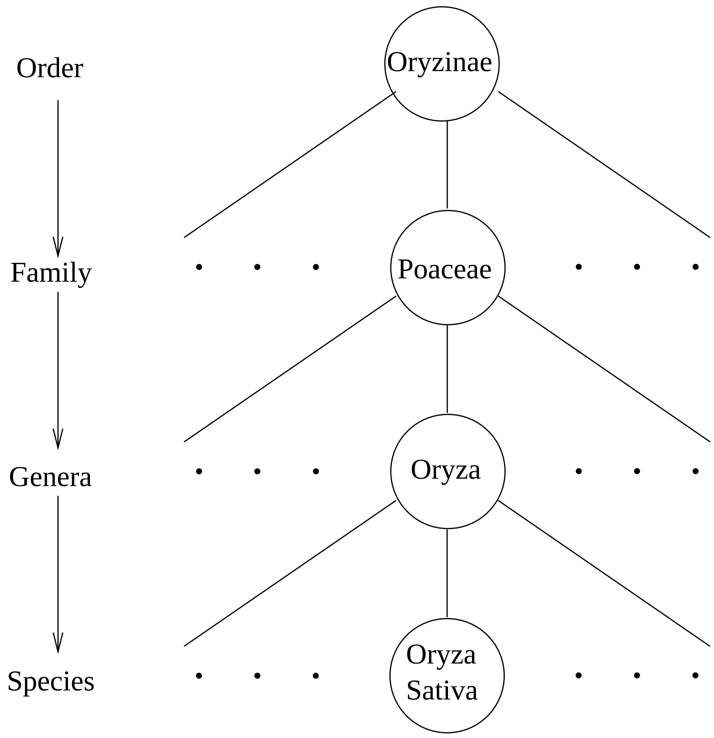

Figure 2.1: The taxonomy of species

Some characteristics of a species help in their identification like leaf, stem, flower and fruit and are therefore called as identifying characters. Other characters, for example, the root character doesn't aid in the identification of a species and are hence called non-identifying characters. **Example:** A flower is composed of several parts namely, petals, sepals, inflorescence, carpels, ovaries, anthers, style and stigma. Each of these characters could be used for identification of a species and hence they are also identifying characters. Hence, every order, family, genus and species is associated with a set of identifying and non-identifying characters which differentiate it from the other orders, families, genera and species respectively. A species may have synonyms (synonyms are commonly used names for species, eg, Rose is a synonym for Rosa Alba), line drawings and a large amount of textual information associated with it. The textual information may include extracts from journals and textbooks. A species may also have other species as its predators.

2.1.2 Geographic Data

There are essentially four types of geographical organizations – the *bio-geographic provinces* (e.g., Malabar, South Karnataka, etc,.), the *biomes* (e.g., evergreen, deciduous, etc,.), the *administrative organizations* (e.g., district, state, etc,.) and the *forests*. These organizations, along with the botanical gardens and herbariums, represent the geographical data. Every continent, country, state, district and city has a political map associated with it. The environmental conditions (soils, temperature, pressure humidity, etc,.) in the countries, states, districts and cities need to be recorded in two forms. One is the spatial variation (distribution) of the condition and the other is the temporal variation (variation of the condition over a period of time). Similarly, for all the species found in a geographic region, the distribution and temporal variation in population are to be recorded. The information about the distribution of a species could be available at any geographic organisation type. For example, we might have the information that a species is found in the Malabar province or that a species is found usually in evergreen forests or that a species is found in Karnataka. In addition we have to model the bio-diversity along the rivers and roads and also the aquatic bio-diversity, i.e. the plants kingdom in water.

2.1.3 Bibliographic and Experts Data

The bibliographic data includes a collection of bio-diversity related journals and textbook names. Each journal is referred to by its title and date. A journal has a collection of articles which refer to a species or a geographical region. An article has a group of authors.

The experts in the area of bio-diversity, their area of specialization, affiliation and publications to their credit constitute the experts database.

2.2 Features of the Bio-diversity Database

The bio-diversity application requires management of taxonomy data, geographic data, bibliographic data and experts data. A study of the data provided by the *Centre for*

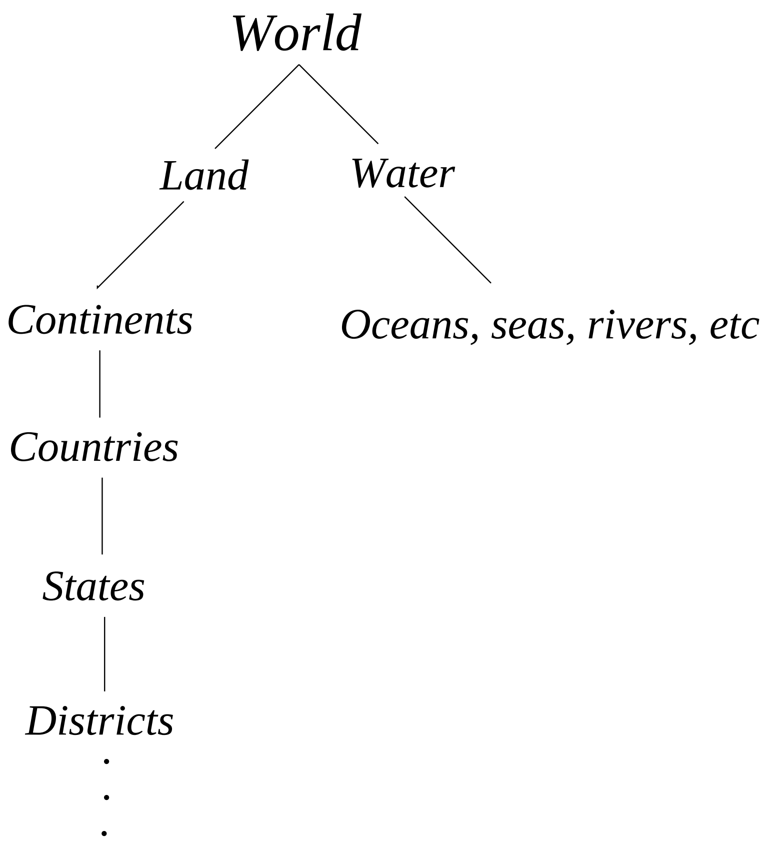

Figure 2.2: The World Hierarchy

Ecological Sciences, *Indian Institute of Science* reveals the following prominent features:

- *Model*

  There is an inherent nesting (aggregation and inheritance) in most of bio-diversity objects. For example, the taxonomy of species results in a four level hierarchy depending on their characteristics. Similarly, the entire world is composed of continents, the continents are composed of several countries and a country is composed of several states. A state is made up of districts, which in turn is composed of cities. Similar hierarchy is also exhibited by the spatial data (polygons, lines, points). The species show extensive sharing of data (characteristics) among them and the characteristics themselves exhibit a hierarchy. For example, a flower is composed of parts such as petals, sepals, stamen, etc,. each of which is a characteristic that could be associated with any species. A species could be found in one or more geographic locations (this is an example of a relationship between the taxonomy data and the geographic data) and a geographic location could be a city, country, state, district, forest or a bio-geographical province.

- *Types of operations*

  Most queries in the bio-diversity database requires picking a single object and navigating through a web of related objects with the path originating from the selected object. For example, displaying the map of a country requires traversing the following objects: the country object, the spatial data (or map) associated with the country (the boundary of the country is usually approximated by a polygon), the points constituting the endpoints of the polygon . Similarly, another useful query on geographical data would be to locate a state which contains a location (specified by its latitude and longitude). This requires searching through the polygons that represent states and applying the *point in polygon* algorithm [Hear90] to each of those polygons. Other spatial queries like identifying regions that satisfy certain conditions would require overlapping two or more polygons [Gunt94]. They might as well involve queries which require a join of points in the database or a rectangle of interest given by two points. [Seq93].

- *Versions*

  Another important requirement of the application is the need for *Versioning* or maintenance of the variations in the state of an object over a certain period. Versioning is required in order to monitor the status of endangered species and to monitor the climatic conditions in geographic locations of interest.

- *Large Volume of Data*

  The amount of bio-diversity data that need to be managed is very large. The taxonomy information includes several thousand species, each having a large number of characteristics. The spatial data which includes the points, polygons and lines that are used to represent the map of a country, also accounts for the large size of the data. Information about the roads, rivers, lakes, oceans, etc add to the size of the database.

- *Infrequent Updates*

  Since the properties of species will not change with time, the bio-diversity database is characterized by infrequent updates. Also the maps of countries, states, districts, cities, forests, etc change infrequently. The design of the system is done taken this point into consideration.

- *Schema Evolution*

  In the bio-diversity application, new object types may evolve with time. For example, new types of characteristics may have to be added to the schema.

2.3 The Queries

In the Section 2.1, we described the types of bio-diversity data and their relationships that need to be maintained in the database. We now focus on the query processing requirements of the bio-diversity database. The following are some of the queries that need to be answered by the system:

- *Select an order, genus, family or species and list some/all of its characters*
- *Identify a species based on one or more of the flower/fruit/seed/leaf characters*
- *Display the map of a geographic location*
- *List all species in a given geographic location*
- *List all species with specific characters in a given geographic location*
- *Find out the experts in a specific field*
- *Display the distribution of a species in a geographic location*

- *Display the distribution of various climatic conditions like temperature, pressure and humidity in a geographic location*
- *List all predators of a given species*
- *Find all the tiger populated areas in the given rectangle, polygon, circle, etc*
- *Display a zoomed region of a particular map*
- *Find all the countries, states through which a particular river flows.*

Summarizing, we can state the requirement of the system as follows :

- The database should incorporate all types of data that are specified and the system should provide facility for data entry.
- The system should be able to efficiently process queries such as the ones listed.

Having identified the goals of the project, We now discuss the steps taken for the design and implementation of the OSHADHI system during its development, that aims to fulfill the specified goals.

Chapter 3

The OSHADHI System

The OSHADHI system was designed for better management of the bio-diversity data and to answer related queries. In this chapter we first discuss the various steps for the development of OSHADHI biodiversity database system and also give a brief overview of OSHADHI-3, the subject of this thesis. Following this we describe the development of OSHADHI-1 and OSHADHI-2. OSHADHI-1 involved identifying the entities in the biodiversity domain, and modeling the domain, designing of OSHADHI-1 architecture, implementing the model, and designing and implementing the graphical user interface [Vid95]. In OSHADHI-2, an SQL like query language *Vaidya* was written. In addition to this some indexing techniques were adopted which retrieved objects efficiently and quickly [Mou97].

We conclude the chapter by describing the features of OSHADHI-3 and some of the design aspects involved. We also discuss the architecture of the system.

3.1 Evolution of OSHADHI

The making of OSHADHI was in three stages as OSHADHI-1, OSHADHI-2, and OSHADHI-3.

- In OSHADHI-1, all the classes required for implementation were identified, aggregations and other associations were designed. An object model was designed and implemented on **Exodus** as backend. The implementation includes *query processing*, which is one of the most important features of a database system.

- In OSHADHI-2, in addition to the above, an SQL like query language was designed and implemented. Some indexing techniques were adopted for quick retrieval of Objects.

- OSHADHI-3, which is built on **SHORE** as backend, highlights the various geographic and spatial queries. The chapters which follow discuss the system architecture followed, the SHORE platform, and the queries which can be answered, in detail.

We will discuss the details of the OSHADHI system in detail in Chapter 3

3.2 Overview of OSHADHI-3

1. **OSHADHI system**

   Before the design of OSHADHI-3, we looked at the previous work i.e. OSHADHI-1 and OSHADHI-2. Following this, the design of OSHADHI-3 was done. It involved the following steps.

   - Identify the classes that would be required for the implementation.

   - Identify indexes and design classes to implement them.

2. **The Platform**

   In order to implement the design, that has been developed, the *system architecture* needed to be designed. System architecture here refers to the overall organization of the database in terms of the back-end *OODBMS*, the *programming language* and the *user interface* through which an user interacts with the database system. This stage involves making the choice of the OODBMS that is most suitable for the application. Various factors need to be considered in making this choice. For example, the query processing requirements impose certain restrictions on the choice of OODBMS. The previous platform **EXODUS** used in OSHADHI-1 and OSHADHI-2 has certain drawbacks and hence a new platform **SHORE** is used in OSHADHI-3. The details will be seen in Chapter 4, where the choice of OODBMS for OSHADHI-3 biodiversity database system is discussed.

3. **Queries Answered**

   The next stage is to implement the various queries which the system should be able to answer. There is a variety of queries that can be answered by the OSHADHI-3 system, which differ from the earlier works like polygons overlapping ie crossing and whether points inside or polygons inside other polygons. Most of these queries are from the benchmark [Seq93]. These queries will be discussed in detail in Chapter 5.

4. **Implementation**

   This is the last stage in the development process when the model is converted to computer programs. This involves implementing the classes and functions associated with these classes, to make query processing simple, using the corresponding data structures of the language. Implementation details are discussed in Chapter 6.

3.3 OSHADHI-1

3.3.1 Object Model of OSHADHI

The object oriented approach to software development presented in [Rumb91] is used in the development of the OSHADHI system. This approach suggests the design of a detailed object model of the system prior to system implementation. The details of the complete object model are given in [Vid95]. The design of OSHADHI-1 system architecture was the next step.

3.3.2 The system design

The following are the various components of the OSHADHI-1 system.

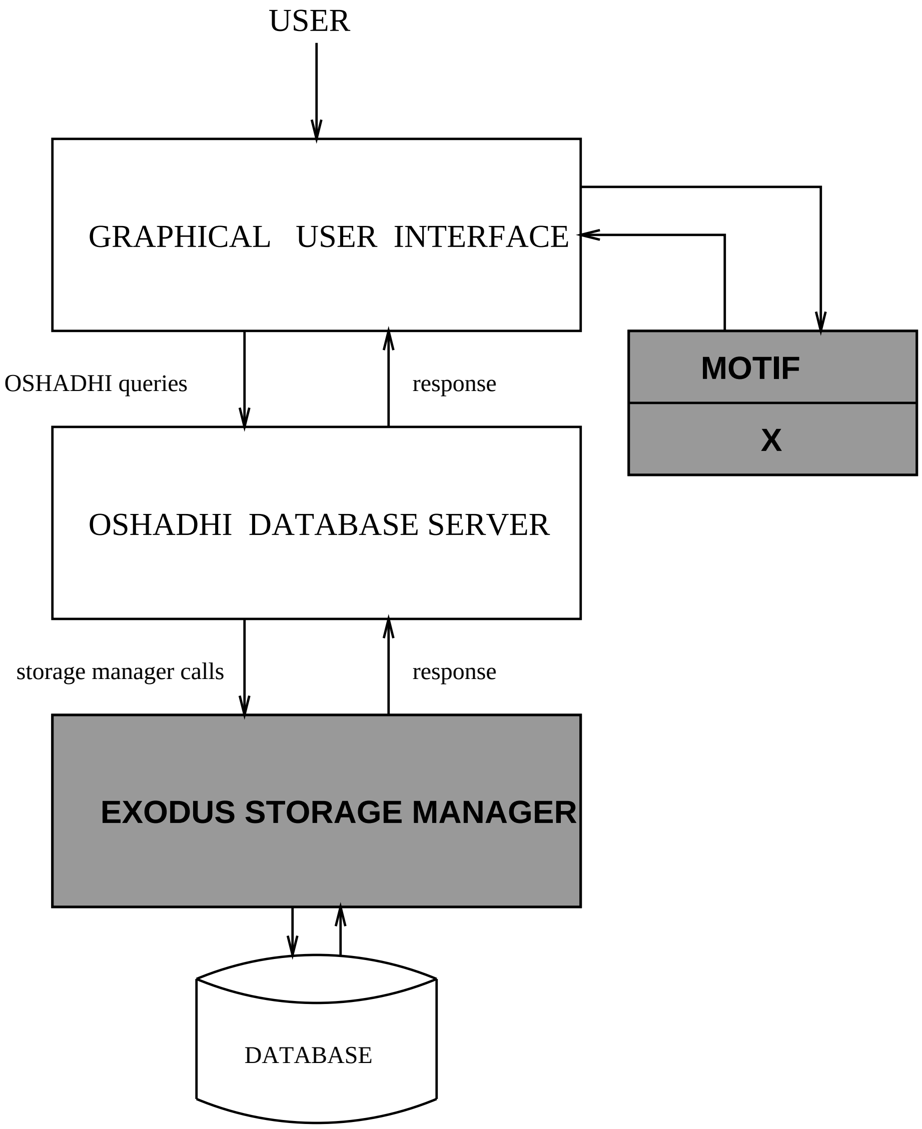

Figure 3.1: System Structure of OSHADHI-1

- **The EXODUS Storage Manager Server**

  The storage manager server of EXODUS [Care86a] forms the lowest layer of OSHADHI. The server receives requests from clients for various operations on objects and collections. The server is responsible for concurrency control, recovery, transactions and storage management.

- **The Database Server**

  The database server acts as a client of the EXODUS storage manager server. The database server written using E programming language. The database server is responsible for query processing and for all the operations on the bio-diversity database. After initialization, the database server waits for a query from its client. The results of the query that are obtained by processing it are sent back to the client.

- **The Interface**

  This is a graphical user interface between the user and the database server. The user interactions with the GUI results in a query which is sent to the database server. The interface waits for the result of the query from the database server and displays it in a suitable form when it is received. The GUI is built using X and Motif [Youn90]. The system architecture of OSHADHI-1 is shown in Figure 3.1.

3.3.3 Implementation

After the completion of the design of bio-diversity object model and the system architecture for OSHADHI-1, the OSHADHI-1 server and GUI were implemented.

3.4 OSHADHI-2

The basic OSHADHI-1 system was updated in the second phase of the project i.e. OSHADHI-2. A query language Vaidya was designed based on the suggestion given by students of *Centre for Ecological Sciences*, *Indian Institute of Science*. The performance of the system is improved by incorporating hcC tree and Nested indexing methods.

The architecture of the OSHADHI-2 is shown in Figure 3.2

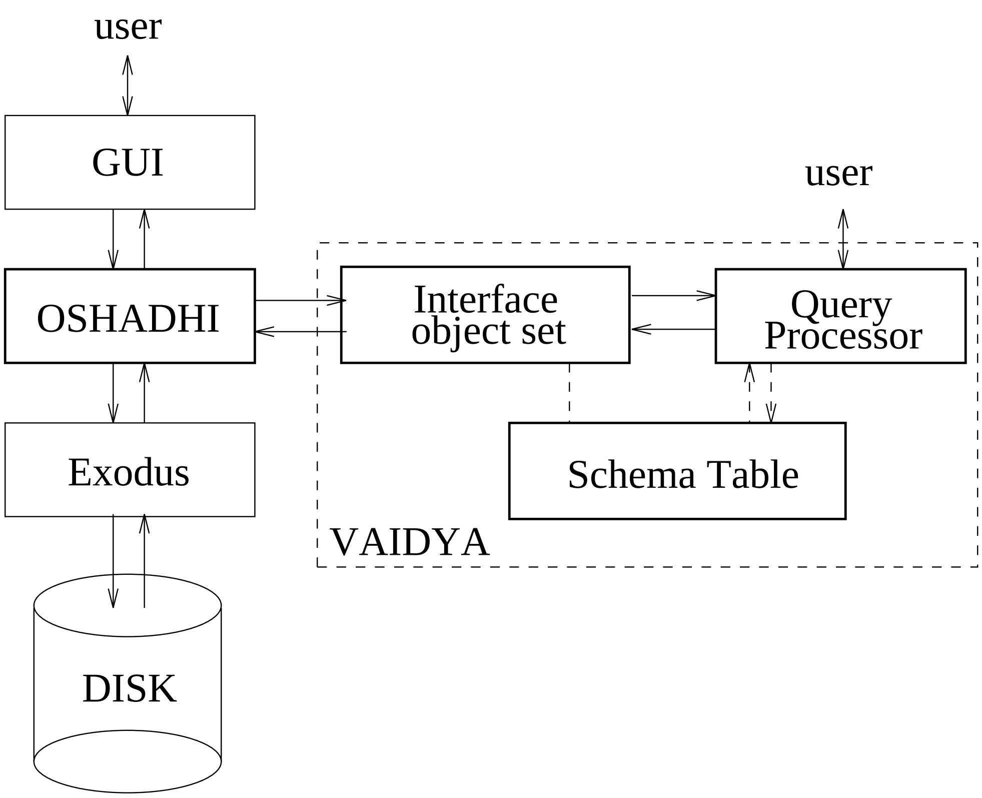

Figure 3.2: The architecture of OSHADHI-2

3.4.1 Vaidya Language

- **SQL Like Features**

  The aim of designing the language Vaidya was to give the user more querying capability yet maintaining the simplicity of the language. The Vaidya language is simple to learn and use. Vaidya provides keywords *Insert*, *Delete*, *Select* and *Relate*, for inserting an object, for deleting an object, for querying and for creating a relationship between two objects, respectively.

- **Member Functions**

  Member function call is an important functionality provided by Vaidya. The user is allowed to call some member functions of the classes in the object model. Some member function calls are *IsPointInside* and *IsCompletelyInside*.

3.4.2 Access Methods

In object oriented database systems, query processing involves traversing complex structures represented by the object oriented query model. For example, object oriented query processing systems need to support queries on objects forming a hierarchy.

- **Simple value based querying**
  - B^+ tree
  - Extendible hashing
- **Aggregation based Access Methods**
  - The Nested Index
  - The Path Index
  - Multi index
  - Multi index with path configuration
- **Inheritance based access methods**
  - The CH tree
  - The H-tree
  - The hcC-tree

3.5 The EXODUS Platform

Both OSHADHI-1 and OSHADHI-2 are built on EXODUS. EXODUS achieves extensibility by providing the basic components of a DBMS. EXODUS is composed of the five major components [Care86a] – The storage manager, the E programming language and its compiler, a library of access methods, a query optimizer generator and Tools for constructing the query language front-ends.

