The Spatial Perspective of Oshadhi

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

by

Madhava Rao Cheethirala

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

Jan 2000

Dedicated to my loving parents

Contents

List of Figures

Acknowledgements

I would like to thank my project guides, *Dr. Jayant Haritsa* and *Dr. Ramesh Hariharan* for their constant support and encouragement through out the course of the project. The Database Systems Lab with its comfortable environment, vast library of books and journals, the seminars and the hours long discussions will never be forgotten.

I would like to specially thank Srikanta, Amitabh, Satheesh and Rajarao, who are the team members of the Oshadhi project, provided me with suggestions and constructive criticism which served to streamline ideas. I would like to thank my other lab mates Vikram and Maya for their help and enjoyable company. I would also like to thank the former members of the Oshadhi project Vidya, Mouli and Manjusha for creating the initial environment to this project.

I take this opportunity to thank my course advisor, *Dr. C. E. Veni Madhavan*, who helped me in taking several important academic decisions.

Cheers to Murty, Krishna, Siddha, Srikanth, Harinadh and Swamy who have made my stay in IISc a memorable one. Thanks to Giri, Anju and Srinu for their encouragement, understanding and guidance which has always helped me in taking right decisions.

Finally I would like to thank my parents, sister, brother-in-law and brothers for their love and encouragement.

Abstract

Spatial data management forms a major component in a biological information management system. The spatial data types along with their spatial operations are needed to model the locational information associated with various biological objects in the system like the distribution of various species in different geographic locations, the location of herbariums, the boundary of a forest or state etc. Also, compound spatial data-types are useful to represent spatially related collections of objects like the division of a country into a set of bio-geographic provinces. The efficient retrieval and processing of such spatial information greatly influences the overall performance of the system. This necessitates support for multi-dimensional indexing strategies.

In this project, we discuss the design and implementation issues of spatial data management in Oshadhi, an object oriented biological information system under development at Indian Institute of Science. Issues such as the spatial data types provided in Oshadhi along with their operations are discussed. Apart from the management of the spatial data, Oshadhi handles various queries on the spatial relationships of these data. Different access methods such as Hilbert R-tree and R*-tree are provided in Oshadhi to speed up the spatial queries.

Chapter 1

Introduction

1.1 Motivation

India is biologically rich with enormous number of plant species spread all over the country. The rich economic potential and use of these plant species make it essential for their *bio-diversity* to be conserved. The spatial component of bio-diversity data consist of information about the distribution of various species in different geographic locations, location of herbariums and botanical gardens, the environmental conditions, political and physical boundaries etc., and are important for bio-conservation [Vid95]. The queries on these data include point query (find all objects that contain a given search point), region query (find all objects that overlap a given search region) and query over different collections of objects (join of sets of spatial objects). The limited set of data-types viz., integer, float, character and date offered by traditional database systems makes the modeling of these real world spatial applications extremely difficult. Hence there is a requirement for a full-fledged database management system which supports these multi-dimensional spatial objects as first class objects and provides efficient query and retrieval facilities over these data.

Having decided to implement a biological information system, we have to model the spatial data and design efficient access methods. The basic object oriented spatial data model has already been implemented in Oshadhi-1 by Vidya [Vid95]. Due to the limitations that will be explained in chapter 2, we need to extend that model to have much richer functionality. The spatial model has to handle both single objects as well as spatially related collections of objects. The operations on these collections have to guarantee integrity of data (for example, in a map describing the boundaries of the states of India, if the boundary of one state changes, it should lead to changes in the boundaries of some of its neighboring states too). To speed up the point and region queries we need fast multi-dimensional access methods. These access methods have to index both multi-dimensional point data as well as objects which have extensions in multi-dimensional space, hence R-tree [Gut84] based index structures are well suited [B$^+$96]. We have chosen Hilbert R-tree [KF94] index among the variants of R-tree based methods, because of the reasons that will be explained later in chapter 3. All these issues are discussed in detail in the remainder of the thesis.

1.2 The Oshadhi System

The goal of Oshadhi system is to design and prototype a new database system for effectively supporting the management of bio-diversity data. The development of the Oshadhi system has taken place in different phases. In the first phase an initial prototype (Oshadhi-1) was developed and some advanced features like SQL like query language are added during the following phases of the project.

This project extends the data model provided in Oshadhi by offering some primitive spatial data types along with efficient implementation of the queries on their relationships and some compound spatial data types to represent spatially related collections of objects. It also provides spatial access methods viz., Hilbert R-tree and R*-tree to speed up the processing of spatial queries.

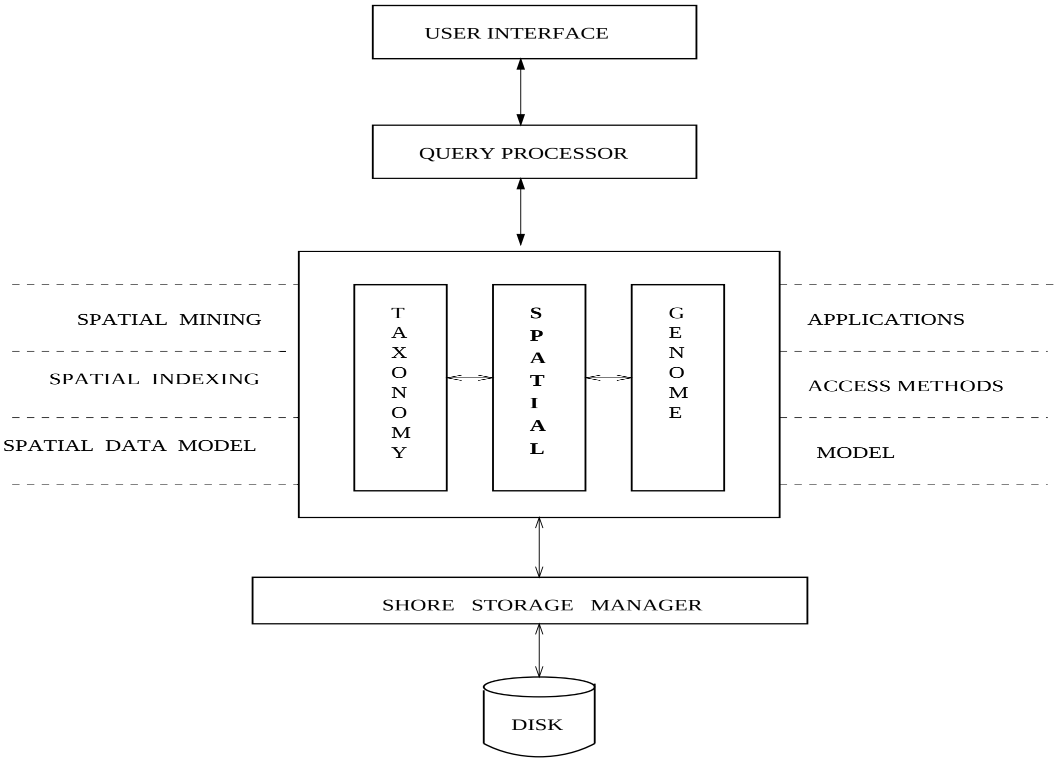


Figure 1.1: Oshadhi system architecture

1.2.1 The Oshadhi Architecture

Oshadhi has been built on top of an object-oriented micro kernel Shore (Scalable Heterogeneous Object REpository) [C+94]. Shore provides the basic database features like transactions, recovery etc. Oshadhi mainly consists of 5 modules. They are *spatial, taxonomy*, *genome*, *query processor* and *user interface* as shown in Figure 1.1. The spatial, taxonomy and genome modules are responsible for implementing the corresponding Oshadhi data model, providing operations on that data, supporting efficient access methods and useful applications like mining. For example, spatial module is responsible for implementing spatial data model, efficient spatial access methods like Hilbert R-tree and applications like spatial mining. These modules may interact with each other and with the query processor. Query processor does the processing and optimization of the queries. End user can either use canned queries or issue custom queries through the user interface.

1.3 Organization of the Report

The rest of the report is organized as follows. Spatial data types and associated operations are explained in chapter 2. Different spatial access methods to efficiently retrieve the spatial data are discussed in chapter 3. Features of the underlying micro-kernel Shore are described in chapter 4. Chapter 5 discusses some of the implementation details of the system. We present the current status and possible future extensions in chapter 6 and finally conclude in chapter 7.

Chapter 2

Modeling of Spatial Data in Oshadhi

In this chapter, we look at the modeling of spatial data in Oshadhi. The simple and compound data types offered to support spatial data are discussed in detail. We will also discuss the relationships among these data types that are important for bio-diversity applications.