Some of the functionalities in EXODUS are as follows:

- **The Storage Manager**

  It is responsible for creating and operating on storage objects. The storage manager also ensures concurrency and recovery on the storage objects.

- **The E programming language**

  The E programming language [Rich89a, Rich89b] is provided for implementing application specific access methods and operator methods for application specific query language. E extends C++ [Stro91] by adding features like persistence, iterators and certain other database constructs.

- **Facilities**

  - Persistence

    E provides database types called *DB Types*. Objects of dbtypes are similar to their non-db counterparts in C++ except that an object of a non-db type is volatile (whose lifetime is bound by the program run) whereas an object of db-type could be declared **persistent**. A persistent object survives across program runs and crashes.

  - Iterators

    This feature is particularly useful in performing operations on a group of related objects since the iterator function could be written independent of the client loops.

- **Collections**

  E provides a way of grouping related objects into *Collections*. A collection may also be defined to be persistent.

- **Transactions**

  E provides built-in functions to begin, commit and abort transactions.

The features of E discussed above provide a neat interface between the storage manager and the user. These features are, however, inadequate and hence we have chosen a new platform SHORE on which OSHADHI-3 has been built. The drawbacks of EXODUS and the advantages of SHORE over EXODUS have been discussed in the Chapter 4.

3.6 OSHADHI-3

The OSHADHI-3 system follows OSHADHI-1 and OSHADHI-2. While OSHADHI-1 and OSHADHI-2 mostly deal with the Taxonomy queries and a very minimal of Geographic queries, OSHADHI-3 fully concentrates on Geographic and Spatial queries. Moreover, OSHADHI-3 is built on a different platform SHORE about which we will be discussing in detail in Chapter 4. The system architecture for OSHADHI-3 is as shown in Figure 3.3.

The OSHADHI module has been enhanced by adding these set of geographical queries. Most of these queries are discussed in [Seq93] and will be discussed in detail in the Chapter 5.

- **The SHORE Platform**

  It forms the lowest layer of the OSHADHI-3 system. Shore is a collection of cooperating data servers, with each data server containing typed persistent objects. To organize this universe of persistent Shore objects, a Unix-like name space is provided. Requirements are sent to SHORE by the Database and objects are requested.

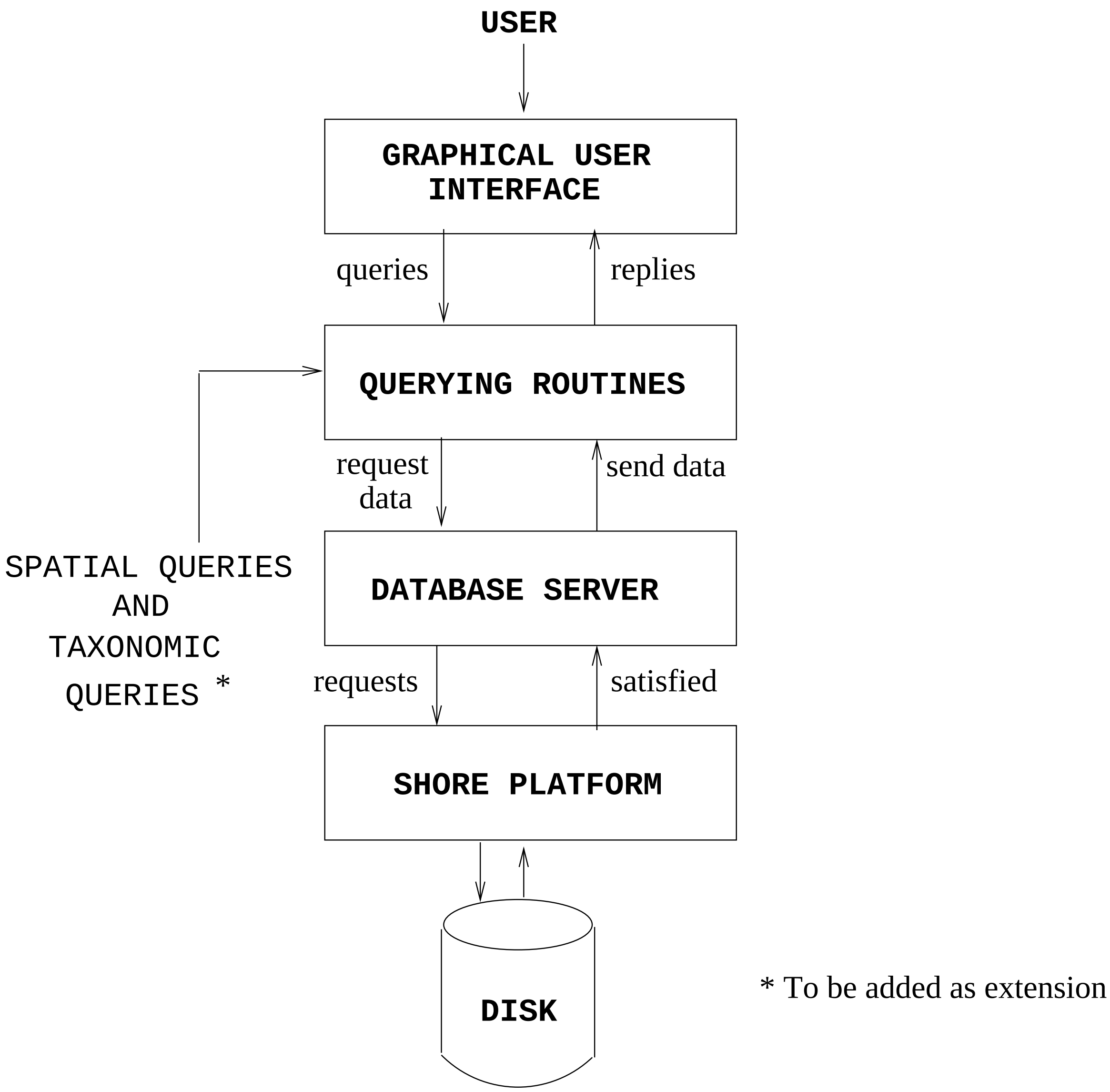

Figure 3.3: The architecture of OSHADHI-3

- **The Database Server**

  The database server sends requests to the SHORE namespace and acts like a client for the SHORE server. It is responsible for the query processing and the other operations on the database. The request is sent to the server, the query is processed and it sends back the results to the user or displays the results on a Canvas.

- **The GUI**

  All the queries in the OSHADHI-3 system are done as form based querying. This is done by a Graphical User Interface in X [XWin98] and Motif [Dan94], [Pau94]. The user sends his queries to the database server by means of certain forms and awaits for the results from the database. The results are either displayed on the drawing area or the Canvas described in Section 6.4.1 or they are displayed on a pop-up dialog.

Chapter 4

SHORE

The Shore (Scalable Heterogeneous Object REpository) is a new persistent object system that represents a merger of object-oriented database (OODB) and file system technologies. In this chapter, we describe features of the SHORE (or Shore) system, which forms the backend database server of the **OSHADHI-3** database system for storing Spatial Objects and parasitics related information.

Most of the material in this Chapter is taken from [Arun96] and from [Over97].

4.1 Introduction

While the past few years have seen significant progress in the OO databases area, most applications have not chosen to leave file systems behind in favour of OODBMSs. Some of the reasons are:

1. Many current OODBMSs are restricted to a single language while large scale applications often require multilingual data access.

2. With most current OODBMSs, application programmers face an either/or decision - either they put their data in the OODBMS, in which case all their file based applications must be rewritten, or they leave their data unchanged in files.

3. Most current OODBMSs have strong client server architectures, and are thus inappropriate for execution in peer-to-peer distributed systems.

4.2 Drawbacks with EXODUS

- EXODUS storage objects are untyped arrays of bytes; correct interpretation of their contents is the responsibility of application programs. No type information is stored about the object. This leads to the following disadvantages:
  - It is too easy to access objects under the wrong type, because of programming or configuration errors such as version mismatch.
  - Restricting type support to the compiler locks users into single-language solutions.
  - Sharing data between applications is difficult.
  - Lack of stored types prevents the DBMS from providing such facilities as support for heterogeneous hardware platforms, data browsers, or garbage collectors.
- A second limitation of the EXODUS storage manager (ESM) is its client-server architecture. EXODUS does not provide server-to-server communication facilities, which is essential for efficient parallelism.
- A third limitation of EXODUS is its lack of support for access control. Furthermore, EXODUS allows client processes to manipulate objects directly in cached copies of database pages, so an errant pointer can destroy not only client data but also metadata.

Finally, while EXODUS objects are similar to Unix files (they are untyped sequences of bytes), the interface for manipulating them is completely different. As a result, existing applications built around Unix files cannot easily use EXODUS.

SHORE tries to retain the good features of the EXODUS Storage Manager (such as transactions, performance, and robustness) while eliminating some of these limitations.

4.3 How SHORE differs from EXODUS

Each object in Shore contains a pointer to a type object that defines its structure and interface. The Shore Data Language provides a single language-neutral notation for describing the types of all persistent data.

Shore's process architecture is different from that of EXODUS in two key ways.

- Shore has a symmetric, peer-to-peer structure. Every participating processor runs a Shore server process regardless whether it has local disks. A client process interacts with Shore by communicating with the local Shore server.
- Shore supports the notion of a "value-added" server. The server code is modularly constructed to make it relatively simple for users to build application-specific servers without facing the "client-level server" problem.

4.4 Basic Shore Concepts

Shore is a collection of cooperating data servers, with each data server containing typed persistent data objects. To organize this, a Unix-like namespace is provided. As in Unix, named objects can be directories, symbolic links or individual typed objects (the counterpart of Unix files). Shore allows each object to be accessed by a globally unique Object Identifier (OID). The OID consists of an 8-byte volume identifier and an 8-byte serial number. The former is designed to be large enough to be globally unique, while the latter is large enough to avoid reuse of values under any operating conditions. Shore also introduces a few new types of object classes, including *types* and *pools*. The type system is language neutral, supporting applications in any programming language for which a language binding exists. For objects whose primary data content is textual or untyped binary data, Unix file system calls are provided to enable legacy applications to access their data content in an untyped manner.

4.5 Shore Object Basics

The Shore object model consists of *objects* and *values*. Every persistent datum is an object and each object has an identity denoted by a unique object identifier or OID and an object is a container for a value. Every object has a type, and all Shore objects are tagged with a reference to a type object that captures this information.

The *core* of an object is described by its type. To support the flexibility of dynamic structures with the efficiency of logically contiguous blocks on secondary storage, Shore allows each object to be extended with a variable-sized *heap*(Figure 4.1). The heap is used by the system to store variable-sized components of its value such as strings, variable arrays and sets.

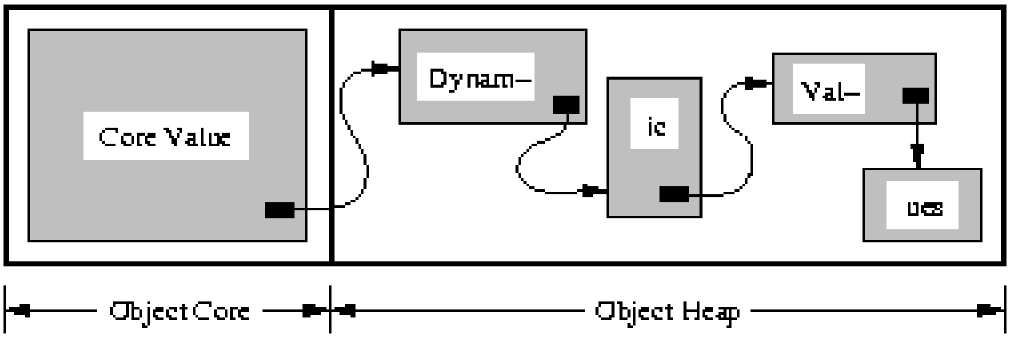

Figure 4.1: Shore Object Structure

4.6 File System Features

From a file system standpoint, Shore provides two major services: First, to support object naming and space management in a world with many persistent objects, Shore provides a flexible object namespace. Second, to enable legacy Unix file based applications to continue to exist while new Shore applications are being developed, mechanisms are provided that permit Shore object data to be accessed via Unix file system calls. However, this method of access does not guarantee the database system features of concurrency control and recovery.

4.6.1 Shore Object namespace

Shore provides a tree-structured, Unix like namespace in which all persistent objects are reachable either directly or indirectly from a distinguished root directory. The namespace extends the set of familiar Unix object types (directory, symbolic links, regular files) with *cross references, pools, modules* and *type objects*. Directories and the objects they contain are called *registered* objects. Each registered object contains a superset of the Unix attributes: ownership, access permissions, and timestamps. Shore also introduces a new kind of registered object called a *pool*. Typically, the contents of one pool form a single database. Members of a pool, called *anonymous objects*, are clustered near one another. Anonymous objects do not have path names, but they can be accessed by OID like any other object. The *registered* property is orthogonal to type: any kind of object can be created either in a pool (as an anonymous object) or in a directory (as a registered object). In a typical SHORE database, the vast majority of objects will be anonymous, with a few registered objects serving as entry points to graphs of anonymous objects. By introducing these object types, Shore gives users a framework in which to register both individual persistent objects as well as the roots of large persistent data structures. Thus, Shore provides a much richer naming environment than the single-level "persistent root" directory found in current OODBMSs.

Shore introduces three more fundamental kinds of objects, *modules*, *type objects* and *cross references*. Modules and type objects are similar to pools and anonymous objects, respectively, but have different deletion semantics to preserve the existence dependency from objects to their types. Cross references are similar to symbolic links in that they provide a way to insert an alias for an object into the directory name space. While a symbolic link contains a path name for a registered object, a cross reference contains the OID of an arbitrary object.

4.6.2 Legacy Unix tool support

While Shore provides a richer environment than traditional file systems, there are many situations where tools designed to be used on files need to be invoked on database objects.

The solution followed in Shore is to provide a special *Unix compatibility* feature. The Unix compatibility feature may be utilized by designating a range of bytes as its *text* field. A compatibility library provides versions of Unix file system calls.

For applications that cannot even be relinked, Shore provides an NFS file server. An entire subtree of the Shore namespace can be "mounted" on an existing Unix file system. When applications attempt to access files in this portion of the name space, the Unix kernel generates NFS protocol requests that are handled by the Shore NFS value added server.

4.7 Object Oriented Database System features

4.7.1 The Shore Type System

The Shore type system is embodied by the Shore Data Language, SDL, the language in which Shore types are defined. SDL is quite similar to the Object Definition Language (ODL) proposal from the ODMG consortium [Cat93], which is descended from OMG's Interface Description Language (IDL), a dialect of the RPC interface language used in OSF's Distributed Computing Environment.

All objects are instances of *interface types*, types constructed with the *interface* type constructor. Interface types can have methods, attributes and relationships. The attributes of an interface type can be one of the primitive types (e.g., integer, character, real) or they can be of constructed types. Shore provides the usual set of type constructors: enumerations, structures, arrays, and references (which are used to define relationships). In addition, Shore provides a variety of *bulk types*, including sets, lists and sequences, that enable a Shore object to contain a collection of references to other objects. Finally, Shore provides the notion of *modules*, to enable related types to be grouped together for name scoping and type management purposes.

4.7.2 Shore Language Bindings

Shore is intended to allow databases built by an application written in one language (e.g., C++) to then be accessed and manipulated by applications written in other object-oriented languages as well (e.g., CLOS). This capability will be important for large scale applications such as VLSI CAD; C++ might be used for efficiency in simulating large chips, while CLOS (or perhaps Smalltalk) might be more convenient for writing the associated design rule checking and user interface code. In Shore, the methods associated with SDL interfaces can therefore be written using any of the languages for which a Shore language binding exists. Currently, only the C++ binding is operational.

4.7.3 Other OODB-Like services

Shore provides support for concurrency control via locking and crash recovery through logging. Shore provides users also with a choice of lower levels of consistency and recovery.

4.8 The Shore Architecture

Shore executes as a group of communicating processes called Shore servers. Shore servers constitute exclusively of *trusted* code, including those parts of the system that are provided as part of the standard Shore release, as well as code for Value Added Servers (VASs) that can be added by sophisticated users to implement specialized facilities (e.g., a query shipping SQL server). Application processes manipulate *objects*, while servers deal primarily with fixed-length *pages* allocated from disk *volumes*, each of which is managed by a single server.

The Shore server plays several roles. First, it is the page-cache manager. Second, the server acts as an agent for local application processes. When an application needs an object, it sends an RPC request to the local server, which fetches the necessary pages and returns the object. Finally, the Shore server is responsible for concurrency control and recovery. A server obtains and caches locks on behalf of its local clients. The owner of each page is responsible for arbitrating lock requests for its objects as well as logging and committing changes to the page.

4.9 Shore Software Components

The main software components of Shore (Figure 4.2) constitute the Shore server and the Language Independent Library. We next discuss these components in greater detail.

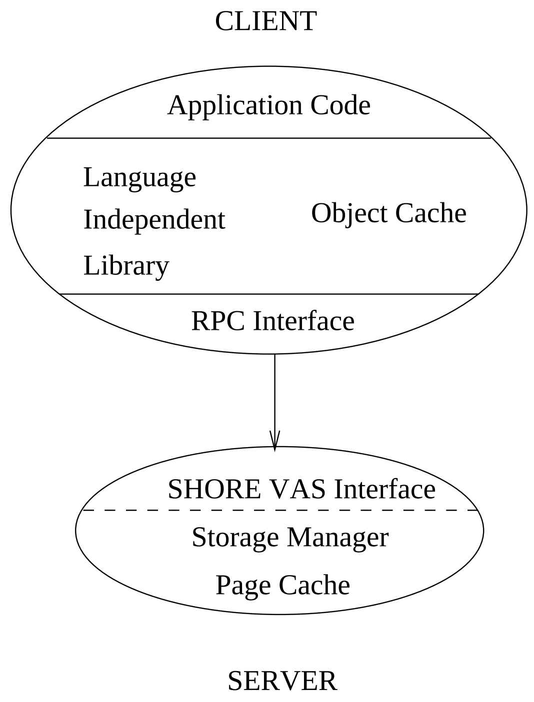

Figure 4.2: Application - Server Interface

4.9.1 The Language Independent Library

The process of converting object references on disk to main memory addresses is called *swizzling*. When an application attempts to dereference an “unswizzled” pointer, the language binding generates a call to the object-cache manager in the *language independent library* (LIL). If the desired object is not present, the LIL sends an RPC request to the local server, which fetches the necessary pages by reading from the local disk.

To reduce paging, the object cache manager locks the cache in memory and uses LRU replacement if it grows too large. All OIDs in the cache are swizzled to point to entries in an *object table.* This level of indirection allows objects to be removed from memory before the transaction commits, without the need to track down and unswizzle all pointers to them. Finally, the LIL is responsible for authenticating the application to the server using the Kerberos authentication system.

4.9.2 The Shore server

The Shore server (Figure 4.3) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

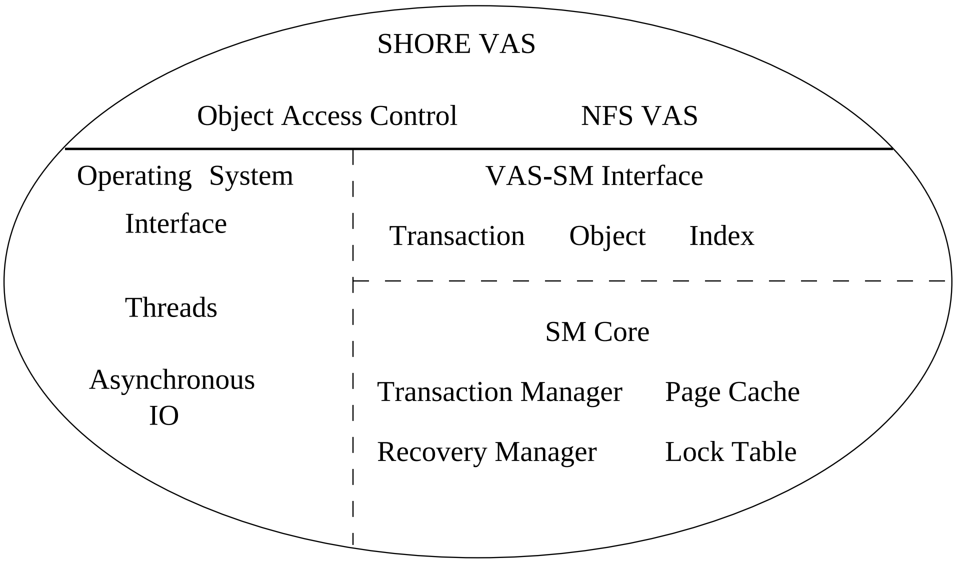

Figure 4.3: System Architecture

The Server Interface is responsible for providing access to Shore objects stored using the SM. It manages the Unix-like namespace. When an application connects to the Shore server, the server associates Unix-like process state with the connection such as user ID and a current directory name. User ID information is checked against registered objects when they are first accessed to protect against unauthorized access. As in Unix, the current directory name information provides a context for converting path names into absolute locations in the name space.

The Shore server code is modularly constructed so that users can build application-specific servers, thus supporting the notion of "value-added" servers (VAS). The Server Interface is an example of one such VAS. Another VAS is the NFS file server described earlier. Each VAS provides an alternative interface to the storage manager. They all interact with the storage manager through a common interface that is similar to the RPC interface between applications and the server. It is thus possible to write a new VAS as a client process and then migrate it into the server for added efficiency. Below the server interface lies the Storage Manager (SM). The SM can be viewed as having three sub-layers. The highest is the VAS-SM interface, which consists primarily of functions to control transactions and to access objects and indexes. The middle level comprises the core of the SM. It implements records, indexes, transactions, concurrency control and recovery. At the lowest level are extensions for distributed server capabilities. In addition to these layers, the SM contains an operating system interface that packages together multithreading, asynchronous I/O and inter-process communication.