2.1 The Object Model

The analysis and design of the spatial data management in Oshadhi was done by Vidya [Vid95], a former student who worked on this project. She had collected the details regarding the types of data that need to be maintained in the database, their constraints and relationships. She had developed an object model for the spatial data which involved various stages such as identifying the classes, their associations, attributes and operations, and finally simplifying the classes using inheritance. Figure 2.1 shows the Rumbaugh notation [R$^+$91] of the object model of the geo-spatial data observed in biological applications. The containment and inheritance relationships between the various entities country, state, district etc., are shown. The locational aspects of these entities may need to be represented (for example, a country or a continent can be represented by its boundary and a river by the path it flows). Also, these entities may contain a collection of similar entities (for example, a country contains a collection of states or a collection

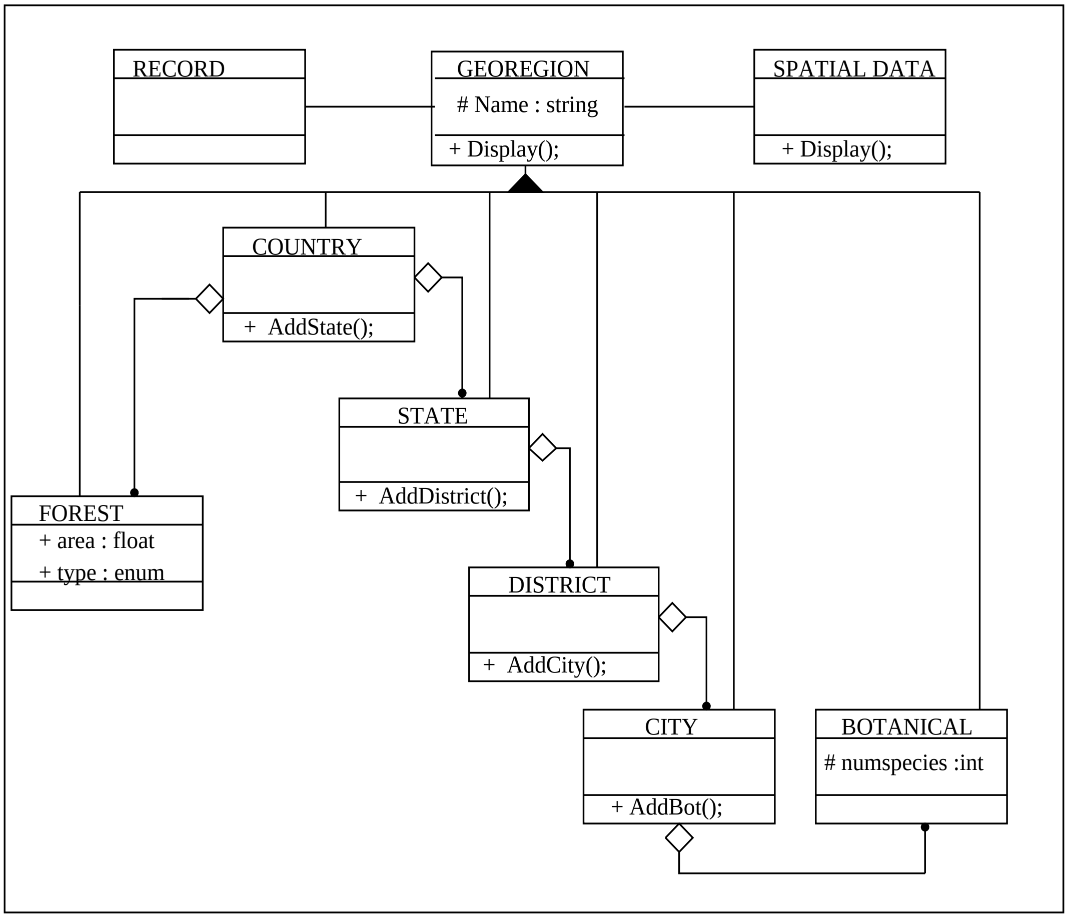


Figure 2.1: Object Model of the geo-spatial data observed in biological applications.

of rivers). Queries over this data may be on the spatial or non-spatial relationships of these entities. The spatial queries of interest will be like list all the states adjacent to Karnataka, select all the states in India where river Ganga flows etc. An important issue is to efficiently represent the locational aspects of these entities and the relationships among them.

2.2 Spatial Data Types and Relationships

Oshadhi offers a set of generic data types with some primitive data types to represent single objects like country, state, river etc., and some compound data types to represent spatially related collection of objects like the set of countries contained in a continent, the set of rivers flowing through a country etc. It also provides efficient implementation of the operators to query on the spatial relationships of interest among the entities.

2.2.1 Primitive Data Types

For modeling single objects, three primitive data types viz., point, line, and polygon as suggested in [Güt94] are offered. Figure 2.2 shows the three abstractions for single objects.

- A *point* represents (the geometric aspect of) an object for which only its location in space, but not its extent, is relevant. For example, IISc campus may be modeled as a point in a model describing a large scale map, say India.
- A *line* (or a curve in space, usually represented by a poly-line) is the basic abstraction for facilities moving through the space, or connections in space (for example, roads, rivers, cables for phone, electricity).
- A *polygon* is the abstraction for something having an extent in 2-D space. A polygon may have holes in it and may also consist of several disjoint pieces (for example, a country, a lake, or a park).

2.2.2 Compound Data Types

For modeling spatially related collections of objects two compound data types, layer and network, are offered as suggested in [Güt94]. Figure 2.3 shows these two compound data types.

- A *layer* can be viewed as a set of polygon objects that are required to be disjoint. The adjacency relationship is of particular interest, that is, there are often pairs of polygon objects with a common boundary. In Oshadhi, layers can be used to represent thematic maps (for example, map of Western Ghats with all districts).

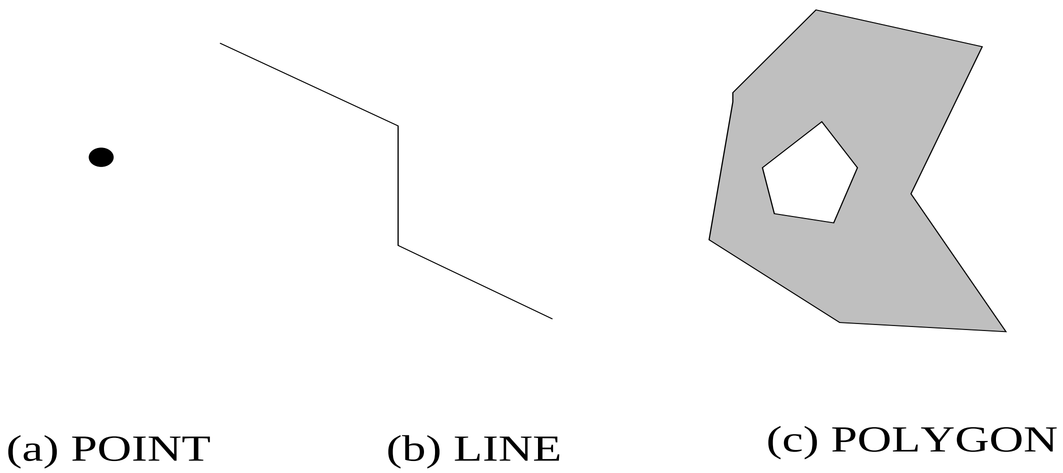


Figure 2.2: Three basic abstractions to represent single spatial objects

- A *network* can be viewed as a set of line objects. Networks are ubiquitous in geography (for example, highways, rivers, public transport, or power supply lines) and hence can be used in Oshadhi to represent collection of rivers, roads etc.

These primitive and compound spatial data types are important for representing spatial objects in bio-diversity applications. The Rumbaugh [R+91] notation of these data types are shown in Figure 2.4. We can observe that the Line and Polygon are collections of Points. Also, Layer and Network are specialized collections of Polygon and Line respectively. Now we will look at the different spatial relationships among these data types.

2.2.3 Spatial Relationships

Spatial relationships are the most important among the operations offered by spatial algebras [Güt89]. The following is the classification of spatial relationships [M.89].

- *Topological relationships*, such as *adjacent, inside*, and *disjoint* are invariant under topological transformations like translation, scaling, and rotation.

- *Direction relationships*, for example *above, below*, or *north-of, southwest-of*.

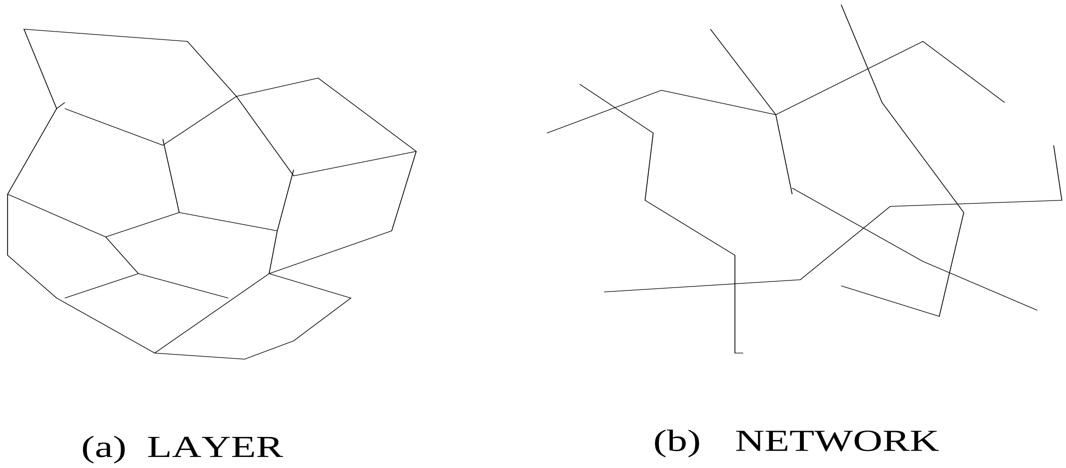


Figure 2.3: Two compound spatial data types to represent spatial collections

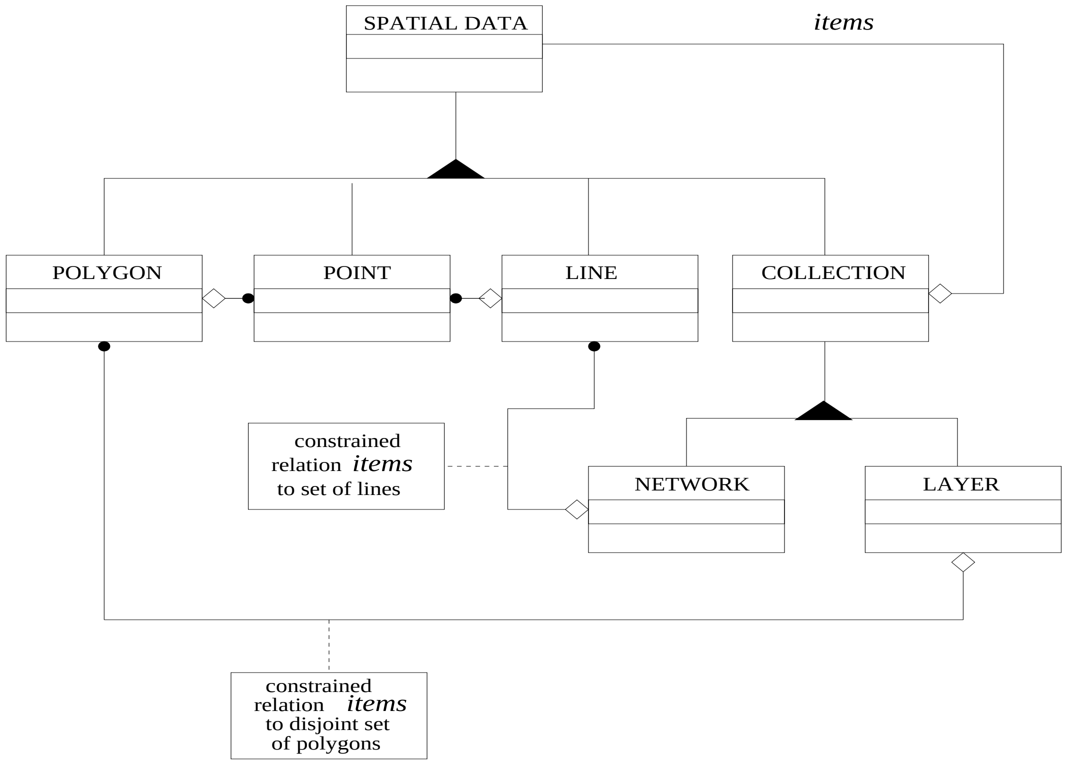


Figure 2.4: Object Model of Oshadhi Spatial Data Types

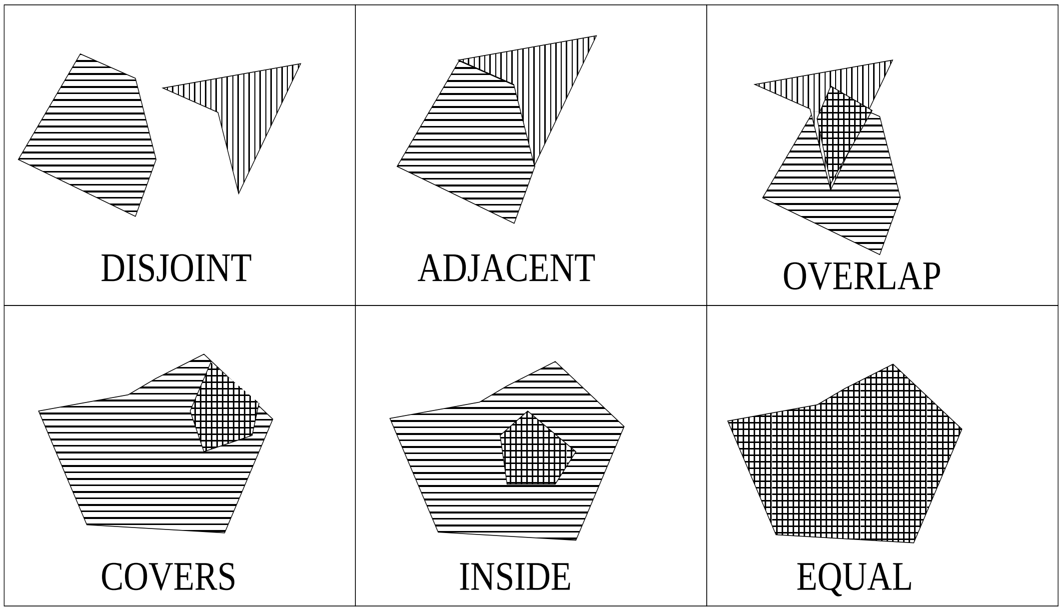


Figure 2.5: Spatial Relationships

- *Metric relationships*, for example, "*distance* < 100".

Among the relationships mentioned above, according to [Güt94], there are mainly six different relationships, called *disjoint, in, touch, equal, cover* and *overlap* are shown in Figure 2.5, which are important for geo-spatial applications. All these relationships are being supported in Oshadhi. The basic spatial data model provided in Oshadhi-1 by [Vid95] offers only a subset of features stated above. So, it has become essential for us to extend that model to add these additional features.

In this chapter we have seen the modeling of the spatial data in Oshadhi. The support for spatial data by offering primitive and compound data types are discussed. The important relationships among these data types are also identified. This completes modeling phase and in the next chapter, we will look at the spatial access methods to speed up the queries over these data.

Chapter 3

Spatial Access Methods

Access methods are a key feature of every database system. They help in improving the performance by retrying a subset of the data in the database. We next discuss the access methods that are part of the Oshadhi system for handling spatial queries.

3.1 Introduction

In order to handle spatial data efficiently, a spatial database system needs an index mechanism that will help it retrieve data items quickly according to their spatial locations. However, traditional indexing methods are not suited in multi-dimensional spaces as all operations on such data are one-dimensional. So, structures using one-dimensional ordering of key values such as B-trees do not work [SK88]. Structures based on exact matching of objects such as hash tables are also not useful because range searches are required. [SK88] discusses the three basic schemes for providing multi-dimensional indexing. They are *clipping, transformation* and *overlapping region* schemes. Among them the overlapping region schemes are widely used [SK88].

In *Overlapping Region Schemes*, each d-dimensional object is represented by its minimal bounding box. The region of a node is the minimal bounding box of all the rectangles belonging to the node. These schemes allow data nodes to have overlapping regions. The R-tree family of structures, well known in these schemes use B^+-tree [Com79] as the underlying point access method. The advantage of the overlapping schemes is that storage utilization depends only on the underlying point access method, since every rectangle is uniquely represented in the structure. Thus these structures inherits the guarantee of at least 50% storage utilization from B^+-tree. Another nice property is that d-dimensional points and d-dimensional rectangles can be organized together in one structure. Because of these advantages of overlapping methods, R-tree family of index structures are our choice.

3.2 R-Tree

The R-tree is a balanced tree generalizing the B^+-Tree [Com79] using the technique of overlapping regions. Leaf nodes in the R-tree contain index record entries of the form

(B, id)

where B is an n-dimensional minimal bounding rectangle and 'id' refers to the object in the database. Intermediate nodes of the tree will have entries of the form

(R,p)

where p is a pointer referring to a child and R is the minimal bounding rectangle of all rectangles in the corresponding child. The minimum number of entries in any node other than the root can be fixed at some constant m to achieve the desired space utilization. Figure 3.1 illustrates the structure of an R-tree and the partition of the corresponding data space. As demonstrated, the regions S_2 and S_3 have an overlap. Some important operations on R-tree are,

- *Search*: search descends the tree from the root in a manner similar to a B^+-tree. However, more than one subtree under a node visited, may need to be searched. Hence it is not possible to guarantee good worst-case performance.

- *Insertion*: Inserting index records for new data tuples is similar to that of a B^+-tree in that the new index records are added to the leaves. Since there are overlapping entries in the non-leaf nodes, the entry which causes *minimum expansion of the area*