4.10 Conclusion

Shore is an integration of file system and OODB concepts and services. From the file system world, Shore draws object naming services and an object access mechanism for use by legacy Unix file-based tools. From the OODB world, Shore draws data modeling features and support for associative access and performance acceleration features.

Chapter 5

Querying

The geographical data consists of information about bio-geographic provinces, biomes, administrative organizations, forests, botanical gardens and herbariums. Each of the administrative organizations, forests, etc., has a map associated with it. The maps are stored as polygons which approximate the region. Processing of queries on these maps requires mathematical computations. These computations are done in member functions of the *Polygon* class. Query Processing involves calling these member functions. All the queries are called by the user using the Graphical User Interface. The user gives his requests through a form which he is supposed to fill in according to his need. The querying routines read the requests from the user and opens a SHORE *PoolScan*. This PoolScan traverses through the entire *Pool* of graphs to find if the graph mentioned in the query is present. If the graph is found in the pool, the querying routines perform the necessary action. The routines check whether the request can be satisfied by calling appropriate functions from the *Polygon* class. If the request is satisfied, the result is returned to the user, on either a message Dialog or on a Pop-up Selection Box. Some of the results are also displayed on the Canvas defined in Section 6.4.1.

In this chapter we discuss all the queries that can be answered by the system and how these queries will be useful for the bio-diversity database or in any other generic database which involves geographic data. All the queries discussed here are according to *The Sequoia 2000 benchmark* [Seq93], which have been implemented in OSHADHI-3.

OSHADHI-1 and OSHADHI-2 did not take this into benchmark into consideration.

In addition to the queries discussed in the benchmark, we provide a general query which can handle all the benchmark queries and more. This is discussed in Section 5.5.

5.1 Spatial Queries

The typical queries over the geo-spatial dataset are those that select maps that are either contained by a polygon, or which intersect some other polygon. This querying requires clipping the polygons, finding the overlapping regions or finding whether all the points of the desired polygon are inside the given polygon. Some of these queries also involve displaying the results on the Canvas.

Let us discuss all these queries in detail and see how they can be used by the biodiversity database.

5.1.1 Load

The first query that any Database must perform is load the database. In our case the geographic data required for the maps must be filled in the database. The data that must be entered is *vectored* data. Vector data here refers to the longitude and latitude form of data for the maps. The user can either directly give the filename of the file which contains the specified data on command line or select the file on executing the application. The application checks for the format of the file, only if it matches the file is loaded.

The format of the file is as shown in the table 5.1.1 below:

Continent	Country	State	Name	Polygon	Type	Open
			Asia	110 5 ...	continent	
			Africa	-40 -80 ...	continent	
0			India	110 5	country	
1			Zimbabve	-30 -60	country	
0	0		Karnataka	110 5	state	
0	0		Ganga	110 5	river	o
0	0	0	Bangalore	120 10	city	

The column *Open* here refers to whether the map is open or closed. If the graph is open, example rivers, roads, lakes, the open field is filled with 'o' or else it is left blank. The meaning of other fields is conveyed by the names of the columns. This query loads the database with the geographic data. The database was loaded with the data from a data file for some of the maps. The picture of the globe is shown in Figure 5.1 . Now all kinds of operations can be performed on this database.

5.1.2 Inside Queries

This set of these queries answer if a Point or Polygon, is inside another polygon.

IsPointInside

query: Is point.x, point.y inside polygon *name*.

This query finds whether the point given by longitude and latitude, is inside the given polygon.

To answer this query, we need an algorithm which finds whether a point is inside a polygon [Hear90]. We have implemented an algorithm which can efficiently do this and we discuss about it in the Chapter 6.

An example of the query is shown in Figure 5.2.

IsCompletelyInside

query: Is polygon name 'x' inside polygon name 'y'.

This query finds if the polygon 'x' is completely inside polygon 'y'.

5.1.3 OverlappingPolygon

This set of queries find maps which overlap the given polygon. There are different ways of providing the given polygon. Also, for convenience of asking the query, we provide some specific queries as well. For such queries, the Polygon, with which all the overlapping maps are to be found can be a square, a rectangle, circle or a river. These issues are discussed below.

IntersectRectangle

query: Find if a particular polygon intersects a specific bounding rectangle.

The given polygon here is the bounding rectangle. We want to find whether the boundary we have given crosses this rectangle or not.

This query finds if the map intersects the rectangle. The rectangle is specified as the diagonal end points. The result of the query is shown in Figure 5.3.

Point Join

Many Biodiversity queries require the ability to correlate (or join) multiple kinds of data. Example: Find polygons that intersect a square of interest. The square is defined to have a centre and the width of the square is provided. The centre point is the part of the database.

Thereby the input to the query is: the centre of the square and the distance of the side from the centre. The query constructs the square and finds the maps overlapping this square. Finding the overlapping maps with a square by providing such data is called as the Point Join in the Sequoia 2000 benchmark [Seq93].

query: Find all the polygons which overlap the square, given by the centre and the width.

This query first constructs the square with the centre and the width given and returns all the maps which overlap this square. Example of the query is shown in Figure 5.4.

PolylineCrosses

This query will find all the *open* polygons which cross a polygon.

query: Show all the rivers that cross polygon *name*.

This query finds all the rivers which cross the polygon and displays the results on the Canvas, highlighting the points of intersection.

The result is shown in Figure 5.5.

CrossingCircle

query: Find all the polygons which overlap the circle given by the radius and the centre of the circle.

This query returns the names of all the maps that overlap the given circle.

The example of the query is shown in Figure 5.6.

IntersectPolygon

query: Find all the polygons which intersect the polygon given by its end points.

This query returns all the polygons which overlap the given polygon. The query processing routine, finds out whether any line of the polygon crosses any line of the given polygon. If any of the lines cross it, implies that the polygon overlaps the given polygon. The names of the overlapping maps are provided to the user on a Selection Dialog.

The result of the query is shown in Figure 5.7.

PolygonJoin

Apart from Point Join we also require Polygon Join. This is to find all the maps which overlap with a polygon which is a part of the database. This query is described as Polygon Join in the Sequoia 2000 benchmark [Seq93].

We facilitate the user to draw such polygons by providing him with a form as shown in Figure 5.8. The user has drawn a polygon as shown in Figure 5.9. This polygon must be stored in the database and further querying can be done on it.

query: Find all the polygons which cross the polygon.

This query checks if the given polygon exists in the database. If so it returns all the polygons which intersect the polygon and displays them on the Canvas.

Refer Figure 5.10. to check the results of the query.

Inside Query using the Polygon

In addition to the Polygon Join query discussed above, the user can also find all the maps that are inside the polygon.

query: Find all maps which are inside the polygon.

Result is shown in Figure 5.11.

A conjunction of the above two queries can be made.

query: Find all maps which are inside and which cross the polygon.

Result is shown in Figure 5.12.

5.2 GIS Queries

These queries include *The Geographic Information System* queries like zoom, etc. These queries can also be shortcuts to some queries like

- Find if a point is inside another polygon. Just click on the Canvas. It will show you the map which contains the point.

- Find all the rivers of a country. For this all that has to be done is display the country on the Canvas and just use the *Rivers* option. Getting the country of interest on the Canvas is a trivial task. It can be accomplished by just clicking on the Canvas.

- Sometimes the user will want to go back to the previous map and then see a different child of the map. This is possible by using the *Parent* option. It takes the user to **Asia** if he were looking at **India**.

query **Zoom**: The zoom function zooms whatever is on the screen. The zoom query asks the user the diagonal end-points of the rectangle within which the region of the map to be zoomed lies. These end-points are set as the end-points of the Canvas.

The click action is an application of Polygon contains point. The Zoom option is an additional facility which allows the user to set the end points as he desire.

5.3 Queries using indexes

A region index has been provided in the OSHADHI-3 system. This index helps in quick retrieval of maps that are within the given range. The region quad tree index has been built for the spatial queries and will be further discussed in detail in the Chapter 6.

Some of the queries which can use this index are:

- IntersectRectangle
- Overlapping Polygon.
- PointJoin.
- PolygonJoin.

Also the insert and delete routines can use this index to find the neighbours while updating them.

5.4 Drawing

The system also allows the user to draw polygons. Example is shown in Figures 5.8 and 5.9. Although these maps are stored as temporary maps, they are quite useful from the users point of view. He can now draw the polygon he wishes to find out which maps are inside or crosses it. The maps that the user makes this way are also stored in the Database informing it that they are temporary polygons. This way the user can store his query and execute it at a later time. If he wishes later he can delete these maps.

5.5 General Query

In addition to the above queries, we provide the user with a general query interface. The user can use the query for any conjunction and disjunction operation.

For example, typical queries could be:

- Select all the maps from map Asia. Results are shown in Figure 5.13.
- Select all the maps from map Asia **and** crosses river Ganga. Refer Figure 5.14.
- Select all the rivers. Figure 5.15 shows the results.
- Select all the rivers which flow in India. Figure 5.16 shows the results.
- Select all the roads which *crosses* any river and display their point of crossing or display all the 'open' maps. Results are shown in Figure 5.17.
- Select all the maps which are completely *inside* the polygon (as was constructed for Polygon Join query in subsection 5.1.3). Results shown in 5.11
- Select all countries which are in Asia **and** *crosses* river Ganga or all the rivers or all the countries in Europe. Figure 5.18 shows the results.

In this way any combination of conjunctions and disjunctions in the query can be provided to the system. This query, thus takes care of any boolean expression that a user might want to perform.

5.6 Uses of the Queries

Having seen the various queries that can be answered by the system, let us see how these queries can be useful in the biodiversity database and also in any other database involving geo-spatial data.

5.6.1 Biodiversity

Here are some of the queries related to Biodiversity which can be easily answered using the set of the queries described above.

- Find all the tiger populated areas in India. This query can be easily answered using the PolygonJoin query and setting an extra condition on the species of the region as "tiger".
- Find the vegetation along the banks of river *Ganga*.
- Find all the river that cross Karnataka.
- List all the countries which are in Asia and where river *Ganga* flows.
- Find all the states which cover the Malabar coast.
- My region of interest is drawn on the Canvas, Show me all the maps which cross this region.

There are many such queries which can be answered by the system. The list can go on and on. These are some of the basic queries where the queries implemented in OSHADHI-3, discussed in Sections 5.1, 5.2, 5.3, 5.4, 5.5 can be used.

5.6.2 Other Databases

There are various Databases which require the use of geo-spatial data in their databases. All databases which involve geography related queries can use the above designed queries.

For example:

- The Public Works department might want to find out all the bridges in the country or a state or a city. All that is required is to find all the roads which cross all the rivers.
- The Meteorology department might want to find all the cities which are in Karnataka and through which river Kaveri flows.

There is an end-less list of all such requirements from various database systems.

Our system is capable of answering most of the spatial queries. All, these queries need to be integrated with the existing OSHADHI model to form a full fledged biodiversity system.

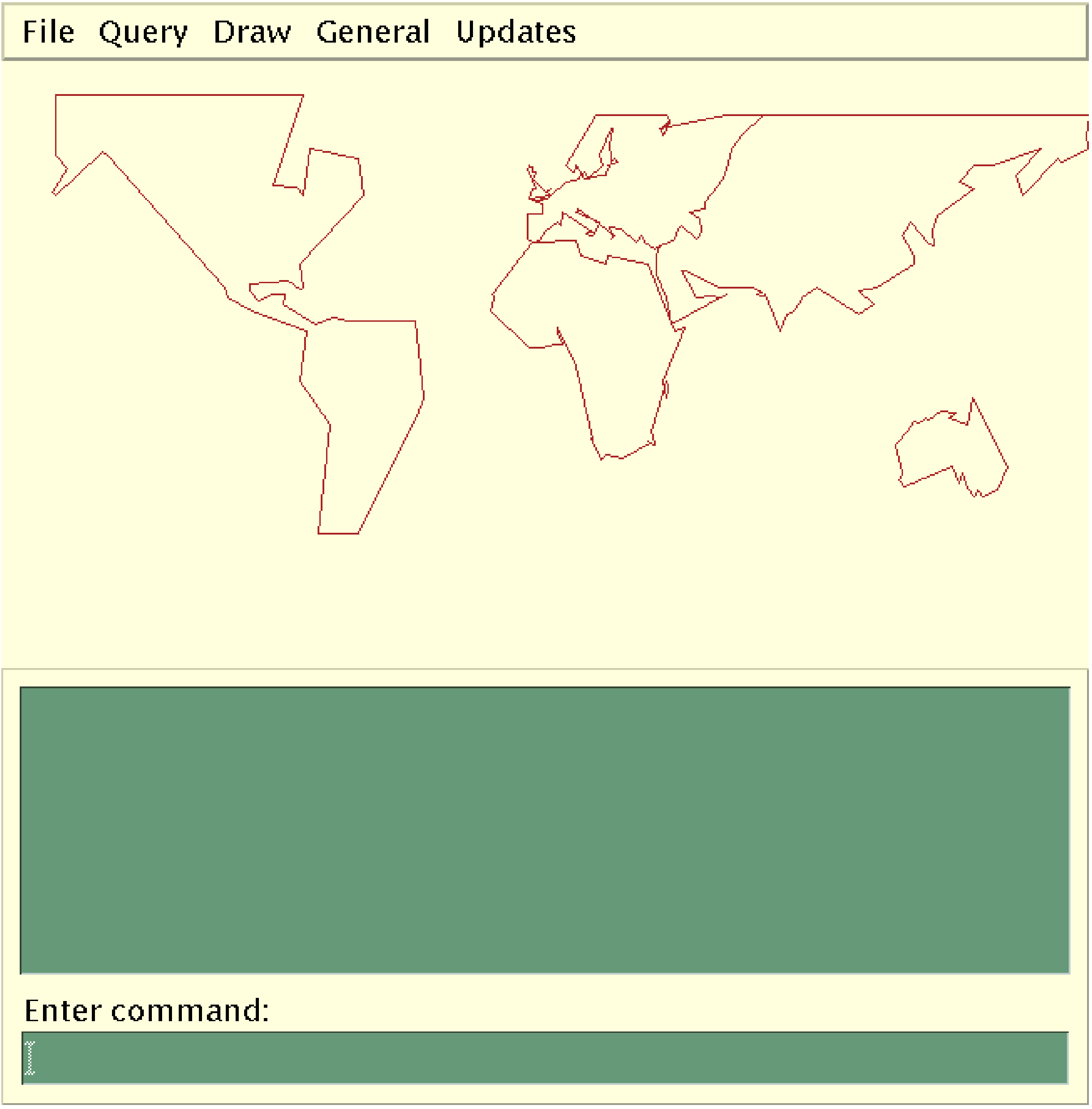

Figure 5.1: Globe

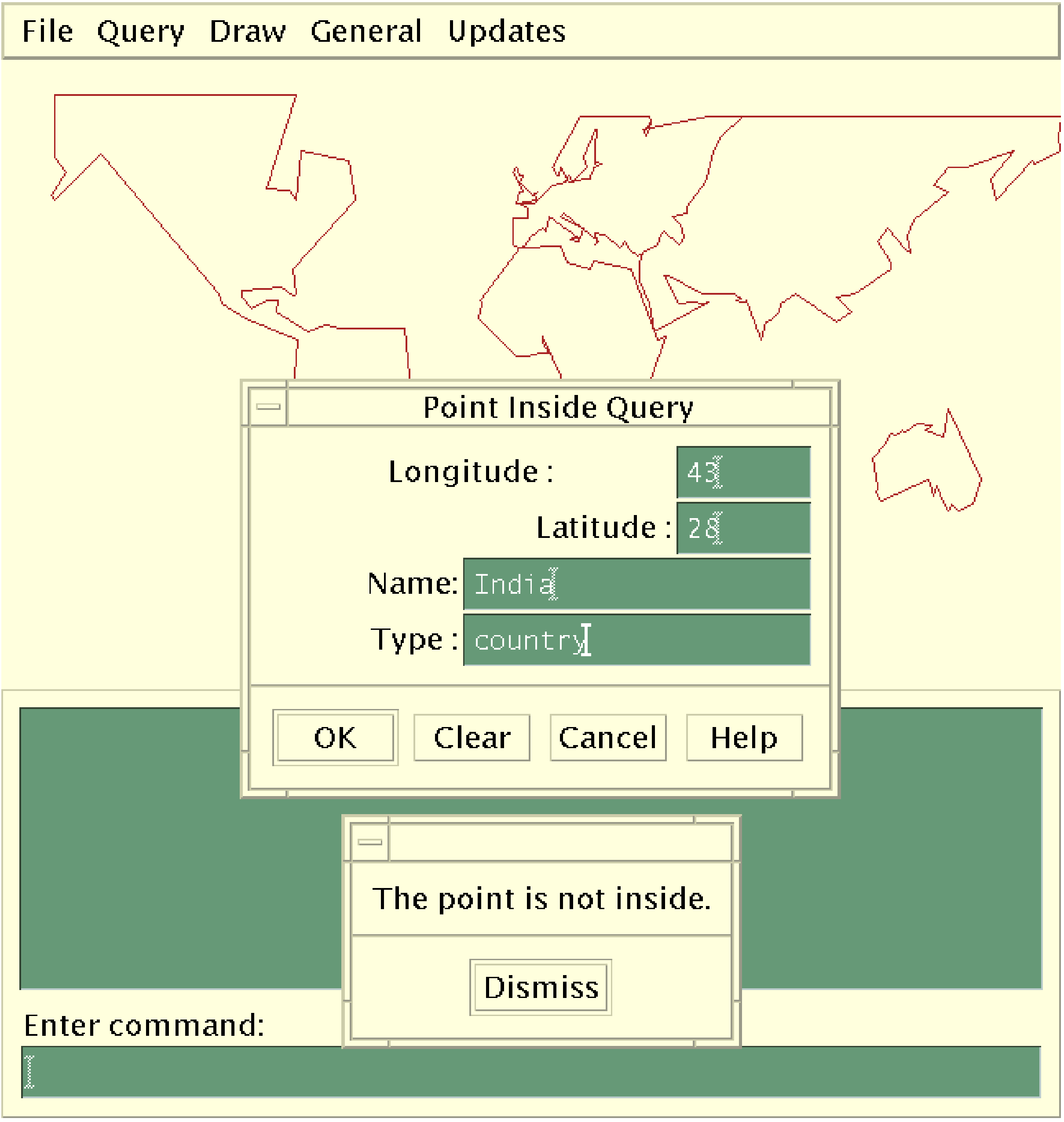

Figure 5.2: IsPointInside
Query: Find if Point 43, 28 is inside country "India".

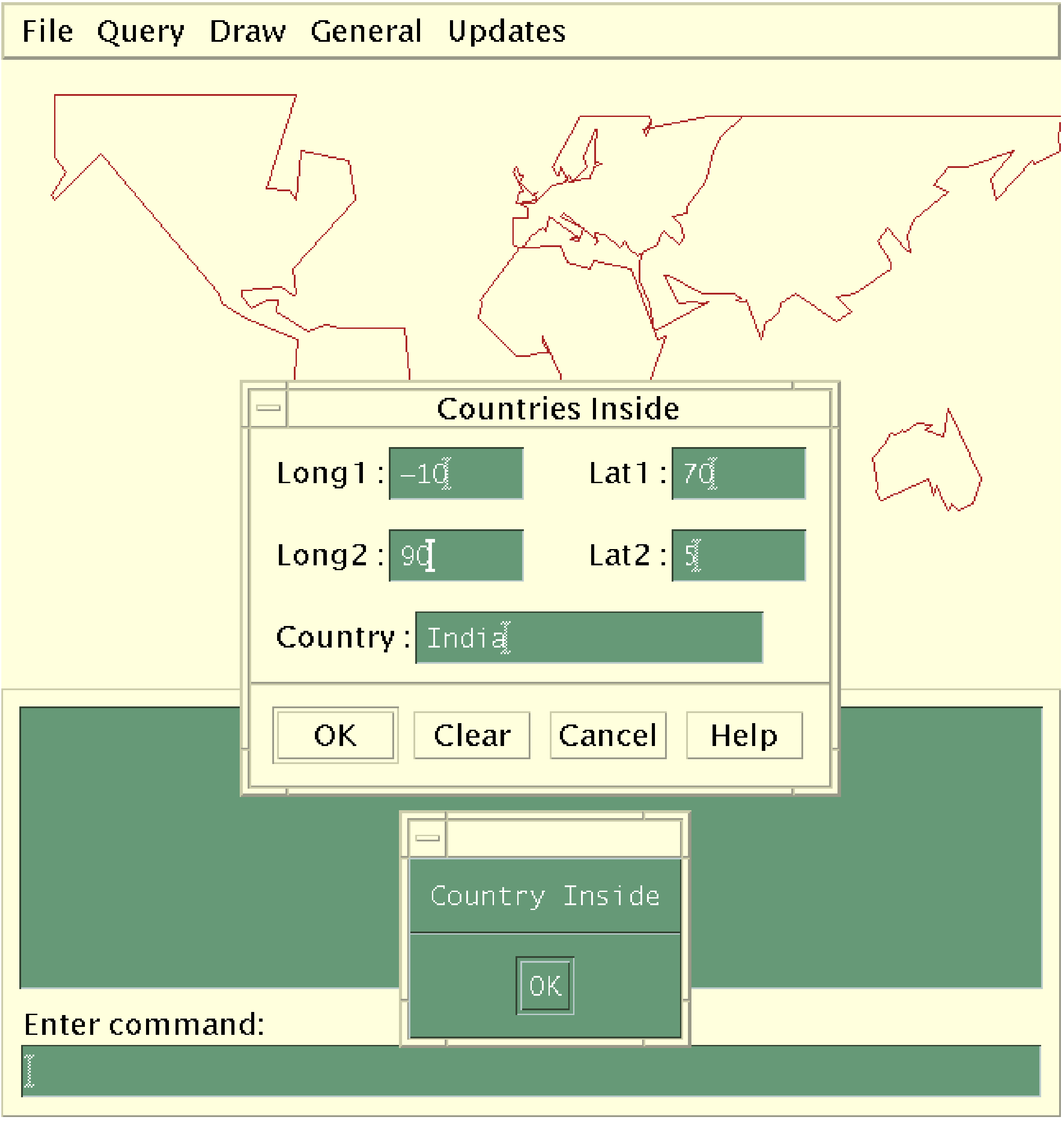

Figure 5.3: IsCountryInsideRectangle
Query: Find if "India" is inside the *bounding rectangle*
given by the diagonal end-points -10, 70 and 90, 3.

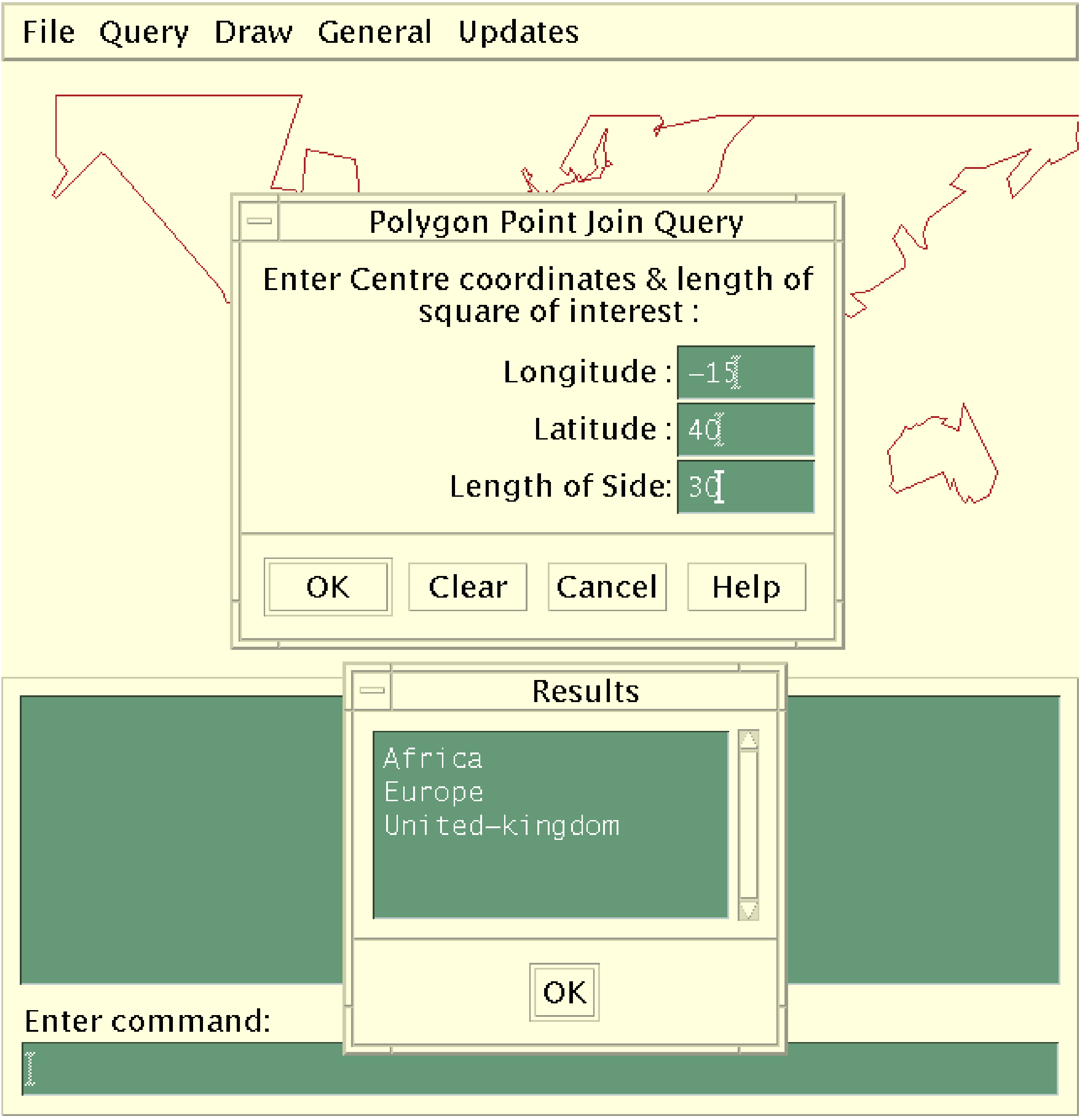

Figure 5.4: Polygon Point Join Query
Query: Find all maps which overlap the square given by the centre co-ordinates -15, 40 and the width 30.

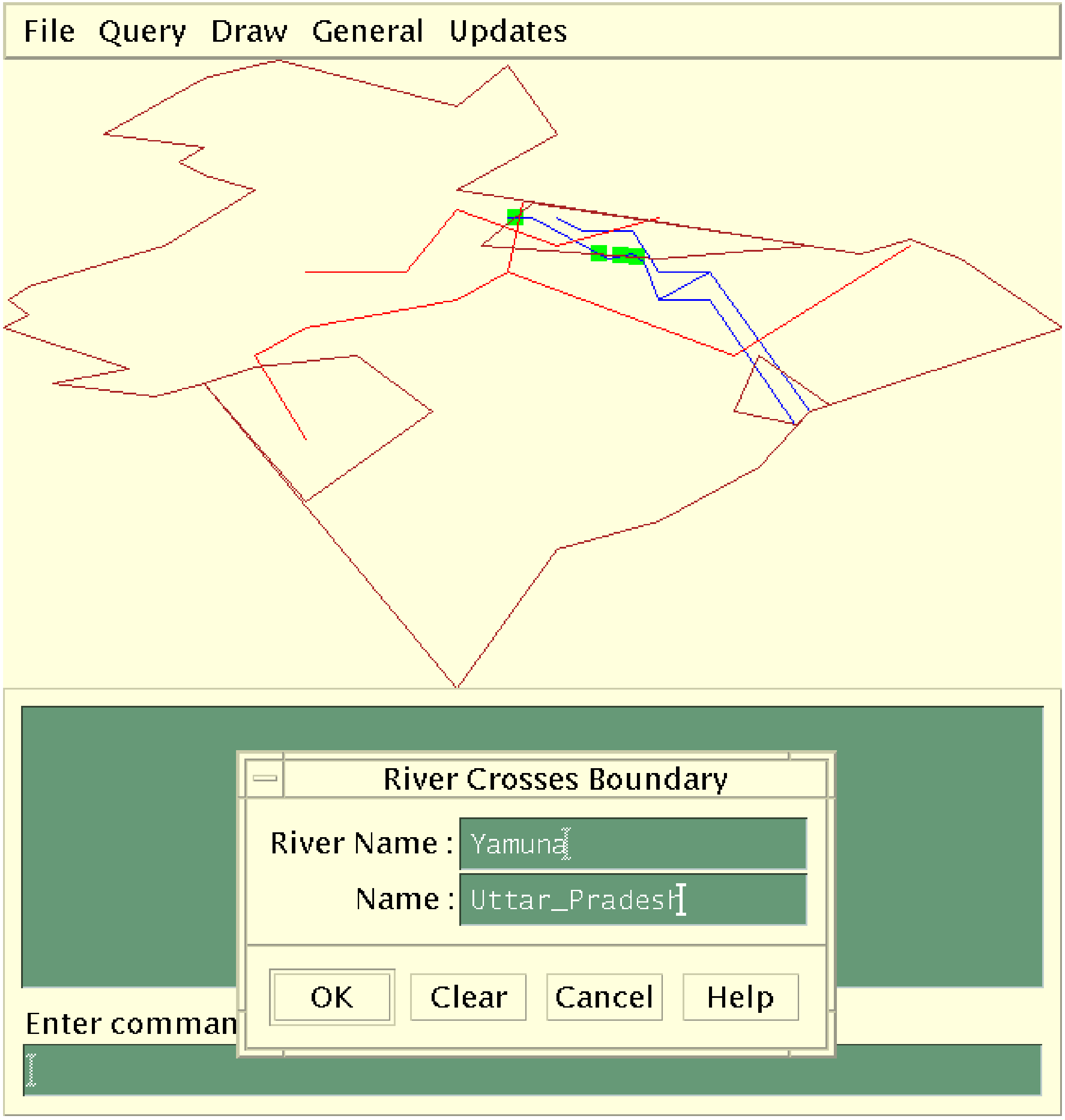

Figure 5.5: River Crosses Boundary
Query: Find if river "Yamuna" crosses "Uttar_Pradesh" and
if it does show the points of intersection.

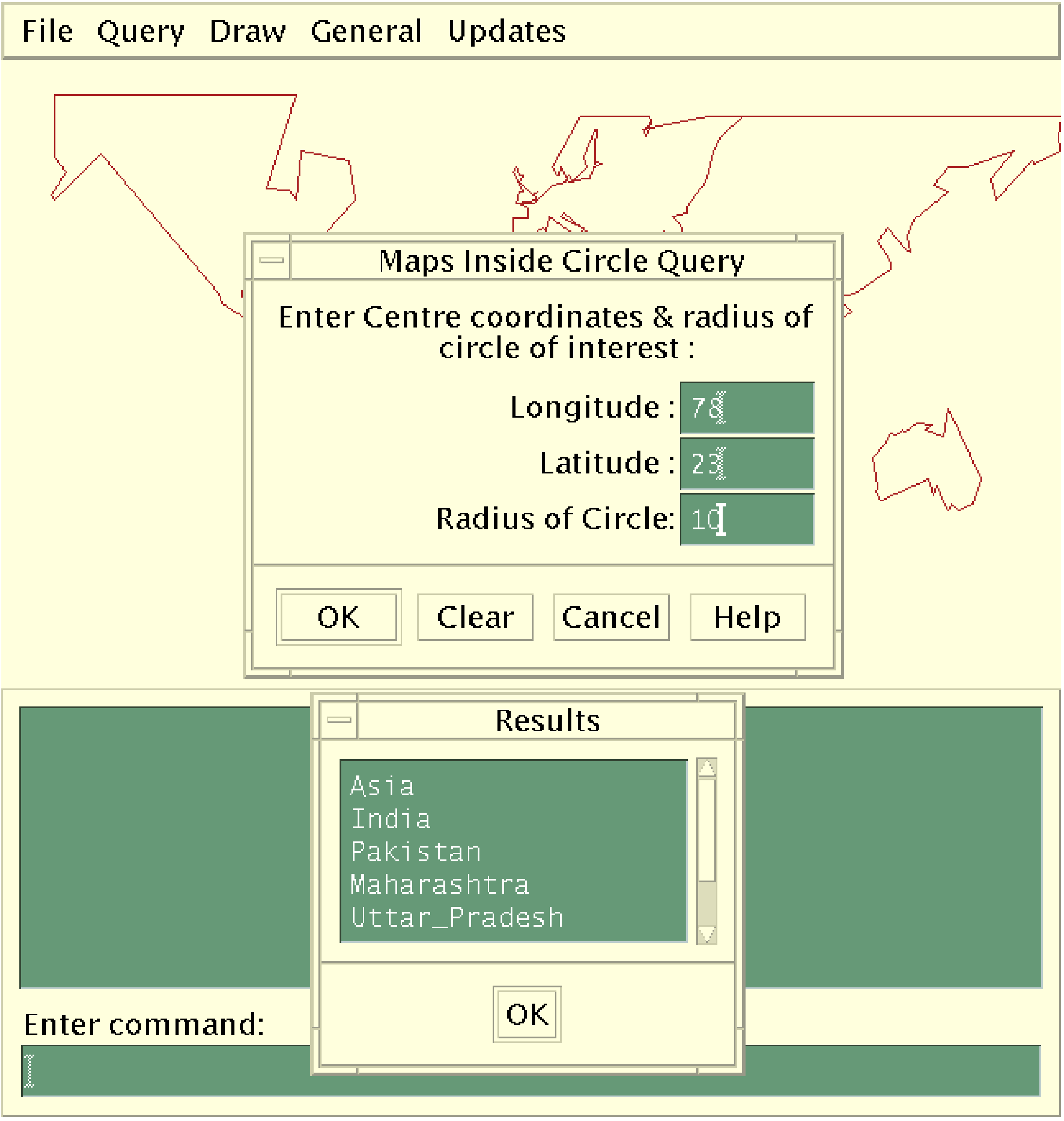

Figure 5.6: Maps Overlapping Circle
Query: Find all maps which overlap the circle given by the centre co-ordinates 78, 23 and the radius 10.

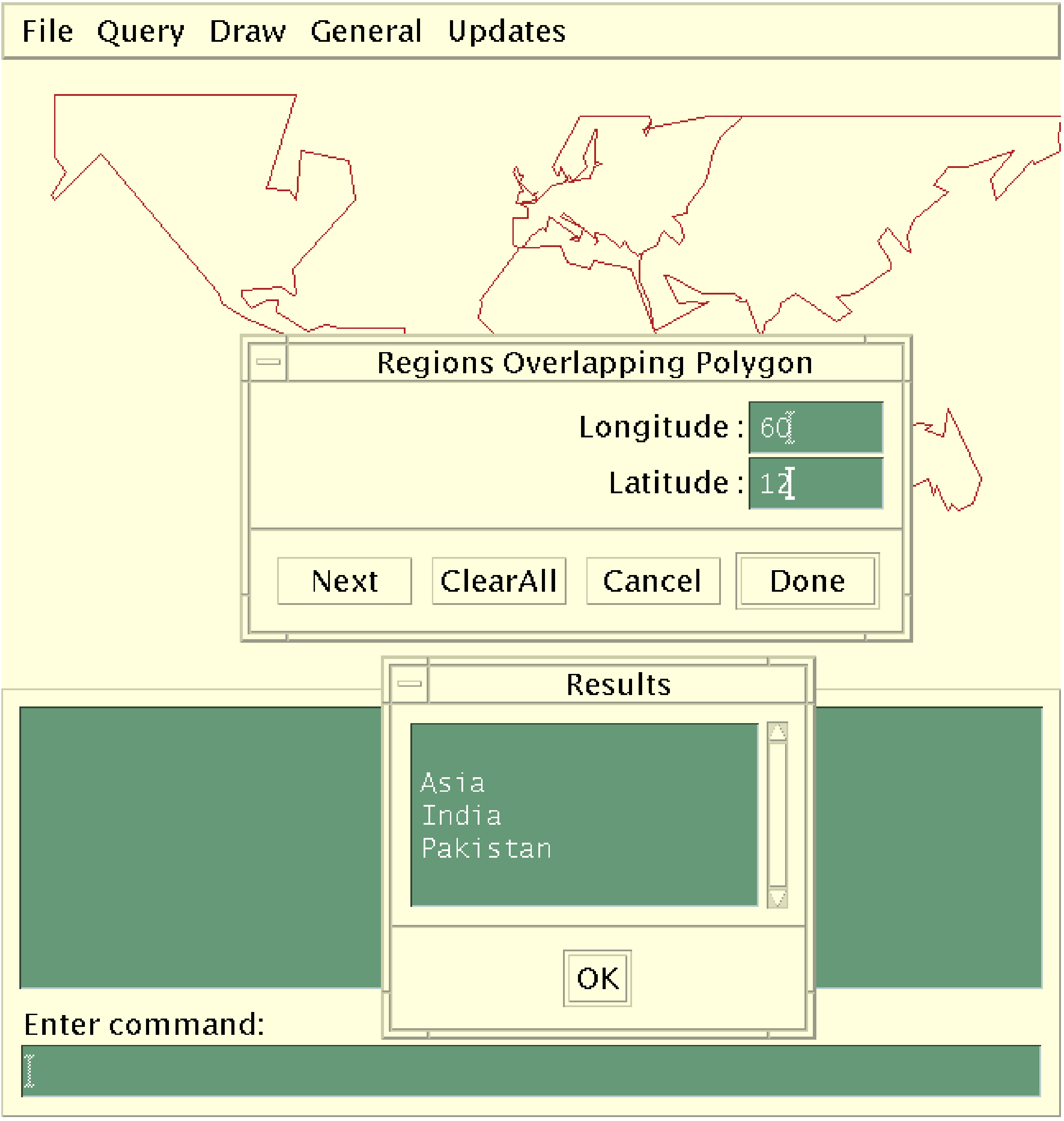

Figure 5.7: Regions Overlapping Polygon
Query: Find all maps which overlap the polygon constructed by the given points

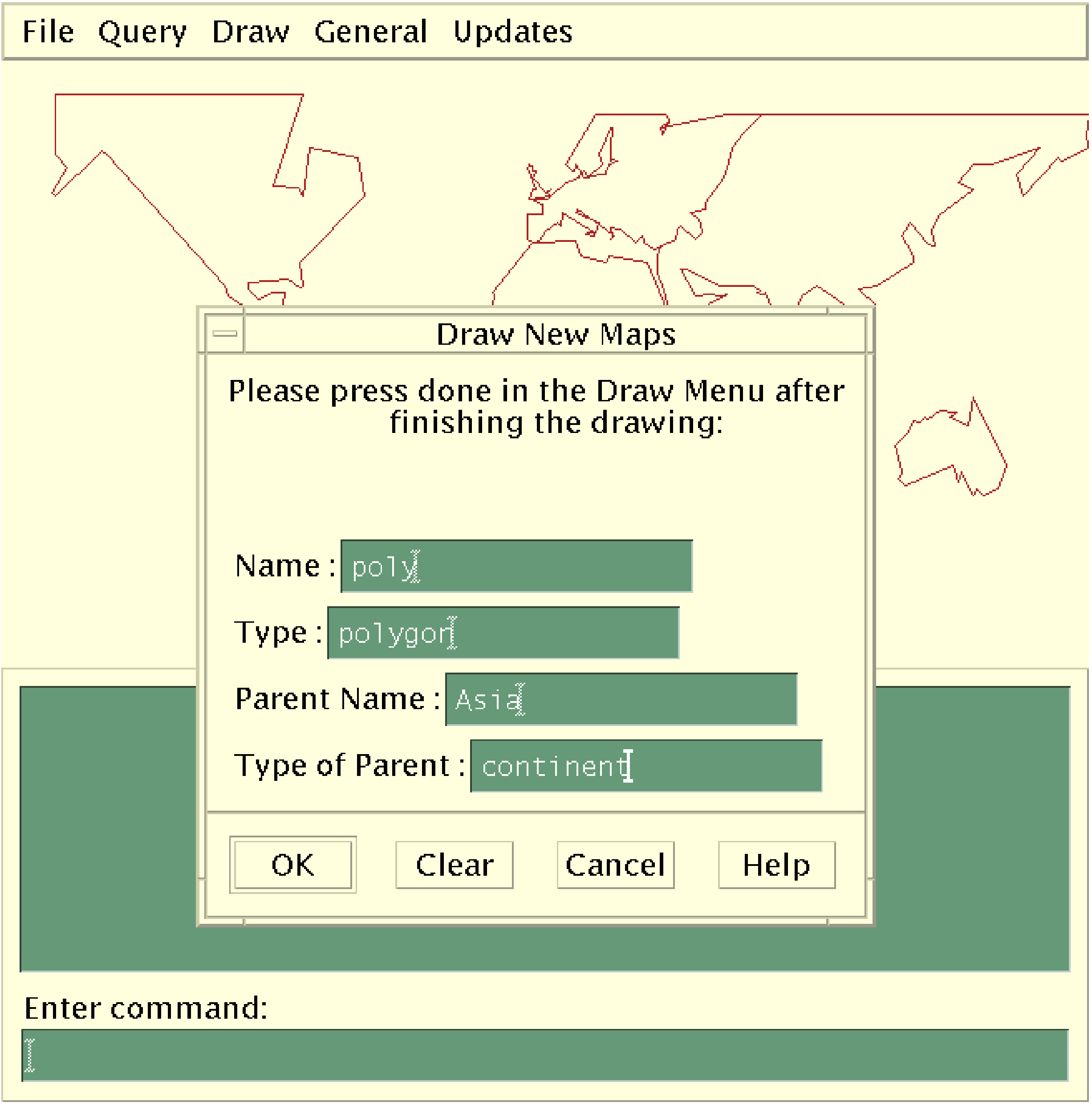

Figure 5.8: Draw New Map
Query: Give the specifications for drawing the new polygon

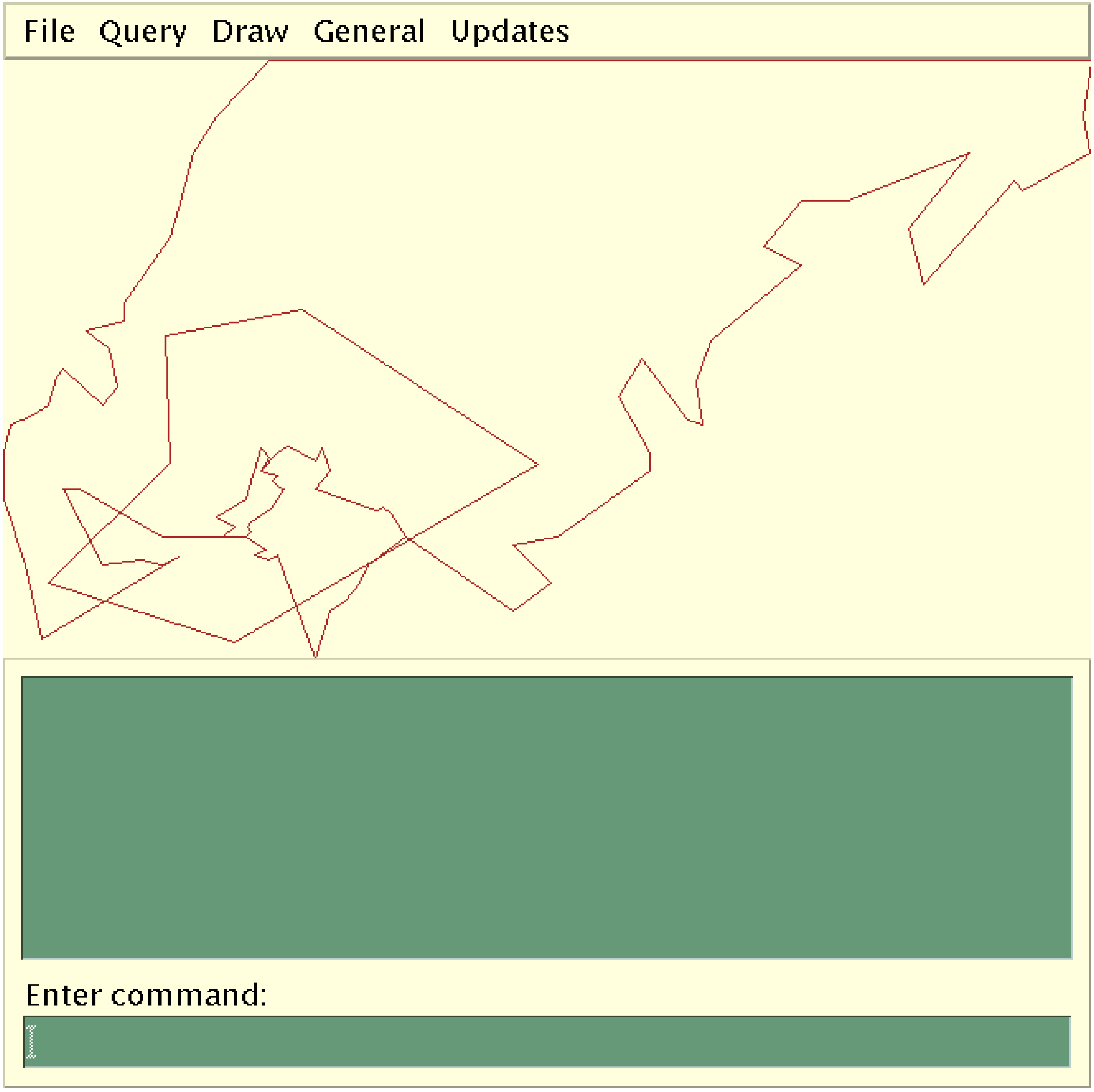

Figure 5.9: Start the Drawing and say *done* when finished
Query: Draw the polygon with the given specifications
Save it in the database.