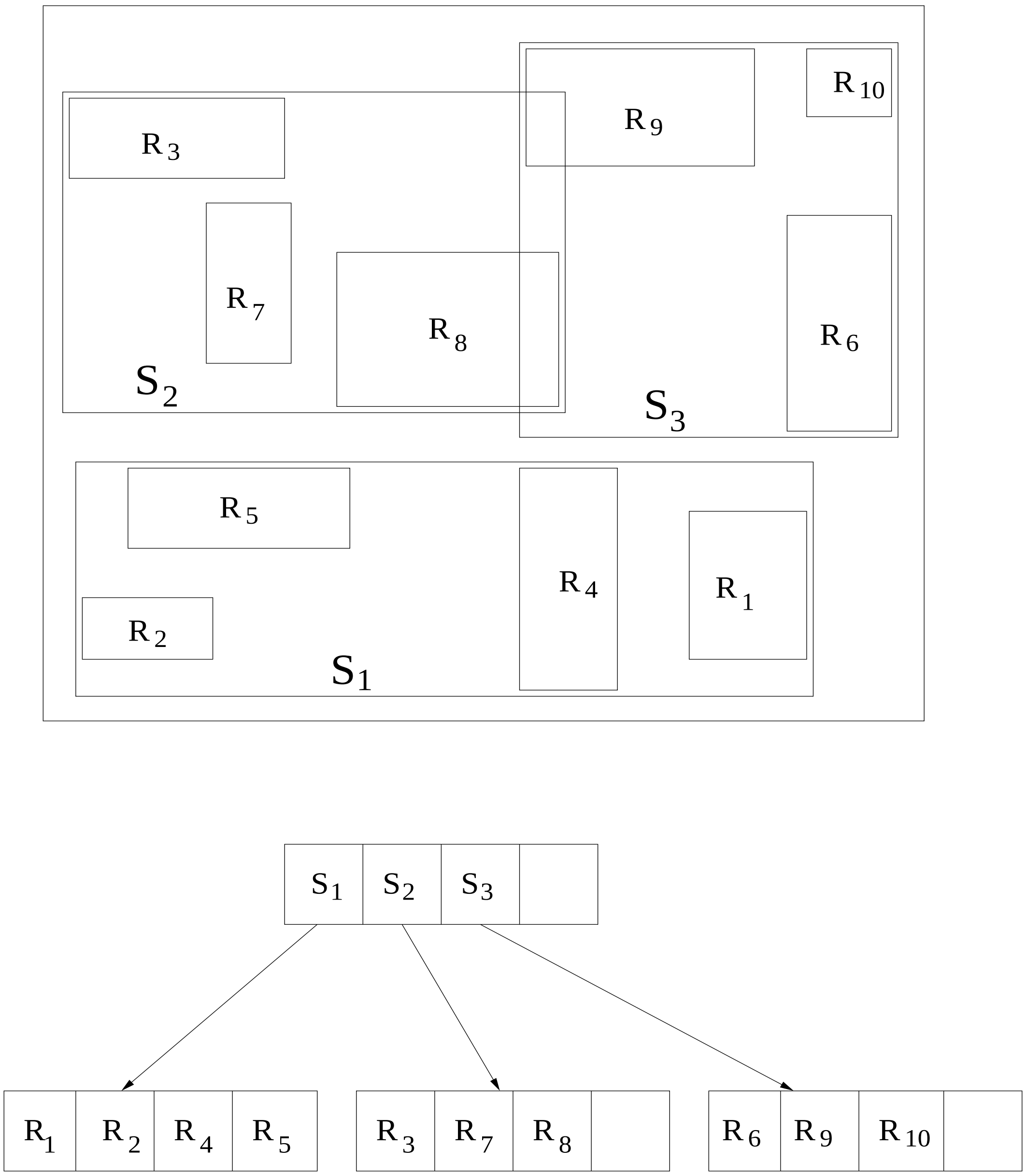


Figure 3.1: The file structure of R-tree.

of its bounding box with insertion, is selected as the target. One can have different policies here. Nodes that overflow are split which may propagate up to the root.

- *Deletion*: Deleting an index record from the tree requires searching for the entry, as explained above, then removing it from the leaf, and propagating the changes upward. Deletion may cause the nodes to underflow. This requires a different treatment compared to that of the B^+-tree, where the orphaned nodes are adjusted among the siblings since there is an ordering among the entries in B^+-tree which guarantees good clustering of entries. Here there is no such ordering for spatial objects. So the orphaned nodes are re-inserted to get better spatial clustering.

R-Tree is only the start of the overlapping structures and there were many variants of R-Tree proposed, viz., R^*-tree [BKSS90], Hilbert R-tree [KF94], X-tree [B^+96] etc. They differ from R-tree mainly in their policy for overflow and underflow treatment. Among them, Hilbert R-tree is our choice. Because, with the deferred splitting policy provided in Hilbert R-tree, one can achieve desired space utilization of up to 100%.

3.3 Hilbert R-tree

Hilbert R-tree is a variant of R-tree proposed by [KF94]. The heart of the idea is to facilitate the deferred splitting approach in R-tree. This is done by proposing an ordering on the R-tree nodes. This ordering has to be 'good', in the sense that it should group 'similar' data rectangles together, to minimize the area and perimeter of the resulting minimum bounding rectangles (MBRs).

They proposed to use Hilbert space filling curve to impose a linear ordering on the data rectangles. A space filling curve visits all the points in a k-dimensional grid exactly once and never crosses itself. [FR89] shown experimentally that the Hilbert curve achieves the best clustering among the other space filling curves. The path of a space filling curve imposes a linear ordering on the grid points. The Hilbert value of a rectangle is defined as the Hilbert value of its center. Given the ordering, every node has a well-defined set of

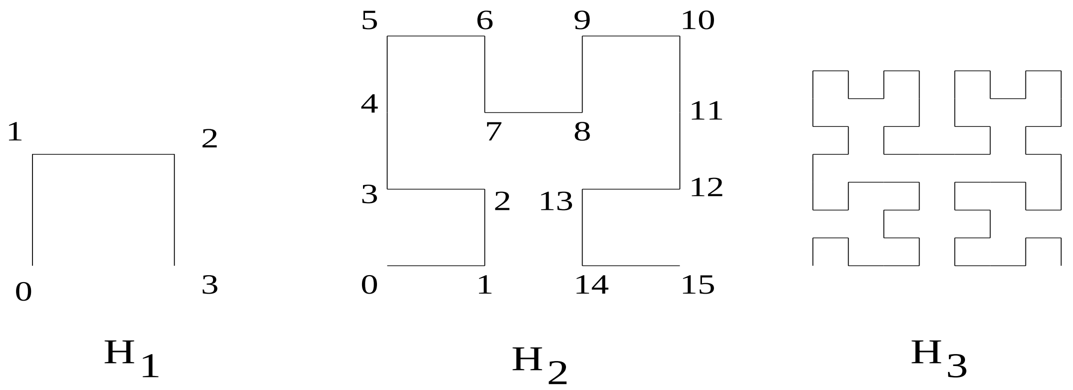


Figure 3.2: Hilbert Curves of order 1, 2 and 3.

sibling nodes; thus, we can use deferred splitting. By adjusting the order of the deferred splitting, the Hilbert R-tree can achieve a high space utilization of upto 100%. .

3.3.1 Hilbert Curve

The basic Hilbert curve on a 2×2 grid, denoted by H_1. is shown in Figure 3.2. To derive a curve of order i, each vertex of the basic curve is replaced by the curve of order i - 1, which may be appropriately rotated and/or reflected. Figure 3.2 also shows the Hilbert curves of order 2 and 3. When the order of the curve tends to infinity, the resultant curve is a *fractal*, with a fractal dimension of 2 [KF94]. The Hilbert curve can be generalized for higher dimensionalities. Algorithms to draw the two-dimensional curve of a given order, can be found in [Jag90].

The path of a space filling curve imposes a linear ordering on the grid points. Figure 3.2 shows one such ordering for a 4×4 grid (see curve H_2). For example the point (0,0) on the H_2 curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2.

3.3.2 Description of Hilbert R-tree

The main idea of Hilbert R-tree is to create a tree structure that can

- behave like an R-tree on search

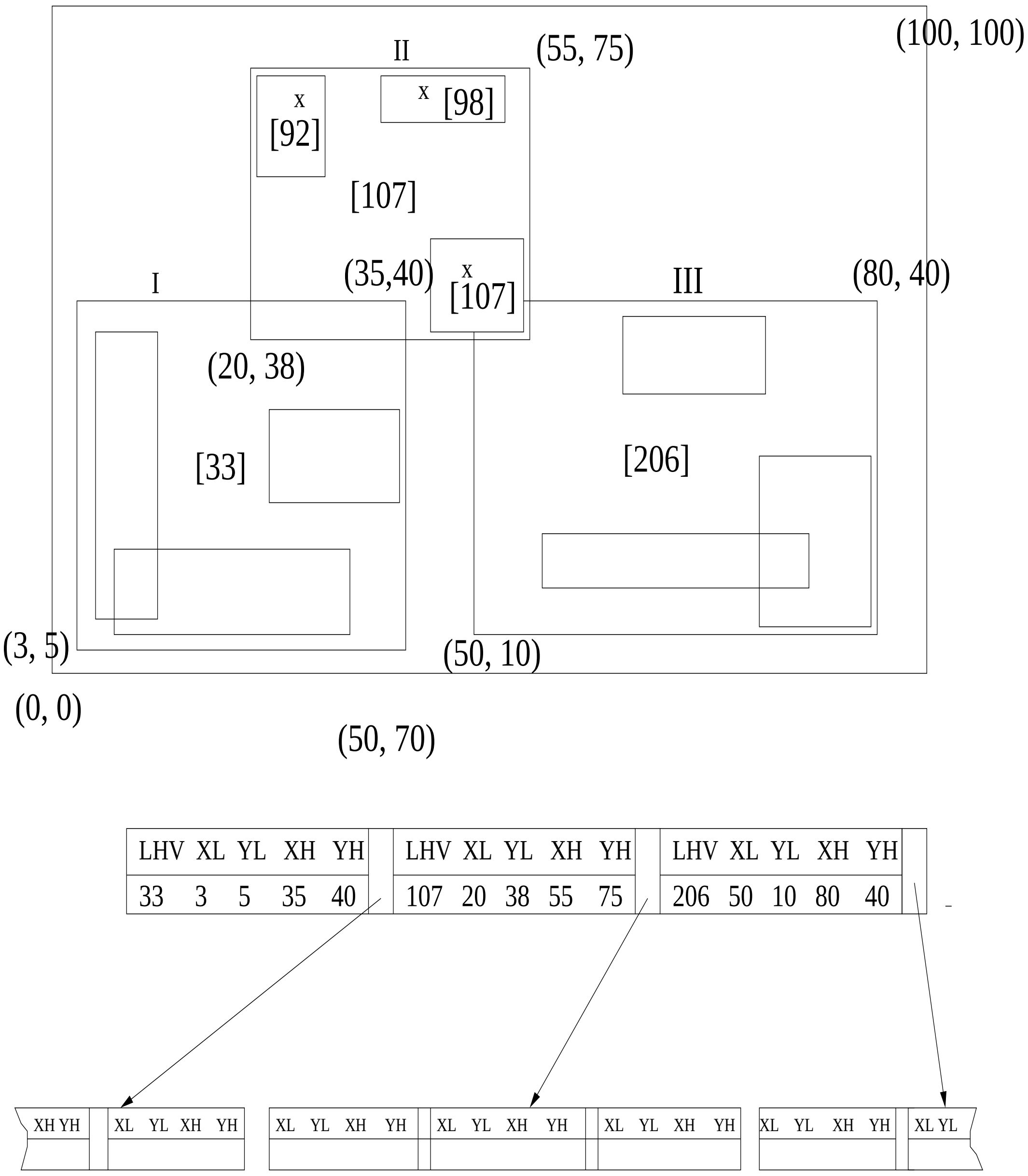


Figure 3.3: An example of the file structure of Hilbert R-tree.