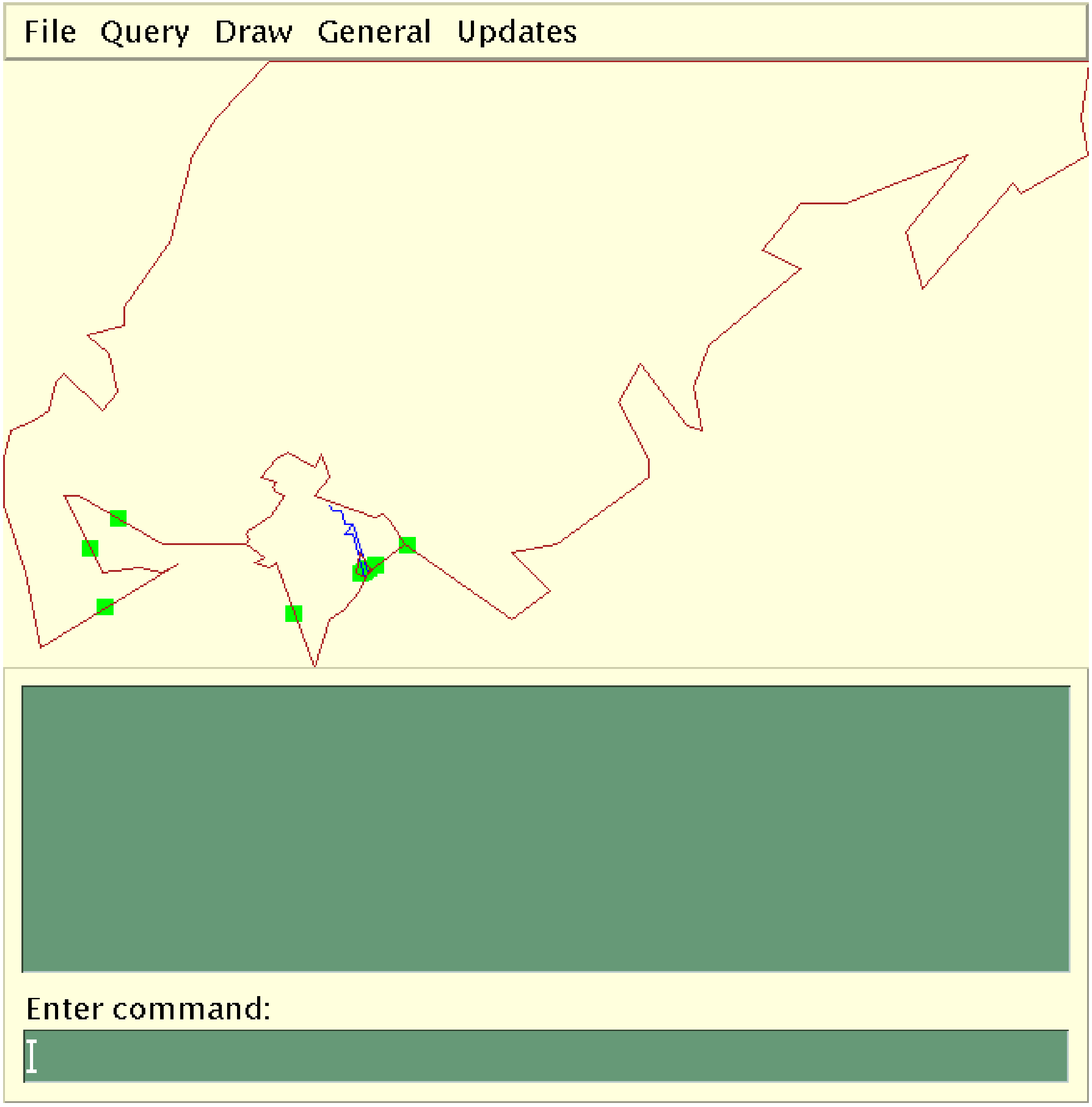

Figure 5.10: Maps crossing polygon
Query: Display all the points where this polygon crosses

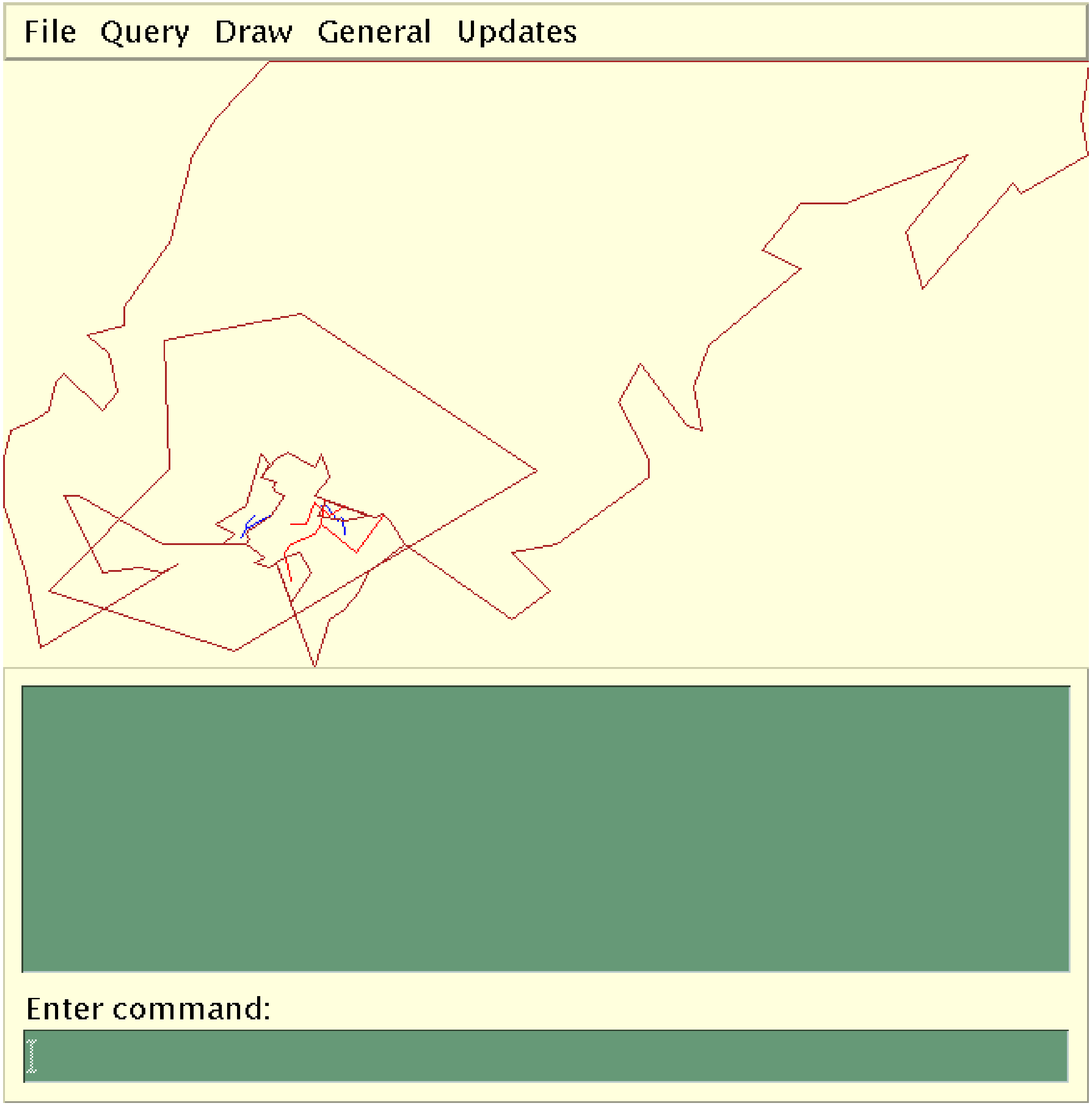

Figure 5.11: Maps inside polygon
Query: Display all the maps which are inside this polygon

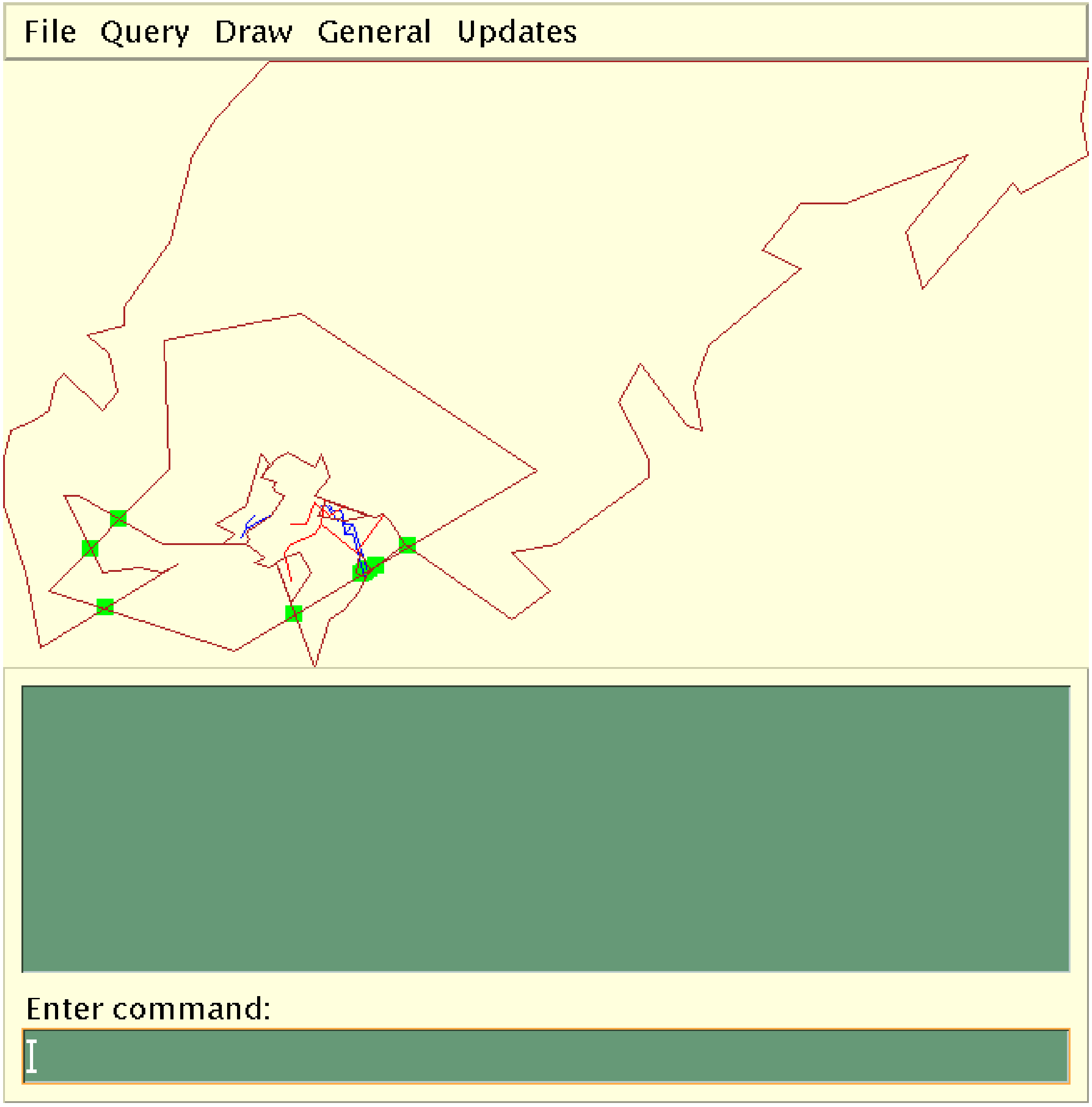

Figure 5.12: Maps inside or crosses polygon
Query: Display all the points where the polygon crosses or all the maps which are inside this polygon

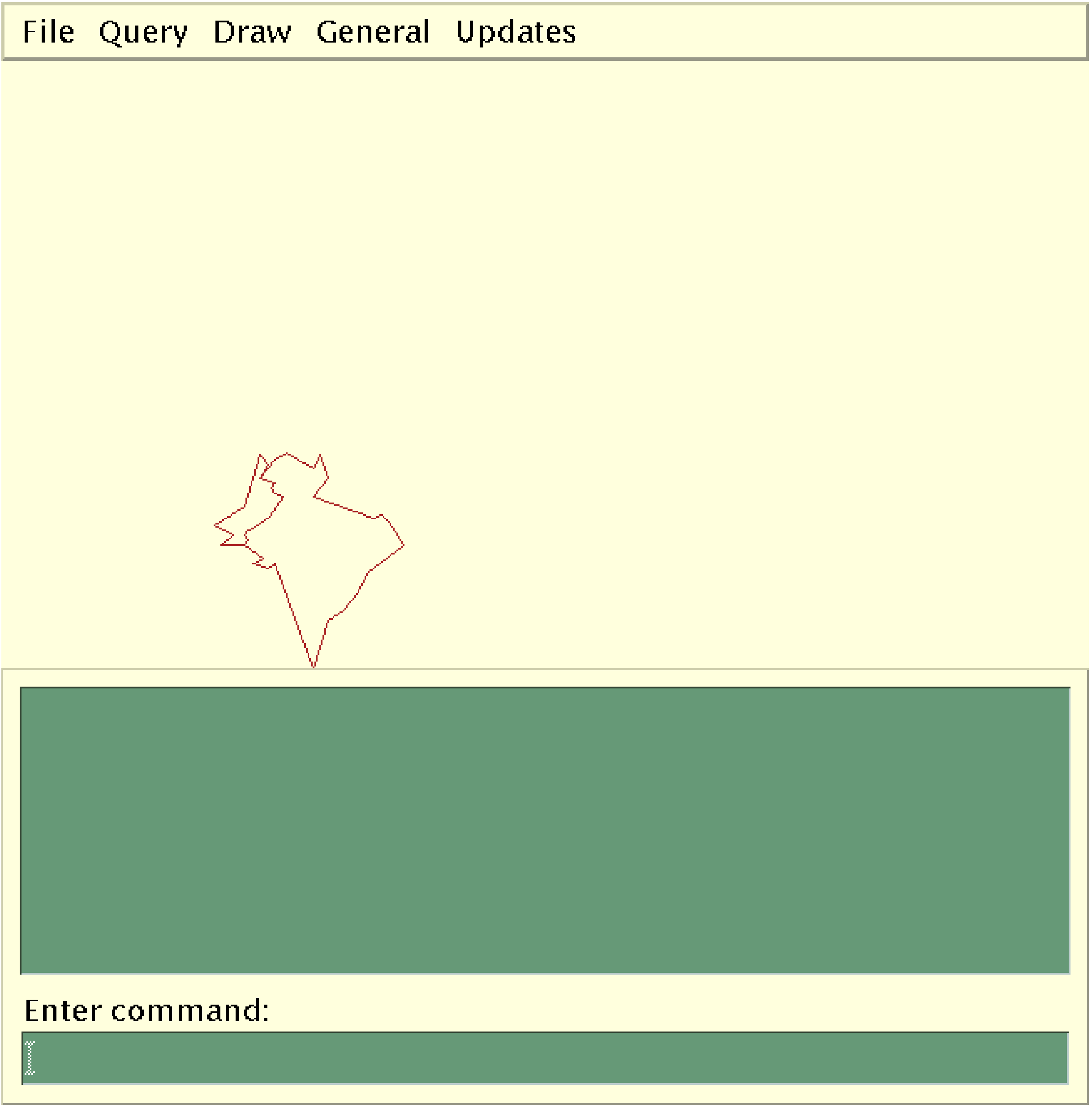

Figure 5.13: General Query No. 1
Query: Select all the maps whose parent is "Asia"

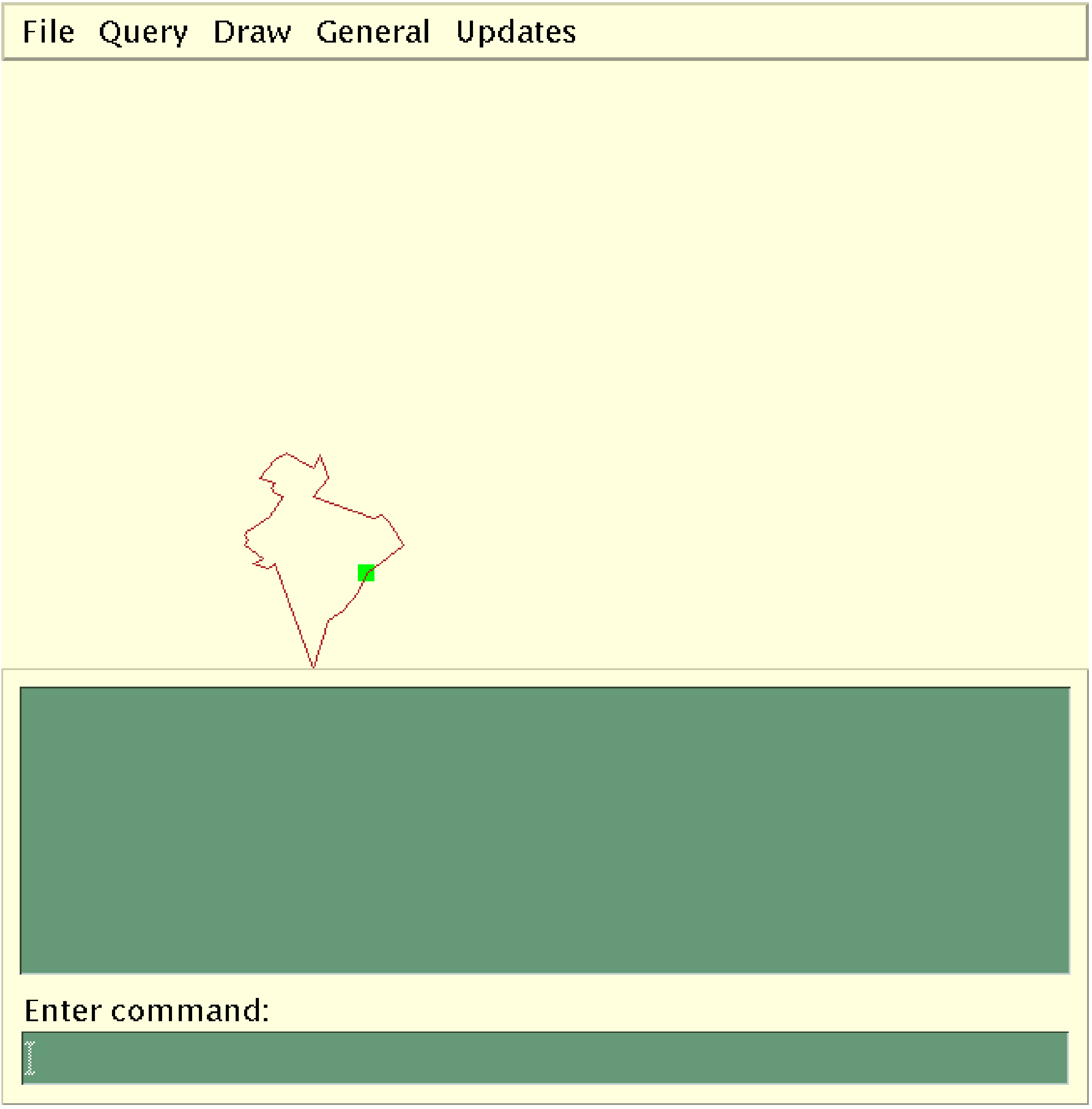

Figure 5.14: General Query No. 2
Query: Select all the maps whose parent is "Asia"
and which cross river "Ganga"

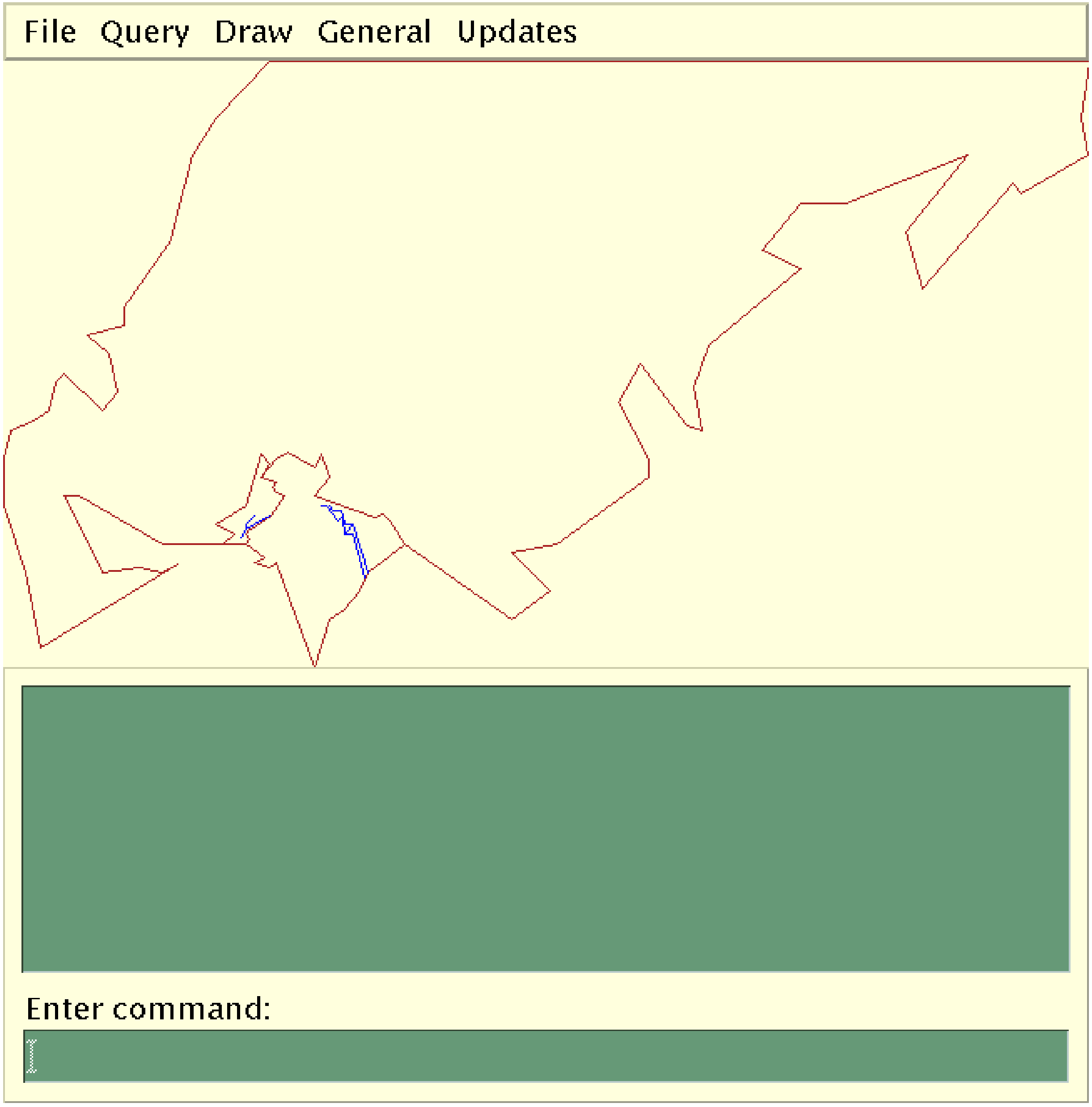

Figure 5.15: General Query No. 3
Query: Select all rivers

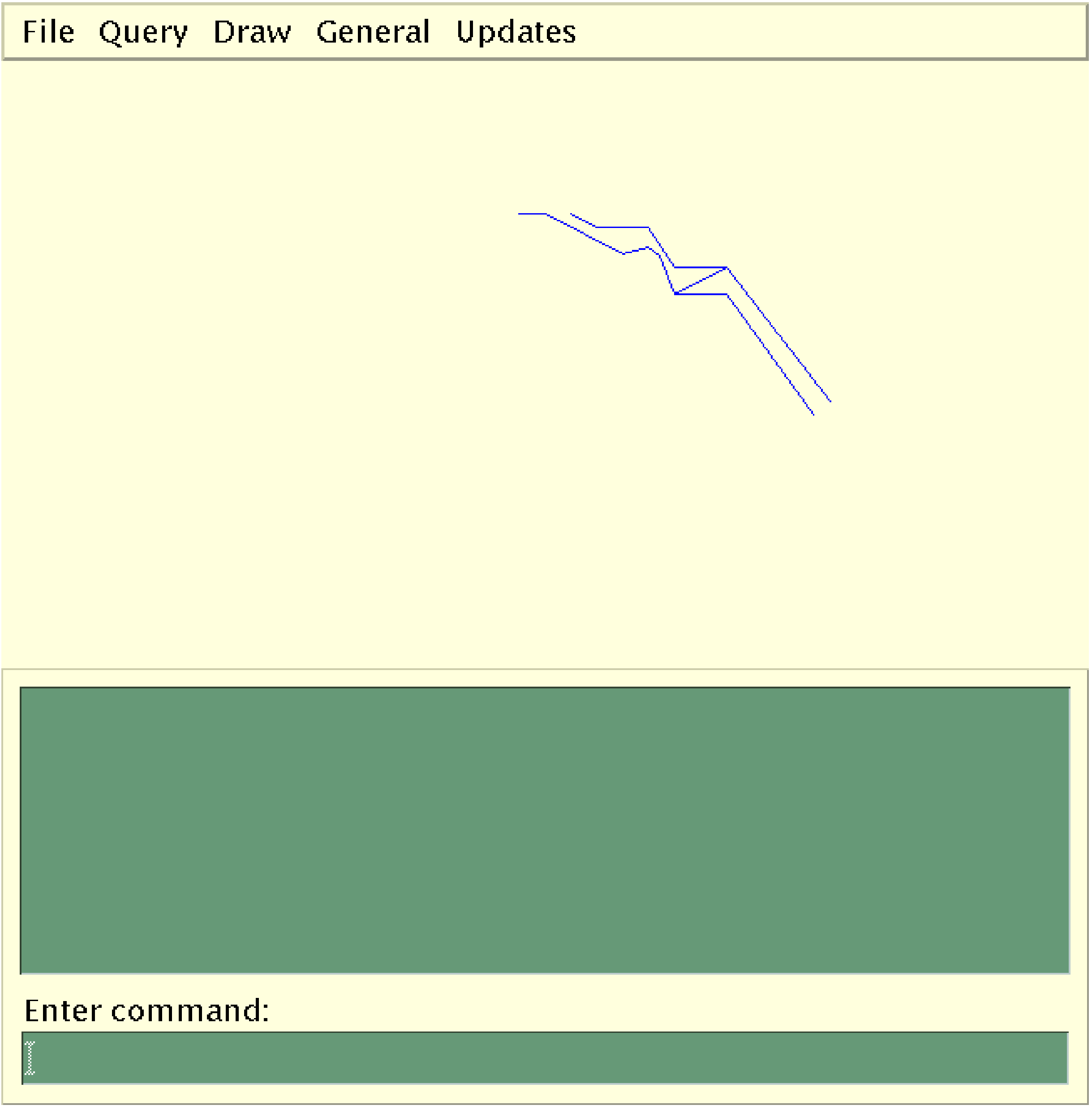

Figure 5.16: General Query No. 4
Query: Select all rivers which flow in "India".

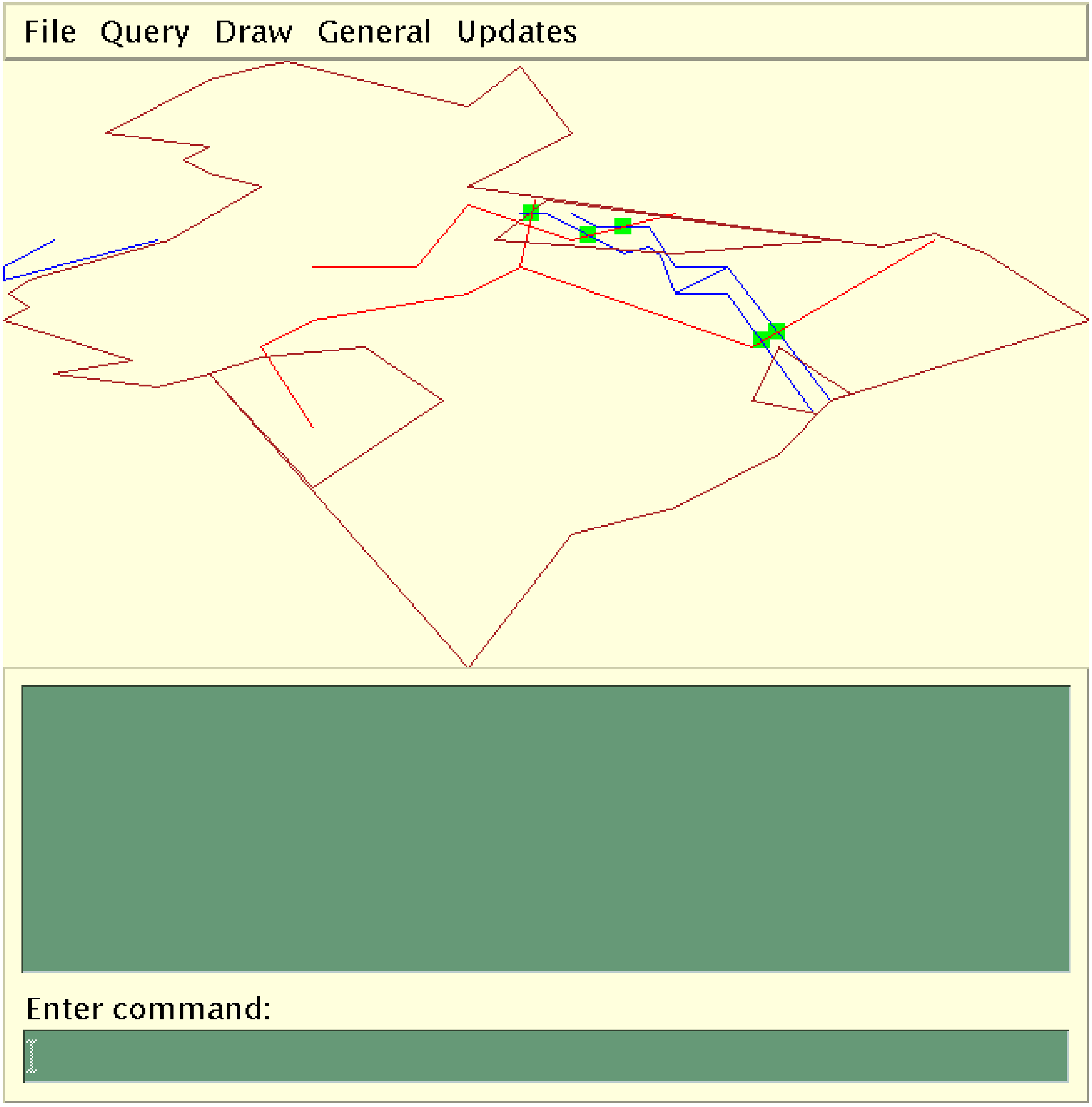

Figure 5.17: General Query No. 5
Query: Find the points where any river crosses any road
or select all maps which are of category open

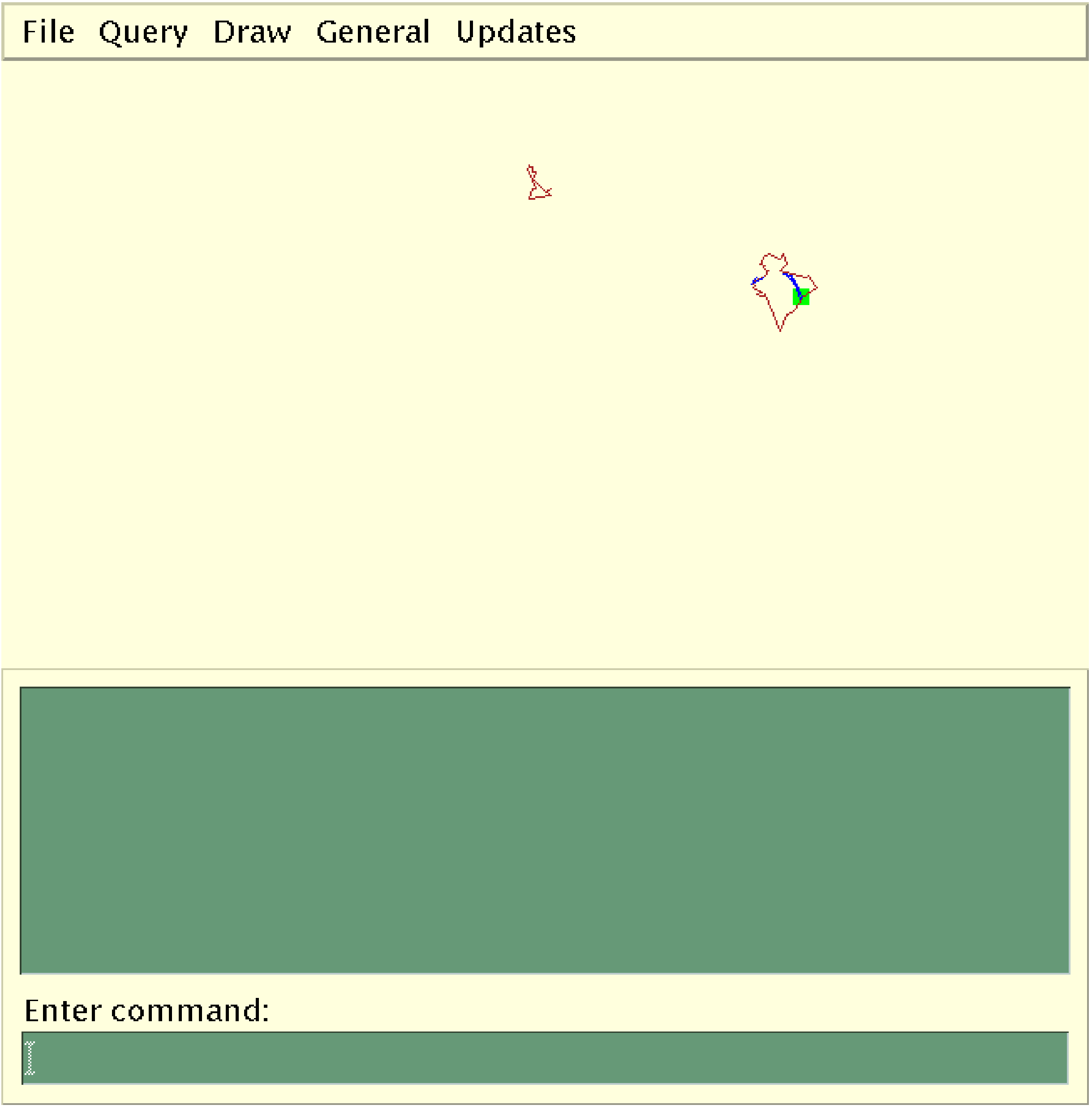

Figure 5.18: General Query No. 6
Query: (Show all the countries which are in Asia **and** which cross Ganga) **or** all the rivers **or** all countries which are in Europe

Chapter 6

Implementation

The building blocks of the OSHADHI-3 system constitute the Database server, the Graphical User Interface, the Platform and the Querying routines. The architecture of the system discussed in Chapter 3 is shown in Figure 6.1. The Database Server and the Graphical User Interface are the two components of the OSHADHI-3 biodiversity database system which are built on the SHORE Platform discussed in Chapter 4. The database server is responsible for processing the queries while the graphical user interface provided, is responsible for getting the queries from the user interactions and displaying the results of the queries to the user. In this Chapter we discuss the implementation of the database and the various algorithms that are used by the database system and the querying routines for the queries supplied by the user.

6.1 Implementation of the Database

The first step in the design of the Database is to define a schema of the database system. This is done by using the *interface* types in the *SHORE Data Language*. Interface types are types constructed with the *interface* keyword. The attributes of the interface types can be any of the primitive types as were discussed in Section 4.7.1. Interfaces also contain member functions.

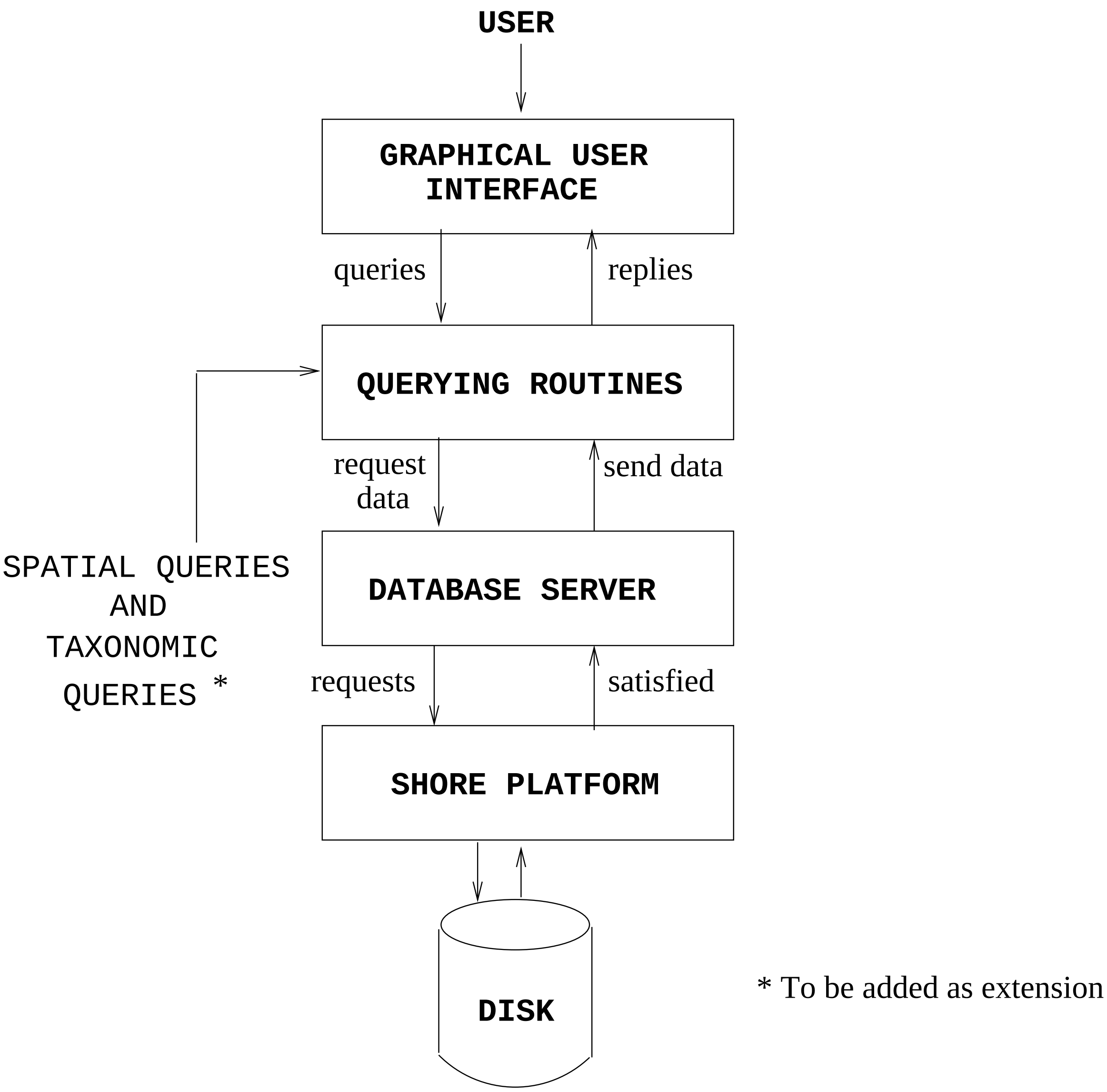

Figure 6.1: The architecture of OSHADHI-3

6.1.1 Shore Data Language

The Shore Data Language (SDL) is a language for specifying the types of persistent objects. Although the design of SDL is heavily influenced by C++, it is intended to be independent of any particular target language. Definitions written in SDL are compiled by the SDL compiler (see the manual page sdl [Sman97]) into objects stored in the database. These objects are then used to generate data declarations appropriate to a particular language, called a language binding and is discussed in Chapter 4.

SDL is derived from and inspired by the object-definition language ODL being developed by the Object Database Management Group (ODMG) as a proposed standard for object-oriented databases.

6.1.2 Description

Using SDL involves the following steps:

- Create type descriptions for the Shore objects. These type descriptions are written in the SDL language, and were discussed in the Chapter 4.
- Compile the type descriptions with the SDL compiler. (For this, a Shore server should be running [Sman97]). The effect of this step is that metatypes (type objects) are created in the Shore database. A metatype is a Shore object that contains a complete description of a type.
- Create a C++ language binding for the types, by running sdlcxx.
- Write implementations for member functions of the types.
- Compile the method implementations along with the language binding, using a C++ compiler; link them and run the application.

6.1.3 Data

The main hierarchy observed for the geo-spatial database is that shown below in Figure 6.2a. The Rumbaugh Notation [Rumb91] is used to show the hierarchy. This figure shows the containment relationship between the various classes *continent, country*, etc. That is a continent contains country and so on.