- support deferred splitting on insertion, using the Hilbert value of the inserted data rectangle as the primary key.

These goals can be achieved as follows: for every node n of the tree, we store (a) its Minimum Bounding Rectagle (MBR), and (b) the *Largest Hilbert Value* (LHV) of the data rectangles that belong to the subtree with root n.

Specifically, the Hilbert R-tree has the following structure. A leaf node contains at most C_l entries of the form

$$(R,\ OID)$$

where C_l is the capacity of the leaf, R is the MBR of the real object (x_{low}, x_{high}, y_{low}, y_{high}) and *OID* is a pointer to the object description record. A non-leaf node in the Hilbert R-tree contain at most C_n entries of the form

$$(R,\ ptr,\ LHV)$$

where C_n is the capacity of a non-leaf node, *R* is the MBR that encloses all the children of that node, *ptr* is a pointer to the child node, and *LHV* is the largest Hilbert value among the data rectangles enclosed by R. Note that for the intermediate nodes, we never calculate the Hilbert value. Figure 3.3 illustrates some rectangles, organized in a Hilbert R-tree. Every data rectangle in node I has Hilbert value ≤ 33; everything in node II has Hilbert value greater than 33 and ≤ 107 etc.

A plain R-tree splits a node on overflow, turning 1 node to 2, called as 1-to-2 splitting policy. In Hilbert R-tree they proposed to defer the split, waiting until they turn 2 nodes into 3, known as 2-to-3 split. In general we can have s-to- (s+1) splitting policy, where s is the *order of the splitting policy.* To implement the order-s splitting policy, the overflowing node tries to push some of its entries to one of its s-1 siblings; if all of them are full, then we have an s-to- (s+1) split. The s-1 siblings are referred as *co-operating siblings* of the given node.

Next, we will describe in detail the algorithms for searching, insertion, and overflow handling.

3.3.3 Searching

The searching algorithm is similar to the one used in other R-tree variants. Starting from the root, search descends the tree examining all nodes that intersect the query rectangle. At the leaf level it reports all entries that intersect the query window w as qualified data items.

Algorithm Search (node *Root*, **rect** w) :

S1. *Search non-leaf nodes* :

invoke Search for every entry whose MBR
intersects the query rectangle window w.

S2. *Search the leaf nodes* :

Report all the entries that intersect the query
window w as candidates.

3.3.4 Insertion

To insert a new rectangle r in the Hilbert R-tree, the Hilbert value h of the center of the new rectangle is used as the key. In each level we choose the node with minimum LHV greater than h among the siblings. When a leaf node is reached the rectangle r is inserted in its correct order according to h. After a new rectangle is inserted in a leaf node N, **AdjustTree** is called to fix the MBR and LHV values in upper level nodes.

Algorithm Insert (node *Root*, **rect** r) :

/*insert a new rectangle r in the Hilbert R-tree. h is its Hilbert value. */

I1. *Find the appropriate leaf node* :

Invoke **ChooseLeaf** (r, h) to select a leaf node L in which to place r.

I2. *Insert r in a leaf node* L :

if L has an empty slot, insert r in L in the appropriate place
according to the Hilbert value and return.
if L is full, invoke **HandleOverflow**(L, r),
which will return new leaf if split was inevitable.

I3. *Propagate changes upward* :
form a set that contain L, its cooperating siblings
and the new leaf(if any).
invoke **Adjust Tree(S)**

I4. *Grow Tree Taller* :
if node splits propagation caused the root to split,
create a new root whose children are the resulting nodes.

Algorithm ChooseLeaf (rect r, **int** h) :

/*Returns the leaf node in which to place a new rectangle r. */

C1. *Initialize* :
Set N to be the root node.

C2. *Leaf Check* :
If N is a leaf, return N.

C3. *Choose Subtree* :
If N is non-leaf node, choose the entry (R, ptr, LHV) with
the minimum LHV value greater than h.

C4. *Descend until a leaf is reached* :
set N to the node pointed by ptr and repeat from C2.

Algorithm AdjustTree(set S) :

/* S is a set of nodes that contain the node being updated, its cooperating siblings (if overflow has occurred) and newly created node NN(if split has occurred). The routine ascends from leaf level towards the root, adjusting MBR and LHV of nodes that cover the nodes in S siblings. It propagates splits (if any). */

A1. if reached root level stop.

A2. *Propagate node split upward*

let N_p be the parent node of N.

if N has been split, let NN be the new node. insert NN in N_p in the correct order according to its Hilbert value if there is room. Otherwise, invoke **Handle Overflow(** N_p, **MBR**(NN)).

if N_p is split, let PP be the new node.

A3. *adjust the MBR's and LHV's in the parent level* :

let P be the set of parent nodes for the nodes in S.

Adjust the corresponding MBR's and LHV's appropriately of the nodes in P.

A4. *Move up to next level.*

Let S become the set of parent nodes P, with NN = PP, if N_p was split.

repeat from A1.

3.3.5 Deletion

In Hilbert R-tree we do not need to re-insert orphaned nodes, whenever a father node underflows. Instead, we borrow keys from the siblings or we merge an underflowing node with its siblings. We are able to do so, because the nodes have clear ordering (Largest Hilbert Value LHV); in contrast, in R-trees [Gut84] there is no such concept of sibling node. Notice that, for deletion, we need s cooperating siblings while for insertion we need *s-1*.

Algorithm Delete(r) :

D1. *Find the host leaf* :

Perform an exact match search to find the leaf node L that contain r.

D2. *Delete r* :

Remove r from node L.

D3. *handle underflow* :

if L underflows borrow some entries from s cooperating siblings.

if all the sibling nodes are ready to underflow, merge s+1 to s nodes, adjust the resulting nodes.

D4.*adjust MBR and LHV in parent levels* :

form a set S that contain L and its cooperating siblings(if underflow has occurred).

invoke **AdjustTree(S)**.

3.3.6 Overflow handling

The overflow handling algorithm in the Hilbert R-tree treats the overflowing nodes either by moving some of the entries to one of the s - 1 cooperating siblings or splitting s nodes to s + 1 nodes.

Algorithm HandleOverflow(node N, **rect** r **)** :

/* return the new node if a split occurred */

H1. let E be a set that contain all the entries from N and its s - 1 cooperating siblings.

H2. add r to E.

H3. if at least one of the s - 1 cooperating siblings is not full, distribute E evenly among the s nodes according to the Hilbert value.

H3. if all the s cooperating siblings are full, create a new node NN and distribute E evenly among the s + 1 nodes according to the Hilbert value.

return NN.

3.3.7 Advantages of Hilbert R-tree

The advantages of the Hilbert R-tree can be summarized as follows:

- Since every rectangle is uniquely represented in the structure, the underlying access structure B^+-tree inherits at least 50% storage utilization.

- Multi-dimensional points data as well as multi-dimensional rectangle data can be organized in the same structure. Hence we don't need to use two different structures for each of these data.

- By varying the order of deferred splitting one can achieve the desired space utilization of up to 100%.

These advantages motivated us to choose Hilbert R-tree index as the access method for spatial data in Oshadhi.

This chapter finishes the design phase of the project and in the following chapters we will look at the implementation phase first by looking at the details of the underlying platform and then the exact implementation details of the spatial data model and the spatial access methods.

Chapter 4

The Platform - 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 back-end database server of the Oshadhi database system for storing spatial objects and related information.

Most of the material in this chapter is taken from [C+94].

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 favor of OODBMSs. Some of the reasons are:

1. Many current OODBMSs are restricted to a single language while large scale applications often require multi-lingual 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 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 name-space 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.

4.3 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.

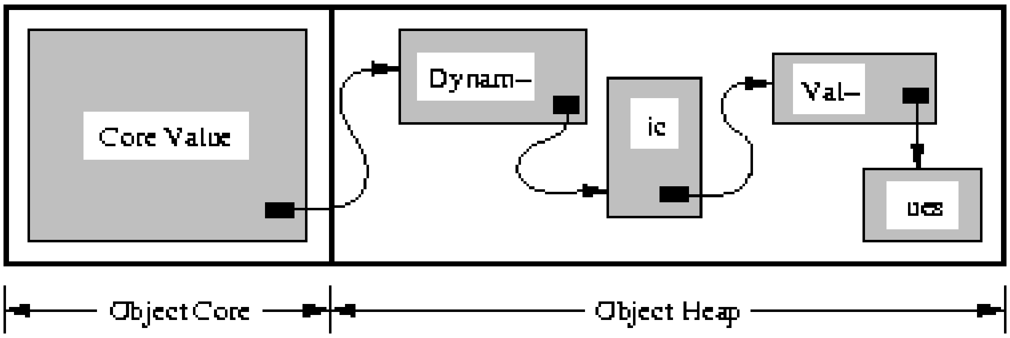


Figure 4.1: Shore Object Structure

4.4 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 name-space. 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.

4.4.1 Shore Object name-space

Shore provides a tree-structured, Unix like name-space in which all persistent objects are reachable either directly or indirectly from a distinguished root directory. The name-space 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 time-stamps. 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.

4.4.2 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 re-linked, Shore provides an NFS file server. An entire subtree of the Shore name-space 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.

4.5 Object Oriented Database System features

4.5.1 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 [?], 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.

4.5.2 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.

4.5.3 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.

4.6 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.

4.7 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.