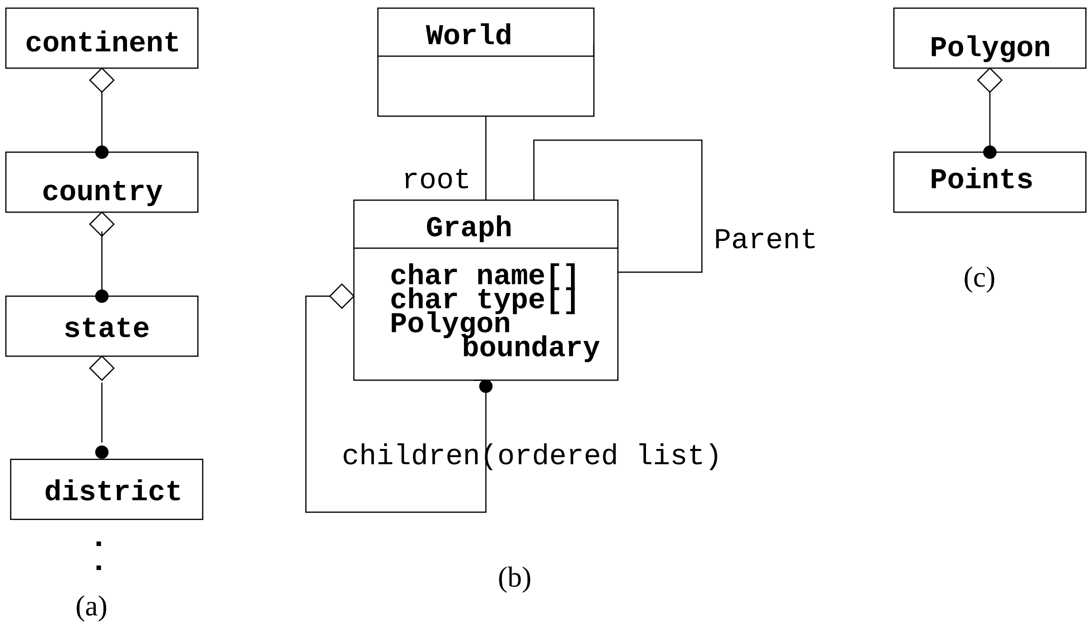

Figure 6.2: The hierarchy for the Classes in Geo-data

This means that the world contains continents, the continents contain countries, countries contain states and so on. However, the structure of these classes will be the same. Hence, instead of implementing continent, country, state as different classes we have implemented it as a single class called graph as shown in Figure 6.2b. This class has attributes namely *name, type, open, boundary, parent* and *children*. The name field stores the name of the graph, the type field stores the type such as continent, country, river etc. The open field tells whether the graph is open or closed i.e. whether the boundary is a polygon or a polyline. The boundary field stores the boundary as a sequence of points. The parent is a Pointer to the parent of the graph. Example Asia would be the parent of India. We maintain an ordered list of children in the graph. Example India, Pakistan, etc would be the children of Asia. Similarly, oceans can be called the children of world. The ambiguity, however, arises for rivers. That is because rivers could flow across countries. This ambiguity is resolved by saying that a river is a child of a continent if it flows across countries, otherwise its a child of a country. Same technique can be applied to roads as well. Same will follow for countries like Russia which belong to both Asia and Europe, then we say that it is a child of the world.

Thus a uniform data structure is used for all the maps. Hence, we have is a single SHORE *interface* discussed in Section 6.1 for all kinds of maps that exist. Each map is identified as below:

```
interface Graph
{
      public:
      attribute char type[128];
      attribute char name[128];
      attribute char open;
      attribute ref<Graph> parent;
      attribute ref<Polygon> boundary;
      attribute sequence<ref<Graph> > children;
};
```

Why is this interface called *Graph*? Why not *maps*? The reason is again “river”, “roads”, “lakes”, etc. All maps are polygons and polygons means they have to be closed but rivers , “roads”, “lakes”, etc cannot be closed polygons, they are collections of polylines [Seq93], represented in our model as open graphs. Hence, we distinguish the maps to be either open or closed, depending on the type of the map. Hence, the class is called Graph and not maps. This feature makes the implementation very flexible.

Hence, each graph has a name, a type, category i.e. whether it is open or closed, it stores a pointer to its parent, it has a boundary and the list of its children.

The boundary here is a polygon. The polygon is a collection of points. The Rumbaugh notation [Rumb91] is shown in Figure 6.2c. In the hierarchy here we ignore the line class. The polygon is actually a collection of lines but lines in turn contain points, hence we directly store the end points of the polygon. This avoids storing the points twice, as the endpoints of two adjacent lines.

Some additional classes required for the implementation are as follows:

- Points
- Line
- Polygon
- World

Note: The Line interface is not used in our implementation. It is, however, provided for future changes in the way of implementation.

All these interfaces form a part of a *module* called **Spatial** and the entire world is stored in a **Pool** called **graphs**.

This design has been done keeping in mind the infrequent updates that would be done to the geo-spatial database.

6.1.4 Member Functions

Along with the data types, the interfaces also contain some member functions which are necessary to answer the various queries and are called by the querying routines. Since they are the member functions of the interface, they can be easily called by the Querying Routines without having to generate any instance of these interfaces.

Most of the functions which the querying routines call are in the Polygon class. This is because all the maps have a boundary which is a Polygon and hence these functions can be easily called. The Polygon interface contains some functions like:

- contains: This function finds out whether a given point is inside the polygon. The algorithm is based on the algorithm in [Hear90].This algorithm is discussed in detail in subsection 6.2.1
- lineCrosses: This function finds whether the two lines cross and returns their point of intersection. It is useful in finding whether polygons intersect or overlap. Also rivers crossings are found using this function.
- polycrosses: This function uses the lineCrosses function to find whether any polygon crosses the given polygon. To determine if the two polygons cross, we check if any of the lines of one polygon crosses any of the lines of the other.
- inside: This function finds whether the requested polygon is completely inside the given polygon. This algorithm checks if any of the lines in requested polygon cross any of the lines in the given polygon. If so the polygon cannot be inside the given polygon. In the next step the algorithm uses the contains function to find if all the points of the requested polygon are inside the given polygon. If all the points of the requested polygon are inside the given polygon, then it is completely inside.

6.2 Implementation of Queries

The queries mentioned in Chapter 5, are implemented using the algorithms discussed in the following subsections.

6.2.1 IsPointInside

We use the odd-even rule to find whether the given point is inside the polygon. We apply the odd even rule, also called the odd parity rule or the even parity rule, by conceptually drawing a line from the given point P to a distant point outside the coordinate extents of the object and counting the number of crossings along the line. If the number of polygon edges crossed by this line is odd, then P is interior otherwise P is exterior.

Our algorithm involves the following steps. Let the required Point be (x' , y') as shown in the figure 6.3

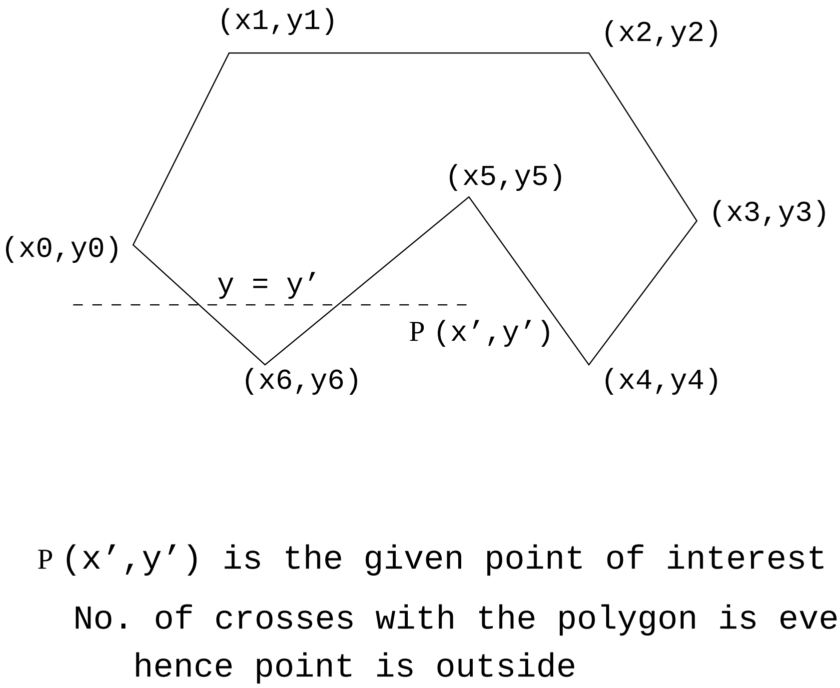

Figure 6.3:

1. Conceptually draw a line through y', thus having equation **y = y'**.
2. Construct a line with the adjacent points of the polygon.
3. Consider the first two points of the polygon - (x_0, y_0) and (x_1, y_1).

If (! (((y_0 < y') && (y_1 < y')) ||((y_0 > y') && (y_1 > y'))) || (((x_0 < x') && (x_1 < x')) || ((x_0 > x')&& (x_1 > x'))))

3a. if ((x_0 < x')&& (x_1 < x'))

3b. i.e. x' lies between x_0 and x_1.

3c. increment crosses count.

4. Find the point of intersection of the two lines in 1 and 2. Lets call the point of intersection (x", y").

5. If (((($x_0 \leq$ x") && ($x_1 \geq$ x")) || (($x_0 \geq$ x") && ($x_1 \leq$ x"))) && ((($y_0 \leq$ y") && ($y_1 \geq$ y")) || (($y_0 \geq$ y") && ($y_1 \leq$ y"))) && (x" $\leq$ x'))

$\Longrightarrow$ (x'', y'') point lies between the endpoints of the line constructed in 2.

5a. If(y" $= y_0$) (Compare with the y-coordinate of previous point)

5a(i). If $(((y_1 < y'')$ && $(y_{-1} > y''))$ || $((y_1 > y'')$ && $(y_{-1} < y'')))$

$\Longrightarrow$ *Line* crosses the polygon Hence, increment the crosses count.

5b. else if(y" $= y_1$)(Compare with the y-coordinate of next point)

5b(i). If $(((y_2 < y'')$ && $(y_0 > y''))$ || $((y_2 > y'')$ && $(y_2 < y'')))$

$\Longrightarrow$ *Line* crosses the polygon Hence, increment the crosses count.

6. If 5. is not true, surely it crosses and hence increment crosses count.

7. Repeat from step 2 for all the points in polygon.

8. If the crosses count is *odd*, the point is inside else it is outside.

6.2.2 LineCrosses

This algorithm checks if two lines are intersecting each other. In our implementation, the two lines are given by the longitudes, latitudes of the endpoints of the lines. The algorithm involves the following steps:

1. Find the point of intersection of the two line segments. Let the point of intersection be denoted by x_0, y_0. Let the endpoints of the two lines be given by $x_{11}, y_{11}, x_{12}, y_{12}$ and $x_{21}, y_{21}, x_{22}, y_{22}$. The calculation of the point of intersection requires to check the following.

1a. if the lines are perpendicular

$x_0 = x_{11}$

$y_0 = y_{21}$

if $x_{11} == x_{12}$ and $y_{21} == y_{22}$

or $x_0 = x_{21}$

$y_0 = y_{11}$

if $x_{21} == x_{22}$ and $y_{11} == y_{12}$

1b. if one of the lines is vertical

$$x_0 = x_{11}$$
$$y_0 = ((x_{11} - x_{21}) * (y_{22} - y_{21}))/(x_{22} - x_{21}) + y_{21}$$
if $y_{11} == y_{12}$

or

$$x_0 = x_{21}$$
$$y_0 = ((x_{21} - x_{11}) * (y_{12} - y_{11}))/(x_{12} - x_{11}) + y_{11}$$
if $y_{21} == y_{22}$

1c. if one of the lines is horizontal

$$x_0 = ((y_{11} - y_{21}) * (x_{22} - x_{21}))/(y_{22} - y_{21}) + x_{21}$$
$$y_0 = y_{11}$$
if $x_{11} == x_{12}$

or

$$y_0 = y_{21}$$
$$x_0 = ((y_{21} - y_{11}) * (x_{12} - x_{11}))/(y_{12} - y_{11}) + x_{11}$$
if $x_{21} == x_{22}$

1d. If any of the above are not true then

$$\begin{aligned} x_0 = ((&x_{21} * (x_{12} - x_{11}) * (y_{22} - y_{21}) - x_{11} * (x_{22} - x_{21}) * (y_{12} - y_{11})) \\ &-((y_{21} - y_{11}) * (x_{12} - x_{11}) * (x_{22} - x_{21}))) \\ &/((y_{22} - y_{21}) * (x_{12} - x_{11}) - (x_{22} - x_{21}) * (y_{12} - y_{11})); \end{aligned}$$
$$y_0 = ((x_0 - x_{21}) * (y_{22} - y_{21}))/(x_{22} - x_{21}) + y_{21};$$

2. If the point of intersection i.e. (x_0, y_0) is between the endpoints of both the lines, then the two lines cross each other.

6.2.3 IsCompletelyInside

This algorithm finds if the polygon is completely inside the required polygon. The steps for this algorithm are

1. For all the points in the polygon, construct a line with adjacent end points of the polygon.

2. If any of these lines cross the required polygon, then the polygon is not inside.

3. If none of the sides cross the required polygon, then for all the points in the polygon, check if they are Inside the required polygon.

4. If step 3, is true then Polygon is inside the required polygon.

6.3 Access Methods

The access method provided for the spatial data set is the region quad tree index [Same89].

6.3.1 Quad Tree Indexing

For each graph in the database, we store the end points of its bounding rectangle in the quadtree. This index enables for the quick retrieval of maps which are within a specific rectangle given by its diagonal endpoints. The entire two dimensional space is divided into four regions the North-East, South-East, North-West, and South-West. Correspondingly each node has a set of four children, which are also quadtree nodes, namely **NE, SE, NW, SW**. Depending on the point stored in the node(This point is one of the endpoints of the map), a suitable path is followed along the tree.

Each node of the quad tree consists of seven fields as discussed in [Same89].

The structure of each node is as below:

```
interface QNode
{
      attribute double x, y;
      attribute ref<QNode> SWNode;
      attribute ref<QNode> NWNode;
      attribute ref<QNode> SENode;
      attribute ref<QNode> NENode;
      attribute Info info;
};
```

The field Info is a struct which contains information about the point stored in the node, and is as below:

```
struct Info
{
        ref<Graph> ElemPtr;
        ref<QNode> otherpt;
        long type;
};
```

The ElemPtr contains a reference to the graph, the QNode otherpt is the QNode that corresponds to the other end of the graph. Type corresponds to whether we are referring to the top left point or the bottom right point. The endpoints of the bounding rectangle of the graph are obtained by using the function called **getEndpoints** which finds the endpoints of the graph at the time of its insertion in the quad tree and accordingly inserts them at the proper position in the Quadtree.

Since, more than one graph could have the same end points repetitions are allowed in the index we have implemented.

The two end points of the graph are thus stored in the quadtree along with the references to each other. Hence, we know exactly within what region our graph lies. While searching for the graph within the region just compare the endpoints of the graph with the endpoints of the given region and accordingly follow the direction. When a match is found we return.

Thus, the region quad tree discussed finds the graphs within the range specified.

6.4 The Graphical User Interface

The Graphical User Interface was developed using X [XWin98] and Motif [Dan94], [Pau94].

6.4.1 Drawing Area

The Motif widget *drawing area* and elementary X drawing routines are used to draw the maps. The drawing area and other data structures such as the *Graphic Context* (GC) are embedded in a structure called as **Canvas**. The Canvas also contains the extreme points of the map and drawing area. This information is used to convert longitudes and latitudes to pixel positions on the drawing area. It is also later used for zooming.

6.4.2 The Query Interface

The user can ask all the queries through a simple graphical user interface. The queries are menu driven and the user has to select the query of his choice. Once he selects his query a form will be provided to him asking for his requirement. When the user gives his requirement, the query will be processed and the result will be returned to the user on the Graphical User Interface itself. The answer could either be displayed on the Canvas where the maps are drawn or they could be shown to him on a Selection Pop up dialog which shows the names of various maps which satisfy the query. Examples are given in Chapter 5.

6.4.3 The General Query

In addition to the queries, mentioned in Chapter 5, we also provide the user with a general query interface. This query can answer any conjunction and disjunction of queries the user might want to ask. The user is required to frame his query in disjunctive normal form. The user is provided with a form to add disjunctions. For each term in the disjunction, a separate form is provided to add a set of conjunctions. Each conjunction is made of a set of requirements. A requirement form is then provided in which the user can specify his where condition consisting of name, type, category, etc. He can include conditions on the parent, for example: query *find all maps whose parent is United kingdom or whose parent crosses river Ganga* and also spatial requirements such as crosses and inside. If he leaves any field in the requirement empty, the application assumes that the user has requested for all the maps which match his query, without considering that field. That is, a blank is equivalent to “any”.

To implement this query we have three classes Conjunction, Disjunction and Requirement. The Disjunction class contains an array of Conjunction which in turn contains an array of Requirements. The Conjunction and Disjunction classes have the satisfied function which satisfies the query according to the AND/OR logic. The Requirement class contains the requirement of the user such as the name, type, category of the maps and the required condition. To satisfy the requirement we have a satisfies function which opens a *PoolScan* and matches the query.

The Requirement class is as below.

```
struct Requirement
{
    char *name;
    char *type;
    char category;             //open/closed/*
    Requirement* parent;
    Requirement* crossboundary;
    Requirement* inboundary;
    bool satisfies(Ref<Graph> g);
};
```

The results are returned back to the user.

6.5 Error Messages and other features

While the user is entering his requirement or query, the user can make several mistakes. For example leave a field empty or provide a name which does not exist in the database. The OSHADHI-3 biodiversity system is capable of checking the various errors which a user might make. It immediately displays a message, informing the user about the error that he has made. This makes the system more robust and prevents errors that SHORE might generate.

6.6 Some advantages

The implementation of the OSHADHI-3 biodiversity database system is quite flexible. Any type of map can be added on to the system. No special interface has to made while adding any other type of map. Unlike, the previous OSHADHI biodiversity database system i.e. OSHADHI-1 and OSHADHI-2, OSHADHI-3 does not have to rely on socket communication. We, thus, have a tight integration of the Graphical User interface and the Database server. Thus, the entire system runs on a single machine.

Chapter 7

Comparison

The previous chapters in this thesis gave an overview about the biodiversity data, the queries that need to be answered, the implementation of these queries, and how it can be used to answer the various biodiversity queries. In this Chapter we compare the OSHADHI-3 database system with other comparable Geographic Information Systems (GIS) like the *ArcView* [Arc97] and *Paradise* [Par97]. This Chapter reflects on the issues such as how our system differs from the other two GIS systems mentioned here, and the features supported in other GIS systems.

7.1 Paradise: Geo-Spatial DBMS

We first compare our system with *Paradise*. Paradise, an object−relational database system, built at the University of Wisconsin [Par97], is a parallel geo-spatial DataBase Management System. Let us see the overview of the system and the different features that it can provide.

7.1.1 Data Format

The type of data used by the Paradise Project is the *raster* data. All the querying is done on these raster images obtained from satellites. Raster data records spatial information in a matrix organised as a set of rows and columns. Hence, maps can be directly displayed using these raster images. Paradise provides data types to store and retrieve image(raster) data. The key focus of the Paradise DBMS has been the development of techniques for dealing with large satellite images and arrays of scientific data.

7.1.2 Data Structures

Paradise provides a set of spatial data types like points, polygon, polyline, circle, etc. The standard attribute types such as integers, floats, strings are provided as well. Time data is also modeled in the Paradise project. Hence, queries related to time can be asked.

Paradise used various data structures for modeling different map-elements like roads, cities, land cover, drainage(rivers, etc). The shape of these maps are polyline, polygon, polygon, polyline respectively.

7.1.3 Queries

Paradise supports all the queries that are discussed in the *Sequoia 2000 benchmark* [Seq93]. In addition to these, Paradise supports a new set of queries which test geo-spatial functionality. These queries are :

- *Find all lakes, rivers which cross a road.*
- *Find the closest road to a given point.*
- *Find the closest drainage feature (i.e. lake, river) to every large city.*

There are two other queries which can be answered by Paradise but are not discussed in Sequoia 2000 benchmark. These are related to raster data and hence are not mentioned here. For a detailed information of these queries, refer to [Par97].

The querying routines call the member function of the various data structures that are stored as Abstract Data Types. For example the *IsPointInside* is implemented as a member function of the Polygon class. The spatial data types, discussed above in the subsection 7.1.2, provide a set of spatial operators like

```
 clip(Polygon), makeBox(length), closest(shape,
POINT),
```

etc. These operators can be directly accessed from the query language. The query language is an extended version of SQL.

7.1.4 Architecture

Paradise provides graphical frontend for browsing, querying on Windows NT and UNIX. The Paradise database system consists of one Query Coordinator process and one or more Data Server processes. The Query Coordinator and the Data Server processes communicate via a communication infrastructure that automatically selects the transport mechanism.

Clients (applications or GUIs) connect to the Query Coordinator process. After parsing, the Query Coordinator controls the parallel execution of the query on the Data Server. Results are collected by the Query Coordinator for delivery back to the client.

Paradise system has been designed for Parallel architecture. Please refer [Par97] for further details.

Paradise provides standard relational operators. The set of aggregate operators include the SQL operators as well as the spatial operators like "closest".

7.1.5 Backend

The Query Coordinator and Data Server are implemented on top of SHORE (Scalable Heterogeneous Object REpository) [Sho97] Storage Manager. The backend provides storage volumes, files of untyped objects, B^+ trees and R^* trees. SHORE provides recovery mechanisms as well.

7.1.6 Indexing Techniques

Paradise uses the R^* trees [Gunt94] as the indexing method for the spatial data set. Spatial data involves objects which are multi-dimensional. Hence, one-dimensional database indexing structure like B^+ trees are not suitable for these objects. R^* trees can represent data in several dimensions. It is a height balanced tree with index records in the leaf node. The methods for insertion, deletion, searching, etc are discussed in [Gutt84].

7.1.7 Additional Features of Paradise

Apart from those mentioned in the above subsections, Paradise provides some additional features. Some of these are discussed below :

- Paradise is a parallel geo-spatial DBMS. Some techniques in parallel object-relational database have been implemented in this Project. Paradise has been designed to handle new aspects of parallel geo-spatial database. For example: Geo-spatial database systems allow spatial objects as their attributes. Applying parallelism to queries over such objects requires spatial partitioning. However, spatial queries can not be localized to single spatial partition. They must be evaluated by a combination of replication of data and operations. This introduces some trade-offs which are not present in typical relational applications. Paradise provides such features for parallel system.

- Paradise is scalable to multi-processor systems.

- The spatial queries can be answered through an extended version of SQL. The spatial operators like *clip* can be called directly from the query language.

7.2 ArcView GIS

The ArcView is a tool that brings geographic information to the users desktop. ArcView helps to query and analyze spatial data. ArcView is made by Environmental Systems Research Institute (ESRI).

Most of the material in this Section is taken from [AHelp97].

7.2.1 Data Format

The ArcView GIS stores the spatial data as the vector data . However, the attribute data is stored in tables known as attribute tables, storing data in tabular form makes table joins simple and hence SQL queries can be answered using the tabular information. Using ArcView's connection feature, the user can connect to a database server such as Oracle or Sybase, and run a SQL query to retrieve records from it.

ArcView uses image(raster) data as well for displaying the maps. Image data includes satellite images, air photographs and other remotely sensed or scanned data.

7.2.2 Data Structures

The data structures that are used for storing maps involves points, lines, polygons, etc classes. The vector data (i.e points) is used to form lines, polygons, etc which are used to construct maps.

The attribute data, however, is stored as tables as discussed in subsection 7.2.1.

7.2.3 Queries

Arcview can answer the spatial queries as are discussed in Chapter 5 including the spatial joins. In addition it can query on the attribute data and display the results in the form of tables and bar graph displays.

ArcView can, however, answer SQL queries as discussed above in subsection 7.2.1

7.2.4 Indexing Techniques

Indexing techniques are adapted both for the relational data as well as the spatial data. There is a provision to Create and Remove an index in the ArcView GIS. Unfortunately, we were unable to explore the index types used in the ArcView.

7.2.5 Additional Features of ArcView GIS

Lets discuss the special features of the ArcView GIS.

- *Views*: With ArcView the user can work with geographic data in interactive maps called Views. Views are saved in the ArcView *project*(discussed below) that the user is currently working in.

- *Tables*: ArcView works with tabular data. If the user clicks on the view, their records are highlighted in the table showing their attributes. If he selects a feature record in the table, their features are highlighted on the view.

- *Charts*:. ArcView provides charts for the user. The user can click on features on a view to add them to the chart. ArcView lets the user work simultaneously with geographic, tabular and chart representations of his data.

- *Layouts*: ArcView's layouts lets the user create full color maps. Layouts a have a live link to the data they represent. Hence, all the changes to the data are reflected in the layout automatically.

- *Scripts*: ArcView scripts are macros written in Avenue, ArcView's programming language and development environment. With this feature the user can write his own scripts which he might want to use.

- *Projects*: All the components of your ArcView session: views, tables, charts, layouts, and scripts are conveniently stored in one file called a project.

7.3 OSHADHI-3

We now discuss the various aspects involved in OSHADHI-3 as compared to the above mentioned systems.

7.3.1 Data Format

As in ArcView, OSHADHI-3 is built using *vector* data. The advantage of using vectored data over raster data is vector data can record accurately the boundaries of the locations, buildings, etc. Raster data is useful for displaying the maps but no querying, example

IsPointInside, can be done effectively with the raster data. Hence, raster data has to be converted to polygonal data. Since, OSHADHI-3 does not support raster data, queries related to raster data cannot be answered as can be done in Paradise.

7.3.2 Data Structures

OSHADHI-3, Paradise and ArcView provides a set of spatial data types like points, polygon. However, Paradise contains a data type called polyline. This data structure is absent in OSHADHI-3. However, OSHADHI-3 has uniformity in data structures. We use the same data structure for all kinds of maps. We distinguish between a polygon and polyline by defining whether the map/polygon is closed or open depending on the type. The details are mentioned in the Chapter 6.

7.3.3 Queries

OSHADHI-3 answers the queries discussed in the *Sequoia 2000 benchmark* [Seq93] except the ones involving raster data. Refer to Chapter 5 for the details of the queries. Paradise supports all the queries that are discussed in this benchmark. Along with these Paradise supports a new set of queries discussed in 7.1.3. Our system is is also capable of handling the query *Find all lakes, rivers which cross a road.* Our system can not answer the query *Find the closest drainage feature (i.e. lake, river) to every large city*, but it can find out any river, lake crossing a region.

The other two queries discussed above in 7.1.3 which can be answered by Paradise, cannot be handled by our system.

The querying routines in Paradise as well as OSHADHI-3 call the member function of the various data structures that are stored as Abstract Data Types. For example the *IsPointInside* is implemented as a member function of the Polygon class.

OSHADHI-3 can perform GIS related operations like zooming. It can also provides *Views* similar to ArcView where the user can store in the region of his interest and further querying can be done at a later time.

7.3.4 Backend

Similar to Paradise, OSHADHI-3 uses SHORE (Scalable Heterogeneous Object REpository) [Sho97] as the backend for data storage. The various features of SHORE, that are described in Chapter 4 makes it the most suitable backend for object-oriented databases.

7.3.5 Indexing Techniques

Unlike Paradise, OSHADHI-3 uses Quadtree indexing technigues. Traditional indexing techniques such as B^+ trees can not be used for spatial indexing since it is one-dimensional indexing structure. Spatial data objects cover areas in multiple dimensions. For example, map objects like countries occupy regions of non-zero size in two dimensions. Therefore spatial search involves finding regions within, say, longitude x_1,latitude y_1 and longitude x_2,latitude y_2. Region Quad tree enables range search in the area of interest. Hence, this indexing method is implemented in our system.

7.3.6 Additional Features of OSHADHI-3

OSHADHI-3 system is a system built to answer the geographic and spatial queries in the OSHADHI Biodiversity Database System. It answers most of the queries discussed in the Sequoia 2000 benchmark which is a subset of the queries answered by the Paradise Project as discussed above in subsection 7.1.3. Here are some of the features unique to OSHADHI-3 system.

- The uniformity in data structures for the maps enables us to store Parent and children (It is an ordered list of maps example Asia will contain the list of its children as India, Pakistan, etc). for a map. Hence, the parent and children of a map are easily accessible. Hence, all the map attributes can be specified using a single class definition. We call it *Graph* and the details were given in Chapter 6.

- The user might want to create some temporary Polygons (regions) and find the join of this polygon with other maps in the database. OSHADHI-3 allows the user to store such temporary regions. He can ask back the same query at a later time if he desires.

- OSHADHI-3 has a User friendly, Graphical User Interface. This makes querying simpler for the user. Moreover help topics are provided to enable the user to ask correct format of the queries. In addition to this a general query interface has been provided. This enables the user to ask queries containing conjunction and disjunction operations.

7.4 Conclusion

This Chapter compared the OSHADHI-3 system with some earlier projects and products in GIS. This comparison gives us an overview of what possible extensions could be made to the existing OSHADHI biodiversity database. These extensions are discussed further in the next Chapter.

Chapter 8

OSHADHI: Today and Tomorrow

In the previous Chapters we discussed the evolution of OSHADHI biodiversity database system, steps taken to form the system, the queries answered in OSHADHI-3 system, and we compared the system with some previous projects like Paradise [Par97] and products like ArcView GIS [Arc97]. Having done this, lets us now discuss the special features of OSHADHI-3. Also, we will discuss the possible extensions that can be done to the existing system.

8.1 Special Features of OSHADHI-3

The OSHADHI biodiversity Information system is built using the object oriented model. We, thus get the benefits of an object oriented approach. One important benefit is *Extensibility*. We shall consider an example to see how extensibility is achieved in OSHADHI-3. Since all the maps belong to a single class *Graph*, any map can be added very easily to the existing database. Any new hierarchy of maps can also be easily made by providing the parent-children information. This avoids the need of modifying the schema as well. Thus, without affecting the schema we can easily add new map types. All that is required for forming a new hierarchy is to create a new *Pool* in the SHORE namespace to store the objects in the hierarchy. This requires a minimum amount of coding to be done. Hence, all the types of geographic data can be handled by the system. There would be no alterations to the querying routines except that the corresponding *Pool* should be specified for searching.

The geo-spatial data that is used in our system is the vectored data. Vector data records spatial information as x, y coordinates in the rectangular (planar) coordinate system. The points store the longitude and latitudes of the maps. This makes the maps storage a trivial job. Vector data is very well suited to recording the location of discrete geographic features with precise locations of boundaries. This would not be possible if the data was stored in raster format. The associated data with the maps can also be easily stored. Storing this kind of data makes the implementation simple, since the parent, children information is stored along with the maps boundary. Storing the images in pixel format makes processing of spatial operators difficult.

Any type of information related to the maps, example population of a region, average temperature, soil, etc can easily be embedded. The querying routines for such data also have to be altered only in a minimal way i.e. by taking care of the condition given by the user.

Since the end users of the system are going to be biologists, no query language has been developed and interaction is through Graphical User Interface.

The Graphical User Interface, the querying routines and the Database Server are all integrated and thus can be run on a single machine. This eliminates the need of communicating with the Database Server through a socket in the client-server form. Hence, the complexity associated with socket programming is eliminated. However, the modification in the querying routines is very less, on modification of the GUI.

The Graphical User Interface provides with a general query interface which can answer any conjunction and disjunction operations in a query, requested by the user. This facilitates the user to ask any complex query on the existing database. However, if the database is extended and any kind of extra information is added, there would be slight alterations in the interface of the general query and the querying routines associated with them. By and large, the effort required for altering these querying routines is minimal compared to the benefits associated with it. The most important benefit associated with the *General Query* is its ability to answer any query. Thus, the general query all by itself can answer any query and there is no need to make separate interfaces for different queries.

8.2 Extensions and Future Work

- **More Information**

  The OSHADHI-3 biodiversity database systems contains data about the maps, however it does not contain any other additional information like the population of the region, temperature, soil etc. This data needs to be added, so that similar regions with respect to certain attributes can be found.

  Also, the taxonomy data should be added to the existing geographic data, and accordingly querying routines should be added. This would make the OSHADHI biodiversity system a complete system capable of answering the Geographic as well as Taxonomic data.

- **Platform**

  The Platform used for the OSHADHI-1 and OSHADHI-2 biodiversity database was EXODUS. Before we can answer the Taxonomy queries on the OSHADHI-3 biodiversity database system, the OSHADHI-1 and OSHADHI-2 biodiversity systems should be built on SHORE platform. The work on this is being done in the *Database Systems Lab*.

- **Query Language**

  OSHADHI-3 system has form based querying. Hence, separate forms have to be provided for different queries. The system does provide a general query interface which is capable of handling queries containing any combination of conjunction and disjunction operations. By and large, a complete DBMS must have a query language through which the user can interact with the system.

The Vaidya [Mou97] query language is present, which was built as a part of OSHADHI-2 system, but it can handle only taxonomy related queries. It cannot handle the queries developed in OSHADHI-3 biodiversity database system. Hence, a query language should be built which can help in calling the spatial operators directly.

- **Raster Data**

  The system only supports vectored data and hence all the data has to be fed manually. Vector data can accurately record the actual ground location features. However, vector data is highly dependent on the number of (longitude, latitude) coordinate points that are chosen. Hence, the system should be capable of handling raster(image) data. Raster data records spatial information in a matrix organised as a set of rows and columns. If this feature is added, it would automate the data entry process. This feature enables reading the maps directly from gif images. The spatial operators would be useful for this kind of data as well. However, the contours should be extracted first, and all the associated data with the maps must be added to the raster data. Thus, raster data and images can be integrated with the system.

- **Additional GIS features**

  The current system provides GIS features but these are minimal. Some advanced GIS features could be added on such as colours associated with each map. The associated data or the attribute data should be added on to the system so that charts, graphs, bar-graphs associated with spatial data can be displayed on the maps. These attributes cover most of the features of the ArcView GIS.

Chapter 9

Conclusion

In this report, we have addressed a problem which many environmental scientists and biologists face - the requirement of a single source of various kinds of biodiversity related data, and a system to manage this data efficiently and process a variety of queries on them. We have designed and developed OSHADHI, an object oriented biodiversity database, with the purpose of meeting the requirement of the researchers and scientists at the *Centre for Ecological Sciences.* The work that is addressed in this thesis, "OSHADHI-3, The Biodiversity Database System", mostly focuses on the geographic and the spatial data part of the whole biodiversity data unlike OSHADHI-1 and OSHADHI-2 which focussed mostly on the taxonomy part of the biodiversity data.

Since traditional database systems fail to meet the modeling and processing requirements of the the biodiversity database, we have adopted an object oriented approach to develop this database.

The modeling and processing requirements of the geographic and the spatial data were studied and an object model was designed. The system architecture of OSHADHI-3 biodiversity database system has the SHORE as the back-end. The database server which processes the queries, is built on top of SHORE. The Shore Data Language has a language independent library. The Language Independent Library also contains the Unix compatibility library, with procedures that emulate common file system calls such as open, read, and seek. The Shore Data Language (SDL) is a language for specifying the types of persistent objects. The Shore platform also has additional features such as persistence, Unique Object-Identifiers for the objects. To store these objects a Unix like namespace is provided. In addition to these Shore also has database features, i.e., **BeginTransaction()**. Recovery mechanism is provided which is a must for all database systems. SHORE also has built in indexing techniques like B^+ trees and R^* trees.

All interactions between the user and the database system are through a graphical user interface. The results are also returned to the user on Pop-up dialogs or they are displayed on the Canvas.

OSHADHI-3 uses access methods like region quadtrees (for spatial data) and the pointer intersection technique for fast retrieval. This indexing technique retrieves the area of interest in the given range.

OSHADHI-3 provides data entry facility for the geographic database through files. The filename can either be given as the command line argument or could be selected in the load query option. Queries are also entered through menus and text entry. OSHADHI-3 provides a simple graphical interface that allows the user to enter data and communicate with the database easily. The *General Query* of the OSHADHI-3 system helps the user to have conjunction and disjunction operations in his query.

The OSHADHI-3 database is composed of the geographic and spatial data. However, Taxonomy, Bibliographic, and Experts data is not present in our system. However, our system can handle complex spatial queries. For example: Show (all the countries which are in Asia AND which cross river Ganga) OR (all those countries whose parent is Europe) OR (All the rivers).

There is always room for extensibility in the system. Any kind of maps can be added on to the geographic database of the OSHADHI-3 biodiversity system. Apart from maps, any kind of attributes associated with the regions could be added on to the database.

The OSHADHI-3 system covers most of the geographic related queries for the biodiversity database. However, it lacks all other kinds of data i.e. the Taxonomy data, Experts data, Bibliographic data. The previous work on OSHADHI i.e. OSHADHI-1 and OSHADHI-2 covered the Taxonomy part. What is required is intergration of the three systems and addition of Experts and Bibliographic data. Also, there is a lack of a formal query language which could make the system complete in terms of database. These limitations could be overcome by extending the OSHADHI system. This involves adding the features that were discussed in Chapter 7.

With these extensions provided, we expect OSHADHI to be a good first step in serving the needs of biologists and scientists in the field of environmental sciences.

Bibliography

[AHelp97] GeoInformation International, *Getting to Know ArcView GIS – the geographic information system for everyone*, On line Help topics.

[Arc97] GeoInformation International, *Getting to Know ArcView GIS – the geographic information system for everyone, 2nd Edition 1997*, Environmental Systems Research Institute, Inc. *http://www.esri.com*

[Arun96] Arun N. S., "Design and Implementation of an OODBMS for VLSI Interconnect Parasitic Analysis", *MSc (Engineering) Thesis, CSA Dept, Indian Institute of Science*, 1996. *http://dsl.serc.iisc.ernet.in/thesis/arun.ps.gz*

[Baye72] Bayer, R., "Symmetric Binary B-Trees, Data Structures and Maintenance Algorithms", *Acta Information* 1(4), 1972.

[Bert91] Bertino, E., and Martino, L., "Object Oriented Database Management Systems : Concepts and Issues", *IEEE Computer*, April 1991.

[BK89] E. Bertino, W. Kim, "Indexing techniques for queries on nested objects", *IEEE Trans. on Knowledge and Data Engineering*, 1989.

[Booc94] Booch, G., *Object Oriented Analysis and Design with Applications, 2nd Edition*, Benjamin/Cummings, 1994.

[Care86a] Carey, M., et. al., "The EXODUS Extensible DBMS Project : An Overview", Technical report, University of Wisconsin, 1986.

[Care86b] Carey, M., et. al., "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[Care88] Carey, M., De Witt, D., and Vandenberg, S., "A Data Model and Query Language for EXODUS", *Proceedings of the 1988 SIGMOD Conference*, Chicago, IL, June 1988.

[Cat93] R. Cattell., *The Object Database Standard: ODMG-93.*, Morgan Kaufmann, San Mateo, CA, 1993.

[Dan94] Dan Heller, and Paula M. Fergusson, *Motif Programming Manual* O'Reily and Associates, Inc, Second Edition, February 1994.

[DBE87] *Database Engineering* 10(2), Special Issue on Extensible Database Systems, M. Carey, ed., June 1987.

[Decc95] Noronha, F., "Plant Genesis of Western Ghats", *Deccan Herald*, January 22, 1995.

[Deux91] Deux, O., et. al., "The O2 System" *Communications of the ACM* 34(10), October 1991.

[Exod93] "Overview of the Exodus Storage Manager", unpublished, included in the EXODUS storage manager software release, April 1993.

[Gunt94] Gunther, O., and Lamberts, J., "Object Oriented Techniques for Geographic Data Management", *The Computer Journal*, 37(1), 1994.

[Gutt84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 SIGMOD Conference*, Boston,MA, May 1984.

[Hear90] Hearn and Baker, "GRAPHICS", Wiley Eastern, 1990.

[Kast89] Kasturi, R., et. al., "Map Data Processing in Geographic Information Systems", *IEEE Computers*, December 1989.

[Kim89] *Object Oriented Concepts, Databases and Applications*, Kim, W., and Lochovosky, J., Eds., Addison Wesley, 1989.

[Kim90] Kim, W., "Object Oriented Databases : Definition and Research Directions" *IEEE Transactions in Knowledge and Data Engineering*, September 1990.

[Kort91] Korth, H.F., and Silberschatz, A., "Database System Concepts", Second Edition, McGrawHill, 1991.

[Mou97] Chandramouli. B. N. "OSHADHI-2: A Biodiversity Information System", *ME Thesis, CSA Dept, Indian Institute of Science*, 1997. *http://dsl.serc.iisc.ernet.in/thesis/mouli.ps.gz*

[Orac88] *SQL Language Reference Manual, ORACLE RDBMS Version 6.0*, 1988.

[Over97] *An Overview of Shore*, *The Shore Project*, The Shore Project Group, Computer Sciences Department, UW-Madison, Madison WI, Version 1.1, 1997., *http://www.cs.wisc.edu/ shore*

[Par97] *Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation*, Jignesh Patel, JieBing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellmann, Jim Kupsch, Shelly Guo, Johan Larson, David Dewitt, and Jeffery Naugthon, *Proceedings ACM SIGMOD*, 1997.

[Pau94] Paula, M. Ferguson, David Brennan *Motif Reference Manual*, O'Reily and Associates, Inc, April 1994.

[Poun95] Pountain, D., "The Poison Disc", *BYTE*, May 1995.

[Rich87] Richardson, J.E., and Carey, M.J., "Programming Constructs for Database System Implementation in EXODUS", *Proceedings of ACM SIGMOD Conference*, San Fransisco, CA, May 1987.

[Rich89a] Richardson, J.E., Carey, M.J., and Schuh, D., "The Design of the E Programming Language", Computer Sciences Technical Report, University of Wisconsin-Madison, February 1989.

[Rich89b] Richardson, J.E., and Carey, M.J., "Persistence in the E Language: Issues and Implementation", *Software - Practise and Experience* 19(12), December 1989.

[Rumb91] Rumbaugh, J., et. al., *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[Same89] Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[Samp85] *Proceedings of the Discussion Meeting on Development of Scientific Databases in India*, Sampath, D., Balakrishnan, N., Rajaraman, V., Eds., 1985.

[Seq93] *The Sequoia 2000 Storage Benchmark*, Michael Stonebraker, James Frew, Kenn Gardels and Jeff Meredith., 1996.

[Sho97] *The Shore Project*, The Shore Project Group, Computer Sciences Department, UW-Madison, Madison WI, Version 1.1, 1997., *http://www.cs.wisc.edu/ shore*

[Sman97] *The Shore Project, Shoreman, Manual pages of SDL*, The Shore Project Group, Computer Sciences Department, UW-Madison, Madison WI, Version 1.1, 1997., *http://www.cs.wisc.edu/ shore*

[SS94] B. Sreenath, S. Seshadri, "The hcC-tree: An Efficient Index Structure for object oriented databases", *Proc of 20th Intl. Conf. on VLDB*, 1994.

[Stro91] Stroustrop, B., *The C++ Programming Language, 2nd edition*, Addison-Wesley, 1991.

[Vasi74] Vasishta, P.C., *Taxonomy of Angiosperms*, S. Chand and Co., Delhi, 1974.

[Vid95] R.Vidya, "OSHADHI: A Biodiversity Information System", *ME Thesis, CSA Dept, Indian Institute of Science*, 1995. *http://dsl.serc.iisc.ernet.in/thesis/vidya.ps.gz*

[XWin98] Nabajyoti, Barkakati., *X Window System Programming*, Prentice Hall, February 1998, 2nd Edition.

[Youn90] Young. D.A., *The X Window System* Englewood Cliffs: Prentice Hall, 1990.