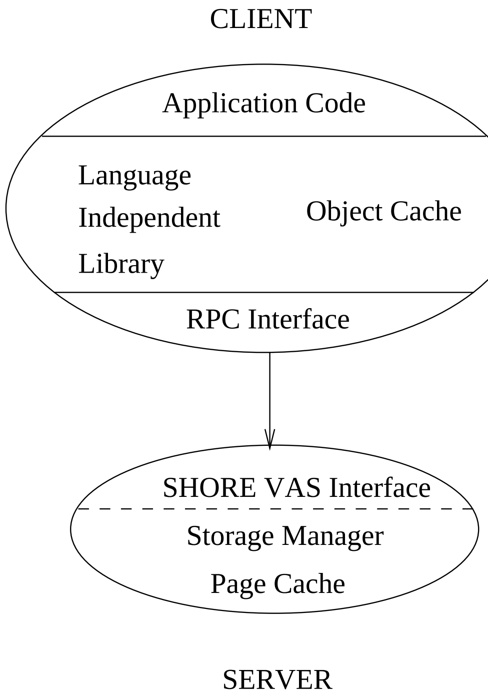


Figure 4.2: Application - Server Interface

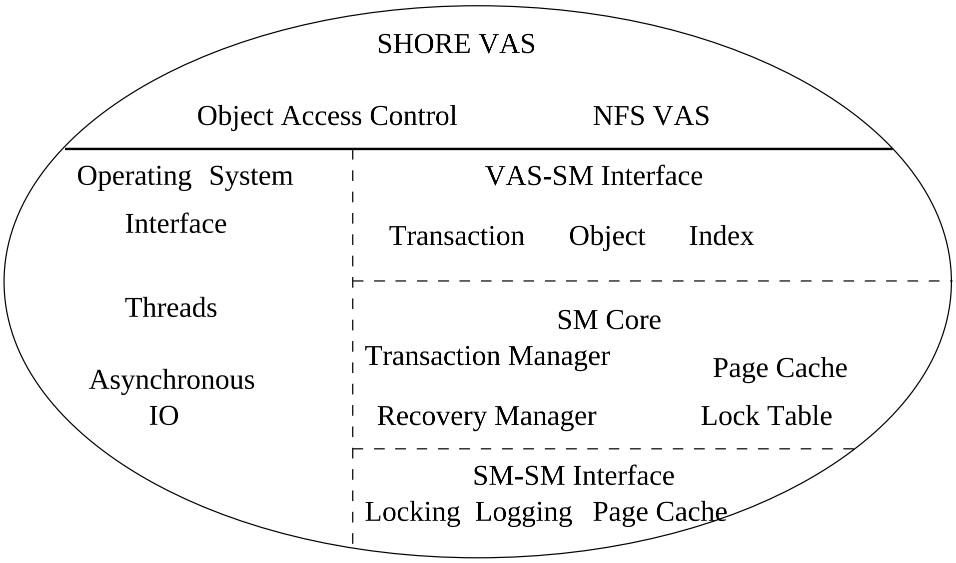
4.7.1 The Language Independent Library

The process of converting object references on disk to main memory addresses is called *swizzling*. When an application attempts to dereference an "UN-swizzled" 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 UN-swizzle all pointers to them. Finally, the LIL is responsible for authenticating the application to the server using the Kerberos authentication system.

4.7.2 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.



∠(90)

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 name-space. 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 multi-threading, asynchronous I/O and inter-process communication.

4.7.3 The Shore Storage Manager

As a database storage engine, the SSM provides the atomicity, consistency, isolation, and durability (often referred to as ACID) properties associated with transactions.

4.7.3.1 Transaction Facilities

- **Transactions**

  A transaction is an atomic set of operations on records, files, and indexes. Updates made by committed transactions are guaranteed to be reflected on stable storage, even in the event of software or processor failure. Updates made by aborted transactions are rolled back and are not reflected on stable storage. Although nested transactions are not provided at this time, the notion of save-points are. Save-points delineate a set of operations that can be rolled back without rolling back the entire transaction.

- **Concurrency Control**

  Transactions are also a unit of isolation. Locking is provided by the SSM as a way to keep a transaction from seeing the effect of another, uncommitted transaction. Normally, locks are implicitly acquired by operations that access or modify persistent data structures, but the SSM interface also provides methods for locks to be acquired explicitly. The SSM uses a standard hierarchical, two-phase locking protocol. For a file, the hierarchy is volume, file, page, record (slot number); for an index, it is volume, index, key-value. Chained transactions are also provided. Chaining involves committing a transaction, retaining its locks, starting a new transaction and giving the locks to the new transaction.

4.7.3.2 Crash Recovery Facilities

The crash recovery facilities of the SSM consist of logging, check-pointing, and recovery management.

- **Logging**
  Updates performed by transactions are logged so that they can be rolled back (in the event of a transaction abort) or restored (in the event of a crash). Both the old and new values of an updated location are logged (so-called "undo/redo logging". This technique supports buffer management policies with the properties called "steal" (a dirty page can be written to disk at any time) and "no force" (dirty page need not be forced to disk at commit time).

- **Check Pointing**
  Checkpoints are taken periodically by the SSM in order to free log space and shorten recovery time. Checkpoints are "fuzzy" and do not require the system to pause while they are completing.

- **Recovery**
  The SSM recovers from software, operating system, and CPU failure by restoring updates made by committed transactions and rolling back all updates by transactions that did not commit by the time of the crash.

4.7.3.3 Thread Management

Providing the facilities to implement a multi-threaded server capable of managing multiple transactions is one of the distinguishing features of the SSM. Some extensions to the thread package are also implemented in SSM. These are not formally part of the thread package, but are found useful enough in building the SSM and the Shore VAS.

- **Latches**

  Latches are a read/write synchronization mechanism for threads, as opposed to locks which are used for synchronizing transactions. Latches are much lighter weight than locks, have no symbolic names, and have no deadlock detection.

- **Thread-Protected Hash Tables**

  It is a hash table in which each entry is protected by a latch and uses them to enforce a protocol in which multiple threads have consistent and concurrent access to the shared resources. When an entry needs to be added and the table is full, and old entry is removed based on the LRU policy. For instance the Shore buffer manager uses such table to manage buffer control blocks.

4.7.3.4 Error Handling

Errors in the SSM (and the rest of Shore) are indicated by an unsigned integer encapsulated in a class that includes stack traces and other debugging aids. The class is w_rc_t (commonly typdefed to rc_t). It is the return type for most SSM methods. When linked with a debugging version of the SSM (compiled with #define DEBUG), the destructor of an w_rc_t object verifies that it was checked at least once. If not, the destructor calls w_rc_t::error_not_checked which prints a warning message. An w_rc_t is considered checked when any of its methods that read/examine the error code are called, including the assignment operator. Therefore, simply returning an w_rc_t (which involves an assignment) is considered checking it. Of course, the newly assigned w_rc_t is considered unchecked.

4.7.3.5 Communication and RPC facilities

Clients (applications) need a way to communicate with servers. The publicly available Sun RPC package is modified to operate with the SSM's thread package. The Shore value-added server uses this package. One of the goals of Shore is that clients only need to communicate with a server on their local machine. The SSM in the local server cooperates with other remote SSMs to cache data locally. Work on this capability is not complete. Eventually, the communication system we built to support this will also be available to VAS writers.

4.7.3.6 Miscellaneous Facilities

- **Statistics**

  The SSM keeps many statistics on its operation such as lock request and page I/O counts. A utility for formatted printing of these statistics is also provided.

- **Sorting**

  The SSM has sorting facilities, but they are still under development. The sorting type can be of any basic type or spatial type (ordered according to the Hilbert value of the spatial object).

- **Data Vectors**

  A data vector is an array of (pointer,length) indicating blocks of data in memory. The array can be arbitrarily long, and methods are provided to comparing and copying data. Data vectors reduce the number of parameters in many SSM methods by combining pointer and length information and allow the code to avoid re-copying data by allowing a sequence of chunks to be treated like one contiguous buffer.

4.8 Conclusions

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

Implementation Details

The implementation of the spatial module is discussed in this chapter. As seen in the architecture of Oshadhi shown in chapter 1, Shore server is the lower most layer of the system. The object model discussed earlier in chapter 2 is implemented using Shore's support for persistent objects. The Hilbert R-tree index described in chapter 3 is then implemented inside the Shore storage manager (SSM). Bulk-loading and iterator facilities to the index are also provided.

5.1 Object Model implementation

The first step in the implementation of a database system is the definition of the *schema*. The Object Definition Language (ODL) [Cat93] is used for this purpose. The major steps involved are:

- Identify the "class" declarations
- Organize groups of related classes into modules
- Identify data that need to be exported to other modules
- Declare operations of the classes

Each class declared for storing persistent data has a corresponding ODL class declaration. All the classes that were identified in the object model are now converted into their corresponding ODL *classes*. Each class is composed of declarations of *properties* and *operations.* The properties model the state of the object while the operations model its behavior. The properties constitute *attribute* declarations, *relationship and ref* declarations which are used to represent the data members and the associations. The operations of the interface constitute the return type, the operation name, and a set of arguments. The arguments may be of *in, out* or *inout* type. The *in* arguments are not modified inside the operation while the *out* arguments are used for data that are changed as a result of the operation. The *inout* operations may be used inplace of either the *in* or the *out* operation. An argument may also be *transient* (not persistent), in which case it is preceded by the keyword *external*. Similarly, the attributes and operations defined earlier in the model are converted into their corresponding ODL declarations. The sample ODL declaration for the *Line* class is shown below. The *Line* is represented as a collection of *points* and has a set of methods. Appendix A has the ODL declarations for the other important classes.

```
class Line
{
    public :
    attribute sequence < Point > points;

    const boolean isPointOnLine(in  Point  p ) ;
    const boolean isLineIntersectingLine( in  Line  line );
    const Point  intersectsLine( in  Line  line );
    const boolean isLineIntersectingPolygon( in  Polygon  boundary );
    const boolean isLineMeetsLine( in  Line  line );
    const boolean isLineMeetsPolygon( in  Polygon  boundary );
    const boolean isLineInsidePolygon( in  Polygon  boundary );
};
```

A logical group of related interfaces form a *module*. Interfaces within the same module share data variables and constant declarations. Sometimes, declarations of one module may need to be accessed in some other module. This is done by *exporting* those identifiers in the module in which they are originally declared and then *importing* them in the other module which needs to access them.

The next step after declaring the interfaces in ODL is to generate the appropriate language binding, in our case C++. The ODL compiler creates a persistent meta-data (data about the data) object in the Oshadhi database corresponding to each module declaration. This meta-data object is useful while evaluating the queries over the objects of these classes. More details on this meta-data can be found in [Kon00]. In addition to creating these module and type objects, the ODL compiler along with SDL compiler generates the C++ language binding in the form of header files containing C++ *class declarations* for the corresponding ODL declarations.

After generating the header files, we next implement the definitions of the methods corresponding to the *operation declarations* of interfaces declared earlier in ODL. This is similar to definitions of member functions in C++, with some differences in the manipulation of persistent objects, as discussed in the following section. The overall flow of the implementation follows the basic stages as shown in Figure 5.1.

5.2 Data Manipulation

Data manipulation includes the following: insertion of new information into the database, retrieval of information from the database, modification of data stored in the database and deletion of information from the database. These features are implemented using C++ as the database programming language or the Object Query Language (OQL)[Cat93]. More details on OQL is out of scope of this report and can be found in [Kon00].

Object Creation

Objects are created using an overloaded form of the *new* operator. This form also allows the programmer to specify where the newly allocated object is to be clustered. Thus, the

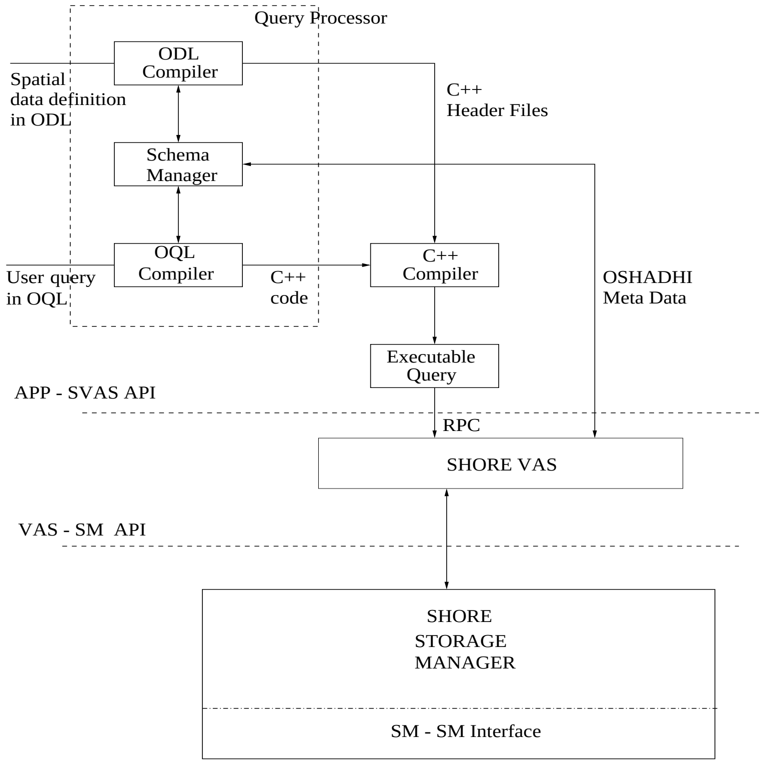


Figure 5.1: The data flow in Oshadhi

new operator can be used to create an object of a class previously declared in ODL. The object may be registered, in which case it is accessible through a fully qualified path in the Shore file system, or anonymous wherein it belongs to a pool and cannot be accessed individually.

Object Deletion

The deletion of previously created objects is done through calls to *Ref<T>::unlink()* or *Ref<T>::destroy()* member functions respectively for registered and anonymous objects.

Operations

Operations are defined as in C++. Operations with transient and persistent objects behave entirely consistent within the operational context defined by standard C++. This includes all overloading, argument passing and resolution, and other compile time rules.

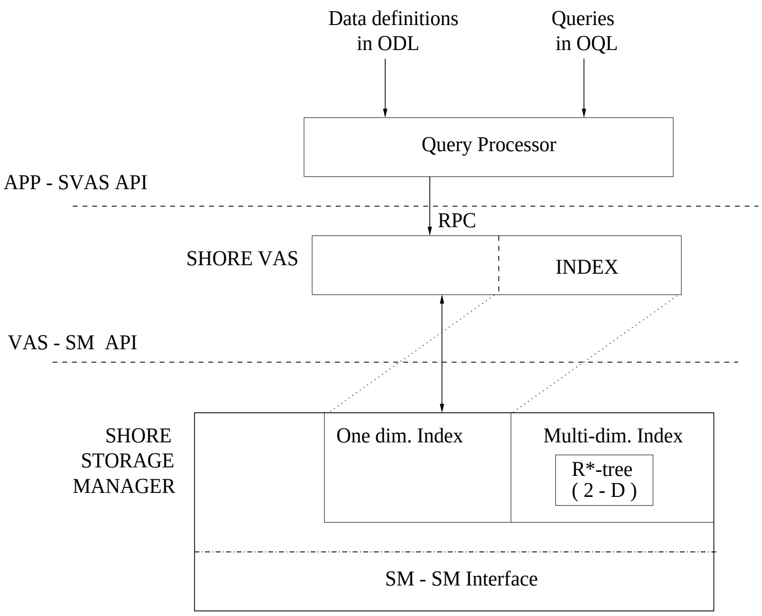


Figure 5.2: System Before Implementation

The list of operations supported in Oshadhi along with examples are discussed in Appendix B.

5.3 Access Methods

The implementation of the access methods for efficient query processing constitutes an important optimization step in efficient data retrieval. As we have already stated that we are building Oshadhi on top of Shore, first we will describe the facilities that Shore supports to implement the indexes.

The underlying micro-kernel Shore provides two different interfaces: one for the programmers who want to write application programs and another, shown as "VAS-SM API" in Figure 5.1, for the developers who may want to add more functionality to Shore by writing "value-added" servers (VAS). Along with these interfaces Shore provides one more interface shown as SM - SM Interface in Figure 5.1. Only modules inside the storage manager can access this interface.

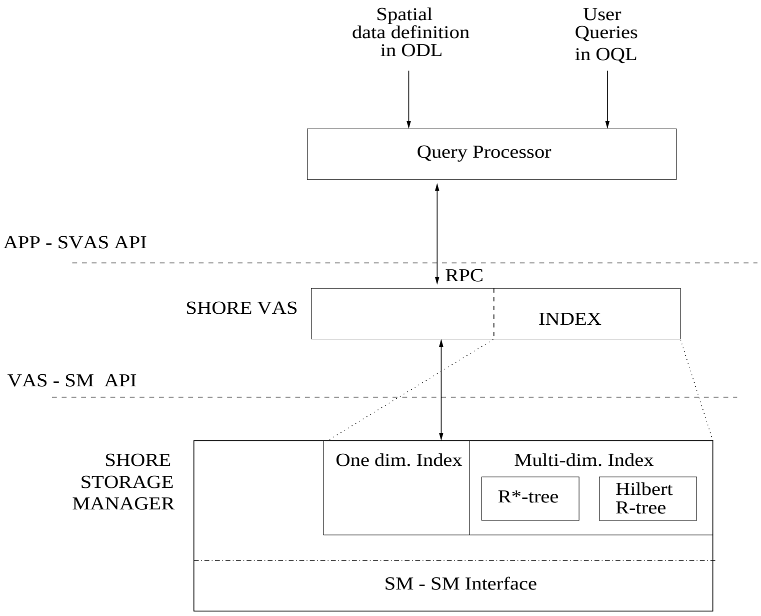


Figure 5.3: System After Implementation

5.3.1 Hilbert R-tree

We have implemented basic routines of the Hilbert R-tree, bulk loading routines and iterator routines for scanning all the objects in the tree in the given range. The Implementation is done is Shore storage manager because of the following reasons: The Shore storage manager interface exported to the VAS developer does not provide control over individual pages. But for efficiency purposes, Hilbert R-tree requires control over these individual pages. Hence its implementation is done at storage manager level using the SM-SM Interface shown in Figure 5.2.

5.3.2 Modifications to Shore built-in R*-tree

: The Shore built-in R*-tree is exported only to the VAS developers but not to the application programmers. So, application programmers cannot make use of this multi-dimensional index structure in their applications. Moreover, the built-in R*-tree functionality is limited to only 2 dimensions as shown in Figure 5.2. Since Oshadhi applications have to deal with spatial objects and multi-attribute queries, it is essential to export the

R^*-tree to the application programmers and extend it to support for any number of dimensions. Because of the above requirements, we have modified the code of R^*-tree to work for any dimensions.

5.3.3 Exporting Multi-Dimensional Indexes to Applications

As already explained above, the multi-dimensional access methods like R^*-tree and Hilbert R-tree in Shore are exported only to VAS developers but not to the applications as shown in Figure 5.2. We exported these indexing routines to the application programmers and also provided a common interface to both the one-dimensional indexes (Unique B-tree, Non-Unique B-tree, Hashing etc.,) and multi-dimensional indexes (Hilbert R-tree and R^*-tree etc.,) as shown in Figure 5.3. This required modifying the code of Shore VAS by adding some more functions to it.

In this chapter we have discussed the implementation details of the spatial data model and indexing mechanisms in Oshadhi. We have also stated the choices and the decisions made regarding the level (layer) at which all these implementations were done. Next chapter presents the current status of the project and the possible future extensions.

Chapter 6

Current Status and Future Directions

The Spatial Data Model and its corresponding member functions are implemented. The Hilbert R-tree indexing has been implemented in the Shore storage manager. Shore built-in R*-tree has been modified to suit the needs of Oshadhi and exported to the application level. To ease the work of query-processor, a uniform interface has been provided to all the spatial and non-spatial indexing mechanisms available in Shore as shown in Figure 5.3. Preliminary testing on Hilbert R-tree implementation is done and extensive tests are to be done to compare the performance of the R*-tree and the Hilbert R-tree. The entire implementation is done in C++ on a SPARC/Solaris-2.5 platform and is about 7,000 lines of code.

In future, Spatial join implementation using Hilbert R-tree can be done to improve the join performance. Applications like spatial mining can be implemented to add more functionality to Oshadhi. Also the spatial data types can be extended to support images also.

Chapter 7

Conclusions

Spatial data management is a major component in any biological information system. In this project, we have addressed how Oshadhi, being a biological information management system, supports these applications by providing spatial abstractions like point, line and polygon along with efficient implementation of their operations. Compound data types viz., layer and network are also offered, to represent spatially related collection of objects like states of a country or the road network of a state etc. For the fast retrieval of spatial information, multi-dimensional indexing technique are provided. We have chosen to provide the Hilbert R-tree index because of its clustering properties and its implementation simplicity . With this structure one can achieve the desired utilization of up to 100% by varying the order of the deferred split. Moreover, if the ordering provided by Hilbert curve is 'good', that is, to group similar rectangles together, then the tree will in addition have nodes with small minimum bounding rectangles, and eventually, fast response times. In future, a join algorithm implementation which makes use the existing Hilbert R-tree index can be done, so that the processing of join will also get the benefits of the good clustering properties of the Hilbert R-tree.

Bibliography

[B+96] Stefan Berchtold et al. The x-tree: An index structure for high-dimensional data. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 28–39, Mumbai, 1996.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Scheider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, May 1990.

[C+94] Michael J. Carey et al. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD Conference*, pages 383–394, Minneapolis, USA, 1994.

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

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2), June 1979.

[FR89] C. Faloutsos and S. Rosemen. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGART-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS)*, pages 247–252, 1989.

[Gut84] Antonin Guttman. R-tree: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, 1984.

[Güt89] R.H Güting. Gral: An extensible relational database system for geometric applications. Technical report, Amsterdam, 1989.

[Güt94] Ralf Hartmut Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[Jag90] H.V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*, pages 332–342, 1990.

[KF94] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th Very Large Data Bases Conference*, pages 500–509, Santiago, Chile, 1994.

[Kon00] Satheesh Konidala. Query Processing and Optimization in Oshadhi. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

[M.89] Egenhofer M. A formal definition of binary topological relationships. In *Proceedings of the Third International Conference on the Foundations of Data Organization and Algorithms*, Paris, 1989.

[R+91] J. Rumbaugh et al. *Object Oriented Modeling and Design.* Prentice Hall, 1991.

[SK88] B Seeger and H.P Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proceedings of the 14th International Conference on Very Large DataBases*, pages 360–370, Los Angeles, 1988.

[Vid95] R. Vidya. OSHADHI A Biodiversity Information System. Master's thesis, Inidan Institute of Science, 1995.

Appendix A

ODL Declarations of the Spatial Data Types

```
module SpatialModule
{

    class Point
    {
        public:
        attribute double x,y;
        attribute int refCount;

        void init(in double xx, in double yy);
        const boolean isPointInsidePolygon(in  Polygon boundary );
        const boolean isPointOnLine( in Line line );
        const double distanceFromPointToPoint( in  Point  point );
        const double distanceFromPointToLine( in  Line  line );
        const double distanceFromPointToPolygon( in  Polygon  boundary );
    };
```

```
class Line
{
    public :
    attribute sequence < Point > points;

    const boolean isPointOnLine(in  Point  p ) ;
    const boolean isLineIntersectingLine( in  Line  line );
    const Point  intersectsLine( in  Line  line );
    const boolean isLineIntersectingPolygon( in  Polygon  boundary );
    const boolean isLineMeetsLine( in  Line  line );
    const boolean isLineMeetsPolygon( in  Polygon  boundary );
    const boolean isLineInsidePolygon( in  Polygon  boundary );
    const double distanceFromLineToPoint( in  Point  point );
    const double distanceFromLineToLine( in  Line  line );
    const double distanceFromLineToPolygon( in  Polygon  boundary );
    const double length( );
    void deleteAll();
    const void print() ;
    int addNewPoint(in  Point  p);
};
```

```
class Polygon
{
    public:
    attribute sequence < Point >  points;

    int addNewPoint(in  Point  p);
    const boolean isPointInsidePolygon( in  Point  point );
    const boolean isLineInsidePolygon( in  Line  line );
    const boolean isPolygoninsidePolygon( in  Polygon  boundary );
    const boolean isPolygonIntersectingLine( in  Line  line );
    const Line  PolygonintersectsLine( in  Line  line );
    const boolean isPolygonIntersectingPolygon( in  Polygon  boundary );
    const Polygon  PolygonintersectsPolygon( in  Polygon  boundary );
    const boolean isPolygonMeetingLine( in  Line  line );
    const boolean isPolygonAdjacentToPolygon( in  Polygon  boundary );
    void plus( in  Polygon  boundary, in Pool point_pool );
    void minus( in  Polygon  boundary, in Pool point_pool );
    Line contour() ;
    const double distanceFromPolygonToPoint( in  Point  point );
    const double distancFromPolygoneToLine( in  Line  line );
    const double distanceFromPolygonToPolygon( in  Polygon  boundary );
    const double perimeter( );
    const double Area() ;
    const void print() ;
    void deleteAll();
    const void getBoundingBox(inout nbox_t box);
    boolean ConvexIntersect(in  Polygon  P,
        in  Polygon  Q, in  Pool  points_pool );
    boolean ConvexPlus(in  Polygon  P,
```

```
            in  Polygon  Q, in  Pool  points_pool );
        boolean ConvexMinus(in  Polygon  P,
            in  Polygon  Q, in  Pool  points_pool );

        private:
        const double Area2() ;
        void copy( in  Polygon  Q );

    };
    class Layer
    {
        public:
        attribute set < Polygon >  regions;
        attribute set < Point >  pointpool;

        boolean addPolygon(in Polygon region);
        boolean deletePolygon(in Polygon region);
    };
    class Network
    {
        public:
        attribute set< Line > lines;
        attribute set< Point > pointpool;

        boolean addLine(in Line line);
        attribute deleteLine(in Line line);
    };
};
```

Appendix B

Spatial Queries

The typical queries supported in Oshadhi for the geo-spatial data set can be on any of the spatial relationships mentioned in chapter 2. These are implemented as the member functions of the spatial data classes as shown in appendix A. This appendix discusses all those member functions with appropriate example queries.

B.1 Inside Queries

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

B.1.1 IsPointOnLine

query: Is point p(x,y) is on the line 'l'.
This query finds whether the point given by longitude and latitude, is on the given line 'l'.

B.1.2 IsPointInsidePolygon

query: Is point p(x,y) inside the polygon 'r'.
This query finds whether the point given by longitude and latitude, is inside the given polygon 'r'.

B.1.3 IsLineInsidePolygon

query: Is the line 'l' inside the polygon 'p'.

This query finds if the line 'l' is completely inside the polygon 'p'.

B.1.4 IsPolygonInsidePolygon

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

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

B.2 Overlapping Queries

This set of queries find all the objects which overlap the given polygon or line. These queries may return either a boolean value or a set spatial objects as the result.

B.2.1 IsLineInstersectingLine

query: Is line 'line1' intersecting the line 'line2'.

This query finds if the two given lines are intersecting.

B.2.2 IsLineIntersectingPolygon

query: Is the line 'line1' intersecting the polygon 'p'.

This query finds if the line is intersecting the given polygon. This can either return a boolean value or a set of points as the result.

B.2.3 IsPolygonIntersectingPolygon

query: Is the polygon 'p1' overlapping the polygon 'p2'.

This query finds whether the two polygons are overlapping. This may return a boolean value or a polygon object as the result.

B.3 Adjacency Queries

This set of queries find objects which are adjacent to the given object. These queries may return either boolean value or a set of objects that are satisfying the query.

B.3.1 IsLineMeetingLine

query: Is the line 'abc' meeting the line 'bde'.
This query finds whether the two lines are meeting. i.e checks whether the end point of one line is on the other line. It can be observed that this is different from the intersection of two lines.

B.3.2 IsLineMeetingPolygon

query: Is the line 'abc' meeting the polygon 'cdefghc'.
This query finds whether any of the end points of the line 'abc' falls on the boundary of the polygon 'cdefghc'.

B.3.3 IsPolygonAdjacentToPolygon

query: Is the polygon 'abcdefgh' adjacent to polygon 'bcdpqrs'.
This query finds whether the polygon 'abcdefgh' is sharing boundary with the polygon 'bcdpqrs'. This can return boolean value or a line.

Along with the above types of queries, the queries to find disjointness, equality and the nearest distance between the spatial objects are also provided in Oshadhi.